## Contents

- **Table of contents**  
i
- **List of figures**  
v
- **List of tables**  
vi
- **List of algorithms**  
vii
- **Preface**  
viii
- **Syllabus**  
ix
- **Lecture schedule**  
xii

### 1 Introduction
1

### 2 Problems and languages
3

### 3 Models of computation
5

#### 3.1 Turing machines
5

##### 3.1.1 Computations
7

##### 3.1.2 Complexity
7

##### 3.1.2.1 Asymptotic notation
8

##### 3.1.3 Programming a Turing machine
9

##### 3.1.3.1 Example of computing a function
10

##### 3.1.3.2 Example of computing a predicate
12

##### 3.1.4 Turing machine variants
12

##### 3.1.5 Limitations of simulations
16

##### 3.1.6 Universal Turing machines
19

#### 3.2 Random access machines
20

#### 3.3 The extended Church-Turing thesis
22
## CONTENTS

4 Time and space complexity classes 23

5 Nonterminism and NP 25
   5.1 Examples of problems in NP 27
   5.2 Reductions and NP-complete problems 28
   5.3 The Cook-Levin theorem 29
   5.4 More NP-complete problems 30
      5.4.1 1-IN-3 SAT 31
      5.4.2 SUBSET SUM and PARTITION 31
      5.4.3 Graph problems 33
         5.4.3.1 Reductions through INDEPENDENT SET 33
         5.4.3.2 GRAPH 3-COLORABILITY 35
         5.4.3.3 Reductions involving Hamiltonian paths 36
   5.5 coNP and coNP-completeness 36
   5.6 Relation to EXP 37

6 Diagonalization 38
   6.0.1 Undecidability of the Halting Problem 38
   6.1 Hierarchy theorems 40
      6.1.1 The Space Hierarchy Theorem 40
      6.1.2 The Time Hierarchy Theorem 42
   6.2 Hierarchy theorems for nondeterminism 46
   6.3 Ladner's Theorem 46

7 Oracles and relativization 50
   7.1 Oracle machines 50
   7.2 Relativization 50
      7.2.1 The Baker-Gill-Solovay Theorem 51
      7.2.2 The Random Oracle Hypothesis 51
      7.2.3 How not to relativize 51
   7.3 The oracle polynomial-time hierarchy 51

8 Alternation 53
   8.1 The alternating polynomial-time hierarchy 53
   8.2 Equivalence to alternating Turing machines 53
   8.3 Complete problems 53
   8.4 Equivalence to oracle definition 54
   8.5 PH ⊆ PSPACE 54
CONTENTS

9 Space complexity .................................................. 56
  9.1 Space and time ................................................. 57
  9.2 \textbf{PSPACE} and \textbf{TQBF} ............................... 57
  9.3 Savitch’s Theorem ........................................... 58
  9.4 The Immerman-Szelepcsényi Theorem ...................... 58
  9.5 Oracles and space complexity .............................. 59
  9.6 \(L = \text{NL} = \text{AL} = \text{P} \) ............................. 60
    9.6.1 Complete problems with respect to log-space reductions 60
      9.6.1.1 Complete problems for \text{NL} .................. 61
      9.6.1.2 Complete problems for \text{P} .................. 62
  9.6.2 \(\text{AL} = \text{P} \) ........................................ 63
10 Circuit complexity ................................................ 65
  10.1 Polynomial-size circuits .................................. 66
    10.1.1 \( \text{P/poly} \) ........................................ 67
    10.1.2 Information-theoretic bounds .......................... 67
    10.1.3 The Karp-Lipton Theorem .............................. 68
  10.2 Uniformity ................................................... 69
  10.3 Bounded-depth circuits .................................... 69
    10.3.1 Parallel computation and \textbf{NC} ...................... 70
    10.3.2 Relation to \text{L} and \text{NL} ....................... 70
    10.3.3 Barrington’s Theorem .................................. 71
    10.3.4 \text{PARITY} \notin \textbf{AC}^0 .......................... 73
      10.3.4.1 Håstad’s Switching Lemma ........................ 73
      10.3.4.2 Application to \text{PARITY} ..................... 75
      10.3.4.3 Low-degree polynomials .......................... 77
  10.4 Monotone circuits .......................................... 80
11 Natural proofs .................................................... 81
  11.1 Natural properties ......................................... 81
  11.2 Pseudorandom function generators ....................... 82
  11.3 The Razborov-Rudich Theorem .............................. 82
  11.4 Examples of natural proofs ............................... 83
12 Counting classes ................................................ 85
13 Descriptive complexity .......................................... 86
14 Toda’s Theorem .................................................. 87
15 Interactive proofs ................................................ 88
CONTENTS

16 Probabilistically-checkable proofs

A Assignments

A.1 Assignment 1: due Wednesday, 2017-02-01 at 23:00 . . . . . . 90
A.1.1 Bureaucratic part . . . . . . . . . . . . . . . . . . . . . 90
A.1.2 Binary multiplication . . . . . . . . . . . . . . . . . . . . 90
A.1.3 Transitivity of $O$ and $o$ . . . . . . . . . . . . . . . . . 93

A.2 Assignment 2: due Wednesday, 2017-02-15 at 23:00 . . . . . . 93
A.2.1 A log-space reduction . . . . . . . . . . . . . . . . . . . . 93
A.2.2 Limitations of two-counter machines . . . . . . . . . . . 94
A better solution: . . . . . . . . . . . . . . . . . . . . . . . . . . . 96

A.3 Assignment 3: due Wednesday, 2017-03-01 at 23:00 . . . . . . 96
A.3.1 A balanced diet of hay and needles . . . . . . . . . . . . . 96
A.3.2 Recurrence . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

A.4 Assignment 4: due Wednesday, 2017-03-29 at 23:00 . . . . . . 98
A.4.1 Finite-state machines that take advice . . . . . . . . . . . 98
A.4.2 Binary comparisons . . . . . . . . . . . . . . . . . . . . . . 98

A.5 Assignment 5: due Wednesday, 2017-04-12 at 23:00 . . . . . . 98
A.6 Assignment 6: due Wednesday, 2017-04-26 at 23:00 . . . . . . 99

Bibliography

Index
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Turing-machine transition function for reversing the input, disguised as a C program</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Recognizing a palindrome using a work tape</td>
<td>13</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Turing-machine transition function for reversing the input</td>
<td>10</td>
</tr>
<tr>
<td>A.1 Transition table for multiplying by 3 (LSB first)</td>
<td>92</td>
</tr>
</tbody>
</table>
List of Algorithms

A.1 Log-space reduction from INDEPENDENT SET to CLIQUE . 95
Preface

These are notes for the Spring 2017 semester of the Yale course CPSC 468/568 Computational Complexity. This document also incorporates the lecture schedule and assignments. Because this is a work in progress, it will be updated frequently over the course of the semester.

Updated versions of these notes will appear at http://www.cs.yale.edu/homes/aspenes/classes/468/notes.pdf. If the Yale CS server becomes inaccessible, a backup copy can be found at https://www.dropbox.com/sh/j98y7z3k7u91o/bAAAglmWGH5gdyKoi3rSJewa?dl=0.

The Spring 2016 version of the course was taught by Joan Feigenbaum, and the organization of this course is in part based on this previous version. Information about the Spring 2016 course, including lecture notes and assignments, can be found at http://zoo.cs.yale.edu/classes/cs468/.

Much of the course follows the textbook, *Computational Complexity: A Modern Approach*, by Sanjeev Arora and Boaz Barak. In most cases you’ll find this textbook contain much more detail than what is presented here, so it is probably better to consider these notes a supplement to it rather than to treat them as your primary source of information.
Syllabus

Description

Introduction to the theory of computational complexity. Basic complexity classes, including polynomial time, nondeterministic polynomial time, probabilistic polynomial time, polynomial space, logarithmic space, and nondeterministic logarithmic space. The roles of reductions, completeness, randomness, and interaction in the formal study of computation. After Computer Science 365 or with permission of the instructor.

Meeting times

Monday and Wednesday 1:00–2:15 in AKW 000.

On-line course information

The lecture schedule, course notes, and all assignments can be found in a single gigantic PDF file at http://www.cs.yale.edu/homes/aspnes/classes/468/notes.pdf. You should probably bookmark this file, as it will be updated frequently.

For office hours, see http://www.cs.yale.edu/homes/aspnes#calendar.

Staff

The instructor for the course is James Aspnes. Office: AKW 401. Email: james.aspnes@gmail.com URL: http://www.cs.yale.edu/homes/aspnes/

The for the course is Stanislaw Swidwinski. Email: stanislaw.swidwinski@yale.edu
Textbook

The textbook for the class is:

The status of this book is a little complicated. It is available on-line in an inconvenient format from Yale campus IP addresses at [http://proquest.safaribooksonline.com/9780511530753](http://proquest.safaribooksonline.com/9780511530753). A draft version in PDF format is also available at [http://theory.cs.princeton.edu/complexity/book.pdf](http://theory.cs.princeton.edu/complexity/book.pdf), but this is not identical to the final published version. The final published version is not currently available in printed form. So we will mostly be working from the PDF draft. Where it makes a difference, in the notes we will cite the PDF draft as [AB07] and the print version as [AB09], and we will try to avoid citing the unavailable print version whenever possible.

Reserved books at Bass library

In addition to the textbook, the following books are on reserve at Bass Library:


The first two are other widely-used computational complexity theory textbooks, which may offer perspectives on various topics that complement Arora-Barak and the course notes. The last is a classic collection of known NP-hard problems, and can be helpful as a starting point for checking if some problem you are interested is also NP-hard.

Other useful resources

- [https://complexityzoo.uwaterloo.ca/Complexity_Zoo](https://complexityzoo.uwaterloo.ca/Complexity_Zoo) On-line catalog of complexity classes.

Course requirements

Six homework assignments (60% of the semester grade) plus a final exam (40%).

Use of outside help

Students are free to discuss homework problems and course material with each other, and to consult with the instructor or a TA. Solutions handed in, however, should be the student’s own work. If a student benefits substantially from hints or solutions received from fellow students or from outside sources, then the student should hand in their solution but acknowledge the outside sources, and we will apportion credit accordingly. Using outside resources in solving a problem is acceptable but plagiarism is not.

Clarifications for homework assignments

From time to time, ambiguities and errors may creep into homework assignments. Questions about the interpretation of homework assignments should be sent to the instructor at james.aspnes@gmail.com. Clarifications will appear in an updated version of the assignment.

In some circumstances, you may be able to get a faster response using Piazza, at http://piazza.com/yale/spring2017/cpsc468. Note that questions you ask there are visible to other students if not specifically marked private, so be careful about broadcasting your draft solutions.

Late assignments

Late assignments will not be accepted without a Dean’s Excuse.
Lecture schedule

As always, the future is uncertain, so you should take parts of the schedule that haven’t happened yet with a grain of salt. Readings refer to chapters or sections in the course notes, except for those specified as in AB, which refer to the course textbook [AB07].

Office hours, lecture times, and assignment due dates can be found at http://www.cs.yale.edu/homes/aspnes#calendar.

2017-01-18 Computational complexity theory. What the course is about.
Languages and complexity classes. One-tape and $k$-tape Turing machines. Review of asymptotic notation. Readings: Chapters 1 and 2, §3.1 up to §3.1.4; AB §1.1 and 1.2.

2017-01-20 Complexity classes $\text{TIME}(f(n))$, $\text{SPACE}(f(n))$, and $P$. Turing machine simulations: one-tape, two-tape, two counters. Crossing sequence arguments and lower bounds on single-tape Turing machines. The extended Church-Turing thesis. How to simulate anything with a Turing machine. Readings: Rest of Chapter 3; AB covers some of this in §1.2.2 but doesn’t give much detail.

2017-01-23 Nondeterminism and $\text{NP}$. Reductions, $\text{NP}$-complete problems, and the Cook-Levin theorem. Some flaming about the consequences of $P = \text{NP}$ if true. Readings: Chapter 5 through §5.3, AB §§2.1–2.3, 2.7. Not required, but if you are a big fan of $P = \text{NP}$, some other resources are Aaronson’s very recent (and very large) survey [Aar17] and a classic paper by Impagliazzo on what happens to cryptography and other fields if $P = \text{NP}$ [Imp95].

2017-01-25 Various $\text{NP}$-complete problems and examples of reductions. $\text{NP}$-complete languages and $\text{coNP}$-complete languages. Readings: §§5.3 and section-coNP; AB §§2.4 and 2.6.1.
2017-01-30 Diagonalization: Undecidability of the Halting Problem, the Space Hierarchy Theorem. Readings: Chapter 6 through §6.1.1; AB §3.2.

It’s probably worth mentioning that in my excitement over restricting $L$ to one-work-tape machines I forgot to have $R^*$ test if $M$ halts (which we can do for space-bounded machines, by using a counter to keep track of whether we have run long enough to repeat a configuration). See §6.1.1 for a working version of the upper-bound construction.

2017-02-01 More diagonalization: the Time Hierarchy Theorem, preview of Ladner’s Theorem. Some impromptu discussion of the polynomial-time hierarchy, which is not related to the THT but which sounds like it should be. Readings: §6.1.2; AB §3.1.


2017-02-08 Proof of Baker-Gill-Solovay: oracles with $P^A = NP^A$ and $P^B \neq NP^B$. Oracles as subroutines and the polynomial-time oracle hierarchy. Readings: §§7.2.1 and 7.3; AB §§3.5, 5.1 and 5.5.

2017-02-13 Alternation. Equivalence of oracle poly-time hierarchy and alternating poly-time hierarchy. Complete problems for various levels. Preview of TQBF and $PSPACE$. Readings: Chapter 8; AB §§5.2 and 5.3.

2017-02-15 Space complexity: $PSPACE$ and $PSPACE$-completeness of TQBF. Savitch’s theorem and the Immerman-Szelepcsényi theorem. The very flat log-space (alternating) hierarchy and the even flatter poly-space hierarchy. Readings: Chapter 9; AB [[To be announced.]]

2017-02-20 A bit about the log-space oracle hierarchy: $NL^{NL} = NL$. $L$ vs $NL$ vs $P$: $NL$-complete and $P$-complete problems, $AL = P$. Readings: Sections 9.5 and 9.6; AB [[To be announced.]]

LEcTURE SCHEDULE

2017-02-27 Restricted circuit complexity classes: $\text{AC}^i$, $\text{NC}^i$, and $\text{NC}$. Branching programs and Barrington’s Theorem. Readings: §10.3 through §10.3.3; AB §§6.5 and 13.4.4.

2017-03-01 Håstad’s Switching Lemma. Readings: §§10.3.4.1; AB §13.1.

2017-03-06 $\text{PARITY} \not\in \text{AC}^0$ (using Håstad’s Switching Lemma). $\text{PARITY} \not\in \text{AC}^0$ plus mod-$p$ gates (using the Razborov-Smolensky approximation by low-degree polynomials). Readings: §§10.3.4.2 and 10.3.4.3; AB §13.2.

2017-03-08 Natural proofs. Readings: Chapter 11; AB Chapter 22.

2017-03-27 Counting classes: PP, BPP, and friends. Readings: Chapter 12; AB [[ To be announced. ]]．

2017-03-29 [[[ To be announced. ]]]

2017-04-03 [[[ To be announced. ]]]

2017-04-05 [[[ To be announced. ]]]

2017-04-10 [[[ To be announced. ]]]

2017-04-12 [[[ To be announced. ]]]

2017-04-17 [[[ To be announced. ]]]

2017-04-19 [[[ To be announced. ]]]

2017-04-24 [[[ To be announced. ]]]

2017-04-26 [[[ To be announced. ]]]

2017-05-06 Final exam, starting at 2:00pm, in a location to be determined by the Registrar. It will be a closed-book test covering all material discussed during the semester.
Chapter 1

Introduction

The basic question that computational complexity theory \footnote{When I started in this business, the field was known as just complexity theory. But “complexity theory” is now often used to refer to the study of complex systems, a field of research strongly associated with the Santa Fe Institute. At its best, this field represents some of the finest minds in physics seeking to cope with the realization that the behavior of much of the universe is nonlinear and cannot be described succinctly. At its worst, it consists of popular-science writers looking at pictures of fractals and saying “Wow, those things are really complex!” We use the term “computational complexity theory” to avoid confusion with this largely unrelated area of research, although when not being careful we will often drop the “computational” part.} tries to answer is:

Given a problem $X$, and a machine model $M$, how long does it take to solve $X$ using a machine from $M$?

Unfortunately we can only rarely answer this question. So the real questions of complexity theory are:

1. How can we classify problems into complexity classes based on their apparent difficulty?
2. What relationships can we established between these complexity classes?
3. What techniques might we be able to use to resolve relationships between complexity classes whose status is still open?

The most famous of these questions center around the $P$ vs. $NP$ problem. Here $P$ consists of problems that can be solved in time polynomial in the size of the input on reasonable computational devices (we will see a more formal definition in Chapter \footnote{When I started in this business, the field was known as just complexity theory. But “complexity theory” is now often used to refer to the study of complex systems, a field of research strongly associated with the Santa Fe Institute. At its best, this field represents some of the finest minds in physics seeking to cope with the realization that the behavior of much of the universe is nonlinear and cannot be described succinctly. At its worst, it consists of popular-science writers looking at pictures of fractals and saying “Wow, those things are really complex!” We use the term “computational complexity theory” to avoid confusion with this largely unrelated area of research, although when not being careful we will often drop the “computational” part.}), $NP$ consists of problems whose solutions can
be verified in time polynomial in the size of the input (Chapter 5), and the big question is whether there are any problems in \( \text{NP} \) that are not also in \( \text{P} \). Much of the development of computational complexity theory has arisen from researchers working very hard to answer this question.

Computational complexity theory also includes questions about other computational resources. In addition to time, we can ask about how much \textbf{space} it takes to solve a particular problem. The space class analogous to \( \text{P} \) is \( \text{L} \), the class of problems that can be solved using space logarithmic in the size of the input. Because a log-space machine can only have polynomially-many distinct states, any problem solvable in \( \text{L} \) is also solvable in \( \text{P} \), and indeed \( \text{L} \) is the largest space complexity class that includes only problems we can expect to solve efficiently in practice. As with \( \text{P} \), a big open question is whether solving problems in log space is any harder than checking the solutions. In other words, is \( \text{L} \) equal to its nondeterministic counterpart \( \text{NL} \)? Other classes of interest include \( \text{PSPACE} \), the class of problems solvable using polynomial space, which contains within it an entire hierarchy of classes that are to \( \text{NP} \) what \( \text{NP} \) is to \( \text{P} \) \(^2\).

In addition to asking about deterministic computations, we can also ask about what happens with randomness. The class \( \text{BPP} \) of bounded-error probabilistic polynomial-time computations, in which we must produce the right answer substantially more often than not in polynomial time, corresponds to typical randomized algorithms. It is an open question whether \( \text{BPP} = \text{P} \) or not. Cryptographers tend to think they are equal (a good pseudorandom number generator will let you simulate \( \text{BPP} \) in \( \text{P} \)), but we don’t really know. A similar question arises for \( \text{BQP} \), the class of problems solvable with bounded error in polynomial time using a quantum computer. Quantum computers seem pretty powerful, but as far as we know, \( \text{BQP} \) could be equal to \( \text{P} \).

Complexity theory is a huge field, probably the most technically developed part of theoretical computer science, and we won’t be able to cover all of it. But the hope is that this course will give you enough of an introduction to the basic ideas of complexity theory that if you are interested, you will be able to pursue it further on your own.

\(^2\)Which is to say: possibly equal but probably not.
Chapter 2

Problems and languages

For the most part, the kind of problems we study in complexity theory are decision problems, where we are presented with an input \( x \) and have to answer “yes” or “no” based on whether \( x \) satisfies some predicate \( P \). An example is GRAPH 3-COLORABILITY: given a graph \( G \), is there a way to assign three colors to the vertices so that no edge has two endpoint of the same color?

Most of the algorithms you’ve probably seen have computed actual functions instead of just solving a decision problem, so the choice to limit ourselves (mostly) to decision problems requires some justification. The main reason is that decision problems are simpler to reason about than more general functions, and our life as complexity theorists is hard enough already. But we can also make some plausible excuses that decision problems in fact capture most of what is hard about function computation.

For example, if we are in the graph-coloring business, we probably want to find a coloring of \( G \) rather than just be told that it exists. But if we have a machine that tells use whether or not a coloring exists, with a little tinkering we can turn it into a machine that tells us if a coloring exists consistent with locking down a few nodes to have particular colors. With this modified machine, we can probe potential colorings one vertex at a time, backing off if we place a color that prevents us from coloring the entire graph. Since we have to call the graph-coloring tester more than once, this is more expensive than the original decision problem, but it will still be reasonably efficient if

---

1It’s conventional to name problems in all-caps.

2Since we can’t necessarily rewrite the code for our graph-coloring tester, this involves adjusting the input graph. The basic idea is that we can add a triangle off on the side somewhere that gives us nodes with our three colors, and force a node to have a particular color by linking it to the two other colors in the triangle.
our algorithm for the decision problem is.

Concentrating on decision problems fixes the outputs of what we are doing. We also have to formalize how we are handling inputs. Typically we assume that an instance $x$ of whatever problem we are looking at has an encoding $\langle x \rangle$ over some alphabet $\Sigma$, which can in principle always be reduced to just $\{0, 1\}$. Under this assumption, the input tape contains a sequence of symbols from $\Sigma$ bounded by an infinite sequence of blanks in both directions. The input tape head by convention starts on the leftmost symbol in the input.

We typically don’t care too much about the details of this encoding, as long as (a) it is reasonably efficient in its use of space (for example, we’d encode a natural number like 19 using its binary representation $\langle 19 \rangle = 10011$ instead of its unary representations $1^{19} = 1111111111111111111$); and (b) it makes the features of the input we want to get at reasonably accessible (if we want to encode two primes, we represent them something like $\langle 17, 19 \rangle = 10001, 10011$ instead of $\langle 17 \cdot 19 \rangle, 323 = 101000011$, even though the latter representation in principle lets us recover the former). The justification for being blasé about the details is that we should be able to convert between reasonable representations at low cost; so if we can solve a problem efficiently for one representation of the input, we should also be able to solve it efficiently for any other reasonable representations, and if we can’t, that’s a sign that there may be something wrong with our problem.

Summing up, on the output side we consider yes/no outputs only, and on the input side we insist that all inputs are encoded as strings of symbols. We can formalize this idea by defining a language as a set of finite strings over some alphabet $\Sigma$. If $x \in L$ then are supposed to answer “yes” and if $x \not\in L$, we are supposed to answer “no.” An implementation of a language is a machine of some sort that does this correctly, and a complexity class will just be a set of languages whose implementations have some particular complexity property.

If we have a Turing machine $M$ that halts in an accepting state on any input $x \in L$ and halts in a rejecting state on any input $x \not\in L$, we say that $M$ decides $L$. Each $M$ that halts on all inputs decides exactly one language $L$, which we write as $L(M)$.

---

3 The technical term here is that we want our problems to be representation independent, which is borrowed from the theory of abstract data types; the idea is that we want the meaning of the problem to depend on $x$ and not a particular choice of $\langle x \rangle$. 
Chapter 3
Models of computation

Many models of computation have been proposed over the years. The Church-Turing thesis is the observation that all of the ones that are sufficiently powerful and that can plausibly be realized in practice are capable of computing the same predicates. The extended Church-Turing thesis says that all reasonable computational models can simulate each other with slowdown at most polynomial in the duration of the computation.

Taken together, these let us ignore the question of precisely which model of computation we are using, and we can settle on whatever model we find convenient (for example, C programs). But when reasoning about specific computations, it can be helpful to settle on a single, mathematically simple model. This usually ends up being a Turing machine.

3.1 Turing machines

A Turing machine (TM for short) consists of one or more a tapes, which we can think of as storing an infinite (in both directions) array of symbols from some alphabet Γ, one or more heads, which point to particular locations on the tape(s), and a finite-state control that controls the movement of the heads on the tapes and that may direct each head to rewrite the symbol in the cell it currently points to, based on the current symbols under the heads and its state, an element of its state space Q.

\footnote{Alternatively, the Church-Turing thesis is a declaration of what models we will consider to be plausibly realizable. This roughly means that we can only do a finite amount of work in a finite amount of time, a restriction that may not hold if, for example, you live very close to a rotating black hole or have the ability to carry out supertasks in apparent violation of the usual laws of physics.}
In its simplest form, a Turing machine has exactly one tape that is used for input, computation, and output, and has only one head on this tape. This is often too restrictive to program easily, so we will typically assume at least three tapes (with corresponding heads): one each for input, work, and output. This does not add any significant power to the model, and indeed not only is it possible for a one-tape Turing machine to simulate a \( k \)-tape Turing machine for any fixed \( k \), it can do so with only polynomial slowdown. Similarly, even though in principle we can limit our alphabet to just \{0, 1\}, we will in general assume whatever (finite) alphabet is convenient for the tape cells.

Formally, we can define a \( k \)-tape Turing machine as a tuple \( \langle \Gamma, Q, \delta \rangle \), where \( \Gamma \) is the alphabet; \( Q \) is the state space of the finite-state controller; and \( \delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, S, R\}^k \) is the transition function, which specifies for each state \( q \in Q \) and \( k \)-tuple of symbols from \( \Gamma \) seen at the current positions of the heads the next state \( q' \in Q \), a new tuple of symbols to write to the current head positions, and a direction Left, Stay, or Right to move each head in.

More formal definitions of a Turing machine add additional details to the tuple, including an explicit blank symbol \( b \in \Gamma \), a restricted input alphabet \( \Sigma \subseteq \Gamma \) (which generally does not include \( b \), since the blank regions of the input tape mark the ends of the input), an explicit starting state \( q_0 \in Q \), and an explicit list of accepting states \( A \subseteq Q \). We will include these details as needed.

To avoid confusion between the state \( q \) of the controller and the state of the Turing machine as a whole (which includes the contents of the tapes and the positions of the heads as well), we will describe the state of the machine as a whole as its configuration and reserve state for just the part of the configuration that represents the state of the controller.

Because we allow the Turing machine to do nothing, we do not need to include an explicit halting state. Instead, we will define the machine to halt if it reaches a configuration where it does not move its heads, change its state, or change any of the tape symbols.

---

2This definition mostly follows the one in §1.2 of Arora and Barak [AB07]. One difference is that we allow the machine to write to all \( k \) of its tapes, while Arora and Barak reserve the first tape as an input tape and thus define the transition function as \( \delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^{k-1} \times \{L, S, R\}^k \). The advantage of the more general definition is that it allows for a one-tape machine where the single tape function in all three roles of input, storage, and output. The disadvantage is that if we do want a read-only input tape, which is important when defining sublinear space complexity classes like \( L \), we must explicitly require that \( \delta \) always write to the first tape whatever symbol is already there.
3.1.1 Computations

A computation of a predicate by a Turing machine proceeds as follows:

1. We start the machine in an initial configuration where the first tape contains the input. For convenience we typically assume that this is bounded by blank characters that cannot appear in the input. The head on the first tape starts on the leftmost input symbol. All cells on all tapes, other than the cells representing the input, start off with a blank symbol. The controller starts in the initial state $q_0$.

2. Each step of the computation consists of reading the $k$-tuple of symbols under the heads on each of the tapes, then rewriting these symbols, updating the state, and moving the heads according to the transition function.

3. This continues until the machine halts, which we defined above as reaching a configuration that doesn’t change as a result of applying the transition function.

3.1.2 Complexity

The time complexity of an execution is the number of steps until the machine halts. Typically we will try to bound the time complexity as a function of the size $n$ of the input, defined as the number of cells occupied by the input, excluding the infinite number of blanks that surround it.

