Contents

Table of contents i
List of figures iv
List of tables v
List of algorithms vi
Preface vii
Syllabus viii
Lecture schedule xi

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 13
   3.1.5 Limitations of simulations 17
   3.1.6 Universal Turing machines 18
   3.2 Random access machines 19
   3.3 The extended Church-Turing hypothesis 21
CONTENTS

4 Time and space complexity classes 22

5 Nontermination and NP 24
  5.1 Examples of problems in NP 25
  5.2 Reductions and NP-complete problems 26
  5.3 The Cook-Levin theorem 27
  5.4 More NP-complete problems 27

6 Diagonalization 28
  6.1 Uncomputability 28
  6.2 Hierarchy theorems 28

7 Relativization 29

8 Descriptive complexity 30

9 L vs NL 31

10 PSPACE 32

11 The polynomial-time hierarchy 33

12 Randomized classes 34

13 Circuits 35

14 Natural proofs 36

15 Toda’s Theorem 37

16 Interactive proofs 38

17 Probabilistically-checkable proofs 39

A Assignments 40
  A.1 Assignment 1: due Wednesday, 2017-02-01 at 23:00 40
     A.1.1 Bureaucratic part 40
     A.1.2 Binary multiplication 40
     A.1.3 Transitivity of $O$ and $o$ 41
     A.1.4 Vector balancing 41
  A.2 Assignment 2: due Wednesday, 2017-02-15 at 23:00 41
  A.3 Assignment 3: due Wednesday, 2017-03-01 at 23:00 42
List of Figures

3.1 Turing-machine transition function for reversing the input, disguised as a C program ........................................... 11

3.2 Recognizing a palindrome using a work tape .............................. 13
List of Tables

3.1 Turing-machine transition function for reversing the input . . 10
List of Algorithms
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/aspeness/classes/468/notes.pdf](http://www.cs.yale.edu/homes/aspeness/classes/468/notes.pdf). If the Yale CS server becomes inaccessible, a backup copy can be found at [https://www.dropbox.com/sh/j98y7z3k7u91obh/AAAg1mWHGHs5gdyKoi3rSJewaa?dl=0](https://www.dropbox.com/sh/j98y7z3k7u91obh/AAAg1mWHGHs5gdyKoi3rSJewaa?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/](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.
SYLLABUS

- \(\text{http://www.scottaaronson.com/papers/pnp.pdf}\)  Survey of current state of the \(P\) vs. \(NP\) problem.

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 \text{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 \text{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.

2016-01-18 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.

2016-01-20 Complexity classes $\text{TIME}(f(n))$, $\text{SPACE}(f(n))$, and $\text{P}$. Turing machine simulations: one-tape, two-tape, two counters. Crossing sequence arguments and lower bounds on single-tape Turing machines. The extended Church-Turing hypothesis. How to simulate anything with a Turing machine