The space complexity is the number of tape cells used by the computation. We have to be a little careful to define what it means to use a cell. A naive approach is just to count the number of cells that hold a non-blank symbol at any step of the computation, but this allows cheating (on a multi-tape Turing machine) because we can simulate an unbounded counter by writing a single non-blank symbol to one of our work tapes and using the position of the head relative to this symbol as the counter value. So instead we will charge for any cell that a head ever occupies, whether it writes a non-blank symbol to that cell or not.

An exception is that if we have a read-only input tape and a write-only output tape, we will only charge for cells on the work tapes. This allows space complexity sublinear in $n$.

---

3This counter abstraction only supports the operations increment, decrement, and test for zero, but two such counters are enough to simulate an unbounded ordinary memory using a construction of Minsky [Min61], and indeed Minsky’s original construction is described in terms of a 2-tape TM with only one non-black cell per tape.
CHAPTER 3. MODELS OF COMPUTATION

3.1.2.1 Asymptotic notation

In computing time and space complexities, we want to ignore constant factors and performance on small inputs, because we know that constant factors depend strongly on features of our model that we usually don’t care much about, and performance on small inputs can always be faked by baking a large lookup table into our finite-state controller. As in the usual analysis of algorithms, we get around these issues by expressing performance using asymptotic notation. This section gives a brief review of asymptotic notation as used in algorithm analysis and computational complexity theory.

Given two non-negative functions \( f(n) \) and \( g(n) \), we say that:

- \( f(n) = \Theta(g(n)) \) if there exist constants \( c_1 > 0 \), \( c_2 > 0 \), and \( N \) such that \( c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \) for all \( n \geq N \).

- \( f(n) = O(g(n)) \) if there exist constants \( c > 0 \) and \( N \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq N \).

- \( f(n) = \Omega(g(n)) \) if there exist constants \( c > 0 \) and \( N \) such that \( f(n) \geq c \cdot g(n) \) for all \( n \geq N \).

- \( f(n) = o(g(n)) \) if for any constant \( c > 0 \), there exists a constant \( N \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq N \).

- \( f(n) = \omega(g(n)) \) if for any constant \( c > 0 \), there exists a constant \( N \) such that \( f(n) \geq c \cdot g(n) \) for all \( n \geq N \).

Note that we are using the equals sign in a funny way here. The convention is that given an asymptotic expression like \( O(n) + O(n^2) = O(n^3) \), the statement is true if for all functions we could substitute in on the left-hand side in each class, there exist functions we could substitute in on the right-hand side to make it true.

4This limitation is convenient for talking about time and space complexity, because we don’t know how to make anything run using negative time or space. In other contexts, like in analysis (the branch of mathematics), it’s common to allow \( f(n) \) or \( g(n) \) to be negative or even complex-valued, and just put in absolute values everywhere to make the definitions make sense.

5This particular claim happens to be true: to prove this, we would need to show that if \( f(n) = O(n) \) and \( g(n) = O(n^2) \), then there is some \( h(n) = O(n^3) \) such that \( f(n) + g(n) = h(n) \). The good news is that we don’t have to guess what \( h(n) \) is once we see \( f(n) \) and \( g(n) \). The bad news is that, after unpacking the definition of \( O(\cdot) \) in each case, we have at least six different constants to wrangle to show that \( h(n) = O(n^3) \).
CHAPTER 3. MODELS OF COMPUTATION

Intuitively, it may help to think of the five asymptotic operators $o$, $O$, $\Theta$, $\Omega$, and $\omega$ as mapping to the five comparison relations $<$, $\leq$, $=$, $\geq$, and $>$. When we say $f(n) = o(g(n))$, we mean that $f(n)$ grows strictly more slowly than $g(n)$; $f(n) = O(g(n))$ means it grows no faster than $g(n)$; $f(n) = \Theta(g(n))$ means it grows at about the same rate; etc.; where in each case when we talk about rate of growth we mean the rate of growth ignoring constant factors and small inputs.

If you are familiar with limits, it is also possible to define $f(n) = o(g(n))$ as $\lim_{n \to \infty} f(n)/g(n) = 0$, and similarly $f(n) = \omega(g(n))$ as $\lim_{n \to \infty} g(n)/f(n) = 0$, with the caveat that bad things may happen if $f(n)$ or $g(n)$ are ever 0. This doesn’t work so well for something like $f(n) = O(g(n))$. For example, the function

$$f(n) = \begin{cases} n & \text{when } n \text{ is odd} \\ 2n & \text{when } n \text{ is even} \end{cases}$$

is $\Theta(n)$ (it’s always between $n$ and $2n$), but $\lim_{n \to \infty} f(n)/n$ doesn’t exist since $f(n)/n$ oscillates between 1 and 2.

When expressing complexities in asymptotic form, we usually try to keep the function inside the big $O$ as simple as possible. This means eliminating constants and terms that are dominated by other terms. So complexities like $O(n)$, $O(n \log n)$, $O(n^5)$, and $O(2^n \cdot n^2 \log^5 n)$ are all things you are likely to see in actual papers, but $O(3n)$, $O(n^2 + n)$, and $O(2^n + n^2)$ are not.

3.1.3 Programming a Turing machine

Although one can in principle describe a Turing machine program by giving an explicit representation of $\delta$, no sane programmer would ever want to do this. I personally find it helpful to think about TM programming as if I were programming in a C-like language, where the tapes correspond to (infinite) arrays of characters and the head positions correspond to (highly-restricted) pointers. The restrictions on the pointers are that we can’t do any pointer operations other than post-increment, post-decrement, dereference, and assignment through a dereference; these correspond to moving the head left, moving the head right, reading a tape cell, and writing a tape cell.

The state of the controller represents several aspects of the program. At minimum, the state encodes the current program counter (so this approach only works for code that doesn’t require a stack, which rules out recursion and many uses of subroutines). The state can also be used to hold a finite number of variables that can take on a finite number of values.
Table 3.1: Turing-machine transition function for reversing the input. In lines where $x$ appears in the input tape, it is assumed that $x$ is not zero. Note this is an abbreviated description: the actual transition function would not use variables $x$ and $y$ but would instead expand out all $|\Sigma|$ possible values for each.

### 3.1.3.1 Example of computing a function

For example, Figure 3.1 is a C program that reverses its input tape to its output tape. The assumption is that blank symbols (including the ends of the input) are represented by null characters. Because this program uses no local variables other than the head position pointers, the state only needs to represent the program counter. A representation of the corresponding transition function is given in Figure 3.1.

Realistically, nobody would ever write out either of these representations, unless they were really trying to be careful about counting states. Instead, a claim that a Turing machine can reverse its input would probably be described in terms of a less formal algorithm:

1. Move the input and output heads to the right until the input head reaches the end of the input.
2. Move the output head back one cell to the left.
3. Move the input head back to the leftmost cell of the input.
4. Copy the input to the output, one cell at a time, moving the input head right and the output head left after each step.

---

6Note that this function won’t work on standard C strings, which are (a) not null-terminated on both sides, and (b) not stored in arrays that are infinite in both directions. Objection (a) is easily dealt with by demanding null-termination at the start from the caller. Objection (b) can in principle be dealt with using a clever dynamically-allocated data structure and C++ overloading magic. As far as I know, nobody has ever bothered to do this.
void reverse(char *input, char *work, char *output) {
    /* move output head to past last position of output */
    /* state 0 */
    while(*input != 0) {
        output++;
        input++;
    }

    /* pull output head left one cell to actual last position */
    /* start moving input head to leftmost input symbol */
    /* state 0 */
    output--;
    input--; 

    /* return input head to leftmost input symbol */
    /* state 1 */
    while(*input != 0) {
        input--;
    }

    /* state 1 */
    input++;

    /* copy input to output in reverse order */
    /* state 2 */
    while(*input != 0) {
        *output = *input;
        input++;
        output--;
    }

    /* HALT */
}

Figure 3.1: Turing-machine transition function for reversing the input, disguised as a C program
CHAPTER 3. MODELS OF COMPUTATION

As long as it’s clear that each of these steps only requires reading and writing cells under the current head positions, each move only moves a head at most one cell left or right, and the number of states needed to keep track of everything is finite,

3.1.3.2 Example of computing a predicate

Here’s an example of a predicate computed by a Turing machine. Note that here we don’t use the output tape. The halting state determines whether we accept the input or not.

1. Copy the input from the input tape to the work tape, leaving both the input and work tape heads on the blank cell to the right.

2. Move the work tape head back to the blank cell to the left of the copy of the input.

3. Finally, walk the work tape head right across the copy while walking the input tape head left across the input. If we see mismatched symbols before reaching a blank, halt and reject. If we reach the blanks at the end, halt and accept.

Note that most of the “steps” in this algorithm are not steps in the sense of a single move of the TM; instead, we rely on the reader’s sense of how to program a TM to interpret statements like “Write $L$ one cell to the right of the rightmost $R$” in terms of a sequence of states and transitions that move the work-tape head to the appropriate location and update the cell. Doing this involves the usual wager in mathematical writing: the writer is betting both their own and the reader’s time against the possibility of embarrassment if a procedure that has an obvious implementation does not in fact work.

In this case, the algorithm is simple enough that we can also write out the transition table: see Figure 3.2.

3.1.4 Turing machine variants

A lot of early working on Turing machines and related models went into showing that each could simulate the others. Most of these simulations are not very exciting, but knowing they exist can sometimes simplify arguments about what can or cannot be computed by a Turing machine. The following lemma gives a few of the more useful reductions:
Lemma 3.1.1. Each of the following models computes the same predicates and functions as a standard \( k \)-tape Turing machine:

1. A machine with a writable input tape and a readable output tape.

2. A machine that allows more than one head per tape.

3. A machine whose tapes have a left boundary, making them infinite in only one direction.

4. A machine that has only a single tape.

5. A machine with tape alphabet \{0, 1\}.

6. A machine that has no work tape, but instead has at least two counters supporting increment, decrement, and test-for-zero operations. (Equivalently, a Turing machine with at least two non-writable work tapes that each contain exactly one non-blank symbol.)

Proof. 1. The trick here is to add two extra work tapes to the standard machine. The first extra tape holds a copy of the input, the second a copy of the output. It is straightforward to modify the transition function (with a few extra states) so that the first thing the machine does is copy the non-writable input tape to the first extra work tape, and the last thing the machine does is copy the second extra work tape to the output tape. This incurs a time and space overhead linear in the combined size of the input and output, which is usually not a problem unless we are considering machines with very restricted space complexity.
2. For this we extend the tape alphabet to include markers for the simulated head positions, so that instead of $\Gamma$ we are using $\Gamma \times \mathcal{P}([\ell])$ where $[\ell] = \{0, 1, \ldots, \ell - 1\}$ and $\ell$ is the number of heads per tape. We assume, based on the previous construction, that the input and output tapes are writable.

To execute a step of the multi-head machine, we first send the real heads across each tape to collect all symbols under the simulated heads. This takes $O(T)$ time in the worst case, where $T$ is the running time of the simulated machine. We then compute the new simulated state, moves of the simulated heads, and symbols written, and again send the real heads across each tape to make these updates. The actual mechanics of implementing this are pretty tedious, but it is straightforward to see that only finitely many extra states are needed to keep track of the finite vector of symbols that have been, the finite vector of symbols that need to be written, and which heads still need to move and in which direction. So this is something that a standard Turing machine can do.

There is no overhead in space complexity (except possibly dealing with the issue of a non-writable input tape with multiple heads), but the time complexity of a computation can easily go from $T$ to $O(T^2)$.

3. Here we take each standard doubly-infinite tape and fold it in half, turning it into a tape that infinite in only one direction.

The idea is that the $i$-th cell of the folded tape holds cells at positions $\pm i$ on the original tape, with a special mark in cell 0 to indicate the left edge. If we use an extra bit for this mark, this requires expanding the tape alphabet from $\Gamma$ to $\Gamma^2 \times \{0, 1\}$. We must also expand the state space to indicate for each head whether it is on the positive or negative half of the simulated tape, and adjust the transition function to keep track of this information, move each head in the appropriate direction, and update whichever side of the simulated tape is appropriate at each step. These changes will still leave the size of the alphabet, state space, and transition table constant, and there is no slowdown or increase in space complexity as a result of the simulation.

4. To reduce from $k$ tapes to a single tape, pack the tapes consecutively one after each other, with markers as in the previous construction for the $k$ distinct heads. A simulated step consists of scanning the entire tape to find the values under the heads $O(T(n))$ steps, computing the result of applying $\delta$, and then rewriting new values and moving the
head markers. This last step takes $O(T(n))$ time if we don’t have to increase the space used by any of the work tapes. If we do, we can copy values to the right of the places where we are expanding to make room; doing this from right-to-left requires $O(1)$ storage and $O(T(n))$ time. In each case the total cost of simulating one step of the $k$-tape machine is bounded by $O(T(n))$, and the total cost of simulating $T(n)$ steps is bounded by $O((T(n))^2)$. This turns out to be optimal, see §3.1.5.

5. To reduce to a two-character alphabet, encode the original alphabet in binary. Let $\Gamma$ be the original alphabet, and let $k = \lceil \log |\Gamma| \rceil$ be the minimum number of bits needed to represent each element of $\Gamma$ uniquely. We will represent each cell of the simulated machine with $k$ cells in the simulating machine, and represent each symbol in $\Gamma$ with a distinct sequence of $k$ bits (note that this requires cooperation from whoever is supplying our input!)

At the start of each simulated step, we assume that each head is parked on the leftmost symbol of a $k$-wide block. To read the simulated symbols under the heads, we move the heads across the blocks collecting bits as they go, then move them back; this takes $2(k - 1)$ steps and requires expanding the state space to track what we are doing, but the state space will still be finite. We then compute the transition and store the new symbols to write in the finite-state controller. A second $2(k - 1)$-step walk across the blocks writes the new symbols. Finally, we take $k$ steps to move the heads left or right $k$ cells as determined by the simulated transition function.

6. To reduce a TM to two counters, we use a construction of Minsky [Min61].

The first idea is that we can replace a doubly-infinite tape with a head in the middle with two stacks. To move the head right, we pop from the right stack and push to the left; to move left, we do the reverse.

Next, we can implement a stack with two counters, supporting increment, decrement, and test-for-zero operations. A stack containing symbols $x_0, x_1, \ldots$ is represented by the number $X = \sum_{i=0}^{\infty} s^i x_i$, where $s = |\Gamma|$ and we assume that a blank symbol is represented by 0 to keep the total finite. To read the top symbol $x_0$, we must compute $X \mod s$, which we can do by copying the counter holding $x$ to a second counter starting at 0 (using $X$ many decrement and increment operations), and tracking the remainder mod $s$ as we go. To pop the stack, we compute $\lfloor X/s \rfloor$ by incrementing the output counter once after every $s$
decrements we manage to do on the input counter, throwing away any remainder at the end. To push a new symbol $z$, compute $X \cdot s + z$ by incrementing the output counter $s$ times for each time we decrement the input counter, then add an extra $z$ on the end. All of these operations can be managed by a finite-state controller, since we only need to be able to count up to $s$.

Applying both techniques turns $k$ tapes into $2k$ counters. To reduce to just two counters, use Goedel numbering \footnote{Gö31, §V}: pick $2k$ distinct primes $p_1, p_2, \ldots, p_{2k}$ and encode the vector $\langle X_1, \ldots, X_{2k} \rangle$ as $Y = \Pi_{i=1}^{2k} p_i^{X_i}$. We can now test if any particular $X_i$ is 0 by testing if $Y$ is not divisible by $p_i$ (which can be done by computing a remainder while copying $Y$ from one counter to the other), and can increment or decrement $X_i$ by multiplying or dividing by $p_i$. Again, everything requires a finite amount of state to manage.

The blow-up in time complexity for this simulation is exponential. Going from $k$ tapes to $2k$ counters by itself makes each step cost as much as $O(ks^T)$ counter operations. Each counter operation in turn can take up to $O(p_1^{T_2})$ steps on the two-counter machine, where $p_{2k}$ will be $\Theta(k \log k)$ (a constant!) if we are parsimonious in our choice of primes. Combining these gives a time per step of the original machine of $O(k(s^{2k})^T)$, which argues for applying this technique only to machines with few tapes and small alphabets, if we have to apply it at all. On the other hand, it shows that even very limited (but unbounded) storage devices, together with a finite-state controller, can compute anything computed by a standard Turing machine or any model it can simulate. This makes general-purpose computation very easy to achieve if we have unbounded space and don’t care about time too much.

\begin{flushright} \blackslug \end{flushright}

\section{Limitations of simulations}

Except for tightly constrained models like Minsky’s two-counter machines, most of the models we consider can simulate each other with at most polynomial blow-up. A typical case is our ability to simulate a $k$-tape Turing machine that runs in $O(T(n))$ time using a one-tape Turing machine that runs in $O((T(n))^2)$ time. In some of these cases, we can show that this blow-up is unavoidable.

For example, we have previously seen \footnote{3.1.3.2} that it is possible to decide the language \textsc{Palindrome} = $\{ x \mid x = x^R \}$ using a Turing machine with
a work tape in $O(n)$ steps. A classic lower bound of Hennie [Hen65] shows that a one-tape machine must take $\Omega(n^2)$ steps to recognize this language.\footnote{Strictly speaking, Hennie shows [Hen65, Theorem 5] the same lower bound for the language $x2^n x$, where $x \in \{0, 1\}^n$. Here the second copy of $x$ is not reversed, but the lower bound argument is pretty much the same.}

We give an explanation of this result below.

Given a computation of a one-tape Turing machine $M$ on input $x$, we can consider the sequence of steps that send us back and forth between cells $i$ and $i + 1$ on the tape. The crossing sequence $C_i(x)$ is defined as the sequence of states $q_1 q_2 \ldots q_k$ that the finite-state controller holds in each configuration just before moving from $i$ to $i + 1$ or from $i + 1$ to $i$. The crossing sequence characterizes what information is carried from the left side of the tape to the right side of the tape and vice versa. When drawing a crossing sequence, we’ll often put in arrows indicating which direction the head is moving at each point in the sequence, but this is redundant: we know that the head will be on the left side at the beginning, and each crossing changes which side it is on, so the odd positions in the sequence always correspond to left-to-right transitions and the even positions always correspond to right-to-left transitions.

The length of a crossing sequence $C_i(x)$ may depend on $x$, since different inputs may result in fewer or more crossings. What makes the crossing sequences useful is that $\sum_i C_i(x)$ is a lower bound on the number of steps taken by the Turing machine.\footnote{It is not exact because the sum doesn’t include steps where the head doesn’t move.} So showing that a computation takes a long time requires “only” showing that it has many long crossing sequences. We can do this for PALINDROME, as well as many similar languages like $\{xx\}$ where recognizing a member of the language requires comparing a lot of information on widely-separated parts of the input tape.

The key step is observing that a crossing sequence fully characterizes what information flows across the boundary between cells $i$ and $i + 1$. More explicitly, we show that if two inputs give the same crossing sequence, we can swap the parts of the input on either side of the boundary without changing the outcome of the computation:

**Lemma 3.1.2.** Let $M$ be a one-tape Turing machine. Consider two inputs $st$ and $uv$, where $|s| = |u| = i$. Then if $C_i(st) = C_i(uv)$ and $M(st) = M(uv)$, then $M(sv) = M(st)$.

**Proof.** The idea is that we can paste together parts of the computation on $st$ with parts of the computation on $uv$ to get a computation on $sv$ that behaves the same as $st$ on the $s$ side of the boundary and as $uv$ on the $v$
side. Divide the st computation into a sequence of \( k = C_i(st) + 1 \) fragments \( \alpha_0 \alpha_1 \ldots \alpha_k \), where the split between each \( \alpha_j \) and the following \( \alpha_{j+1} \) is at the point where the TM head crosses the \( i-(i+1) \) boundary. Similarly divide the uw computation into fragments \( \beta_0 \ldots \beta_k \).

We now argue that \( \alpha_0 \beta_1 \alpha_2 \beta_3 \ldots \gamma_k \) describes the sv computation, where \( \gamma_k \) is either \( \alpha_k \) or \( \beta_k \) depending on whether \( k \) is even or odd. The argument is a straightforward induction on the step count so far, with the induction hypothesis stating that at the \( \ell \)-th step of the \( j \)-th segment, the active side of the tape and the finite-state controller are both in the same state as in the corresponding step of the appropriate unmixed execution, and the passive side of the tape is in whatever state it would be at the end of the segment it just completed in its own unmixed execution. This is easily shown for transitions that don’t cross the boundary. For transitions that cross the boundary, we appeal to \( C_i(st) = C_i(uw) \) to argue that the finite-state controller’s state is the same coming out of the active side as it should be going into the previously passive side.

It follows that when the machine halts on the s side, it halts in the same state as in \( M(st) \); if it halts on the v side, it halts in the same state as in \( M(uw) \). In either case it accepts or rejects as in the original unmixed execution.

How does this help us with PALINDROME? Fix some machine \( M \) for recognizing PALINDROME, and consider the set \( S_n \) of inputs of the form \( x0^nx^R \), for all \( x \in \{0,1\}^n \). Let \( C_i(S_n) = \{C_i(y) \mid y \in S\} \). We will argue that \( |C_i(S)| \) is large for all \( i \in n \ldots n2 - 1 \), and thus that most crossing sequences across these boundaries are also large. Since each element of each \( C_i(y) \) crossing sequence corresponds to a step of \( M(y) \), this will give us an \( \Omega(n^2) = \Omega(|y|^2) \) lower bound on \( M(y) \) for some \( y = x0^nx^R \).

To show \( |C_i(S)| \) is large, suppose that there are strings \( x \) and \( y \) in \( \{0,1\}^n \) such that \( C_i(x0^nx^R) = C_i(y0^ny^R) \). Since these two strings are both palindromes, Lemma 3.1.2 says that we can split them at \( i \) and paste the results together to get a new string \( x0^ny^R \) that \( M \) accepts. This implies that \( x = y \), making \( C_i \) injective on \( S \). So \( |C_i(S)| = |S| = 2^n \).

Because a crossing sequence is just a list of states, there are at most \( |Q|^t \) possible crossing sequences of length \( t \), and at most \( \frac{|Q|^{t-1}}{|Q|-1} \leq |Q|^t \) crossing sequences of length strictly less than \( t \). Let \( t = \lceil \frac{n-1}{\log |Q|} \rceil = \Omega(n) \). Then \( |Q|^t \leq 2^{n-1} \), and at least half of the \( y \) in \( S \) give crossing sequences \( C_i(y) \) that have length at least \( t = \Omega(n) \). If we choose an element \( y \) of \( S \) uniformly at random, the expected number of positions \( i \in \{n \ldots 2n - 1\} \)
for which \( C_i(y) \geq t \) is at least \( n/2 \). It follows that there is some \( y \) such that \( C_i(y) \geq t \) for at least \( n/2 \) values if \( i \), giving a total length over all \( i \) of at least \( tn/2 = \Omega(n^2) \).

### 3.1.6 Universal Turing machines

One of Turing’s most striking observations about Turing machines was that even though any particular Turing machine has a fixed transition table, you can build a universal Turing machine \( U \) that simulates any other Turing machine \( M \), given a description \( \langle M \rangle \) of that machine on its input tape. This is true even if the simulated machine has a larger tape alphabet than \( U \) (although the input will need to be encoded so that \( U \) can read it) or uses more tapes.

Specifically, \( U \) is universal if \( U(\langle M \rangle, \langle x \rangle) = M(x) \) for any Turing machine \( M \) and input \( x \), where \( \langle M \rangle \) and \( \langle x \rangle \) are appropriate encodings of \( M \) and \( x \) in \( U \)’s input alphabet.

By an appropriate encoding of \( M \), we want something that specifies:

1. The size of \( M \)’s state space \( Q \), tape alphabet \( \Gamma \), and input alphabet \( \Sigma \).
2. The number of work tapes available to \( M \).
3. The transition table for \( M \). To simplify things, it’s usually easiest to assume a standardized form of the transition table, where the states in \( Q \) are encoded as binary numbers in the range \( 0 \ldots |Q| - 1 \), with 0 encoding the initial state \( q_0 \), and the alphabet \( \Gamma \) is similarly encoded as \( 0 \ldots |\Gamma| - 1 \), with 0 representing the blank symbol, \( 0 \ldots |\Sigma| - 1 \) representing the input alphabet.

Actually programming \( U \) is a bit of a nuisance, but if we are not too worried about time complexity, we can store \( M \)’s work tapes consecutively on a single work tape, using the techniques from Lemma 3.1.1 to simulate having a larger alphabet than \( U \) and separate heads for each simulated tape. This may require copying large section of \( U \)’s work tape from time to time to expand the storage allocated to a particular simulated tape, but each of these copying operations will only take time \( O(S \log |\Gamma|) \) where \( S \) is the space complexity of \( M \)’s computation (which is bounded by the time complexity \( T \) of this same computation). To execute a step of \( M \), we gather up the symbols under \( M \)’s heads onto a second work tape (which we also use to store \( M \)’s state), and then look for a matching element of \( Q \times \Gamma^k \) in \( M \)’s transition table. This requires scanning our entire storage tape, although for
the input we can just use a copy of the original input tape. We then copy the new state, cell contents, and head movements onto the second work tape, and finally run up and down the first work tape to rewrite cells and move the simulated heads. Any output is written directly to the output tape. All of this takes \( O(|\Lambda M \Lambda| + T \log |\Gamma|) \) time assuming the simulated work tapes are reasonably well packed together. The total time to simulate \( T \) steps of \( M \) is thus \( O(CT^2) \) where \( C \) is a constant that depends on \( M \).

Using the clever amortized data structure of Hennie and Stearns (see Lemma 3.1.1), we can replace the consecutive representations of \( M \)'s work tapes by interleaved representations and reduce the cost to \( O(CT \log T) \), where \( C \) is again a constant that depends on \( M \). This improvement in efficiency will turn out to be important when we look at the time hierarchy theorem in §6.1.2.

3.2 Random access machines

A traditional C programmer, presented with the modified version of the language from §3.1.3.1, might initially be overjoyed to realize that having infinite arrays means no possibility of segmentation faults. But their joy would turn to ashes quickly once we reveal that basic array operations like \( a[i] \) are denied them. If we want to market our models to programmers, we will need to give them more power. Typically this is done using some version of a random access machine (RAM).

A RAM looks a lot like a typical modern computer in that it has a controller with registers and a memory that looks like a giant unbounded array, except that (as in a Turing machine) its program is encoded directly into the transition table of its finite-state controller. Each register and each memory location holds an integer. While different RAM models use different instruction sets, the basic idea is that we can do arithmetic on registers, compare registers, and load or store a value from a memory location addressed by a particular register all as a single step. Each such step is determined by the current state, and the transition function specifies the next step (possibly based on the outcome of a test in the case of comparing registers).

We can implement a RAM using a Turing machine by storing a binary representation of the registers and memory on work tapes. The simplest way to do this is probably to assign a separate work tape to each register (there

---

9 Using a copy means that we don’t need to use the input head to mark where we are in \( x \), and can instead use it to scan through \( \Lambda M \Lambda \).
are only finitely many); if this is too profligate, we can use the simulation from Lemma 3.1.1 to reduce to a single work tape. For the memory, we use a single work tape organized as an association list: a non-empty memory location \( i \) holding a value \( x \) is represented by a sequence \( i \rightarrow x \) where \( i \rightarrow \) and \( x \rightarrow \) are binary representations of \( i \) and \( x \) and \( \rightarrow \) is a separator. The list elements are themselves separated by a different separator.

Arithmetic operations on registers are implemented using standard Turing machine programs (possibly using an extra work tape or two). For addition, subtraction, and comparison, this will take \( O(\log M) \) time when working on values \( x \) with \( |x| \leq M \).

Memory operations more painful. To read a memory cell \( i \), we have to scan the entire memory tape to find \( i \rightarrow \), then copy the corresponding \( x \rightarrow \) to the desired register tape. To write \( x \) to \( i \), we must again scan for the current value of \( i \) (if any) and remove it by copying any subsequent association-list pairs down. We can then append the new pair \( i \rightarrow x \) to the end of the list. Both of these operations take time linear in the length of the memory, which will be \( O(T \log C) \) if \( C \) is an upper bound on the absolute value of any register during the computation. For typical computations, \( C \) will be polynomial in \( T \), giving a slowdown of \( O(T \log T) \).

The simulation in the other direction is trivial: given a Turing machine, we can assume (based on the simulations in Lemma 3.1.1 that it has a single, half-infinite tape. Store this in memory, one cell per memory location, and use a register to track the position of the head.

Even though RAMs are more natural to program than Turing machines, as a mathematical model they are a bit annoying. The big problem is that the state of a RAM is complicated, which makes it tricky to simulate a RAM using other models, and allowing unbounded values in the registers and memory makes defining space complexity tricky as well. So we will generally think of Turing machines as our fundamental model, and appeal to simulations like the one above (or the extended Church-Turing thesis more generally) to transform algorithms written for more powerful models into something that can run on a TM.

A second objection to the RAM model is that the ability to access arbitrary memory locations in \( O(1) \) steps is not physically realistic. Assuming each bit of memory requires some minimum volume of space (a few thousand cubic nanometers using current integrated circuit technology, or something related to the Planck length based on quantum mechanics), we can only pack \( O(1) \) bits of memory within any constant distance of the CPU, and in general we can only pack \( O(\ell^3) \) bits within distance \( \ell \). This means that accessing a memory of size \( S \) without faster-than-light communication will
require $\Theta(S^{1/3})$ time in the worst case. Turing machines enforce a worse restriction implicitly, since we have to run the head down the tape. This makes a TM arguably a better representation of what a very large computer could do than a RAM.

### 3.3 The extended Church-Turing thesis

There are many other models of computation we could consider, but with the possible exception of quantum computers, all the ones we can currently imagine implementing fall under the **extended Church-Turing thesis**, which says that

\[\text{Claim 3.3.1. Any language that can be decided by a physically realizable computing device } M \text{ in time } T(n) \text{ can be decided by a Turing machine in time } O(T(n)^k \text{ for some fixed } k \text{ that depends only on } M.}\]

In other words, all physically realizable computing devices are equivalent in power to a Turing machine, up to polynomial slowdown.

The argument for this is that we can imagine that any physically realizable computing device can be simulated by a 1977 TRS-80 Model I home computer equipped with an unboundedly large external storage device\(^{10}\) and using a construction similar to the one sketched for random access machines we can imagine that the TRS-80 Model I can be simulated by a Turing machine. So the original device can be simulated by a Turing machine. By itself this claim is just the **Church-Turing hypothesis**; the extended version says that this simulation involves only polynomial slowdown, which appears to be true for everything we’ve managed to come up with so far.

Note that polynomial slowdown means that things like $O(n)$ or $O(n^2)$ time may depend on our choice of computational model. But if we just talk about polynomial time, this is robust against changes in the model. This is why we are so interested in classes like $P$ (what we can decide in polynomial time) as opposed to classes like $\text{TIME}(n)$ (what we can decide in linear time). But these less robust classes are still well-defined as long as we are careful to specify our model.

\(^{10}\)If this seems implausible, substitute one of the Zoo computers equipped with an unboundedly large external storage device.
A complexity class is a set of languages that are similarly hard in some sense. For example, the class $\textbf{P}$ is the set of all languages that can be decided by a Turing machine in polynomial time, that is, in $O(n^k)$ time for some $k$.

To formalize the class of languages that can be decided within some time bound, we need a technical definition to exclude time bounds that produce weird results (say, by being themselves uncomputable). A function $f(n)$ is time-constructible if there is a Turing machine that, given input $1^n$, computes $1^{f(n)}$ in $O(f(n))$ steps. Similarly, a function $f(n)$ is space-constructible if there is a Turing machine that, given input $1^n$, computes $1^{f(n)}$ in $O(f(n))$ space. To avoid the issue of reading all the input, we generally also require $f(n) \geq n$ in both cases.

Given a time-constructible function $f(n)$, the complexity class $\textbf{TIME}(f(n))$ consists of all languages $L$ for which there exists a Turing machine $M$ that decides $L$ while always halting after at most $O(f(n))$ steps. Similarly, given a space-constructible function $f(n)$, the class $\textbf{SPACE}(f(n))$ consists of all languages $L$ for which there exists a Turing machine $M$ that decides $L$ while always using at most $O(f(n))$ space.

As observed previously, both $\textbf{TIME}(f(n))$ and $\textbf{SPACE}(f(n))$ may depend on the specific details of the computational model we are using. For example, it is known that recognizing a palindrome can be done in $O(n)$ time on a Turing machine with a separate input and work tape (easy exercise) but requires $\Omega(n^2)$ time on a machine with just one tape. For this reason we often work with more robust classes.
The most important class of all, which is generally taken to correspond to what is computationally feasible, is the class

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

This consists of all languages that are in \( \text{TIME}(n^k) \) for some finite \( k \).

The extended Church-Turing thesis says that \( P \) is robust in the sense that it contains the same languages for any reasonable model of computation. This is not a theorem (although it can be taken as a definition of a reasonable model); instead, it is a hypothesis that follows from the fact that all of the plausible-looking models of computation that have been invented over the years all have the ability to simulate each other up to polynomial slowdown.
Chapter 5

Nonterminism and NP

Historically, a nondeterministic Turing machine has been defined as one where the transition function is replaced a transition relation: instead of each configuration leading directly to a unique successor configuration, a configuration may have more than one successor configuration. In its simplest form, this means that the next state $q'$ of the Turing machine controller is replaced by two next states $q_0'$ and $q_1'$, and the Turing machine can choose between them. In order to avoid having to think about how this choice is made, we imagine that in fact the machine makes both choices: we give it the magical ability to split into two copies, each with a different bit telling it what to do next. These copies can then split further into exponentially many copies, or branches. If any of the branches accepts, then we say the machine as a whole accepts; if none do, the machine rejects.

This definition is a little awkward to work with, so nondeterminism is now typically represented by giving a machine an extra input, the certificate or witness. This corresponds to the original definition by having the witness provide the sequence of choices that lead to the accepting branch (if there is one). But instead of having to imagine a ghostly parade of branches, we just think about a single computation that happens to get very lucky in the choice of witness.

Formally, a language $L$ is decided by a nondeterministic machine $M$ if, for every $x \in L$, there exists a witness $w$ such that $M(x, w)$ accepts, and for every $x \notin L$, there does not exist a witness $w$ such that $M(x, w)$ accepts.

This definition is not quite as symmetric as it looks: saying that $x \notin L$ means there is no witness is the same as saying that all $w$ cause $M(x, w)$ to reject. So for a “yes” instance of $L$, one good witness (equivalently, one accepting branch) is enough, but for a “no” instance, all witnesses must be
bad (all branches must reject).

Analogous to \( \text{TIME}(f(n)) \) and \( \text{SPACE}(f(n)) \), we have the nondeterministic time complexity classes \( \text{NTIME}(f(n)) \) and nondeterministic space complexity classes \( \text{NSPACE}(f(n)) \). These consist of all languages \( L \) that are decided by a nondeterministic Turing machine in \( O(f(n)) \) time or \( O(f(n)) \) space, respectively. Here the time or space complexity of machine \( M \) on input \( x \) is defined as the maximum time or space, respectively, taken by \( M \) over all choices of witness \( w \). Note that \( |w| \) does not count toward \( n \), which is just \( |x| \).

One complication with bounded space complexity is that the time complexity (and thus the number of branchings represented by the witness string) may be much larger than the space complexity (and thus how much of the witness the machine can remember). If we supply the witness on a standard two-way input tape, this allows the machine to go back and revisit its earlier choices in a way that might not be possible in the original branching version of nondeterminism. To avoid this, we supply the witness on a second read-only input tape, whose head can only move right. This also justifies, somewhat, not counting the length of the witness string as part of the input size \( n \).

Like the deterministic time and space complexity classes, nondeterministic time and space complexity classes may depend on the specific details of the model being used. For this reason, we are generally most interested in nondeterministic classes that are more robust against changes in the model. The most important of these is

\[
\text{NP} = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k),
\]

the set of all languages that can be decided in \textbf{nondeterministic polynomial time}. As far as we know, this class may or may not be the same as \( \text{P} \), and the most important outstanding problem in complexity theory is showing whether or not \( \text{P} = \text{NP} \).

There’s an alternative definition of \( \text{NP} \) which puts the input and witness on the same input tape. Here we have to put a bound on the size of the witness to avoid, for example, ending up with a time complexity exponential in \( |x| \) because an otherwise-useless \( w \) makes \( n \) too big. For this definition, we let \( L \in \text{NP} \) if there is a polynomial \( p \) and a polynomial-time \( M \) such that for all \( x, x \in L \) if an only if there exists \( w \) of length \( p(|x|) \) such that \( M(x, w) \) accepts. Note that the running time of \( M \) is now polynomial in \( |x| + |w| \), but this is still polynomial in \( |x| \) because \( |w| \) is.

This approach doesn’t work as well for \( \text{NTIME}(f(n)) \) in general, because the size of \( w \) would have to be both smaller than \( n \) (if it is counted in \( n \)
and at least $f(n)$ (if it represents nondeterministic choices that can occur at each of $f(n)$ steps.

5.1 Examples of problems in $\text{NP}$

It’s trivial to show that any language $L$ in $\text{P}$ is also in $\text{NP}$: take a poly-time machine $M(x)$ that decides $x \in L$, and convert it to a nondeterministic poly-time machine $M'(x, w)$ that decides $x \in L$ by the simple expedient of ignoring $w$. But there are a large class of problems that can easily be shown to be in $\text{NP}$ that we don’t know how to solve in $\text{P}$.

Typically these are problems where we are asked if some solution exists, and checking the solution (provided as the witness) can be done efficiently. What makes this nice from the point of view of the programmer is that finally we have a logical quantifier that is on our side. No longer must we face the worst-case input, supplied by our adversary $\forall x$, alone. Instead, our good friend $\exists w$ comes to our aid after the adversary makes its play.

For example, suppose we want to solve GRAPH 3-COLORABILITY. On an ordinary Turing machine, we could try out all possible colorings; but for an $n$-node, $m$-edge graph, there are $3^n$ of them. With a nondeterministic Turing machine, we simply summon $\exists w$ and demand it provide us with the correct coloring. This is trivial to check in $O(n + m)$ time, and if for some reason the existential quantifier betrays us, we will be able to recognize in that same time that $w$ is no good. So the beauty of having a nondeterministic machine is that all the hard work of designing an actual algorithm is taken over by whoever provides $w$; we just need to be able to specify what a correct solution would look like, and write an efficient program to verify candidate solutions.

Many other problems have a similar structure. Want to know if a graph has an INDEPENDENT SET of size $k$? Have $\exists w$ guess the list of nodes in the independent set. Can your TRAVELING SALESMAN visit every node in a weight graph using a path of total weight $W$? Have $\exists w$ guess the path. Is your Boolean formula SATISFIABLE? Have $\exists w$ guess the satisfying assignment. In each case the problem of verifying that the guess is correct is straightforward, and we can easily argue that it can be done in polynomial (often only linear) time. So all of these problems are in $\text{NP}$. Which means that if $\text{P} = \text{NP}$, and we interpret membership in $\text{P}$ as meaning a problem is easy, then all of these problems are easy. Sadly, there is a very strong chance that they are not easy.
5.2 Reductions and NP-complete problems

A **polynomial-time many-one reduction** from a language $L$ to a language $L'$ is a deterministic polynomial-time computable function $f$ such that $x \in L$ if and only if $f(x) \in L'$. If there exists a polynomial-time many-one reduction from $L$ to $L'$, we write $L \leq_P L'$.

The idea behind writing a reduction as an inequality is that if we can efficiently reduce $L$ to $L'$, then $L$ is no harder than $L'$, because, given an efficient algorithm $M$ that decides membership in $L'$, then $M \circ f$ is an efficient algorithm that decides membership in $L$. The $P$ subscript specifies what complexity class the reduction $f$ lives in; in some cases, we will replace this with other classes to indicate different restrictions on $f$.

Proving a reduction $L \leq_P L'$ generally involves 3 steps:

1. You have to come up with the mapping $f$ and show that it runs in polynomial time.
2. You have to show that if $x \in L$, then $f(x) \in L'$.
3. You have to show that if $x \not\in L$, then $f(x) \not\in L'$; or that if $f(x) \in L'$, then $x \in L$. The second version is just the contrapositive of the first, but is sometimes easier to see how to do, especially when $f$ is a one-to-one mapping that is easily inverted.

A language $L$ is **NP-hard** if $L' \leq_P L$ for any language $L'$ in NP. A language $L$ is **NP-complete** if it is both in NP and NP-hard. The NP-complete languages are the hardest languages in NP, in the sense that if we can recognize any NP-complete language in polynomial time, then $P = NP$.

If $P \neq NP$, then there are languages in NP that are not NP-complete (for example, all the ones in P). Under this assumption, there are even languages in NP – P that are not NP-complete, but they are not very interesting languages.

The NP-complete languages, on the other hand, are very interesting: given any NP-complete $L$, the $P \not\equiv NP$ question is equivalent to $L \not\in NP$. So instead of having to consider all possible languages in NP, it’s enough to pick one particular NP-complete languages, and show that it either does or does not have a polynomial-time algorithm.

Alternatively, if we believe that $P \neq NP$, then this immediately tells us that any NP-complete language (more generally, any NP-hard language) will not have a polynomial-time algorithm. Even if we aren’t sure if $P \neq NP$, we still know that we have no examples so far of a polynomial-time algorithm.
for any problem that is \textit{NP}-hard. So proving that a particular problem is \textit{NP}-hard means that we can be reasonably confident that we won’t find a polynomial-time algorithm for it without some surprising breakthrough.

5.3 The Cook-Levin theorem

None of this is useful unless we can point to an example of an \textit{NP}-complete problem. The \textbf{Cook-Levin theorem} gives one such problem, called \textit{3SAT}. This is the problem of testing whether there exists a satisfying assignment to a Boolean formula in \textit{conjunctive normal form} (CNF) where each clause contains exactly three literals. A formula in this form is an AND of clauses, each of which is an OR of three variables \(x_i\) or their negations \(\neg x_i\). The proof of the theorem is a construction that translates any problem in \textit{NP} into such a 3CNF formula.

\textbf{Theorem 5.3.1.} \textit{3SAT} is \textit{NP}-complete.

\textit{Proof.} The essential idea is that we can encode the entire computation of an \textit{NP} machine \(M\) on a given input \(x\) as a single gigantic (but polynomially large) SAT formula.

We’ll start by setting up some variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Q^t_q)</td>
<td>Finite-state controller is in state (q) at time (t)</td>
</tr>
<tr>
<td>(H^t_{ij})</td>
<td>Head (i) is in position (j) at time (t)</td>
</tr>
<tr>
<td>(T^t_{ijs})</td>
<td>Tape (i), cell (j), holds symbol (s) at time (t)</td>
</tr>
</tbody>
</table>

If the machine runs for \(T\) time, we get \(O(T^3)\) variables total. For the tapes, we are relying on the fact that the infinite parts more than \(T\) away from the initial head positions don’t need to be represented.

We now add some consistency conditions to the SAT formula. For example, to enforce that the machine is in exactly one state at any time, we include the OR-clauses \(\bigvee_q Q^t_q\) and \(\neg Q^t_q \lor \neg Q^t_{q'}\) for each \(t\), \(q\), and \(q'\). Similar clauses enforce that each head is in exactly one position and each tape cell contains exactly one symbol. The total size of these clauses is again polynomial in \(T\).

For the transition relation, we consider each combination of tape head positions, cell symbols under those positions, and finite-state controller state, and write a clause for each consequence of this combination. For example, if the machine has a single tape, and changes from state 0 to either state 1 or state 2 (it’s nondeterministic) when it sees a \(b\) at location 5 on this tape, we could write this condition at each time \(t\) as \((Q^t_0 \land H^t_{1,5} \land \neg H^t_{2,5})\) or \((Q^t_1 \land H^t_{1,5} \land \neg H^t_{2,5})\) or \((Q^t_2 \land H^t_{1,5} \land \neg H^t_{2,5})\).
$T^t_1, 5, b) \Rightarrow (Q^{t+1}_1 \lor Q^{t+1}_2)$, which conveniently transforms into the OR clause $\neg Q^t_0 \lor \neg H^t_{1,5} \lor \neg T^t_1, 5, b \lor Q^{t+1}_1 \lor Q^{t+1}_2$.

There are a lot of these clauses, but for a fixed number of tapes their total size is still polynomial in $T$.

Finally, we need to assert that the final state accepts (an OR over all accepting states $q$ of $Q^T$) and that the initial configuration of the machine is correct (many one-variable clauses asserting $Q^0_0$, $H^0_0$, $S^0_{ijx_1}$, $S^0_{ijb}$, etc.). The intermediate states are left unconstrained, except by the sanity clauses and transition-relation clauses already described.

If we can satisfy this enormous (but polynomial) formula, then there is a nondeterministic computation by the original machine on the given input that accepts. If we can’t, there isn’t. So we have successfully reduced the question of whether $M(x)$ accepts, and thus whether $x \in L(M)$, to SAT.

The last step is to replace each clause with more than three literals with a clause with three literals (or fewer, since we can always duplicate a literal to get to exactly three). The trick is that $x_1 \lor x_2 \lor \ldots \lor x_k$ is satisfiable if and only if both of $z \lor x_1 \lor x_2$ and $\neg z \lor x_3 \lor \ldots \lor x_k$ are, where $z$ is a new variable introduced solely for the purpose of splitting up this clause. If $z$ is true, at least one of $x_3$ through $x_k$ must also be true, making the original clause true; if instead $z$ is false, then at least one of $x_1$ or $x_2$ must be true. When we do this split, we new clauses of size 3 and $k - 1$. Iterating until we get $k$ down to 3 gives us a 3CNF formula.

### 5.4 More NP-complete problems

Once we know that 3SAT is NP-complete, showing any other language $L$ is NP-complete requires only (a) showing that $L \in \text{NP}$ and (b) showing that there is a reduction $3\text{SAT} \leq_P L$. The reason why (b) is enough is that $\leq_P$ is transitive: given some language $L' \in \text{NP}$, we take some candidate member $x$ of $L'$, run it through one polynomial-time function $f$ to get and instance $f(x)$ of 3SAT, and then run $f(x)$ through another polynomial-time function $g$ to get an instance $g(f(x))$ of $L$, and we will have $x \in L$ if and only if $g(f(x)) \in L$. The first function $f$ exists because of Theorem 5.3.1. The second function $g$ exists because we showed $3\text{SAT} \leq_P L$.

We don’t have to reduce from 3SAT to show $L$ is NP-hard. Starting from any known NP-hard language $L'$ also works. So it’s helpful to have a few known NP-hard languages around to get this process started.

Many of the problems below are from Karp’s 1972 paper [Kar72] extending Cook’s result for SAT to a variety of other combinatorial problems. (Note
that the reductions may not be exactly the same.) A more extended source of core NP-complete problems is the classic book of Garey and Johnson \[GJ79\].

5.4.1 1-IN-3 SAT

An unfortunate feature of 3SAT is that each satisfied clause can have any of 1, 2, or 3 true literals. This turns out to be awkward when we try to reduce to problems that involve exact totals. Fortunately, we can show that 3SAT reduces to its more restrictive cousin 1-OF-3 SAT, defined as the set of all 3CNF formulas that have a satisfying assignment that makes exactly one literal in each clause true.

We do this by converting our original 3CNF formula one clause at a time. This involves adding a few extra variables specific to the representation of that clause.

To show how this works, let’s start with $x_1 \lor x_2 \lor x_3$. We’ll replace this clause with a new clause $y_1 \lor y_2 \lor y_3$, where $y_i \Rightarrow x_i$ but not necessarily conversely. These $y_i$ variables are new, and we use a separate trio for each original clause. This allows us to pick just one of the true $x_i$ literals to appear in the $y_1 \lor y_2 \lor y_3$ clause if there is more than one. To enforce that $y_i$ can be true only if $x_i$ is also true, we add three new 1-in-3 clauses, with three new variables: $\neg x_1 \lor y_1 \lor z_1$, $\neg x_2 \lor y_2 \lor z_2$, and $\neg x_3 \lor y_3 \lor z_3$. If any $x_i$ is false, this makes $\neg x_i$ in the corresponding clause true, so $y_i$ must also be false: so we can only satisfy $y_1 \lor y_2 \lor y_3$ if we satisfy $x_1 \lor x_2 \lor x_3$. But if $x_i$ is true, we have a choice between making $y_i$ true or $z_i$ true. The $z_i$ variables (which appear nowhere else) act as a sink for excess truth that would otherwise violate the 1-in-3 property.

Since this works for every clause, the output formula is in 1-IN-3 SAT if and only if the input formula is in 3SAT. Since the reduction is obviously polynomial (it’s linear), this gives 3SAT $\leq_P$ 1-IN-3 SAT, making 1-IN-3 SAT NP-hard. But it’s also in NP, since in polynomial time we can easily guess the satisfying assignment and check that it has the 1-in-3 property. So 1-IN-3 SAT is NP-complete.

5.4.2 SUBSET SUM and PARTITION

SUBSET SUM is the language $\{ \langle x_1, x_2, \ldots, x_m, k \rangle \in \mathbb{N}^{m+1} \mid \exists a \in \{0, 1\}^m \sum a_i x_i = k \}$. This is trivially in NP since we can just guess the vector of coefficients $a$, and add up the lucky values $x_i$ with $a_i = 1$ to see if we get $k$.\footnote{This is an $O(n \log n)$ operation if $n$ is the total length of the $x_i$ in bits.}
that it is also \( \mathbf{NP} \)-hard, since we can reduce from 1-IN-3 SAT. This makes SUBSET SUM \( \mathbf{NP} \)-complete.

Suppose we are given an instance of 1-IN-3 SAT with \( n \) variables \( y_1, \ldots, y_n \) and \( m \) clauses \( C_1, \ldots, C_m \). We will encode each variable \( y_i \) with two natural numbers \( x_i \) and \( x'_i \) written in base 4. The \( j \)-th digit of \( x_i \) is 1 if \( y_i \) appears in \( C_i \) and 0 otherwise, and the \( j \)-th digit of \( x'_i \) is 1 if \( \neg y_i \) appears in \( C_i \) and 0 otherwise. We also set the \((n + j)\)-th digit of both \( x_i \) and \( x'_i \) to one, and make \( k \) be the number written as \( n + m \) ones in base 4.

The idea is that the extra ones shared between each \( x_i \) and \( x'_i \) force us to pick exactly one of them (corresponding to having to choose either \( y_i \) or \( \neg y_i \) to be true), while the ones mapping to clauses force us to choose exactly one literal per clause to be true. This works because we never have more than 3 ones added together in a single position, so we get no carries in base 4, essentially reducing addition to vector addition.

For example, the suspiciously symmetric formula \((y_1 \lor \neg y_2 \lor \neg y_3) \land (\neg y_1 \lor y_2 \lor \neg y_3) \land (\neg y_1 \lor \neg y_2 \lor y_3)\), which happens to be in 1-IN-3 SAT because we can set all the \( y_i \) to be true, would be represented by the SUBSET SUM problem

\[
\begin{align*}
x_1 &= 001001 \\
x'_1 &= 001110 \\
x_2 &= 010010 \\
x'_2 &= 010101 \\
x_3 &= 100100 \\
x'_3 &= 100011 \\
k &= 111111
\end{align*}
\]

This SUBSET SUM problem has the solution \( x_1 + x_2 + x_3 = 001001 + 010010 + 100100 = 111111 = k \), from which we can even read off the solution to the original 1-IN-3 SAT problem if we want to.

With a bit of tinkering, we can reduce SUBSET SUM to the even more terrifying \( \mathbf{NP} \)-complete problem PARTITION. This asks, given a sequence of natural numbers \( \langle y_1, \ldots, y_n \rangle \), if it is possible to split the sequence into two subsequences that add up to exactly the same total. Given an instance \( \langle x_1, \ldots, x_n, k \rangle \) of SUBSET SUM, let \( s = \sum x_i \). If \( k \leq s \), construct the sequence \( \langle x_1, \ldots, x_n, 4s - k, 3s + k \rangle \). This forces PARTITION to put \( 4s - k \) and \( 3s + k \) on opposite sides, because if both are on the same side, it will be at least \( 7s \) while the other side will be at most \( s \) even if we put all the \( x_i \)
there. But then each side must sum to half of \( s + (4s - k) + (3s + k) = 8s \).

We can only bring the side with \( 4s - k \) up to \( 4s \) by adding \( x_i \) to it that sum to \( k \), which we can do if and only if the original SUBSET SUM problem has a subsequence of the \( x_i \) that sum to \( k \).

I personally find this problem frightening because it seems like dividing a pile of numbers into two equal-sum piles should not be all that hard. To be fair, for reasonably-sized numbers, it isn’t\(^2\). One of the things that happens along this path of reductions is that the numbers involved get awfully big, since each has a number of digits linear in the size of the output of the Cook-Levin reduction. This is polynomial in the size of the original problem input, but “I have a polynomial number of digits” is not something small numbers tend to say.

### 5.4.3 Graph problems

Many (but not all!) graph problems that ask if a graph has a subgraph with a particular property turn out to be \( \text{NP} \)-complete. It’s trivial to show that such a problem is in \( \text{NP} \) as long as testing a subgraph for the property is in \( \text{P} \), since we can just guess the winning subgraph. Showing that these problems are \( \text{NP} \)-hard usually requires an explicit reduction.

#### 5.4.3.1 Reductions through INDEPENDENT SET

**INDEPENDENT SET** is the problem of determining, given \( G \) and \( k \), whether \( G \) contains an independent set of size \( k \). An independent set is a subset \( S \) of the vertices of \( G \) such that no edge has both endpoints in the subset. We can show INDEPENDENT SET is \( \text{NP} \)-hard by reducing from 3SAT.

The idea is that any clique in \( G \) can’t contain more than one element of \( S \), so if we can partition \( G \) into \( k \) non-overlapping cliques, then each of these cliques must contain exactly one element of \( S \). We use this constraint to encode variable settings as 2-cliques (also known as edges): for each \( x_i \), create nodes representing \( x_i \) and \( \neg x_i \), and put an edge between them. We’ll call these the master copies of \( x_i \) and \( \neg x_i \).

We can then represent a clause \( C_j = x \lor y \lor z \) as a 3-clique of copies of \( x \), \( y \), and \( z \) (which may be negated variables). Here the member of \( S \) in the representation of \( C_j \) indicates which of the three literals we are using to

---

\(^2\)There is a simple dynamic-programming algorithm that solves SUBSET SUM in time \( O(nk) \), which looks polynomial but isn’t, since the value of \( k \) can be exponentially large as a function of its size in bits.
demonstrate that $C_j$ is satisfiable. These are per-clause copies; we make a separate vertex to represent $x$ in each clause it appears in.

The remaining step for the graph is to make sure that whatever literal we chose to satisfy in $C_j$ is in fact assigned a true value in the 2-clique representing the corresponding $x_i$ or $\neg x_i$. We do this by adding an extra edge from the per-clause copy in $C_j$ to the master copy of the opposite value in the 2-clique; for example, if we have a copy of $x_i$ in $C_j$, we link this copy to the node representing $\neg x_i$, and if we have a copy of $\neg x_i$ in $C_j$, we link this copy to the node representing $x_i$.\footnote{Essentially the same construction works for SAT, since we just replace the 3-clique for clause $C_j$ with a $|C_j|$-clique, but it doesn’t change the ultimate result reducing from 3SAT instead, and it saves worrying about the size of the clauses.}

Finally, we set $k = n + m$, where $n$ is the number of variables (and thus the number of master-copy 2-cliques) and $m$ is the number of clauses (and thus the number of clause 3-cliques). This enforces the one-element-per-clique requirement.

It is easy to see that this reduction can be done in polynomial (probably even linear) time. It remains to show that it maps satisfiable formulae to graphs with independent sets of size $k$ and vice versa.

Let’s start with a satisfiable formula. Put the master copy of $x_i$ in the independent set if $x_i$ is true, otherwise put $\neg x_i$ in. This gets us one element per 2-clique. For the clauses, pick some literal in each clause that is assigned the value true and put its copy in the independent set. This gets us our one element per 3-clique, putting us up to $k$.

However, we still have to worry about marking both endpoints of an edge that crosses between cliques. For each such edge, one of its endpoints is a copy of some $x_i$, and the other of $\neg x_i$, and we can only put these copies in the independent set if the corresponding variable is true. Since $x_i$ and $\neg x_i$ can’t both be true, we are fine.

In the other direction, suppose we have an independent set of size $k$. We can read off the corresponding variable assignment directly from the 2-cliques. For each clause $C_j$, there is an independent set element corresponding to some literal in that clause. But we know that this literal is true because it is linked to the master copy of its negation. So every clause is satisfied by at least one literal and the formula is satisfiable.

Knowing that INDEPENDENT SET is \textbf{NP}-hard instantly gets us some closely-related problems. These are \textbf{CLIQUE}, the problem of determining given $(G, k)$ whether $G$ contains a clique of size $k$, which reduces from
INDEPENDENT SET by replacing the input graph by its complement; and VERTEX COVER, the problem of determining given \((G,k)\) whether \(G\) contains a set of \(k\) vertices such that every edge is incident to at least one vertex in the set, which is equivalent to asking \(G\) contains an independent set of size \(n - k\). In both cases the reduction is straightforward and obviously polynomial.

5.4.3.2 GRAPH 3-COLORABILITY

The GRAPH 3-COLORABILITY language consists of all (undirected) graphs that are 3-colorable: there exists an assignment of colors to vertices so that only three colors are used and no edge has the same color on both endpoints.

It’s easy to see that GRAPH 3-COLORABILITY is in \(\text{NP}\); just guess the coloring. The tricky part as usual is finding a reduction from a known \(\text{NP}\)-hard problem. We will use 3SAT for this.

The main technique is a widget that allows us to build logic gates out of colorings. Consider this graph:

\[
\begin{align*}
A & \quad -- \quad C \\
| & \quad \backslash \\
| & \quad E \\
| & \quad / \\
B & \quad -- \quad D
\end{align*}
\]

Suppose that we want to color the graph using colors red, green, and blue, and we happen to color both \(A\) and \(B\) red. Exactly one of \(C\), \(D\), and \(E\) must be red, and since \(E\) is the only node not adjacent to the two red nodes \(A\) and \(B\), \(E\) must also be red.

Alternatively, if at least one of \(A\) or \(B\) is not red, then \(E\) doesn’t have to be red either.

If we call red false, we’ve just built an OR gate\(^4\). Make \(E\) one of the inputs to a second copy of the widget, and we get an OR gate over three inputs, enough to handle a single clause of a 3SAT formula.

Colorings are symmetric with respect to permutations of the colors, so we can’t actually insist that red is false or green is true. But what we can do is put in a triangle \(rgb\) somewhere that acts as a master color wheel, where we just call the color applied to vertex \(r\) red, on vertex \(g\) green, and on vertex \(b\) blue. We can then build variables that are guaranteed to have \(x\) be red.

\(^4\)Sort of. If red is false and green is true, having one false and one true input allows any of the three possible colors on the output. But for 3SAT it will be enough to force the output to be false if both inputs are false.
and \(\neg x\) green or vice versa by building a triangle \(x, \neg x, b\) for each \(x\) in the formula, where \(b\) is the ground vertex from the color wheel.

The last step is to hook up the variables to the OR gates. For each clause \(x \lor y \lor z\), build a 3-input OR widget and make its input vertices coincide with vertices \(x, y,\) and \(z\). To force the output not to be red, run an edge from the output vertex to vertex \(r\) on the color wheel. Now there exists a coloring of the graph if and only if we can assign colors red and green to each \(x\) and \(\neg x\) so that no 3-input OR widget has all-red inputs. This translates back into being able to assign values false and true to each \(x\) and \(\neg x\) so that no clause has all-false inputs, which is equivalent to the original formula being satisfiable. We have thus reduced 3SAT to GRAPH 3-COLORABILITY, which completes the proof that GRAPH 3-COLORABILITY is NP-complete.

5.4.3.3 Reductions involving Hamiltonian paths

\[
\text{[[ draw pictures of the widgets ]]} \\
\text{[[\text{DIRECTED HAMILTONIAN PATH, HAMILTONIAN PATH,}]} \\
\text{\text{mention cycle versions and TSP }]]]
\]

5.5 coNP and coNP-completeness

The class coNP consists of the complements \(\overline{L}\) of all languages \(L\) in NP. By complement we mean that \(x \in L\) if and only if \(x \not\in \overline{L}\).

We don’t know if coNP = NP or not. If coNP = P or NP = P, then coNP = NP = P, since P is closed under complement. So for coNP \neq NP, we do need P \neq coNP. However, it is possible for coNP = NP \neq P, although this does collapse the polynomial-time hierarchy (see §8.1).

A language \(L\) is coNP-complete if \(L\) is in coNP and \(L' \leq_P L\) for every \(L'\) in coNP. It is not hard to show that \(L\) is coNP-complete if and only if \(\overline{L}\) is NP-complete. This instantly gives us many unnatural-looking coNP-complete problems like GRAPH NON-3-COLORABILITY and a few natural-looking ones like TAUTOLOGY (the complement of SATISFIABILITY).

---

A technical issue here is what to do with “junk” inputs that don’t actually map to an instance of \(L\) because of problem with the encoding. For any reasonable encoding, we can detect a junk input in polynomial tie and reject them in our \(L\) machine, but it seems wrong to then accept them with our \(\overline{L}\) machine. So we will treat the complement as the complement with respect to all correctly-encoded inputs. This won’t affect any results about coNP, under the assumption that we can recognize junk inputs efficiently, since we can always map junk inputs to \(L\) to some specific rejecting input to \(\overline{L}\).
CHAPTER 5. NONTERMINISM AND NP

We can also characterize problems in \(\text{coNP}\) in terms of witnesses, but the quantifier changes: \(L\) is in \(\text{coNP}\) if there is a polynomial-time \(M\) and a polynomial \(p(|x|)\) such that \(x \in L\) if and only if \(\forall w \in \{0, 1\}^{p(|x|)}, M(x, w)\) accepts. An equivalent definition in terms of a nondeterministic Turing machine is \(L\) is in \(\text{coNP}\) if there is some nondeterministic Turing machine for which every branch accepts an input \(x \in L\) and at least one branch rejects an input \(x \notin L\).

5.6 Relation to \(\text{EXP}\)

The class \(\text{EXP} = \bigcup_{k=1}^{\infty} \text{TIME}(2^{n^k})\) consists of all languages decidable in \textit{exponential time}. Similarly, the class \(\text{NEXP} = \bigcup_{k=1}^{\infty} \text{TIME}(2^{n^k})\) consists of all languages decided by a nondeterministic Turing machine in exponential time.

It’s easy to show that \(\text{NP} \subseteq \text{EXP}\), since we can just try all \(2^{n^k}\) witnesses.

A more indirect connection between the classes is that if \(\text{EXP} \neq \text{NEXP}\), then \(\text{P} \neq \text{NP}\). This makes more sense if we consider the contrapositive: \(\text{P} = \text{NP}\) implies \(\text{EXP} = \text{NEXP}\). The reason is that we can take any language \(L\) in \(\text{NEXP}\), replace it with the \textit{padded} language \(L' = \{x; 1^{2^{|x|}} \mid x \in L\}\). So now if \(L\) can be decided by a nondeterministic Turing machine in time \(O(2^{|x|^k})\), then \(L'\) can be decided by a nondeterministic Turing machine in time \(O(|x; 1^{2^{|x|}|^k})\), putting \(L'\) in \(\text{NP}\). Under the assumption \(\text{P} = \text{NP}\), this also makes \(L'\) decidable by a deterministic Turing machine in time polynomial in \(2^{|x|}\), or exponential in \(|x|\). But a machine in \(\text{EXP}\) can simulate such a Turing machine running on just \(x\) by first copying \(x\) to a work tape and then constructing the suffix \(1^{2^{|x|}}\) for itself. This puts \(L\) in \(\text{EXP}\) and shows \(\text{EXP} = \text{NEXP}\).

Note this doesn’t necessarily work in the other direction: for all we know, \(\text{P} \neq \text{NP}\) but \(\text{EXP} = \text{NEXP}\). The problem is that while we can always pad a short input to make it a long one, we can’t “unpad” a long input to make it a short one.
Chapter 6

Diagonalization

The basic idea of diagonalization goes back to Cantor’s argument that there is no surjective map from any set $S$ to its powerset $P(S)$. The proof is that, given any map $f : S \to P(S)$, the set $T = \{ x \in S \mid x \notin f(x) \}$ cannot equal $f(x)$ for any $x$ without producing a contradiction. (If $T = f(x)$, and $x \in T \leftrightarrow x \notin f(x)$, then $x \in T \leftrightarrow x \notin T$.) A similar trick was used by Gödel to prove undecidability of the Halting Problem. It’s the last version of diagonalization that we will make the most use of, so we’ll start by describing how it works.

6.0.1 Undecidability of the Halting Problem

The Halting Problem asks whether it is possible to construct a Turing Machine $H$ that, given a representation $\langle M \rangle$ of a Turing machine $M$, and an input $x$, accepts if $M(x)$ halts and rejects if $M(x)$ runs forever. Turing’s argument involves constructing a machine that provably does the opposite of what $H$ predicts.

This is the equivalent of demonstrating that your local fortune-teller can’t really predict the future because they can’t tell whether the next thing you say is going to be “yes” or “no.” In both cases, the trick only works if you can wait for the predication before choosing what to do. But if $H$ exists, we can do this.

The bad machine $M$ we are going to construct takes as input a description $\langle M' \rangle$ of some machine $M'$, runs $H(\langle M' \rangle, \langle M' \rangle)$, then halts if and only if $H(\langle M' \rangle, \langle M' \rangle)$ rejects. It’s not hard to see that we can implement $M$ given $H$, since the extra work is just a matter of copying the input twice, and instead of halting when $H$ does, using the halting state of $H$ to decide
whether to really halt (if \( H \) rejects) or not to halt at all (if \( H \) accepts), perhaps by moving to a state that just moves one of the tape heads off to the right forever.

So now what happens if we run \( H(\langle M, \lambda \rangle, \langle M, \lambda \rangle) \)? This should accept only if and only if \( M(\langle M, \lambda \rangle) \) halts. But \( M(\langle M, \lambda \rangle) \) halts if and only if \( H(\langle M, \lambda \rangle, \langle M, \lambda \rangle) \) rejects. So we’ve constructed a specific machine \( M \) and input \( \langle M, \lambda \rangle \) where \( H \) gives the wrong answer. So we have:

**Theorem 6.0.1 (Halting Problem).** There does not exist a Turing machine \( H \) that always halts, such that \( H(\langle M, x \rangle) \) accepts if and only if \( M(x) \) halts.

In other words, the Halting Problem is **undecidable**—there is no Turing machine that decides it.

This turns out to have consequences that go beyond just testing if a machine halts or not. Just as polynomial-time reductions from know \( \text{NP} \)-hard problems can show that other problems are \( \text{NP} \)-hard, computable reductions from the Halting Problem can show that other problems are also undecidable.

A very general result that follows from this is **Rice’s Theorem**. This says that testing any non-trivial semantic property of a Turing machine is also undecidable, where a semantic property depends only whether the machine halts or produces a particular output for each input, and not on the details of the computation, and a property is non-trivial if there is at least one machine for which it holds and at least one for which it doesn’t.

**Corollary 6.0.2 (Rice’s Theorem [Ric53]).** Let \( P \) be a non-trivial semantic property of Turing machines. Then \( P \) is undecidable.

**Proof.** Suppose \( P \) is true for a machine that never halts. Let \( M_0 \) be a machine for which \( P \) is false (such a machine exists because \( P \) is non-trivial). Suppose there is a machine \( M_P \) that decides \( P \). We can use \( M_P \) as a subroutine to solve the halting problem.

Let \( \langle M, x \rangle \) be a machine-input pair for which we want to know if \( M(x) \) halts. Construct a machine \( M' \) that takes an input \( y \) and runs \( M \) on \( x \). If \( M \) doesn’t halt, neither does \( M' \), and so \( M' \) has property \( P \). If \( M \) does halt, \( M' \) switches to running \( M_0 \) on \( y \), producing the same output (or failing to halt on the same inputs) as \( M_0 \), giving \( M' \) property \( P \). So to decide if \( M(x) \) halts, construct \( M' \) and run \( M_P \) on it. The Turing machine that does this then solves the Halting Problem, contradicting Theorem 6.0.1.

If \( P \) is false for a machine that never halts, pick a machine \( M_1 \) for which \( P \) is true, and apply essentially the same construction. \( \Box \)
Note that Rice’s Theorem is only about inputs and outputs. There are non-semantic properties (like “does this machine run in polynomial time on input $x$?”) that are easily decidable even though they are not non-trivial. It also only works because we consider machines that might not halt.

### 6.1 Hierarchy theorems

In complexity theory we don’t care too much about machines that might not halt, because all of our complexity classes only include machines that always halt. But we can use the essentially the same proof as for the Halting Problem to show that there are functions that cannot be computed within a given space or time bound. The tricky part is showing that having a bit more space or time makes these functions computable.

For both theorems the idea is to build a language $L$ consisting of machines with inputs reject within some resource constraint, show that deciding $L$ in less than the resource constraint gives a contradiction, then show that with more resources we can use a universal TM to decide $L$. Since we didn’t do universal machines yet, we’ll have to describe them now.

We’ll start with the Space Hierarchy Theorem, because the construction is less fiddly.

#### 6.1.1 The Space Hierarchy Theorem

The **Space Hierarchy Theorem** says that getting more than a constant factor more space is enough to solve more problems:

**Theorem 6.1.1.** If $g(n)$ is a space-constructible function that is $\Omega(\log n)$, and $f(n) = o(g(n))$, then $\text{SPACE}(f(n)) \subset \text{SPACE}(g(n))$.

To prove this, we need to find a language $L$ that is in $\text{SPACE}(g(n))$ but not $\text{SPACE}(f(n))$. The following language will have this property:

$L = \{ \langle \_M, x \rangle \mid M \text{ rejects } \langle \_M, x \rangle \text{ using at most } g(n) \text{ space, one work tape, and } |\Gamma| = 2 \}$

The restriction on $M$’s work tape alphabet $\Gamma$ avoids some annoyances in trying to simulate machines with larger alphabets.

First we’ll show that $L \notin \text{SPACE}(f(n))$. Let $R$ be a machine that supposedly decides $L$ using $O(f(n))$ space. From Lemma 3.1.1 there is another machine $R'$ that produces the same output as $R$ in $O(f(n))$ space (with a bigger constant) using one work tape and a two-bit alphabet (not counting the blank symbol). We’ll use $R'$ to get a contradiction.
Let $x$ be any string long enough that the space complexity of the execution of $R'$ on $\langle \downarrow R', x \rangle$ is less than $g(n)$. We know that such a string exists, because $R'$ takes $O(f(n))$ time, and whatever the constants hiding in the $O$ this must drop below $g(n)$ for sufficiently large $n$. Now let us ask what $R'(\langle \downarrow R', x \rangle)$ returns.

If it accepts, then $R'(\langle \downarrow R', x \rangle)$ does not reject and so $\langle \downarrow R', x \rangle$ is not in $L$: $R'$ (and thus $R$) computes the wrong answer.

If it rejects, then $R'(\langle \downarrow R', x \rangle)$ rejects using at most $g(n)$ space. By construction, $R'$ also uses one work tape and two non-blank work tape symbols. So $\langle R', x \rangle$ is in $L$. Again, we get the wrong answer. It follows that $R$ gives the wrong answer for at least one input $\langle M, x \rangle$, and so $R$ does not decide $L$.

Next, we’ll show that $L \in \mathbf{SPACE}(g(n))$. This requires constructing a universal Turing Machine to simulate $M$ on $x$. We also need to be able to detect if $M$ uses more than $g(n)$ space, or violates its parole in some other way.

Our machine $R^*$ will use several work tapes:

1. A tape of size $O(\log n)$ to store a binary representation of $M$’s state. The argument for $O(\log n)$ being enough is that $|\downarrow M|$ will be at least linear in the number of states, because of $\delta$.

2. An input pointer tape of size $O(\log n)$. This will store a binary representation of the offset into $x$ corresponding to the position of the simulated input tape head. We need this because we will be using our real tape head to run back and forth between reading $\downarrow M$ and $\downarrow x$, and we can’t write to the input tape to mark our position.

3. A second pointer tape that remembers the index of the current head position on the input, relative to the start of $x$. We need this to know when we have reached the right position to match the first pointer.

4. A copy of $M$’s work tape. We don’t have to do anything clever with this, since we can just use our own head to keep track of $M$’s head.

5. A tape of size $g(n)$ holding (after an initial computation) the string $1^{g(n)}$. Whenever we move the simulated work tape head, we’ll move this tape’s head as well, so if it goes off the end of the string, we’ll know that $M$ is trying to use too much space. The very first thing our simulation does before starting up $M$ is to compute $1^{g(n)}$, which we can do (possibly using an addition $g(n)$ space somewhere, although I think we can borrow the work tape for this) since $g(n)$ is space-constructible.
6. A binary counter of size \( g(n) + \log |Q| + \lceil \log g(n) \rceil + \lceil \log n \rceil + 1 = O(g(n) + \log n) \) initialized to all ones. We use this to avoid loops; if we try to decrement the counter and find that it is

Everything else is constant size so we can store it in the finite-state controller. Simulating a step of \( M \) consists of:

1. Moving the input head across \( x \) until the two input pointers match, and collecting the input symbol.

2. Moving back to \( \downarrow M \downarrow \) and searching for an entry in \( \downarrow \delta \downarrow \) that matches (a) the state \( \downarrow q \downarrow \) on the state tape, (b) the input symbol stored in the finite-state controller, and (c) the work-tape symbol under the head on the simulated work tape. Having found this entry, we copy the new state \( q' \) to the state tape, do a write and move on the work tape if we need to, and increment or decrement the input-tape-head pointer as needed to simulate moving the input tape.

3. Decrementing the binary counter. If the counter is already at 0 or the head on the space-bound tape moves onto a blank, reject. In the first case, the machine has run for at least \( 2 \cdot 2^{g(n)} g(n) |Q| \) steps, meaning that somewhere during its execution the contents of its work tape, the position of its work tape head, the position of its input tape head, and its state have repeated: it’s looping and will never reject. In the second case, it used too much space.

4. If after many such steps the simulation rejects, accept. If it accepts, reject.

This might take a while (I think we can get it each simulated step down to \( O(g(n) \log n) \) time if we are careful, and the initialization looks like it takes \( O(g(n)) \) time altogether), but the important thing is that at no time do we use more than \( O(g(n) + \log n) \) space, and \( R^* \) decides \( L \). So \( L \in \text{SPACE}(g(n) + \log n) = \text{SPACE}(g(n)) \) assuming \( g(n) = \Omega(\log n) \).

6.1.2 The Time Hierarchy Theorem

The **Time Hierarchy Theorem** is similar to the Space Hierarchy Theorem, but the gap is wider:

**Theorem 6.1.2.** If \( g(n) \) is a time-constructible function, and \( f(n) = o(g(n)) \), then \( \text{TIME}(f(n)) \not\subseteq \text{TIME}(g(n) \log g(n)) \).
CHAPTER 6. DIAGONALIZATION

By analogy to the proof of the Space Hierarchy Theorem, a first try at a language for this one would be

\[ L = \{ \langle L, \cdot, x \rangle \mid M \text{ rejects } \langle L, \cdot, x \rangle \text{ using at most } f(n) \text{ time and a tape alphabet of size } 2 \} \]

This will give us something to start with, but getting the full-blown theorem will require some more tinkering. The main issue is that building a time-efficient universal Turing machine is harder than building a space-efficient one (this also accounts for the extra \( \log g(n) \) factor), and we’ve learned from the SHT proof that the diagonalization side of the argument is pretty robust to small changes in \( L \), so it will be helpful to adjust the definition of \( L \) a bit after we see the hard parts of the upper bound part of the argument in order to make that part easier.

First, though, the diagonalization part. Suppose \( R \) decides \( L \), and \( R \) runs in \( o(g(n)) \) time, then running \( R(\langle R, \cdot, x \rangle) \) gives a contradiction when \( x \) is large enough that \( R \)'s running time drops below \( g(n) \) exactly. If \( R(\langle R, \cdot, x \rangle) \) rejects in such a case, then \( \langle R, \cdot, x \rangle \) is in \( L \) (by the definition of \( L \)), meaning that \( R \) just gave the wrong answer on this input. But the same thing happens if it rejects. So we get a contradiction either way, and \( R \) either does not decide \( L \) or it doesn’t run in \( o(g(n)) \) time.

But now we need to show that a machine that does decide \( L \) runs in a reasonable amount of time. We’ll start with a simple, direct simulation of the input machine \( M \), and then worry about improving the running time later\[1\]

Here’s the easy approach: Build a machine \( R^* \) that has:

1. A tape storing \( x \). We’ll copy \( x \) to this tape at the start, costing time \( O(n) \). This exists mostly so we can have one tape head pointing into \( \langle L, \cdot, x \rangle \) and another (on a separate tape) into \( x \), so we don’t have to waste time moving back and forth between two simulated heads.

2. A tape storing \( M \)'s \( k \) work tapes. As in the SHT proof, we’ll just store these consecutively, with markers for the \( k \) heads.

3. A tape storing the input to \( \delta \). This will be a state of \( M \) expressed in binary, plus one symbol for each of the \( k + 1 \) work and input tapes. The total size will be \( O(\log |Q_M| + k) \).

4. A tape storing the output of \( \delta \). Pretty much the same as the previous tape, but we also need \( k \) symbols from \( \{L, S, R\} \) to track how the heads move.

\[1\]This approach is inspired by some lecture notes from Luca Trevisan.
5. A “fuse” tape that holds $01^{g(n)}$. It costs $O(g(n))$ time to set this up, leaving the head on the rightmost cell. Every time we simulate a step of $M$, we move the head one cell to the left, and when it hits the 0, we know that $M$ took more than $g(n)$ steps and we can reject.

A step of $R^*$ is also similar to a step in the SHT proof: we gather up the input to $\delta$ ($O(kg(n))$ time), scan through $M$ to find the matching transition ($O(|M|) = O(n) = O(g(n))$ time), copy the output to the result tape ($O(n)$ time), and then scan through the work tape to update everything ($O(k^2(g(n))^2)$ time, including the cost of copying cells up to $O(k)$ positions to the right to make room for new cells). Somewhere in here we also check the fuse (free, since we can move the fuse head in parallel with something we are doing anyway).

The total cost per step is $O(k^2g(n)) = O(n^2g(n))$, using $n$ as a very crude upper bound on $k$. Since we have to simulate $O(g(n))$ steps, we’ve shown that $\text{TIME}(o(g(n))) \subseteq \text{TIME}(n^2(g(n)))$. This is something, but we can do better.

If we look at the expensive parts of the simulation, the big costs we need to knock down are (a) the $O(kg(n))$ cost to traverse the work tape, and (b) the $O(n)$ cost to traverse $\delta$. For (a), we will use a clever data structure appearing in Hennie and Stearns’ original THT paper [HS66] to knock the amortized cost per operation down to $O(k \log g(n))$, and make a small tweak to $L$ to get rid of the extra $k$. There’s not much we can do about (b), so we will get around this by another tweak to $L$ to insist that $|\cdot M|$ is small enough relative to $n$ that the overhead is dominated by the $O(\log g(n))$ per step that we are already paying.

Here is the data structure. The description below follows the presentation in [AB07, §1.A].

We’ll first map the $k$ work tapes onto one tape by interleaving: for example, with 3 tapes, we will store their cells as 123123 etc. This means that if we park a head on the leftmost of $k$ cells, we can reach any of the other cells in $O(k)$ time, assuming those tapes’ heads are also in the same place. Unfortunately, they probably won’t be after the first step.

We deal with this by moving the contents of the tape instead of the head. Doing this naively (for example, shifting every cell on tape 2 one position to the left) will be expensive. So instead we use a representation with gaps in it that we use to store values that we are pushing in one direction, empty out increasingly large regions in bulk when we need more space. We’ll describe how to do this for a single tape, and then apply the interleaving idea to the representation to handle multiple tapes.
The central cell of the tape we will just store by itself. The right-hand side we divide into zones \( R_1, R_2, \ldots \), where zone \( R_i \) contains \( 2^i \) cells. Similarly, the left-hand side is divided into \( \ldots, L_2, L_1 \). Each zone is either empty (filled with a special not-in-use symbol), half-full (half-filled with that symbol and half with simulated tape symbols), or full (no not-in-use symbols at all). We maintain the invariant that the \( j \)-th cell of the \( L \) side is full if and only if the \( j \)-th cell of the \( R \) side is empty: this means that when \( L_i \) is empty, \( R_i \) is full, etc.

To pop a symbol from \( R \), we look for the leftmost zone \( R_i \) that is not empty. This will either contain \( 2^i \) symbols, in which case we pop one of them and use the remaining \( 2^i - 1 \) symbols to make all of \( R_1 \) through \( R_i \) half-full; or it will contain \( 2^{i-1} \) symbols, in which case we pop one, use the remaining \( 2^{i-1} - 1 \) symbols to make \( R_i \) through \( R_{i-1} \) half-full and leave \( R_i \) empty. In either case we can do the copying (using an extra work tape) in \( O(2^i) \) time.

A push is symmetric (and must be because of our invariant): we look for the first non-full zone, fill it either all the way to full or to half-full as needed to clear out earlier zones to half-full, and then add the symbol to \( R_1 \).

The important thing is that once we fill in or clean out \( R_i \), we don’t do anything to it again until we either fill up or completely empty all zones \( R_1 \) through \( R_{i-1} \). So our \( O(2^i) \)-cost operations on \( R_i \) can only happen every \( O(2^i) \) steps, giving an amortized cost \( O(1) \) per step per zone. We have \( O(\log g(n)) \) zones, so the cost per step of simulating one tape is \( O(\log g(n)) \) and of simulating \( k \) tapes is \( O(k \log g(n)) \), giving a total cost across the entire computation of \( O(kg(n) \log g(n)) \).

If we add in the cost of scanning \( \ulcorner M \urcorner \), we get a total cost of \( O(kg(n)(\log n + |\ulcorner M \urcorner|)) \).

We’ll now adjust \( L \) to get rid of all the extra junk in this expression. Our new \( L' \) will be the set of all inputs \( \langle \ulcorner M \urcorner, x \rangle \) where

1. \( M \) rejects \( \langle \ulcorner M \urcorner, x \rangle \) in time at most \( g(n)/k \), where \( k \) is the number of work tapes used by \( M \);
2. \( M \) has \(|\Gamma| = 2\); and
3. \(|\ulcorner M \urcorner| < \log n \).

Since the extra conditions can be checked in \( O(n) = O(g(n)) \) time, and we can compute \( 1^{g(n)/k} \) in time \( g(n) \) without being especially clever, we can easily decide \( L' \) using the above construction in \( O(kg(n)/k)(\log n + \log n) + g(n) = O(g(n) \log g(n)) \) time. The only thing left is to show we didn’t break the diagonalization argument.
Suppose \( R \) is a machine that decides \( L \) in \( f(n) = o(g(n)) \) time. Then there is also machine \( R' \) that decides \( L \) in \( o(g(n)) \) time with a restricted work-tape alphabet. For any fixed \( k \) (for example, the number of work tapes possessed by \( R' \)), there is some \( n_0 \) so that \( R' \) decides \( L \) in no more than \( g(n)/k \) time for all \( n \geq n_0 \) (this just falls out of the definition of \( o(g(n)) \)). So now we just make \( |x| \geq \max(n_0, 2^{\mathcal{L}_M}) \), and get \( R' \) to accept \( \langle \mathcal{L}_R', x \rangle \) if and only if \( \langle \mathcal{L}_R', x \rangle \notin L' \). So \( L' \notin \text{TIME}(f(n)) \).

### 6.2 Hierarchy theorems for nondeterminism

So far, we have only considered deterministic computation. We can also ask if there are corresponding space and time hierarchy theorems for nondeterministic computation.

For space, a result very similar to Theorem 6.1.1 holds: \( \text{NSPACE}(f(n)) \subsetneq \text{NSPACE}(g(n)) \) whenever \( f(n) = o(g(n)) \), \( g(n) \) is space-constructible, and \( g(n) = \Omega(\log n) \). Pretty much the same proof works, with one complication: to get the upper bound, we need to build contradiction, we have to build a nondeterministic machine that accepts if and only if the nondeterministic machine it is simulating rejects, and doing this directly by just using the simulated machine’s hint doesn’t work. Fortunately, the Immerman-Szelepcsényi Theorem \[9.4\] says that \( \text{NSPACE}(g(n)) = \text{coNSPACE}(g(n)) \) when \( g(n) = \Omega(\log n) \), so we can just build a simulator that accepts \( \langle M, x \rangle \) if \( M \) accepts \( \langle M, x \rangle \), and appeal to Immerman-Szelepcsényi to reverse the output.

For \( \text{NTIME} \) we can’t do this. Instead, we need to use a different technique due to Cook [[[ which we will cite but not actually do, see [AB07, §3.4] if you want to know how this works ]]]

### 6.3 Ladner’s Theorem

Ladner’s Theorem shows that if \( \text{P} \neq \text{NP} \), there are sets in \( \text{NP} \) that are neither polynomial-time computable nor \( \text{NP} \)-complete. The proof is essentially a diagonalization argument, where we construct a single bad set \( A \) by alternating between making it disagree with the output of the next polynomial-time machine \( M_i \) and making it fail to decide \( \text{SAT} \) after running \( \text{SAT} \) instances through the next polynomial-time function \( f_i \). Some additional trickery puts the set in \( \text{NP} \), giving the full result.

**Theorem 6.3.1.** If \( \text{P} \neq \text{NP} \), then there is a set \( A \in \text{NP} \setminus \text{P} \) that is not \( \text{NP} \)-complete.
Proof. This particular proof is similar to Ladner’s original proof [Lad75], but the presentation below follows a later paper by Downey and Fortnow [DF03], although we change the definition of $A$ slightly to avoid a possible bug in that paper.

Let $M_1', M_2', \ldots$ enumerate all Turing machines. Let $M_i$ be the machine that first computes $1^n$ on an extra work tape, then runs $M_i'$ for $n^i$ steps, rejecting if it does not terminate in this time. Since $n^i$ is time-constructible, $M_i$ runs in $O(n^i)$ time, and since every Turing machine appears infinitely often in the $M_i'$ list (because we can always pad with extra unused states), every polynomial-time Turing machine appears as some $M_i$.

Similarly let $f_1, f_2, \ldots$ enumerate all polynomial-time computable functions. The language $A$ is given by \{ $x$ | $x \in \text{SAT}$ and $g(|x|)$ is even \}, where $g$ (which tells us where to put the gaps) is a function to be determined. The idea is that the layers of $A$ corresponding $g$ enforce that some $M_i$ can’t compute $A$ (because otherwise we could use a modified version of $M_i$ to decide SAT in P), while the empty odd layers enforce that some $f_i$ can’t reduce SAT to $A$ (because otherwise we could use a modified version of $f_i$ to decide SAT in NP). We switch to the next layer once we detect that we have knocked out a particular $M_i$ or $f_i$, which may require waiting until $n$ is big enough that we can test all inputs up to the first bad one using brute force in polynomial time. But since $n$ keeps growing forever, we can do this eventually for any fixed $M_i$ or $f_i$ if we wait long enough.

We’ll define $g$ recursively, and show that it is computable in deterministic polynomial time by induction on $n$. Because this construction depends on the running time of some of its own components, it may be easiest to think of it as defining a particular polynomial-time machine and defining $g$ to be whatever that machine outputs.

The issue is that when testing $f_i$ to see if $g(n) = 2i + 1$ should be increased, Downey and Fortnow check all inputs $x$ with $|x| < \log n$ and wait until $\log^{g(n)} n < n$ to ensure that the computation of $f_i(x)$ on each of these inputs is $O(n)$. But it is still necessary to test if $f_i(x)$, which may have size as much as $\log^i |x|$, is in $A$, which may require testing if it is in SAT. Even taking into account the difference between $i$ and $g(n) = 2i + 1$, we still get outputs from $f_i$ that are of size polynomial in $n$, so we can’t just test for membership in SAT by checking all the assignments. We also can’t appeal to the fact that we are ultimately computing $g$ using a nondeterministic machine, because we need to know both when $g$ is positive and when it isn’t. The proof given here avoids the issue entirely by putting a time bound on the testing procedure and arguing that $n$ will eventually get big enough that $M_i$ or $f_i$ fails within the time bound, without attempting to compute the bound explicitly. This idea is pretty much the same as the approach used in Ladner’s original paper.
Start with \( g(0) = g(1) = 2 \). We then compute \( g(n+1) \) by first computing \( g(n) \), and then branching based on whether it is odd or even:

1. If \( g(n) = 2i \), lazily generate a list of all inputs \( x \) with \( |x| < n \) in increasing order by \( x \). For each \( x \), simulate \( M_i(x) \), and compare the output to whether or not \( x \) is in \( A \), which we can test in time exponential in \( |x| \) just by brute-forcing the solution to SAT if \( g(|x|) \) is even.

This procedure is not polynomial-time, but we can truncate it after \( n \) steps. If we find a bad \( x \) for which \( M_i \) gives the wrong answer during those \( n \) steps, then we set \( g(n+1) = g(n) + 1 \) and move on to the next machine. If not, we let \( g(n+1) = g(n) \) and figure that we will catch \( M_i \) eventually when \( n \) is large enough. This works because no matter how long it takes to check each \( x \), as long as that time is finite, eventually \( n \) exceeds whatever time is needed to check all the \( x \) up to the first bad one.

2. If \( g(n) = 2i + 1 \), do the same thing to \( f_i \). Enumerate all \( x \), run each through \( f_i \), and test if \( |f_i(x)| \leq n \) (so we don’t create a cycle trying to compute \( g(|f_i(x)|) \) and \( x \in \text{SAT} \leftrightarrow g(x_i) \in A \). As before we truncate after \( n \) steps.

If we find such and \( x \), \( f_i \) doesn’t reduce SAT to \( A \), so we can set \( g(n+1) = g(n) \) and keep going. If we dont, we set \( g(n+1) = g(n) \) and take solace in the notion that \( f_i \)’s foot shall slide in due time [Edw41].

Since we bound the testing cost for each \( n \) by \( O(n) \), the total cost of computing \( g(n) \) is bounded by the recurrence the form \( g(n+1) = g(n) + O(n) \), which gives \( g(n) = O(n^2) \). So \( A \) is in \( \text{NP} \), because a non-deterministic machine can, in polynomial time, compute \( g(|x|) \) deterministically, and then either reject (if \( g(|x|) \) is odd) or run SAT on \( x \) (if \( g(|x|) \) is even).

We now argue that one of three things happens:

1. If \( g(n) = 2i \) for all \( n \) greater than some \( n_0 \), then \( M_i \) correctly computes SAT in polynomial time for all but a finite number of \( x \) with \( |x| \leq 0 \). Add these cases to \( M_i \) using a lookup table to put SAT in \( \text{P} \).

2. If \( g(n) = 2i + 1 \) for all \( n \) greater than some \( n_0 \), then \( f_i(x) \in A \leftrightarrow x \in \text{SAT} \) for all \( x \), and \( A \) is finite. So make a new machine that runs \( f_i(x) \) and looks up the result in a table of all members of \( A \) to put SAT in \( \text{P} \).

3. The remaining case is when \( g \) is unbounded. Then
   
   (a) No polynomial-time \( M_i \) decides \( A \), so \( A \notin \text{P} \),
(b) No polynomial-time \( f_i \) reduces SAT to \( A \), so \( A \) is not \( \text{NP} \)-complete, and

(c) Some polynomial-time nondeterministic Turing machine decides \( A \), so \( A \in \text{NP} \).

Since the third case is the only one consistent with \( P \neq \text{NP} \), the theorem holds. \( \square \)

If \( P \neq \text{NP} \), Ladner’s Theorem demonstrates the existence of \( \text{NP} \)-intermediate sets, ones that lie strictly between \( P \) and the \( \text{NP} \)-complete sets. Indeed, by applying the construction in the proof iteratively, it is possible to show that there is an entire infinite chain of \( \text{NP} \)-intermediate sets, each irreducible to the next, yet all in \( \text{NP} \setminus P \). But these sets are not especially natural, and it is reasonable to ask if we can point to any practically-motivated sets that might be \( \text{NP} \)-complete.

Ladner \([\text{Lad75}]\), citing Karp \([\text{Kar72}]\), mentions three sets that were candidates for being \( \text{NP} \)-intermediate at the time he proved his theorem: PRIME (is \( x \) prime?), LINEAR INEQUALITIES (is a given linear program feasible?), and GRAPH ISOMORPHISM (can we turn \( G \) into \( H \) just by relabeling the vertices?). All of these were known to be in \( \text{NP} \setminus P \) as of 1975, and none of them were known to be \( \text{NP} \)-complete, but there was no argument that any of them in particular were \( \text{NP} \)-intermediate if \( P \neq \text{NP} \). As of 2017, two of them (PRIME and LINEAR INEQUALITIES \([[[ \text{citations needed} ]]]) are known to be in \( P \). The status of GRAPH ISOMORPHISM is still open. \([[[ \text{Babai’s quasipolytime algorithm} ]]])

Whether there are other potentially \( \text{NP} \)-intermediate problems that are natural depends somewhat on one’s definition of “natural.” See [http://cstheory.stackexchange.com/questions/20930/why-are-so-few-natural-candidates-for-np-intermediate-status](http://cstheory.stackexchange.com/questions/20930/why-are-so-few-natural-candidates-for-np-intermediate-status) for an example of how different people can have very different perspectives on this question.
Chapter 7

Oracles and relativization

[[[ intro ]]]

7.1 Oracle machines

Given a machine $M$ in some model, the machine $M^A$ is obtained by taking $M$ and adding a write-only oracle tape that gives $M$ the ability to determine in a single tape if the string $x$ written to the oracle tape is in $A$. When $X$ and $Y$ are complexity classes, we also write $X^Y$ for the class of languages computable by a machine in class $X$ using an oracle from class $Y$.

This definition makes $X^Y$ at least as powerful as the stronger of $X$ and $Y$, and may give it even more power: for example, if $\text{NP} \neq \text{coNP}$, then $\text{NP} \subset \text{P}^\text{NP}$ since in $\text{P}^\text{NP}$ we can solve any problem in $\text{NP}$ just by asking the oracle for its opinion, but we can also solve any problem in $\text{coNP}$ by asking the oracle for its opinion and giving the opposite answer. And this is all without even taking advantage of the ability to ask multiple questions and have the later questions depend on the outcome of earlier ones.

7.2 Relativization

We say that a proof relativizes if it is not affected by adding an oracle, and that a hypothesis relativizes if there is an oracle that makes it true and another oracle that makes it false. If a hypothesis relativizes, it can only be proved or disproved using a technique that doesn’t relativize.

All of the techniques we have seen so far relativize, because if we only use the fact that we can simulate a machine $M$ using some machine $U$, then it
also holds that we can simulate $M^A$ using $U^A$. This is bad news for resolving $P \overset{?}{=} \text{NP}$, because that hypothesis also relativizes.

### 7.2.1 The Baker-Gill-Solovay Theorem

\[
\text{EXPCOM} = \{\langle M, x, 1^n \rangle \mid M \text{ outputs 1 on } x \text{ within } 2^n \text{ steps} \}
\]
gives oracle for which $P^A = \text{NP}^A$. Proof: Adjust $1^n$ to let EXPCOM search everything. 

Baker-Gill-Solovay: construct oracle $B$ for which $\{1^n \mid \exists x \in B, |x| = n\}$ is in $\text{NP} \setminus P$. Idea is to look for each $M_i$ where it queries, and pick an $n$ large enough that no smaller machine queries it yet and $M_i$ does fewer than $2^n - 1$ queries on $1^n$. Now set the query it didn’t do to be the important one.

### 7.2.2 The Random Oracle Hypothesis

mention Random Oracle Hypothesis of Bennett and Gill and its $\text{IP}^A \neq \text{PSPACE}^A$ counterexample in Chang et al.? Mostly of historical interest but still cute, and explains what happens if you have a model of computation where everybody has access to the same fixed set of actual random bits. Note that this is not the same as generating new random bits for each computation. 

$P^A \neq \text{NP}^A$ for a random $A$, but this doesn’t mean that we can’t solve SAT in real life, because SAT isn’t NP-complete with respect to a random $A$.

### 7.2.3 How not to relativize

talk about what a proof would have to do to not relativize, example would be SAT is NP-complete

### 7.3 The oracle polynomial-time hierarchy

Oracles give us one of several definitions of the polynomial-time hierarchy. Let $\Delta_p^0 = \Sigma_p^0 = \Pi_p^0 = P$. Then define

\[
\Delta_{k+1}^p = P^{\Sigma_k^p}
\]
\[
\Sigma_{k+1}^p = \text{NP}^{\Sigma_k^p}
\]
\[
\Pi_{k+1}^p = \text{coNP}^{\Sigma_k^p}
\]
This gives what we believe to be an infinite tower of complexity classes, with each $\Delta^p_k$ contained in $\Sigma^p_k$ and $\Pi^p_k$, and $\Sigma^p_k$ and $\Pi^p_k$ contained in $\Delta^p_{k+1}$. The union of all of these classes is $\text{PH}$, the polynomial-time hierarchy. For the moment, we will refer to this version specifically as the oracle polynomial-time hierarchy, to distinguish from some other definitions, but these definitions will turn out to be equivalent in the end.
Chapter 8

Alternation

8.1 The alternating polynomial-time hierarchy

The alternating polynomial-time hierarchy is defined by extending the certificate version of the definition of $\mathbf{NP}$. In this definition (see [AB07, Definition 5.4]), a language $L$ is in $\Sigma^p_k$ if there is a polynomial $p$ and a polynomial-time machine $M$ such that $x \in L$ if and only if

$$\exists w_1 \forall w_2 \exists w_3 \ldots Q w_k M((x, w_1, w_2, \ldots, w_k) \text{ accepts},$$

where $Q$ represents either a $\exists$ or a $\forall$ quantifier as appropriate and each $w_i$ has $|w_i| \leq p(|x|)$.

A language is in $\Pi^p_k$ if its complement is in $\Sigma^p_k$, or equivalently if there is an alternating formula for membership in $L$ using $k$ quantifiers starting with $\forall$. Unlike the oracle definition, there does not seem to be a natural way to define $\Delta^p_k$ in terms of alternating formulas.

8.2 Equivalence to alternating Turing machines

8.3 Complete problems

[[[ $\Sigma_k$-SAT ]]]
8.4 Equivalence to oracle definition

We generally don’t distinguish between the alternating and oracle versions of the polynomial-time hierarchy, because they give the same classes.

**Theorem 8.4.1.** For all \( k \), \( \Sigma_k^{p, \text{oracle}} = \Sigma_k^{p, \text{alternating}} \).

**Proof.** By induction on \( k \). When \( k = 0 \), both are \( \text{P} \).

For larger \( k \), suppose that the theorem holds for \( \Sigma_{k-1}^{p} \). The easy direction is \( \Sigma_k^{p, \text{oracle}} \supseteq \Sigma_k^{p, \text{alternating}} \). Recall that \( L \) is in \( \Sigma_k^{p, \text{alternating}} \) if there is a formula \( \exists w \text{P}(x, w) \) that is true when \( x \in L \), where \( w \) has size polynomial in \( x \) and \( \text{P} \) is computable in \( \Pi_{k-1}^{p, \text{alternating}} \). By the induction hypothesis, \( \text{P} \) is also computable by some machine in \( \Pi_{k-1}^{p, \text{oracle}} \). But then a \( \text{NP}^M \) machine can decide \( L \), by guessing \( w \) and verifying it using \( M \). This puts \( L \) in \( \text{NP}^{\Pi_{k-1}^{p}} = \text{NP}^{\Sigma_{k-1}^{p}} = \Sigma_k^{p, \text{oracle}} \).

The other direction requires some trickery. [[[ guess the oracle calls in advance and then check them using \( \Pi \) or \( \Sigma \) formulas based on the guessed answer ]]]

8.5 \( \text{PH} \subseteq \text{PSPACE} \)

Problems in the polynomial-time hierarchy can be powerful, but any of them can be solved in polynomial space.

In fact, we can encode any problem in \( \text{PH} \) by reduction to a problem called **true quantified boolean formula** or \( \text{TQBF} \). \( \text{TQBF} \) consists of all true formulas of the form \( \exists x_1 \exists x_2 \ldots \exists x_n \Phi(x_1, \ldots, x_n) \) where each \( Q_i \) is either \( \forall \) or \( \exists \) and each \( x_i \) represents a Boolean value.

For example, any problem in \( \Sigma_1^p = \text{NP} \) can be encoded as a formula of the form \( \exists x_1 \ldots \exists x_n \Phi(x_1, \ldots, x_n) \), where \( \Phi \) is the SAT formula produced by the Cook-Levin Theorem. More generally, a \( \Sigma_{k+1}^p \) problem \( \exists y M(x, y) \), where \( M \) is in \( \Pi_k^p \), can be encoded as a formula of the form \( \exists y_1 \ldots \exists y_n \Phi(y_1, \ldots, y_n) \Phi(x, y) \), where \( \Phi \) encodes \( M \); the same thing works for \( \Pi_{k+1}^p \) formulas except we get universal quantifiers instead of existential quantifiers. In either case we can recurse within \( \Phi \) until we get down to no quantifiers at all. This makes \( \text{TQBF} \) hard for any \( \Sigma_k^p \), and thus for all of \( \text{PH} \).

It can’t be \( \text{PH} \)-complete unless the hierarchy collapses. The reason it doesn’t immediately collapse is that \( \text{TQBF} \) allows unbounded layers of alternation.
TQBF is computable in $\text{AP}$, alternating polynomial time, which is what we get when we have no restriction on an alternating Turing machine except that each branch must run in polynomial time. It is also computable in $\text{PSPACE}$, since we can build a recursive procedure that given a formula $Q_1x_1Q_2x_2\ldots Q_nx_n\Phi(x_1,\ldots,x_n)$ checks if both assignments to $x_1$ (if $Q_1 = \forall$) or at least one assignment to $x_2$ (if $Q_1 = \exists$) makes the rest of the formula true. So we can decide any language in $\text{PH}$ by reducing to TQBF and then solving TQBF in $\text{PSPACE}$, which shows $\text{PH} \subseteq \text{PSPACE}$.

In fact, TQBF is $\text{PSPACE}$-complete, but we will defer the proof of this to §9.2. A consequence of this is that $\text{AP} = \text{PSPACE}$, which will save us from having to think about $\text{AP}$ as a separate class.
Chapter 9

Space complexity

Understanding space complexity requires giving up some preconceptions about time efficiency, since space complexity classes don’t care about time efficiency, and indeed many of the known results on space complexity involve spending time like water. On the other hand, if the space complexity is low enough, this puts a bound on time complexity even if our use of time is stupidly inefficient.

Important space complexity classes:

- $\text{PSPACE} = \bigcup_{c=1}^{\infty} \text{SPACE}(n^c)$. Contains the entire polynomial-time hierarchy and then some.

- $\text{L} = \text{SPACE}(\log n)$. The class of problems solvable in log-space. Largest practically-implementable space complexity class (it’s contained in $\text{P}$). Popular with theoreticians who think about databases. Might be equal to:

  - $\text{NL} = \text{NSPACE}(\log n)$. Like $\text{P} = \text{NP}$, $\text{L} = \text{NL}$ is open. log-space hierarchy collapse to this class, which is equal to $\text{coNL}$ (§9.4).

Other noteworthy space complexity classes:

- $\text{SC} = \bigcup_{c=1}^{\infty} \text{SPACE}(\log^c n)$. “Steve’s class”, named for Steve Cook. The class of problems solvable in polylog space.

- $\text{RL}$ (randomized log-space). log-space version of $\text{R}$. A language is in $\text{RL}$ if a randomized log-space machine accepts positive instances with constant probability and rejects all negative instances. May be equal to $\text{L}$. [[[ come back to this when we do randomized classes ]]]
CHAPTER 9. SPACE COMPLEXITY

- **SL (symmetric log-space).** Languages accepted by a reversible log-space machine, which has the property that for any transition \( C \to C' \) there is also a reverse transition \( C' \to C \). Proposed [by whom?] as a stepping-stone to proving \( L = \text{NL} \). Now known to be equal to \( L \). [cite result, we probably won’t do this one in class]

9.1 Space and time

Claim: \( \text{TIME}(f(n)) \subseteq \text{SPACE}(f(n)) \). Proof: Can’t use more than \( O(1) \) new tape cells per step.

Claim: \( \text{SPACE}(f(n)) \subseteq \text{TIME}(2^{O(f(n))}) \). Proof: A \( \text{SPACE}(f(n)) \) machine has only \( 2^{O(f(n))} \) distinct configurations, so if it runs for longer, it’s looping and will never halt.

Application: \( L \subseteq \text{P}, \text{PSPACE} \subseteq \text{EXP} \). For all we know, equality might hold in both cases. (Current consensus is that it doesn’t.)

9.2 PSPACE and TQBF

Main result: TQBF is \( \text{PSPACE} \)-complete. This implies \( \text{PH} \subseteq \text{SPACE} \). If they are equal, existence of a \( \text{PSPACE} \)-complete language means that \( \text{PH} \) collapses to a finite level.

Claim: TQBF \( \in \text{PSPACE} \). Proof: game-tree search.

Claim: TQBF is \( \text{PSPACE} \)-hard. Proof: given a language in \( \text{PSPACE} \), reduce to TQBF.

Idea: Space complexity is all about reachability in the graph of configurations. If \( L = L(M) \) for some \( \text{PSPACE} \) machine \( M \), we can test if \( x \in L \) by testing if there is a path from the initial configuration of \( M(x) \) to some accepting configuration. To make our life easier later, we will assume that \( M \) cleans up on the way out (erasing all the work tapes, moving the heads back to their starting positions) so that the accepting configuration is unique.

The formula, naive version: \( C \to^* C' \) expands to \( \exists C'' : C \to^* C'' \land C'' \to^* C' \). This is too big, so instead we use \( \forall \) to test both branches with one formula:

\[
C \to^* C' \equiv \exists C'' \forall C_1, C_2 : ((C_1 = C \land C_2 = C'') \lor (C_1 = C'' \land C_2 = C')) \land C_1 \to^* C_2.
\]

At the bottom we use Cook-Levin to test if \( C_1 \to C_2 \) in a single step. This requires depth logarithmic in the length of the execution, and the length is at most exponential, giving depth at most polynomial. Each stage adds a
polynomial size to the formula (those configurations are big, but they are only polynomial), so total size of the formula is polynomial.

9.3 Savitch’s Theorem

Savitch’s Theorem: If $f(n) = \log n$, then $\text{SPACE}(f(n)) \subseteq \text{NSPACE}((f(n))^2)$.

The proof is similar to the proof of $\text{SPACE}$-hardness of TQBF. If $C_0 \rightarrow \leq T(n) C_{\text{accept}}$, then there is some intermediate configuration $C'$ such that $C_0 \rightarrow \leq T(n)/2 C' \rightarrow \leq T(n)/2 C_{\text{accept}}$. For each possible $C'$, test the left side recursively first, then the right side if the left side worked.

We have to store $C'$ in between these tests, but we can re-use the space we used for the left-side test for the right-side test. So this gives space complexity $S(T(n)) = O(f(n)) + S(T(n/2)) = O(f(n) \log T(n)) = O(f(n) \log 2^{O(f(n))}) = O((f(n))^2)$.

Consequences: $\text{NPSPACE} = \text{PSPACE}$, and in fact the entire poly-space hierarchy collapses to $\text{PSPACE}$. (Unbounded alternation, which boosts $\text{APSPACE}$ up to $\text{EXP}$, [[[ citation needed ]]] may give more power.)

9.4 The Immerman-Szelepcsényi Theorem

Says $\text{NL} = \text{coNL}$. Proof by “inductive counting”: Let $A_i$ be set of all configurations reachable in $i$ steps. Given $|A_i|$, we’ll compute $|A_{i+1}|$ by enumerating all configurations $C$ that might be in $A_i$, and for each $C$ try all possible configurations $C'$ that might be predecessors. For each $C'$, we guess and check a path. If we come up with fewer than $|A_i|$ predecessors with paths, we screwed up: reject this computation path and hope one of the other lemmings does better.

The last step is the same but we only look if $C_{\text{accept}}$ has no path.

Easy consequence: the alternating log-space hierarchy collapses. Use $\text{NL} = \text{coNL}$ to replace last $\forall$ or $\exists$ quantifier with its predecessor and combine them, same as in the proof that $\Sigma^p_k = \Pi^p_k$ implies $\text{PH}$ collapses.

Harder consequence: the oracle log-space hierarchy collapses. This requires some careful looking at how to define oracle access in $\text{NL}$, which we will do in the next section.
9.5 Oracles and space complexity

Just as we have to be careful to restrict the input and output tapes of a space-bounded computation to prevent them from doing double duty as work tapes, we also have to be careful how we define an oracle computation involving a space-bounded machine, particularly if that machine uses nondeterminism. The now-standard definition first appeared in a paper by Ruzzo, Simon, and Tompa [RST84].

The idea is that we have an oracle tape as usual, which we make write-only like the output tape. But we put some restrictions on how a nondeterministic machine can use this tape. These are:

1. Once the controller writes to the oracle tape, it cannot execute any nondeterministic transitions until after it completes and oracle call.

2. An oracle call erases the oracle tape.

Without these restrictions, we would get bizarre results like $\text{NL} \subseteq \text{L}$ being able to solve SAT even if $\text{NL}$ couldn’t on its own: an unrestricted $\text{NL} \subseteq \text{L}$ oracle machine could write the problem and a guess at a satisfying assignment on the oracle tape and ask the $\text{L}$ oracle to verify it [RST84, Example 1]. The Ruzzo-Simon-Tompa definition prevents this, and has the additional advantage of making the contents of the oracle tape, for a $\text{SPACE}(f(n))$ or $\text{NSPACE}(f(n))$ machine, encodable in $O(f(n))$ space provided the decoder has access to the original input.

We illustrate this by showing that $\text{NL}^{\text{NL}} = \text{NL}$ when the oracle calls are subject to the Ruzzo-Simon-Tompa restrictions. Let $M$ be the nondeterministic log-space oracle machine (which is to say the machine that calls the oracle), and let $M'$ be a nondeterministic log-space machine implementing the oracle. We want to build a combined machine $N$ that simulates $M'$.

The idea is that when $N$ simulates an oracle call, instead of writing to an oracle tape that it doesn’t have, it saves a copy of the state of $M$ at the start of the oracle-tape write. This copy now gets called as a kind of co-routine whenever $M'$ looks at a cell on the simulated oracle tape: if $M'$ looks at position $i$ in the oracle tape, $N$ will simulate a fresh copy $M$ starting from its saved state, recording whatever symbols it writes to position $i$, until simulated-$M$ makes an oracle call. Then $N$ can throw away this copy and continuing simulating $M'$ until the next time it looks at the oracle tape. This is, as usual, tremendously wasteful of time, but it still fits in $\text{NSPACE}(f(n))$.

For the specific case of $\text{NL}^{\text{NL}} = \text{NL}$, we need the additional trick of guessing in advance which way the oracle will answer. If the oracle is expected
to answer yes, we just run the straight NL computation, and give up if the answer is no (as usual). If the oracle is expected to answer no, we instead run a coNL computation converted into NL using the Immerman-Szelepcsényi Theorem. Again we give up if the answer is wrong. Since N can do this for each of M’s oracle calls and stay in NL, we get NL^{NL} = NL and the log-space oracle hierarchy collapses to NL.

Some similar issues come up with PSPACE. Here to prevent cheating we restrict the oracle tape to have polynomial size. With this restriction in place, NPSPACE^{NPSPACE} = NPSPACE = PSPACE by applying the preceding construction and Savitch’s Theorem.

9.6 \( L \supseteq NL \supseteq AL = P \)

So far we have seen that \( L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP \). We know from the Space Hierarchy and Time Hierarchy theorems that some of these inclusions must be strict, since \( L \neq PSPACE \) and \( P \neq EXP \). But we don’t know which, and given that L, NL, and P are all in some sense “easy” classes, it is not completely ridiculous to ask if \( L \supseteq NL \) or \( L \supseteq P \).

9.6.1 Complete problems with respect to log-space reductions

As with \( P \supseteq NP \), we can make these problems more concrete by concentrating on specific problems that are complete for NL or P. This notion of completeness will be slightly different from the usual notion of completeness used for NP, since poly-time reductions from problems in P work for just about any target problem: have the reduction do the computation. So instead we will consider log-space reductions.\(^1\)

We write \( L' \leq_L L \) if there is a log-space reduction from \( L' \) to \( L \), meaning that there is a log-space Turing machine \( M \) that transforms instances \( x \) of \( L' \) to instances \( M(x) \) of \( L \), such that \( x \in L' \) if and only if \( M(x) \in L \).

A language \( L \) is NL-complete if \( L \) is in NL and for every \( L' \) in NL, \( L' \leq_L L \). Similarly, a language \( L \) is P-complete if \( L \) is in P and for every \( L' \) in P, \( L' \leq_L L \).

In some contexts, it makes sense to consider different restrictions on the reductions, and we will say things like “\( L \) is P-complete with respect to

\(^1\)These also work for most NP-completeness reductions. For example, the reduction in the Cook-Levin Theorem can easily be made to run in log space, making every language \( L \in NP \) log-space reducible to 3SAT.
log-space reductions” or “L is P-complete with respect to NC^1 reductions.” But for now we will stick with log-space reductions when working with NL- or P-completeness.

9.6.1.1 Complete problems for NL

The classic NL-complete problem is s–t connectivity, abbreviated as STCON. The input to STCON is a triple \( \langle G, s, t \rangle \) where \( G \) is a directed graph and \( s \) and \( t \) are vertices, and \( \langle G, s, t \rangle \) is in STCON if there is a directed path from \( s \) to \( t \) in \( G \).

STCON is in NL because we can just guess each step of the path. To show that any language \( L \in \text{NL} \) reduces to STCON, start with a non-deterministic log-space machine \( M_L \) that decides \( L \) with a unique accepting state, and build a log-space machine that enumerates all possible pairs of states of \( M_L \) (there are only polynomially many, so we can do this in log space), and writes out an edge \( uv \) for each pair of states \( u \) and \( v \) such that \( v \) can follow from \( u \) given the contents of the input tape (also pretty straightforward to test in log space). Then ask if there is a path from the initial state to the accepting state. This gives \( L \leq_L \text{STCON} \), and STCON is \( \text{NL} \)-complete.

We can further reduce STCON to 2SAT to show that 2SAT is NL-complete, which gives a nice symmetry with 3SAT being \( \text{NP} \)-complete. An instance of 2SAT is a collection of two-literal OR clauses \( x \lor y \) and we want to know if there is a truth assignment that makes all the clauses true.

The idea here is that we can encode an edge \( uv \) in our input graph \( G \) as an implication \( u \Rightarrow v \), which is logically equivalent to the OR clause \( \neg u \lor v \). We do this for every edge in \( G \), and add clauses asserting \( s \) (\( s \lor s \) if we are being picky about exactly two literals per clause) and \( \neg t \). It’s not too hard to see that we can do this in log space.

If there is no path from \( s \) to \( t \) in \( G \), then there is a satisfying assignment that sets \( s \) true, that sets every vertex reachable from \( s \) also true, and sets every vertex not reachable from \( s \) (including \( t \) false. But if there is a path from \( s \) to \( t \), we are in trouble. We have to set \( s \) true, but then the implications for the edges mean that we also have to set every literal reachable from \( s \), including \( t \) true. But then we can’t satisfy the \( \neg t \) clause.

If you look closely at this argument, you may have noticed that we didn’t reduce STCON to 2SAT. Instead we reduced s–t non-connectivity to 2SAT,

\(^2\)If we don’t want to insist on a unique accepting state, which doesn’t really constraint \( M_L \) because it can do the usual trick of erasing its work tapes and parking its heads, we can add an extra state \( t \) and run an edge from every accepting state to \( t \).
or equivalently we reduced STCON to 2CNF non-satisfiability. This means that we showed 2SAT is coNL-hard. But coNL = NL, so it’s OK.

Conversely, we can translate a 2SAT problem into its implication graph, where the implication graph contains nodes for every literal (so both $x$ and $\neg x$ for each variable $x$), and there is an edge $uv$ for every pair of literals such that $(\neg u \lor v)$ appears in the 2CNF formula. [[[ win if $x$ and $\neg x$ never in same SCC, tricky issue with backwards paths. We kind of skipped over this in class. ]]]

### 9.6.1.2 Complete problems for $\mathbf{P}$

$\mathbf{P}$-complete problems are similar to $\mathbf{NP}$-complete problems in that they tend to involve encoding a poly-time computation in some other process. The difference is that with $\mathbf{P}$, there is no nondeterminism, which means that the problem we are reducing to generally won’t have many choices left open.

A trivially $\mathbf{P}$-complete language is $\text{PCOM} = \{ \langle M, x, 1^n \rangle \mid M \text{ accepts } x \text{ in } n \text{ steps} \}$. This is in $\mathbf{P}$ because a universal Turing machine can check if $M$ does in fact accept $x$ in $n$ steps, using time poly in $n$ (and thus the size of the input, since we are expressing $n$ in unary). It’s $\mathbf{P}$-complete, because given any particular language $L$ in $\mathbf{P}$ recognized by a machine $M_L$ that runs in $n^c$ steps, we can reduce $L$ to PCOM using a log space machine that writes out $M$, copies $x$, and then computes $1^{|x|^c}$.

Somewhat more interesting is CVP, the circuit value problem. This is defined by given a Boolean circuit $C$ and its input $x$, and asking whether the circuit outputs 1. (Formally, a Boolean circuit is a directed acyclic graph where nodes are labeled with either an input bit or with a function, the value of each node is the value of the function applied to the values of its predecessors, and the value of the circuit is the value of a designated output node.) This is in $\mathbf{P}$ because we can easily write a poly-time program to solve it. It’s $\mathbf{P}$-complete because we can take any fixed poly-time $M$ and translate its execution into a circuit that calculates the contents of each tape cell, etc., at each time, and set the output to whether $M$ accepts.

For more elaborate examples, we need to find some clever way to encode circuit gates in whatever problem we are looking at. One cute example is LINEAR INEQUALITIES, where we are given a matrix $A$ and vector $b$, and want to know if there is a vector $x$ such that $Ax \leq b$. This corresponds to feasibility of a linear program, which is known to be solvable in polynomial time. To show that it is $\mathbf{P}$-complete, take CVP for a circuit built from AND and NOT gates, and translate $y = \neg x$ as $y = 1 - x$ and $z = x \land y$ as $z \leq x, z \leq y, z \geq x + y$. Then if we peg all inputs at 0 or 1, and put
a \(0 \leq x \leq 1\) constraint on any variable representing a gate, the unique satisfying variable assignment will set the output variable \(C\) equal to the output of the circuit. One more constraint \(C \geq 1\) makes the system of linear inequalities feasible if and only if the output is 1.

Many more examples of \(\mathbf{P}\)-complete problems can be found in a classic survey of Greenlaw, Hoover, and Ruzzo [GHR91].

### 9.6.2 \(\mathbf{AL} = \mathbf{P}\)

With bounded alternation, the log-space hierarchy collapses to \(\mathbf{NL}\), which we suspect is properly contained in \(\mathbf{P}\). But unbounded alternation gets us to \(\mathbf{P}\).

**Proof:** Let \(M\) be a single-tape Turing machine that runs in time \(n^c\) for some fixed \(c\). We will show that \(L(M) \in \mathbf{AL}\). Define a **tableau** consisting of cells \(C_{ij}\) where each \(C_{ij}\) describes the state at time \(i\) of position \(j\) of the tape, including whether the head is at position \(j\) and its state if it is there.

Observe that \(C_{ij}\) is completely determined by \(C_{i-1,j-1}, C_{i-1,j},\) and \(C_{i-1,j+1}\). We’ll assume that \(M\) is polite and parks the head on the starting tape position 0 at time \(n^c\), so we can check if \(M\) accepts by checking if \(C_{n^c,0}\) is in an accepting state.

We do this by building a recursive procedure that checks if a given \(C_{ij}\) is in a particular state \(x_{ij}\). For \(i = 0\) we can just check \(x_{ij}\) against the input tape. To test if \(C_{ij}\) is in state \(x_{ij}\) when \(i > 0\), we first guess (using \(\exists\)) states \(x_{i-1,j-1}, x_{i-1,j},\) and \(x_{i-1,j+1}\) for \(C_{i-1,j-1}, C_{i-1,j},\) and \(C_{i-1,j+1}\). If these states are not consistent with \(x_{ij}\), we halt and reject. This leaves only branches with consistent predecessor states. We want to verify that all three of these states are correct. We do this recursively, inside a \(\forall\) quantifier that picks one of the three states to check. Only if all three checks succeed do we accept.

This fits in log space because when we do the recursive check, we only need to store \(i - 1\), the appropriate tape position, and the value \(x_{ij}\). Since \(i - 1\) and the tape position are bounded (in absolute value) by \(n^c\), we can do this with \(O(c \log n) = O(\log n)\) bits. Checking the predecessor states for consistency doesn’t require much additional storage, since we can bake \(M\)’s transition table into the \(\mathbf{AL}\) machine’s finite-state controller and each \(x_{ij}\) has constant size. The result is an alternating log-space machine that accepts an input \(x\) if and only if \(M\) does.

Since we can do this for any language in \(\mathbf{P}\), we get \(\mathbf{AL} \subseteq \mathbf{P}\). In the other direction, a \(\mathbf{P}\) machine can compute the entire state graph for a \(\mathbf{AL}\) machine (it has only polynomially many states), and evaluate whether to accept or
reject at each branching node based on whether the successor nodes accept or reject.

This result generalizes. What we’ve really shown is that $\text{ASPACE}(f(n)) = \text{TIME}(2^{O(f(n))})$, at least for $f(n) = \Omega(\log n)$. (For smaller $f$ we run into issues keeping track of the position of the input head.) So in addition to $\text{AL} = \text{P}$, we also get $\text{APSPACE} = \text{EXP}$. In both cases, unbounded alternation is necessary unless something surprising happens, since bounded alternation gives us either $\text{NL}$ or $\text{PSPACE}$, which we suspect are different from $\text{P}$ and $\text{EXP}$. 
Chapter 10

Circuit complexity

Circuit complexity models computation in terms of Boolean circuits, which are formally represented as directed acyclic graphs where each node is either an input (with no incoming edges) gate (with at least one incoming edge). We think of the edges as wires carrying bits, and typically assume that each input is a bit and each gate computes some Boolean function $f : \{0, 1\}^k \rightarrow \{0, 1\}$. If a Boolean circuit has maximum out-degree 1, it is a Boolean formula. In both cases, we will often just remember that we are sending bits around everywhere and drop the “Boolean” qualifier. The difference between circuits and formulas is that circuits can share computation.

The in-degree of a gate is called its fan-in, and similarly the out-degree is called its fan-out. We will often talk about the fan-in or fan-out of a circuit as a whole, meaning the maximum fan-in or fan-out of any gate. For example, formulas have fan-out 1.

Typically, a single gate will be designated the output of a circuit. A circuit computes a function by setting the inputs to the circuit to the inputs to the function, computing the output of each gate based on its inputs and the function it implements, and taking the output of the output gate as the value of the function.

The depth of the circuit is the length of the longest path from an input to the output. The size of a circuit is the number of gates.

Circuits are attractive as a model of computation because they are in a sense more concrete than Turing machines. They also allow us to model parallel computation, since a low-depth circuit corresponds in a natural way to a highly-parallel computation.

Because each circuit has a fixed number of inputs, circuits can’t generally
compute languages of the sort we are used to studying in complexity theory. So we instead consider circuit families \( \{C_n\} \), where each circuit \( C_n \) is used for inputs of size \( n \). The language decided by a circuit family is \( \{ x \mid C_{|x|}(x) = 1 \} \). Using this approach, we can define complexity classes based on what can be computed by circuit families subject to various size, depth, and fan-in restrictions.

Some of these complexity classes are weak enough that we can prove that actual, non-contrived functions don’t live in them, often using techniques that don’t relativize. Unfortunately, other obstacles come into play when we try to get lower bound results for stronger circuit complexity classes, such as those corresponding to \( \text{P} \).

### 10.1 Polynomial-size circuits

We’ve seen before when looking at \( \text{P} \)-complete problems (§9.6.1.2) that any computation by a Turing machine can be encoded by a circuit made up of basic Boolean logic gates like AND, OR, and NOT. For a computation that takes \( T(n) \) time and uses \( S(n) \) space, the circuit will have size \( O(T(n) \cdot S(n)) \) and depth \( O(T(n)) \). Both of these quantities will be polynomial if \( T(n) \) is polynomial.

This suggests that we can think of \( \text{P} \) as a circuit-complexity class by restricting our attention to families of polynomial-size circuits. This almost works, but as often happens, the simplest definition gives a class that is too powerful.

The reason is that when using a family \( \{C_n\} \) of circuits, we have to ask where these circuits come from. If they are arbitrary, even though each \( C_n \) may be restricted to size \( O(n^c) \), it still may be that \( C_n \) encodes a lot of information about \( n \). For example, there is a family of circuits of size 1 where each \( C_n \) is a constant circuit, and \( C_n \) happens to output 1 if and only if the \( i \)-th Turing machine \( M_i \) halts on an empty input tape. So polynomial-size circuits by themselves are not restricted enough to avoid computing undecidable predicates, but they do give us a starting point for representing \( \text{P} \).

---

\(^1\)There might not seem like there is much room to sneak an oracle into a circuit, but unless we are very careful to specify exactly what gates we allow (and use this restriction in our proofs), individual gates or small groups of gates working together could do all sorts of tricksy things, including making oracle calls. This is why any technique for proving lower bounds on circuits that relativizes won’t help with \( \text{P} \supseteq \text{NP} \), because there are oracles \( A \) and \( B \) that separate small circuits from \( \text{NP} \supseteq \text{NP} \) and that supercharge them so that they can easily solve the wimpy problems in \( \text{NP} \). [Wil85].
10.1.1 \(\text{P/poly}\)

Polynomial-size circuit families decide languages in a complexity class called \(\text{P/poly}\), the class of languages that can be computing in polynomial time with polynomial advice. Advice is a little bit like the certificates or hints given to nondeterministic Turing machines, with the differences that (a) the advice string can only depend on the size of the input \(n = |x|\), and (b) the advice string is trustworthy, in the sense that the Turing machine that handles the advice is not required to do anything sensible if it gets the wrong advice.

We could provide the advice on a separate read-only input tape, but instead we typically just append it to the input. Formally, a language \(L\) is in \(\text{TIME}(f(n))/g(n)\) if there is machine \(M\) and a function \(\alpha : \mathbb{N} \to \{0,1\}^*\) where \(|\alpha(n)| \leq g(n)\) and \(M(x,\alpha(n))\) accepts in \(O(f(n))\) steps when \(x \in L\) and rejects in \(O(f(n))\) steps when \(x \notin L\). The class \(\text{P/poly}\) can then be defined as \(\bigcup_{a,b} \text{TIME}(n^a)/n^b\).

Given a family \(\{C_n\}\) of polynomial-size circuits, the language decided by this family is in \(\text{P/poly}\): let \(\alpha(n)\) be a description of \(C_n\). In the other direction, if \(M\) is a poly-time Turing machine that uses advice, we can simulate it by constructing a family of circuits where each \(C_n\) has \(\alpha(n)\) baked into it. This shows that \(\text{P/poly}\) captures exactly the languages computable by a family of poly-size circuits.

Note that we can always make the advice empty: this gives \(\text{P} \subseteq \text{P/poly}\). In §10.2 we will show how to put some constraints on the advice to get a class of polynomial-size circuits that decide exactly the languages in \(\text{P}\).

10.1.2 Information-theoretic bounds

\(\text{P/poly}\) is pretty powerful, but it’s not all-powerful: almost all functions can’t be computed by polynomial-size circuits.

To keep things simple, let’s assume that our circuits consist only of 2-input AND gates and NOT gates, since we can simulate AND and OR gates with polynomial fan-in using \(O(\log n)\) of these gates and still have a polynomial-size circuit. We can reduce further to 2-input NAND gates (which compute \(\neg(x \land y)\)) since we can build a NOT out of a NAND by wiring both inputs together and build an AND out of two NANDs by using the second one as a NOT to invert the output of the first.

We can specify an all-NAND circuit of size \(f(n)\) by listing which gates or inputs supply the input to each gate. This gives us \((f(n) + n)^2\) choices per
gate, or \((f(n) + n)2^{f(n)}\) choices total.\(^2\) The number of Boolean functions on \(n\) inputs is \(2^{2^n}\). Taking logs of both quantities gives us \(O(f(n) \log f(n))\) bits to represent a circuit of size \(f(n)\) (assuming \(f(n) = \Omega(n)\)) and \(2^n\) bits to represent a function on \(n\) inputs. For \(f(n) = n^c\), we get \(O(n^c \log n) = o(2^n)\) circuits, so most Boolean functions have no circuits of this size.

In fact, this is true for any size bound that is subexponential. On the other hand, at \(O(n \cdot 2^n)\) size, we can build a circuit that consists of \(2^n\) separate circuits that each recognize one input using \(O(n)\) gates, and a gigantic OR that combines the outputs of all the circuits that recognize a positive input. So exponentially-large circuits can do anything, making \(\text{EXP/\exp}\) equal to the set of all functions. For this reason, we will not study \(\text{EXP/\exp}\) much.

\(^2\)We are overcounting a bit here, but it won’t matter.

10.1.3 The Karp-Lipton Theorem

The lower bound in the preceding section is not as exciting as it could be, since it just says that there are many bad languages out there for \(P/\poly\), but it doesn’t say which. The Karp-Lipton Theorem gives evidence that SAT is one of these functions, assuming we don’t believe the poly-time hierarchy collapses.

**Theorem 10.1.1** (Karp-Lipton [KL80]). If \(\text{NP} \subseteq P/\poly\), then \(\text{PH} = \Sigma_2^p\).

**Proof.** \(\square\)

The Karp-Lipton Theorem suggests that even though \(P/\poly\) is in a sense too powerful a class to use in real life, it might still be possible to show \(P \neq \text{NP}\) by proving the just as plausible \(P/\poly \neq \text{NP}\). This approach was pretty popular in the late 1980s, in part because of a proof by Razborov that poly-size monotone circuits could solve CLIQUE [Raz85]. These are circuits with AND and OR gates, but no NOT gates, and (unlike, say, SAT), it is possible to solve CLIQUE using a big enough monotone circuit (take an OR of ANDs over all sets of edges that might be cliques). So the hope was that some sort of clever non-relativizing circuit sneakery could prove a similar result for general circuits. This hope was ultimately dashed by the Razborov-Rudich natural proofs result [RR97], which ruled out a large class of lower bound arguments based on finding testable properties of functions that would imply hardness. (We’ll come back to this in Chapter 11.)
On the other hand, Karp-Lipton gives some interesting conditional results. One of these, which Karp and Lipton attribute to Albert Meyer in their paper, is that if \( \text{EXP} \subseteq \text{P/poly} \), then \( \text{EXP} = \Sigma_2^p \). But then \( \text{P} \neq \Sigma_2^p \) (Time Hierarchy Theorem), which implies \( \text{P} \neq \text{NP} \). So we have the unlikely possibility of being able to prove \( \text{P} \neq \text{NP} \) by showing that \( \text{P/poly} \) is much more powerful than it has any right to be.

10.2 Uniformity

The reason we get \( \text{P/poly} \) instead of \( \text{P} \) out of poly-size circuits is that our circuit families are non-uniform: we get to pick a different circuit \( C_n \) for each input \( n \), and this choice is arbitrary. If we have to actually compute \( C_n \) from \( n \), we can restore some sanity.

A circuit family \( \{ C_n \} \) is logspace-uniform or just uniform if there is a Turing machine \( M \) that computes a description of \( C_n \) from \( 1^n \) in time logarithmic in \( n \). The description will be a labeled graph, so \( M \) needs to output (1) a count of the number of gates in the circuit, equivalent to counting the number of vertices in the graph; (2) an adjacency list that gives the wires in the circuit; and (3) a label for each gate saying which function (AND, OR, NOT) it computes.\(^3\) As with other computations by log-space machines, we can assume either that the machine is explicitly writing the entire description to a write-only output tape or implicitly providing the description as a Boolean function \( f(n, i) \) that gives the \( i \)-th bit of the output on input \( 1^n \).

\[ \text{log-space-uniform circuits of poly size compute P} \]

10.3 Bounded-depth circuits

\[ \text{easier to probe lower bounds on than general poly-size circuits} \]

In addition to restricting the size of a circuit, we can also restrict its depth, the maximum length of any path in the circuit. This gives rise to two important families of complexity classes:

1. \( \text{AC}^k \) is the class of languages computed by polynomial-size circuits with depth \( O(\log^k n) \).

\(^3\)If we are willing to limit ourselves to NAND gates, we can compute anything we can compute with AND, OR, and NOT, and skip the labeling. But having explicit AND and OR gates will turn out to be convenient when we look at various restricted circuit complexity classes.
2. \( \text{NC}^i \) is the class of languages computed by polynomial-size circuits with constant fan-in and depth \( O(\log^i n) \).

The names \( \text{AC} \) and \( \text{NC} \) are short for \textbf{alternating class} and \textbf{Nick’s class}, respectively.

Often we will insist that these circuits are log-space uniform, but this requirement is not part of the standard definitions, and many results about circuit complexity don’t require uniformity.

A common convention is to use De Morgan’s Laws to push all NOT gates to the inputs, and insist on alternating layers of AND and OR gates. This can be done to either \( \text{AC}^i \) or \( \text{NC}^i \) circuits, and at most doubles the depth (since we may have to stick a layer of dummy 1-input AND or OR gates in between layers of gates of the same type).

It holds trivially that \( \text{NC}^i \subseteq \text{AC}^i \), since any polynomial-size circuit of depth \( O(\log^i n) \) with bounded fan-in is also a polynomial-size circuit of depth \( O(\log^i n) \) with unbounded fan-in. It is also not hard to show that \( \text{AC}^i \subseteq \text{NC}^{i+1} \): replace each AND or OR gate with fan-in \( k \) by a tree of AND or OR gates of depth \( O(\log k) \).

### 10.3.1 Parallel computation and \textbf{NC}

The union \( \bigcup_{i=1}^{\infty} \text{NC}^i \) of these classes is called \( \text{NC} \), and is generally taken to be the class of problems that can be computed efficiently in parallel. Note that \( \text{NC} \) is the same as \( \bigcup_{i=0}^{\infty} \text{AC}^0 \).

The \textbf{P}-complete languages (§9.6.1.2) have roughly the same relation to \( \text{NC} \) as the \textbf{NP}-complete languages have to \( \text{P} \): while a \textbf{NP}-complete problem is one that we don’t expect to be able to compute efficiently at all, a \textbf{P}-complete language is one that we expect to be able to compute efficiently, but not in parallel.

### 10.3.2 Relation to \textbf{L} and \textbf{NL}

[\[ \text{NC}^1 \subseteq \textbf{L} \quad \text{(if log-space-uniform): game-tree search again} \]\]

[\[ \text{but doesn’t work for AC}^1 \text{ because too many paths to keep track of, so we need } O(\log^2 n) \text{ space} \] ]

[\[ \text{NL} \subseteq \text{AC}^1 \text{; compute transitive closure by repeated matrix squaring, cite Fischer and Meyer 1971 for the matrix squaring bit?} \] ]
10.3.3 Barrington’s Theorem

Barrington’s Theorem [Bar89] shows that the Boolean functions computable in \( \text{NC}^1 \) are precisely the functions that can be computed by branching programs of width 5.

A branching program is a directed acyclic graph where each non-terminal vertex is labeled by an input bit position, and has exactly two outgoing edges labeled 0 and 1. We execute a branching program by starting at an initial vertex, and at each step we (a) look at the input bit \( x_i \) given by the label on the current vertex, and then (b) move to the successor pointed to by the 0 or 1 edge, depending on the value of \( x_i \). When we reach a terminal vertex, we take the label on that vertex (0 or 1) as the output of the program.

A bounded-width branching program is one in which the vertices can be arranged into layers, so that each vertex in layer \( i \) has successors in layer \( i + 1 \), and there are at most \( k \) vertices in each layer. Here \( k \) is the width of a branching program.

We can think of a bounded-width branching program as a little like a finite-state machine where the transition function changes at each step. In this view, the \( k \) nodes that the program might be in at layer \( t \) are treated as \( k \) states that the machine might be in at time \( t \), and we think of the labeled outgoing edges as representing functions \( f^0_t \) and \( f^1_t \) that map the state \( q_t \) at time \( t \) to the new state \( q_{t+1} \) at time \( t + 1 \). So we can program these things by choosing a sequence of inputs \( x_t \) and functions \( f^0_t \) and \( f^1_t \), and then just iterate applying \( f^t_{x_t} \) to the state each time \( t \) starting with some initial state.

Taking one more step down the rabbit hole, we can choose to make these functions invertible, which makes \( f^0_t \) and \( f^1_t \) both elements of \( S_k \), the symmetric group on \( k \) elements. Composition of these functions corresponds to multiplication of the group elements, so we can think of the entire computation as evaluating a gigantic word \( f^1_{x_1} f^2_{x_2} \cdots f^T_{x_T(n)} \) in \( S_k \). So now our programming problem is one of choosing the individual group elements \( f^0_t \) and \( f^1_t \) and inputs \( x_t \) so that this multiplication does what we want.

We will specifically use \( S_5 \), and use words that evaluate either to cycles or to the identity. A standard convention in group theory writes an element of \( S_k \) as a product of cycles, where each cycle is given by a list of its elements in order in parentheses. For example, \((12345)\) represents the cycle \( 1 \to 2 \to 3 \to 4 \to 5 \to 1 \), while \((1)(2)(3)(4)(5)\) represents the identity \( e \) (which sends each position back to itself; it’s worth mentioning that usually we would omit trivial cycles with just one element). We can multiply group
elements by chasing individual position around the cycles. So, for example, the product \((12345)(13542) = (1)(253)(4) = (253)\), since \(1 \rightarrow 2 \rightarrow 1\), \(2 \rightarrow 3 \rightarrow 5\), and so on.

Each input bit \(x_i\) will be represented by a choice of two group elements \(\xi_{i0}\) and \(\xi_{i1}\), depending on its value. For a function \(f(x_1, \ldots, x_n)\) of the input bits, we will say that a word \(w = \xi_1 \xi_2 \ldots \xi_m\) \(\alpha\)-represents \(f\) if \(w = \alpha\) when \(f = 1\) and \(w = e\) when \(f = 0\). In this representation, we use \(x_i\) to refer to the \(i\)-th bit examined by the branching program rather than the \(i\)-th bit in the actual input; this saves some extra subscripts and allows us to use a different representation for different occurrences of a particular input bit.

If \(\alpha\) and \(\beta\) are cycles, we can convert an \(\alpha\) representation to a \(\beta\) representation by conjugation; there is a group element \(\gamma\) such that \(\gamma \alpha \gamma^{-1} = \beta\), and when \(w = \alpha\) we get \(\gamma w \gamma^{-1} = \beta\) and when \(w = e\) we get \(\gamma w \gamma^{-1} = \gamma \gamma^{-1} = e\). Note that this doesn’t actually increase the length of \(w\), since we can bake \(\gamma\) and \(\gamma^{-1}\) into the representations of the first and last bits examined by \(w\).

Now we want to represent the gates in our circuit by products of words. Negation is easy: if \(w\) \(\alpha\)-represents some sub-circuit \(C\), then \(w \alpha^{-1}\) \(\alpha^{-1}\)-represents \(\neg C\). For AND, we take advantage of the fact that there exist cycles \(\rho = (12345)\) and \(\sigma = (13542)\) in \(S_5\) whose commutator \(\tau = \rho \sigma \rho^{-1} \sigma^{-1} = (12345)(13542)(54321)(24531) = (13254)\) is also a cycle. So now to compute \(C \land D\), we let \(u\) \(\rho\)-represent \(C\), \(v\) \(\sigma\)-represent \(D\), and then \(w = uv^{-1}u^{-1}\) \(\tau\)-represents \(C \land D\), since if \(C = 0\) we get \(w = eue^{-1} = vv^{-1} = e\), if \(D = 0\) we get \(w = ueu^{-1}e = uu^{-1} = e\), and if both are 1 we get \(w = \rho \sigma \rho^{-1} \sigma^{-1} = \tau\).

To convert an entire depth \(d\) circuit, first convert it to a depth-\(d\) formula, then apply the above construction to each subcircuit, adjusting the representation as needed using conjugations. This gives a word \(w\) of length \(4^d\) over \(S_5\) that \(\alpha\)-represents the computation of the original circuit for some \(\alpha\), and we can compute this using a width-5 branching program by taking any initial state and seeing if it gets mapped elsewhere or not by \(w\).

In the other direction, given a branching program of width \(k\), we can represent it as a word over the semigroup of functions from \(\{1 \ldots k\}\) to \(\{1 \ldots k\}\), and use a circuit of depth \(O(\log |w|)\) to compute its value by splitting \(w\) in half, computing the mapping corresponding to each half, then composing the mappings together (which we can do in constant additional depth). This shows that \(O(\log n)\) depth circuits are exactly equivalent in power to polynomial-length width-5 branching programs.

I find this result horrifying. In both directions of the construction, the thing we build has no obvious connection to the thing we started with, and if asked after seeing the state of the branching program coming out of \(w\) what that state means, we would have to admit that we have no ability whatsoever
to decode it without running it back through \( u^{-1}v^{-1} \). So the answer to pretty much any attempted proof of anything in complexity theory that starts with “I understand how programs work, and at this stage of the program the state must mean this” is “Oh yeah? And what about Barrington’s Theorem?”

### 10.3.4 PARITY ̸∈ \( \text{AC}^0 \)

Want to show \( \text{AC}^0 ̸= \text{NC}^1 \) by showing PARITY (odd number of inputs are 1) is not in \( \text{AC}^0 \). This also shows MAJORITY (more than half the inputs are 1) is not in \( \text{AC}^0 \), since we can use pairs of MAJORITY gates to test if \( \sum x_i = k \) for any fixed \( k \) and use an OR of \( n/2 \) of these equality testers to compute PARITY [FSS84].

The first proofs that PARITY ̸∈ \( \text{AC}^0 \) were given by Furst, Saxe, and Sipser [FSS84] and Ajtai [Ajt83]. We’ll give two proofs of this fact, one (§10.3.4.1) based on Håstad’s Switching Lemma [Has86], which simplifies circuits by fixing some of their inputs; and one (§10.3.4.3) based on Razborov’s approximation of \( \text{AC}^0 \) circuits by low-degree polynomials over a finite field [Raz87] together with Smolensky’s proof that such polynomials can’t approximate parity [Smo87].

#### 10.3.4.1 Håstad’s Switching Lemma

First, let’s start with an easy warm-up: A depth-2 circuit can only compute parity if it has at least \( 2^{n-1} \) gates. Proof: If it’s an OR of ANDs, any AND with fewer than \( n \) inputs gives a false positive on some input, and we need at least \( 2^{n-1} \) ANDs on \( n \) inputs to cover all odd-parity inputs. If it’s an AND of ORs, then do the same thing with false negatives.

**Håstad’s Switching Lemma** uses a random restriction to convert an OR of small ANDs into an AND of small ORs. A random restriction fixes a random subset of the inputs to the circuit to random values. By repeatedly switching ANDs with ORs, we can flatten an \( \text{AC}^0 \) circuit down to something that obviously can’t compute parity on the surviving unrestricted variables. This will be be a problem because a random restriction of parity is still parity, and we won’t knock out everything.

Håstad’s Switching Lemma actually shows something stronger. Given a DNF formula of width \( w \) of a DNF formula \( \mathcal{w} \), meaning an OR of ANDs where each AND has at most \( w \) literals, hitting it with a random restriction

---

\(^4\)In fact, applying Barrington’s Theorem is a critical step in many known cryptographic techniques for obfuscating code, although in fairness the real obfuscation comes from disguising the \( S_5 \) elements as more complicated algebraic objects. [GGH+16]
gives us a decision tree of low depth with high probability. This can then be converted into a low-width CNF formula if needed.

Define a random \(s\)-restriction as a choice of random values for a random subset of \(n - s\) variables, where each of the \(\binom{n}{s}2^{n-s}\) possible restrictions are equally likely. (Note that \(s\) indicates how many variables survive.) The restriction \(f|_\alpha\) of \(f\) to \(\alpha\) is the function on \(s\) variables given by fixing the other inputs to \(f\) according to \(\alpha\).

**Lemma 10.3.1** (Håstad’s Switching Lemma). If \(f\) is computed by a \(w\)-DNF, and \(\alpha\) is a random \(s\)-restriction with \(s = \sigma n \leq n/5\), then for any \(d \geq 0\), the probability that \(f|_\alpha\) has no decision tree of depth \(d\) or less is at most \((10\sigma w)^d\).

**Proof.** The proof is by showing that the set of bad restrictions \(\beta\) that require depth more than \(d\) is small. This is done by encoding each such \(\beta\) using an \((s-d)\)-restriction supplemented by a small number of extra bits of information. Since the number \(\binom{n}{s-d}2^{n-s+d}\) of \((s-d)\)-restrictions is much smaller than the number \(\binom{n}{s}2^{n-s}\) of \(s\)-restrictions, and the extra bits don’t add much to the total count, this will show that the proportion of bad restrictions is small.

Given \(f\) and a bad \(\beta\), we’ll define a canonical decision tree that may or may not be the best possible decision tree for \(f|_\beta\), but that is certainly no better than the best possible tree. In particular, the canonical description tree will contain a path of length \(d + 1\) or more, and we’ll use this path to extract our succinct description of \(\beta\).

Order the terms \(T_1 \lor \ldots \lor T_\ell\) in the \(w\)-DNF representation of \(f\) in some standard way. We say that \(\beta\) kills \(T_i\) if \(\beta\) forces \(T_i\) to 0. It fixes \(T_i\) if it instead forces \(T_i\) to 1. Since \(\beta\) is bad, it can’t fix any term or kill all the terms, because then \(f|_\beta\) would be constant and computable by a depth-0 tree.

The canonical decision tree is generated recursively. Start with first term \(T_i\) not killed by \(\beta\), and let \(T_i\) have \(d_i\) free variables. Build a complete depth-\(d_i\) tree that has \(2^{d_i}\) leaves corresponding to each assignment of these variables. Give the leaf of the tree corresponding to a true assignment to \(T_i\) the value 1, and recurse on the terms after \(T_i\) to get subtrees for each of the other leaves. Note that in this recursion, we restrict the variables we’ve already examined, so we may kill (or fix) many of the terms after \(T_i\) depending on which values we saw. But at the end we get a decision tree that computes \(f|_\beta\).

Somewhere in this tree is a path of length \(d + 1\) or more. Let \(P\) be the prefix of this path of length \(d\), and construct a new \((s-d)\)-restriction \(\pi\) by adding to \(\beta\) the values of the variables on \(P\).

For the last step, we’ll show how to recover \(\beta\) from a different \((s-d)\)-restriction \(\gamma\) without too much extra information. What \(\gamma\) does is fix all the
terms $T_i, T_{i2}, \ldots$ whose subtrees are traversed by $\pi$, by setting the variables in each $T_{ij}$ to whatever will make $T_{ij}$ true. To be able to recover $\pi$ and $\beta$ from this, we also write down for each term the set of variables we fixed and what values they have in $\pi$: this requires no more than $d_i(\lg w + 2)$ bits if we are careful, for a total of $d(\lg w + 2)$ extra bits over all terms. The extra information lets us reconstruct $\pi$ from $\gamma$.

To recover $\beta$, we walk through $f|_{\gamma}$ looking for clauses fixed by $\gamma$. For each such clause, the extra bits tell us which variables to remove from $\gamma$ to get $\beta$. This shows that the number of bad restrictions is bounded by

$$(s-d)^{2n-s+d}2^{d(\lg w+2)} = (s-d)^{2n-s+d}(4w)^d.$$

Dividing by the total number of restrictions gives a probability of a bad restriction bounded by

$$\frac{(s-d)^{2n-s+d}(4w)^d}{(n-d)!} \leq \frac{n!}{n!} (s-d)! (8w)^d \leq \frac{(s-d)!}{(n-s+d)!} (8w)^d \leq \frac{(s-d)!}{(n-s+d)!} (8w)^d \leq \frac{(\sigma)^d}{(1-\sigma)^d} (8w)^d \leq \left(\frac{5}{4}\sigma\right)^d (8w)^d \leq (10\sigma w)^d.$$

This shows that an OR of small ANDs usually turns into an AND of small ORs when hit by a random restriction. By negating outputs and inputs, we can apply the same technique to turn an AND of small ORs into an OR of small ANDs. We’ll alternate between these to flatten $\text{AC}^0$ circuits, as explained below.

### 10.3.4.2 Application to PARITY

Now we want to use the lemma to beat up an $\text{AC}^0$ circuit $C$ that claims to compute PARITY.
CHAPTER 10. CIRCUIT COMPLEXITY

To get the process off the ground, we need to restrict the fan-in of our circuit to 1 at the bottom level. We can trivially do this by adding an extra layer of 1-input gates. We will also assume for simplicity that the circuit is a tree, and that we strictly alternate OR and AND levels, with all negations pushed to the inputs, and AND at the bottom. Let $n^b$ bound the size of the resulting circuit.

We now hit the circuit with a random $\sqrt{n}$-restriction, meaning that we set $\sigma = n^{-1/2}$ (this will let us use a union bound over all gates in $C$ later). We'll choose $d$ so that $(10\sigma w)^d = (10n^{-1/2})^d = o(n^{-b})$; $d = 4b$ works, giving a probability of getting depth greater than $4b$ bounded by $10^{4b}n^{-2b} = o(n^{-b})$ since the extra $n^{-b}$ factor starts eating up the constant around $n = 10000$ or so.

If we look at one OR gate at the second level of the circuit and its pile of feeder 1-input AND gates, after applying the restriction we can, with high probability, replace it with an AND of $4b$-input OR gates. All of these ANDs feed into ANDs that used to be at the third level of the circuit, and we can absorb them into their successors without changing the width of the new bottom layer of OR gates. We have just shortened the circuit by one level, at the cost of increasing the width of the input level from 1 to $4b$ and reducing the number of free inputs from $n_0 = n$ to $n_1 = n^{1/2}$.

In general, we’ll do the square-root trick whenever we knock out the bottom level. This gives $n_i = n^{2^{-i}}$ as the number of surviving inputs after each level. For the width, let $w_i = 4^i b$. We’ve already shown that $w_0 = b$ and $w_1 = 4b$ are high-probability bounds on the width after 0 and 1 layers are removed. More generally, the probability that we get more than $w_{i+1}$ width after removing $i$ layers is bounded by

$$(10n_i^{-1/2} w_i)^{w_{i+1}} = (10n_i^{-2-i-1} 4^i)^{4^{i+1} b}$$

$$= (10b)^{4^{i+1} b} 2^{2(i+1)} n_i^{-2-i-1} 4^{i+1} b$$

$$\leq (10b)^{4^{i+1}} 2^{2(i+1)} n_i^{-2^{i+1} b}$$

$$= (10b)^{4^{i+1}} 2^{2(i+1)} n_i^{-2^{i+1} b}$$

Since everything up to the $n$ term is a constant, for sufficiently large $n$ it happens that $n^{-2^{i+1} b}$ not only blows it away but puts what’s left well under $n^{-b}$.

Let $h$ be the height of the original circuit. After $h - 2$ restrictions, we get, with nonzero probability, a depth-2 circuit with its first layer width

\[5\] Which I didn’t do in class on 2017-03-1, leading to much embarrassment.

\[6\] We probably could have used a tighter bound on $w_i$, but $4^i b$ is easy to write down.
CHAPTER 10. CIRCUIT COMPLEXITY

bounded by $4^h b$, a constant. Such a circuit must have size at most $1 + (2n + 1)4^h b$, a polynomial in $n$, since with bounded width we can only get so many combinations of inputs before we start getting duplicate gates we can remove. So if PARITY is in $\text{AC}^0$, this construction gives us circuits for computing parity on arbitrarily large numbers of inputs that have depth 2 and polynomial size. But we’ve already shown this to be impossible. It follows that $\text{PARITY} \not\in \text{AC}^0$.

For the last step, we can encode the argument that a constant-width depth-2 circuit doesn’t compute parity as a further restriction: if we restrict the inputs to the depth-2 circuit to ones that fix one of the bottom-layer gates, we get a constant function. So what we are really showing is that for sufficiently large $n$, any $\text{AC}^0$ function is constant over some reasonably large subcube of the hypercube of all possible inputs of size $n$.

This is an example of a proof that doesn’t relativize. The reason is that random restrictions don’t do anything sensible to arbitrary oracle gates, so we really are using the fact that our original $\text{AC}^0$ circuit is built from AND and OR gates. In the following section, we will see a different proof that also doesn’t relativize, because again we are looking at specific properties of AND and OR gates.

10.3.4.3 Low-degree polynomials

A different proof, due to Razborov and Smolensky, shows that $\text{PARITY} \not\in \text{AC}^0$ based on approximating constant-depth circuits by low-degree polynomials over $\mathbb{Z}_p$ for some prime $p \neq 2$.[7]

The nice thing about this proof is that it (a) gives a lower bound on approximating parity as well as computing parity, and (b) works even if we throw in mod-$p$ gates (that return 0 if the sum of their inputs is a multiple of $p$ and 1 otherwise), for any single prime $p \neq 2$. Throwing in mod-$p$ gates gives us the class $\text{AC}^0[p]$, which is a bit stronger than $\text{AC}^0$ for prime $p$ (for example, $\text{AC}^0[2]$ can compute parity), but probably not by much.

We are going to start by approximating our depth-$h$ circuit by a polynomial that computes a function $\{0, 1\}^n \rightarrow \{0, 1\}$. This 0–1 basis is convenient for constructing the approximation, although when we want to show that the polynomial doesn’t approximate parity it will be handy to switch to a

---

[7] The proof has two parts: Razborov [Raz87] showed that low-degree polynomials over $\mathbb{Z}_p$ can approximate low-depth circuits, and Smolensky [Smo87] shows that low-degree polynomials over $\mathbb{Z}_p$ couldn’t approximate PARITY, which implies that constant-depth circuits can’t either. Both parts are needed to show $\text{PARITY} \not\in \text{AC}^0[p]$, so it has become conventional just to refer to the full result as the Razborov-Smolensky Theorem.
±1 basis, which will involve a change of variables to turn the polynomial into a function \((-1, +1)^n \rightarrow \{-1, +1\}.

Over the 0–1 basis, here is how to encode a depth-\(h\), size \(N\) circuit as a degree \(((p-1)\ell)^h\) polynomial that computes the right answer on any fixed input with probability at least \(1 - N \cdot 2^{-\ell}\):

1. Encode an input \(x_i\) as the variable \(x_i\).
2. Encode \(\neg p\) as \(1 - p\). This does not change the degree.
3. For OR, we will do something sneaky to avoid blowing up the degree too much.

Supposing we are computing \(\bigvee_{j=1}^k p_{ij}\), where each \(p_{ij}\) has degree \(d\).
Take a random subset \(S\) of the \(i_j\), and compute \(\sum_{i_j \in S} p_{ij}\). This has degree \(d\) as well, and if all \(p_{ij}\) are 0, it is 0, and if at least one \(p_{ij}\) is not zero, then including the last nonzero \(p_{ij}\) or not in \(S\) changes the value of the sum, meaning that there is at least a \(1/2\) chance that the sum is nonzero. We can convert this nonzero sum back to 1 using Fermat’s Little Theorem: \(\left(\sum_{i_j \in S} p_{ij}\right)^{p-1} = 1 \pmod{p}\) if and only if \(\sum_{i_j \in S} p_{ij} \neq 0 \pmod{p}\). This gives us a polynomial that is 0 for the all-0 input and 1 with probability at least \(1/2\) for any nonzero input.

Now blow this up by choosing \(\ell\) random \(S_i\) and let \(p = 1 - \prod_{i=1}^\ell \left(1 - \sum_{i_j \in S} p_{ij}\right)^{p-1}\).
This has degree \(d(p-1)\ell\) and computes OR with probability at least \(1 - 2^{-\ell}\).

4. For AND, use De Morgan’s Laws and NOT to reduce to OR; the degree is again \(d(p-1)\ell\) with probability of error at most \(2^{-\ell}\).

5. For mod-\(p\), use \((\sum p_i)^{p-1}\). Degree is \(d(p-1) \leq d(p-1)\ell\) and there is no error.

Applying this over depth \(h\) gives a degree bound of \(((p-1)\ell)^h\), since we go up by at most a factor of \((p-1)\ell\) at each level. Taking a union bound over all gates gives the \(N \cdot 2^{-\ell}\) bound on error. This means that each of the \(2^n\) possible inputs contributes \(N \cdot 2^{-\ell}\) bad outputs on average, for a total of \(2^n \cdot N \cdot 2^{-\ell}\) bad outputs on average. Since this is an average, some specific choice of random subsets must do no worse, showing that there exists a polynomial of degree \(((p-1)\ell)^h\) over \(\mathbb{Z}_p\) that computes the same function as the original circuit on all but \(2^n \cdot N \cdot 2^{-\ell}\) inputs.

Now we need to show that this polynomial can’t approximate parity very well. To do so, it is helpful to switch to a ±1 basis by replacing each
occurrence of $x_i$ with $-(y_i + 1) \cdot 2^{-1} = -(y_i + 1) \cdot \frac{2^i + 1}{2}$ and similarly adjusting
the output of the resulting polynomial $q'$ by changing it to $q = 1 - 2q'$; this
effectively sends every bit $b$ to $(-1)^b$, making parity just the product of all
the inputs. Note that neither step affects the degree of the polynomial: we
still have a degree $((p - 1)\ell)^h$ polynomial that is a good approximation to
the circuit.

Now to show that such a polynomial can’t be a good approximation to
parity, which will show that the circuit doesn’t compute parity. Let $G$ be the
set of inputs $y$ in $\{-1, +1\}^n$ on which our degree-$d$ polynomial $q$ computes
parity. Consider the set of all functions $f : G \rightarrow \{-1, +1\}$. For each such
$f$, there is a polynomial $f_p(y)$ (of degree $n$) over that computes $f$ exactly.
Because the inputs to $f$ are all $\pm 1$, we can assume that each monomial in
$f_p$ is of the form $\prod_{i \in S} y_i$, since any $y_i^2$ that appears can be replaced by 1.
Assuming we only care about inputs in $y$, we can make the same assumption
about $q$. We will now use $q$ to smash down $f_p$ to have degree close to $n/2$, and
argue that there aren’t enough of these lower-degree polynomials to
cover all possible $f : G \rightarrow \mathbb{Z}_p$.

Take any monomial $\prod_{i \in S} y_i$ where $|S| > n/2$. This is equal to $\prod_{i \in S} y_i \prod_{i=1}^n y_i =
\left(\prod_{i \in S} y_i\right) q(y)$ for any $y \in G$. By applying this transformation to all the
high-degree monomials in $f_p$, we get a new polynomial that computes the
same function on $G$ and has degree at most $n/2 + ((p - 1)\ell)^h$.

Let $d = ((p - 1)\ell)^h$ and $\epsilon = N \cdot 2^{-\ell}$. Then we have at least $p^{2n(1-\epsilon)}$ possible
$f$ ($p$ choices for each element of $G$) and only $p^{\sum_{i=0}^d \binom{n}{i}}$ possible degree-$(n + d)$
polynomials. Let $\ell = \frac{N^{1/(2h)}}{p-1}$, so that $d = \sqrt{n}$ and $\epsilon = N \cdot 2^{n^{1/(2h)}/(p-1)} \leq
n^{b-n^{1/(2h)/(p-1)} = o(1)}$ (assuming $N \leq n^b$ for some constant $b$). Then
$|\{f\}| = p^{2n(1-\epsilon)} = p^{2n(1-o(1))}$ but $|\{f_p\}| = p^{c \cdot 2^n}$ for a constant $c < 1$ (since
we are only going out one standard deviation).

At some point the $1 - o(1)$ gets bigger than the $c$, and we can’t cover
all functions $f$ with degree $n/2 + \sqrt{n}$ polynomials any more. This implies
that $q'$ does not in fact work as well as advertised, meaning that our original
circuit fails to compute parity for some input.

With some tinkering, essentially the same argument shows that a $\text{AC}^0[p]$ circuit family can’t compute MOD-$q$ when $p \neq q$ and both are prime (or prime powers). We don’t know much about what happens with circuits with
MOD-$m$ gates where $m$ is not a prime power; polynomials over even $\mathbb{Z}_q$ turn
out to be surprisingly powerful. citeBarringtonBR1994.
10.4 Monotone circuits

[[[ Razborov’s clique bound ]]]
[[[ maybe skip this ]]]
Chapter 11

Natural proofs

The circuit lower bounds in Chapter 10 generally don’t relativize, since oracle gates don’t do anything sensible when hit with random restrictions or reduced to polynomials. But they still are unlikely to generalize to produce stronger results, at least if we believe in cryptography. This follows from the natural proofs theorem of Razborov and Rudich [RR97].

The idea is to rule out certain “natural” classes of proof strategies, which involve showing that there is a property \( \Phi \) such that (a) any Boolean function that has this property is hard for some class, and (b) a particular Boolean function has this property. An example would be PARITY’s property of not becoming the constant function under a large enough restriction, which is not true of functions in \( \text{AC}^0 \). The argument is that if we have such a proof strategy that works and satisfies certain technical requirements, then we can use it to break cryptographic primitives. This means that showing that one thing we think is hard is hard, in this particular way, will involve showing that something else we think is hard is easy.

11.1 Natural properties

A property \( \Phi \) is just a set of Boolean functions. We will use \( \Phi_n \) for the restriction of \( \Phi \) to functions on \( n \) inputs and use \( f_n \) to refer to the restriction of a function \( f \) to \( n \) inputs. A property is constructive if the problem \( f_n \in \Phi_n \) is in \( \text{P} \), where the input to the tester is a string of length \( 2^n \) giving the entire truth table for \( f_n \) and the time to test is thus polynomial in \( 2^n \);

\[\text{1The presentation in this chapter is very, very sketchy, and the presentation in [AB07, Chapter 22] is not much better. I recommend if you are interested in this to look at the original Razborov-Rudich paper [RR97].}\]
large if at least a $2^{-O(n)}$ fraction of functions on $n$ inputs are in $\Phi_n$;  and useful against $\mathbf{P}/\text{poly}$ if $f \in \Phi$ implies $f \not\in \mathbf{P}/\text{poly}$.

If $\Phi$ is both constructive and large, we say that it is natural\footnote{More generally, we say that it is $C$-natural if it is testable in class $C$ and large.} The Razborov-Rudich theorem says a natural property useful against $\mathbf{P}/\text{poly}$ breaks certain pseudorandom function generators, which we now take a brief detour to define.

## 11.2 Pseudorandom function generators

For the purposes of the Razborov-Rudich Theorem, we will consider a class of pseudorandom function generators that are indistinguishable from a random function by a Turing machine running in time $2^{O(n)}$ with oracle access to the function. In this definition, a family of functions $\{f_n\}$ is pseudorandom if, for any machine $M$ running in time $2^{O(n)}$, and $n$ sufficiently large,

$$|\Pr [M(f_n(x) = 1)] - \Pr [M(R) = 1]| \leq 2^{-n^2},$$

where $x$ is chosen uniformly from $\{0, 1\}^n$ and $R$ is a random function. A pseudorandom function generator is a machine $M(s, x)$ that given any polynomial-size seed $s$ becomes a pseudorandom function $f_s^n$ on $x$.

What is interesting about these generators is that it is possible to show that they exist unless problems like FACTORING or DISCRETE LOG can be solved in $O(2^n)$ time for any $\epsilon > 0$.

## 11.3 The Razborov-Rudich Theorem

Here we give a formal statement of the Razbov-Rudich theorem. This version is adapted form [AB07, Theorem 22.9]; the original version in the Razbov-Rudich paper [RR97] is stated more generally.

**Theorem 11.3.1** (Razbov-Rudich [RR97]). If there is a $\mathbf{P}$-natural property $\Phi$ that is useful against $\mathbf{P}/\text{poly}$, then there are no strong pseudorandom function generators as defined in the preceding section.

**Proof.** We’ll use $\Phi$ to distinguish a pseudorandom function generator from a random function $R$.

First observe that for any fixed seed $s$, the function $f_s^n$ is computable in $\mathbf{P}/\text{poly}$. So pseudorandom functions do not have property $\Phi$ (because of usefulness). On the other hand, a random function does have property

\[ |\Pr [M(f_n(x) = 1)] - \Pr [M(R) = 1]| \leq 2^{-n^2}, \]
Φ with probability at least \(1 - 2^{-O(n)}\), since Φ is large. So the probability that Φ distinguishes \(f_n^s\) from \(R\) is at least \(2^{-O(n)} > 2^{-n^2}\). This is a problem because Φ is constructible, so we can use it to distinguish \(f_n^s\) from \(R\).

If we replace \(\mathbf{P}/\mathbf{poly}\) with some other class (like \(\mathbf{AC}^0\)), then we get a similar result for pseudorandom function generators that fool the other class instead of \(\mathbf{P}/\mathbf{poly}\).

11.4 Examples of natural proofs

The random restriction lower bound for \(\text{PARITY}\) given in §10.3.4.2 can be described by a property \(\Phi\) where \(\Phi(f) = 1\) if \(f\) cannot be reduced to a constant function via restricting \(n - n^c\) inputs, where \(c < 1\). This is large since almost all functions have this property, and constructive since we can check all restrictions from the truth table in time polynomial in \(2^n\). It is also useful against \(\mathbf{AC}^0\) because \(\mathbf{AC}^0\) functions don’t have the property.

But because \(\Phi\) is large and \(\mathbf{P}\)-constructible, Theorem 11.3.1 implies that random restrictions won’t be useful against \(\mathbf{P}/\mathbf{poly}\) unless there are no strong pseudorandom function generators.

An example that requires a little more pushing and shoving is Razborov-Smolensky (see §10.3.4.3). The main property used in that proof is that any function \(f : \{-1, 1\} \rightarrow \mathbb{Z}_p\) can be written as \((\prod_{i=1}^n)\ell_1 + \ell_2\) where \(\ell_1\) and \(\ell_2\) are degree-(\(n/2\)) polynomials and the product is the encoding of parity when each bit is represented by an element of \(\{+1, -1\}\). This property is unfortunately unique to parity, which would make it not large. So Razborov and Rudich suggest looking at the set of all functions \(\{+1, -1\}^n \rightarrow \mathbb{Z} - P\) as a \(2^n\)-dimensional vector space and instead using the property \(\Phi(g)\) that the dimension of the set of functions \(\{g \cdot \ell_1 + \ell_2 | \ell_1\) and \(\ell_2\) have degree at most \(n/2\}\) is at least \((3/4)2^n\). This property does not hold for functions approximated by low-degree polynomials (pretty much the same counting argument for the parity case), so it’s useful against \(\mathbf{AC}^0[p]\). At the same time it is large (shown in the paper) and constructible (build a \(m \times n\) matrix whose \(m \leq 2^{2^n}\) rows are all functions in the set and check its dimension using Gaussian elimination in time polynomial in \(2^n\)). So if strong pseudorandom function generators exist, it can’t be useful against \(\mathbf{P}/\mathbf{poly}\).

This second example shows that even proofs that look unnatural might have a natural proof buried inside them. One of the things that Razborov and Rudich do in their paper is go through a catalog of then-known lower bound results, mostly in circuit complexity, and demonstrate that each argument can be interpreted to give a natural property useful against some class. This
means that it is not enough to structure a proof that $\mathbf{P} \neq \mathbf{NP}$ (for example) based on a property only applies to a non-large class of functions or is not obviously constructible: it must also be the case that the proof technique used to show this property is hard can’t generalize to some other property that is both large and constructible. So far we have not had much luck finding such properties, and the Razborov-Rudich paper itself pretty much put an end to most work on circuit complexity lower bounds.\footnote{Curiously, the field of circuit complexity lower bounds can in some sense be traced back to a proof by Minsky and Papert \cite{MP69} that certain classes of depth-2 circuits based on majority gates (perceptrons, a kind of neural network) couldn’t compute PARITY. This put a serious damper on neural network research for a generation and was part of what led to the widespread belief in the “AI winter” era of the late 1980s that artificial intelligence as a field was basically doomed. Nowadays more sophisticated neural networks are the basis of the deep learning methods that have had significant success in solving previously-intractable pattern recognition problems.

And yet these fancier neural networks still can’t compute PARITY. It’s fortunate in retrospect that the problem of recognizing speech or driving a car doesn’t seem to depend on calculating whether the number of one bits in the input is odd or even. Perhaps we can be hopeful that something similar will happen with circuit complexity.}
Chapter 12

Counting classes

[[[ RP, co-RP, ZPP, BPP, PP, RL; Sipser-Gacs ]]]
Chapter 13

Descriptive complexity

[[[ Fagin’s Theorem. Not sure if we want this this early but having a descriptive-complexity version of NL makes applications of Immerman-Szelepcsényi easier. ]]]

[[[ maybe better to do Immerman-Szelepcsényi first then talk about log-space hierarchy with poly-time hierarchy ]]]
Chapter 14

Toda’s Theorem

[[] do it ]]

87
Chapter 15

Interactive proofs

\[\text{IP} = \text{PSPACE}\]
Chapter 16

Probabilistically-checkable proofs

[[[ PCP theorem if we can manage it ]]]
Appendix A

Assignments

Assignments should be submitted in PDF format via the classesv2 Drop Box mechanism. Give your assignment a reasonable filename like hw1.pdf so I can figure out which it is. You do not need to format your assignment using \LaTeX{} or some other document preparation system if you don’t want to, but if your handwriting is particularly illegible or your cell-phone camera is not very good at taking pictures, I may deduct points for details I can’t parse.

A.1 Assignment 1: due Wednesday, 2017-02-01 at 23:00

A.1.1 Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.

2. Your status: whether you are an undergraduate, grad student, auditor, etc.

3. Anything else you’d like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

A.1.2 Binary multiplication

A finite-state transducer (FST) is a Turing machine with a read-only input tape, a write-only output tape, no work tapes, and heads that can
only stay put or move right at each step. We would like to get a finite-state transducer to multiple binary numbers by 3.

1. Suppose that the input and output are both given most-significant-bit (MSB) first: for example, the input 6 is represented by \( \langle 6 \rangle \) = 110 and the corresponding output \( 3 \cdot 6 = 18 \) is represented by \( \langle 18 \rangle \) = 10010. Give a program for a finite-state transducer that multiplies its input by 3 using this representation, or show that no such program is possible.

2. Suppose instead that the input is given least-significant-bit (LSB) first: now 6 is represented by \( \langle 6 \rangle_R \) = 011 and \( 3 \cdot 6 = 18 \) is represented by \( \langle 18 \rangle_R \) = 01001. Give a program for a finite-state transducer that multiplies its input by 3 using this representation, or show that no such program is possible.

**Solution**

1. A finite-state transducer cannot multiple binary numbers by 3 with the MSB-first representation.

Let \( x_k \), for each integer \( k > 0 \), be the input with binary representation \((10)^k\). Then \( x_k = 2 \sum_{i=0}^{k-1} 4^i = 2 \cdot \frac{4^k - 1}{3} \), and \( 3x_k = 2 \cdot (4^k - 1) \), which has the binary representation \( 1^{2k-1}0 \).

Now compare with \( x_k + 1 \): \( \langle x_k + 1 \rangle = (10)^{k-1}11 \) differs from \( \langle x_k \rangle = (10)^{k-1}10 \) only in the last bit, but \( \langle 3(x_k + 1) \rangle = 10^{2k}1 \) already differs from \( \langle 3x_k \rangle = 1^{2k-1}0 \) on the second bit. We will use this fact to argue that any FST for this problem must give the wrong answer for some input \( x_k \) or \( x_k + 1 \).

Fix some FST \( M \). Consider executions of \( M \) on \( x_k \) and \( x_k + 1 \). The second bit in \( M \)'s output is the complement of the last bit it reads, so if it outputs more than one bit before reading the last bit of its input, it will be incorrect in one of the two executions. It follows that a correct \( M \) cannot output more than one bit without reading its entire input.

Now consider executions with inputs \( x_k \), where \( k \) ranges over all positive integers. Let \( q_k \) be the state of the finite-state controller when \( M \) first reaches a configuration where the input head is over the last bit of the input. There are infinitely many \( k \), but only finitely many possible \( q_k \), so there must be two values \( k \neq k' \) such that \( q_k = q_{k'} \). We have previously established that when \( M \) reaches the last input bit in either \( x_k \) or \( x_{k'} \), it is in state \( q_k \) and has output at most one bit. Any
APPENDIX A. ASSIGNMENTS

<table>
<thead>
<tr>
<th>q</th>
<th>read</th>
<th>q′</th>
<th>write</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨0,0⟩</td>
<td>0</td>
<td>⟨0,0⟩</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>⟨0,0⟩</td>
<td>1</td>
<td>⟨0,1⟩</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>⟨0,0⟩</td>
<td>b</td>
<td>⟨0,0⟩</td>
<td>b</td>
<td>S</td>
</tr>
<tr>
<td>⟨0,1⟩</td>
<td>0</td>
<td>⟨0,0⟩</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>⟨0,1⟩</td>
<td>1</td>
<td>⟨1,1⟩</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>⟨0,1⟩</td>
<td>b</td>
<td>⟨0,0⟩</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>⟨1,0⟩</td>
<td>0</td>
<td>⟨0,0⟩</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>⟨1,0⟩</td>
<td>1</td>
<td>⟨1,1⟩</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>⟨1,0⟩</td>
<td>b</td>
<td>⟨0,0⟩</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>⟨1,1⟩</td>
<td>0</td>
<td>⟨1,0⟩</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>⟨1,1⟩</td>
<td>1</td>
<td>⟨1,1⟩</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td>⟨1,1⟩</td>
<td>b</td>
<td>⟨1,0⟩</td>
<td>0</td>
<td>R</td>
</tr>
</tbody>
</table>

Table A.1: Transition table for multiplying by 3 (LSB first)

...
A.1.3 Transitivity of $O$ and $o$

Use the definitions given in §3.1.2.1 to show that:

1. If $f(n) = o(g(n))$, then $f(n) = O(g(n))$.
2. If $f(n) = o(g(n))$ and $g(n) = O(h(n))$, then $f(n) = o(h(n))$.

Solution

1. Fix some $c > 0$. Then $f(n) = o(g(n))$ means that there is some $N$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$. But the existence of $c$ and $N$ with this property means that $f(n) = O(g(n))$.

2. We want to show that for any $c > 0$, there exists $N$ such that $f(n) \leq c \cdot h(n)$ for all $n \geq N$. Fix some such $c$. Let $c_2$ and $N_2$ be such that $g(n) \leq c_2 \cdot h(n)$ for all $n \geq N_2$. Let $c_1 = c/c_2$ and let $N_1$ be such that $f(n) \leq c_1 g(n)$ for all $n \geq N_1$. Then for any $n \geq \max(N_1, N_2)$, we have $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = (c/c_2)c_2 h(n) = ch(n)$ as required.

It’s worth noting that essentially the same argument for part 2 shows that $f(n) = O(g(n))$ and $g(n) = o(h(n))$ together imply $f(n) = o(h(n))$, but one tedious proof is enough.

A.2 Assignment 2: due Wednesday, 2017-02-15 at 23:00

A.2.1 A log-space reduction

The usual definition of \textbf{NP}-completeness uses polynomial-time reductions, where $A \leq_P B$ if there is a polynomial-time computable $f$ such that $x \in A$ if and only if $f(x) \in B$. But we could also consider other kinds of reductions, characterized by a different restriction on $f$.

One of these is a \textbf{log-space reduction}. We write $A \leq_L B$ if there is a function $f$ computable on a standard Turing machine using $O(\log n)$ space such that $x \in A$ if and only if $f(x) \in B$.

Show that \textsc{independent set} $\leq_L \textsc{clique}$, where the input $k$ and $G = (V, E)$ to both problems is given by $1^k$, followed by a delimiter of some sort (say, “;”), followed by $1^{|V|}$, followed by another delimiter, and finally a sequence of pairs of vertex ids representing the edges (in no particular order), with each vertex id represented as a binary number in the range $0 \ldots |V| - 1$, terminated by the delimiter.
You do not need to (and probably shouldn’t, unless you are bored and immortal) give an explicit transition table, but you should describe the workings of a log-space Turing machine that computes $f$ in enough detail that you can argue that it does in fact run in logarithmic space.

Solution

The usual polynomial-time reduction from INDEPENDENT SET to CLIQUE replaces $G$ with its complement $\overline{G}$, which has the same vertices but contains each possible edge $uv$ if and only if $G$ does not contain $uv$. So we would like to implement this mapping in log space.

For $\lfloor k \rfloor$ and $\lfloor |V| \rfloor$, we can just copy the input until we reach the second delimiter. This requires no space beyond a few states in the finite-state controller. So the only difficult part is complementing $E$.

One what to do this is to use three tapes to keep track of binary representations of $v = |V|$, and counters $i$ and $j$ that run from 0 to $|V| - 1$. Observe that using only a finite number of states in the controller, we can do all of the following tasks in space $O(\log |V|)$:

1. Set a counter to 0.
2. Increment $v$, $i$, or $j$ by 1.
3. Compare $i$ or $j$ to $v$.
4. Search $E$ for an edge $ij$ or $ji$.
5. Write edge $ij$ to the output tape.

We can then run the algorithm given in Algorithm A.1. Each of the steps in this algorithm can be done without using any work tape space beyond the space needed to represent $v$, $i$, and $j$, so the total space complexity is $O(\log |V|)$. This is logarithmic in the size of the input because $|V|$ is expressed in unary.

A.2.2 Limitations of two-counter machines

Recall that a two-counter machine consists of a finite-state controller, a read-only input tape, a write-only output tape, and two counters, that

\footnote{It is convenient for us that whoever picked the representation of $G$ made this choice. If $k$ and $|V|$ were presented in binary, we might have to worry about what happens with very sparse graphs.}
Appendix A. Assignments

1. Copy \( \uparrow k \) and \( \uparrow |V| \) to the output tape.
2. \( v \leftarrow 0 \)
3. \( i \leftarrow 0 \)
4. for each 1 in the representation of \( |V| \) do
5.   Increment \( v \)
6. while \( i \neq v \) do
7.     \( j \leftarrow 0 \)
8.     while \( j \neq v \) do
9.         if \( ij \notin E \) and \( ji \neq E \) then
10.            Write \( ij \) to the output
11.       Increment \( j \)
12.      Increment \( i \)

Algorithm A.1: Log-space reduction from INDEPENDENT SET to CLIQUE

Support increment and decrement operations, and that can be tested for equality with zero by the finite-state controller.

Show that if \( f(n) \geq n \) is time-constructible (by a standard Turing machine) then there exists a language \( L \) that can be decided by a two-counter machine eventually, but that cannot be decided by a two-counter machine in \( o(f(n)) \) time.

Solution

Let \( L = \{\langle \downarrow M, x \rangle \mid M \text{ is a two-counter machine that rejects } x \text{ in at most } f(n) \text{ steps} \} \).

Claim: If \( R \) is a two-counter machine that runs in \( o(f(n)) \) steps, then \( L(R) \neq L \). Proof: Let \( x \) be large enough that \( R(\langle \downarrow R, x \rangle) \) runs at most \( f(n) \) steps. Then \( \langle \downarrow R, x \rangle \) is in \( L \) if and only if \( R(\langle \downarrow R, x \rangle) \) rejects. Either way, \( L(R) \neq L \).

Now we have to show that \( L \) can be decided by a two-counter machine without the time bound. We could build a universal two-counter machine directly, but this is overkill. Instead, let’s take this approach to show we can decide \( L \) using a Turing machine:

1. We can translate a representation \( \downarrow M \) of a two-counter machine to a representation of \( \downarrow M' \) of a Turing machine using two work tapes with a single mark to represent zero and the head position to represent the counter value.
2. We can then use one of our previous constructions to show that we can
decide using a Turing machine if \( M' \) rejects \( x \) in \( f(n) \) steps.

This means that there exists a Turing machine that decides \( L \). Since we
can simulate Turing machines using two-counter machines (Lemma 3.1.1),
there is also a two-counter machine that decides \( L \).

**A better solution:** Though the preceding works, it’s overkill, since we
can avoid diagonalizing over two-counter machines entirely.

The Time Hierarchy Theorem says that if \( f(n) \) is time-constructible, then
there is a language \( L \) that cannot be decided by a Turing machine in \( o(f(n)) \)
time but can be decided by a Turing machine eventually.

Since a Turing machine can simulate a two-counter machines with no
slowdown (use two work tapes with a single mark on each for the counter),
if \( L \) can be decided by a two-counter machine in \( o(f(n)) \) time, it can also
be decided by a Turing machine in \( o(f(n)) \) time, but it can’t. So \( L \) is not
decided by a two-counter machine in \( o(f(n)) \) time. On the other hand, it is
decided by some Turing machine eventually, and a two-counter machine that
simulates that Turing machine will also decided it eventually.

A.3 Assignment 3: due Wednesday, 2017-03-01 at 23:00

A.3.1 A balanced diet of hay and needles

Call an oracle \( A \) **balanced** if, for each \( n \geq 1 \), \(|\{x \in A \mid |x| = n\}| = 2^{n-1}\).

Show that there exist balanced oracles \( A \) and \( B \) such that \( P^A = NP^A \)
and \( P^B \neq NP^B \).

**Solution**

Rather than build new oracles from scratch, we’ll show how to encode any
oracle so that it is balanced, and use this to reduce to Baker-Gill-Solovay.

Given an oracle \( A \), define its balanced version \( \tilde{A} = \{x0 \mid x \in A\} \cup \{x1 \mid x \not\in A\} \).

Since every \( x \) with \( |x| = n \) produces exactly one element of \( \tilde{A} \) with
\( |x| = n + 1 \), we get exactly \( 2^n \) elements of size \( n + 1 \), making \( \tilde{A} \) balanced.

\[2\]I would like to thank Aleksandra Zakrzewska for pointing this out in her solution,
unlike the rest of us lemmings who blindly went ahead and diagonalized over two-counter
machines.
Any machine that uses $A$ can be modified to use $\tilde{A}$ instead, by writing an extra 0 to the end of each oracle call, thus replacing a class to $A(x)$ with $A(x0)$. In the other direction, a machine that uses $\tilde{A}$ can be modified to use $A$ instead, by replacing the response to each call to $\tilde{A}(xb)$ with $A(x) \oplus b$. These modifications are easily made to either a $P$ or $NP$ machine.

Now let $A$ and $B$ be the oracles from the Baker-Gill-Solovay Theorem for which $P^A = NP^A$ and $P^B \neq NP^B$. Then $\tilde{A}$ and $\tilde{B}$ are balanced oracles for which $P^{\tilde{A}} = NP^{\tilde{A}}$ and $P^{\tilde{B}} \neq NP^{\tilde{B}}$.

### A.3.2 Recurrence

Call a vertex $u$ in a directed graph $G$ recurrent if a random walk starting at $u$ eventually returns to $u$ with probability 1. (This is true if and only if $u$ is reachable from any node $v$ that is reachable from $u$.) Let $L = \{ \langle G, u \rangle \mid u \text{ is recurrent in } G \}$. Show that $L$ is $NL$-complete with respect to log-space reductions.

#### Solution

We need to show $L \in NL$ and $\forall L' \in NL L' \leq_L L$.

- $L$ is in $NL$. Proof: We can test $u$ in $L^{NL}$ by iterating over all $v$ and checking using an STCON oracle if there is a path from $u$ to $v$ and a path from $v$ to $u$. If we find the former but not the latter, $\langle G, u \rangle \notin L$. This shows $L \in L^{NL} \subseteq NL^{NL} = NL$.

- $L$ is $NL$-hard. Proof: We’ll show it’s $coNL$-hard by log-space reduction from the complement of STCON. Given a graph $G = (V, E)$ and nodes $s$ and $t$, construct a new graph $G' = (V, E')$ where

$$E' = E \cup \{ vs \mid v \in V, v \neq t \} \setminus \{tv \mid v \in V \}.$$ 

Let $u = s$.

Then every $v \neq t$ can reach $u$ in $G'$. So the only way that $u$ is not recurrent is if $u = s$ can reach $t$ in $G'$, which has no outgoing edges. If there is an $s$–$t$ path in $G$, then there is an $s$–$t$ path in $G'$, since if some $G$ path uses an outgoing edge from $t$ we can truncate to the shortest prefix that reaches $t$ and get an $s$–$t$ path that doesn’t. Conversely, any $s$–$t$ path in $G'$ gives an $s$–$t$ path in $G$ by taking the shortest suffix that contains $s$. So $u$ is recurrent in $G'$ if and only if there is no $s$–$t$ path in $G$. 

This makes $L$ coNL-hard. But coNL = NL, so $L$ is also NL-hard.

A.4 Assignment 4: due Wednesday, 2017-03-29 at 23:00

A.4.1 Finite-state machines that take advice

A manufacturer of low-budget Turing machines decides that advice is so powerful, that a machine that uses it might still be useful even without any write heads.

Define the class $\text{FSM/poly}$ to be the set of languages $L$ decided by a Turing machine $M$ with two read-only input tapes, one of which is a two-way tape that contains the input $x$ and one of which is a one-way tape that contains an advice string $\alpha | x$, where both tape alphabets consist of bits. We say that $M$ decides $L$ if and only if there is a family of advice strings $\{\alpha_n\}$ of size polynomial in $n$ that cause $M$ to always output the correct value when presented with $x$ and $\alpha_n$.

(The advice tape is one-way, only allowing its head to move to the right, to keep us from using it as a counter. The input tape is two-way because making it one-way would just be cruel.)

Show that $\text{NC}^1 \subseteq \text{FSM/poly} \subseteq \text{AC}^1$.

A.4.2 Binary comparisons

Suppose that we extend $\text{AC}^0$ by adding binary comparison gates. A $2m$-input binary comparison gate takes inputs $x_{m-1}, \ldots, x_0$ and $y_{m-1}, \ldots, y_0$, and returns 1 if and only if $\sum_{i=0}^{m-1} 2^i x_i > \sum_{i=0}^{m-1} 2^i y_i$.

Define the complexity class $\text{ACBC}^0$ to consist of all circuits of polynomial size and constant depth consisting of unbounded fan-in AND, OR, and NOT gates, and polynomial fan-in binary comparison gates.

Prove or disprove: $\text{PARITY} \in \text{ACBC}^0$.

A.5 Assignment 5: due Wednesday, 2017-04-12 at 23:00

[[] To be announced. ]]]
A.6 Assignment 6: due Wednesday, 2017-04-26 at 23:00

[[] To be announced. ]]
Bibliography


[GGH+16] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability


Index

0–1 basis, 77
±1 basis, 78
NL-complete, 60
NP-complete, 28
NP-hard, 28
P/poly, 67
P-complete, 60
coNP-complete, 36
s–t connectivity, 61
SC, 56
EXP, 37
NEXP, 37
coNP, 36
NP-intermediate, 49
3-colorable, 35
3SAT, 29
advice, 67
alphabet, 4
alternating class, 70
alternating polynomial time, 55
alternating polynomial-time hierarchy, 53
balanced oracle, 96
basis
0–1, 77
±1, 78
binary, 4
binary comparison gate, 98
Boolean circuit, 65
Boolean formula, 65
Boolean function, 65
bounded-width branching program, 71
branch, 25
branching program, 71
bounded-width, 71
canonical decision tree, 74
certificate, 25
Church-Turing hypothesis, 22
Church-Turing thesis, 5
extended, 22
24

circuit

Boolean, 65
monotone, 68
circuit complexity, 65
circuit family, 66
circuit value problem, 62
class

complexity, 4
Steve’s, 56

CLIQUE, 34
CNF, 29
colorable, 35
commutator, 72
complement, 36
complete, 60
NL, 60
P, 60

complex systems, 1
complexity
circuit, 65
INDEX

time, 7
complexity class, 1, 4, 23
complexity theory, 1
computational complexity theory, 1
configuration, 6
conjunctive normal form, 29
connectivity
s-t, 61
constructible
space-, 23
time-, 23
constructive, 81
Cook-Levin theorem, 29
course staff, ix
decide, 4
decision problem, 3
decision tree, 74
canonical, 74
depth, 65
diagonalization, 38
exponential time, 37
extended Church-Turing thesis, 5, 22
family
circuit, 66
fan-in, 65
fan-out, 65
feasible, 24
finite-state control, 5
finite-state transducer, 90
fix, 74
formula
Boolean, 65
function
Boolean, 65
gate, 65
binary comparison, 98
graph
implication, 62
group
symmetric, 71
Håstad’s Switching Lemma, 73
halt, 6
Halting Problem, 38
halting state, 6
heads, 5
hierarchy
polynomial-time, 52
alternating, 53
oracle, 52
hierarchy theorem
space, 40
time, 42
hypothesis
Church-Turing, 22
extended, 22
implication graph, 62
independence
representation, 4
INDEPENDENT SET, 33
independent set, 33
input, 65
instructor, ix
is, 61
kill, 74
Ladner’s Theorem, 46
language, 4
large
property, 82
lemma
Håstad’s Switching, 73
log-space, 56
nondeterministic, 56
randomized, 56
symmetric, 57
log-space reduction, 93
logspace-uniform, 69

machine
Turing, 5
universal, 19, 41
two-counter, 94
many-one reduction
polynomial-time, 28
monotone circuit, 68

natural, 82
natural proof, 68
natural proofs, 81
neural network, 84
Nick’s class, 70
non-uniform, 69
Nondeterministic log-space, 56
nondeterministic log-space, 56
nondeterministic polynomial time, 26
nondeterministic Turing machine, 25

oracle
balanced, 96
oracle polynomial-time hierarchy, 52
oracle tape, 50
output, 65

padded, 37
padding, 37
perceptron, 84
polylog space, 56
polynomial time, 23
nondeterministic, 26
polynomial-time hierarchy, 52
alternating, 53
oracle, 52
polynomial-time many-one reduction, 28

problem
decision, 3
program
bounded-width branching, 71
branching, 71
proof
natural, 68
property, 81
constructive, 81
large, 82
pseudorandom function generator, 82

RAM, 20
random access machine, 20
random restriction, 73
randomized log-space, 56
Razborov-Smolensky Theorem, 77
recurrent, 97
reduction
polynomial-time many-one, 28
reduction
log-space, 93
relation
relation, 25
relativization, 50
relativize, 50
representation independence, 4
restriction, 74
random, 73
Rice’s Theorem, 39
Savitch’s Theorem, 58
seed, 82
semantic, 39
sequence
crossing, 17
set
independent, 33
size, 7, 65
space, 2
polylog, 56
INDEX

space complexity, 7
Space Hierarchy Theorem, 40
space-constructible, 23
staff, ix
state, 6
  halting, 6
STCON, 61
Steve’s class, 56
supertask, 5
Switching Lemma
  Håstad’s, 73
symmetric group, 71
symmetric log-space, 57
tableau, 63
tapes, 5
theorem
  Cook-Levin, 29
  Ladner’s, 46
  Razborov-Smolensky, 77
  Rice’s, 39
  Savitch’s, 58
  space hierarchy, 40
  time hierarchy, 42
time
  exponential, 37
  polynomial
    nondeterministic, 26
time complexity, 7
Time Hierarchy Theorem, 42
time-constructible, 23
TM, 5
TQBF, 54
transducer
  finite-state, 90
  transition function, 6
  transition relation, 25
tree
  decision, 74
  canonical, 74

TRUE QUANTIFIED BOOLEAN FORMULA, 54
Turing Machine
  universal, 41
Turing machine, 5
  nondeterministic, 25
  universal, 19
two-counter machine, 94

unary, 4
undecidable, 39
Undergraduate Learning Assistant, ix
uniform, 69
  logspace-, 69
  non-, 69
universal, 19
universal Turing Machine, 41
universal Turing machine, 19

VERTEX COVER, 35

width
  of a DNF formula, 73
witness, 25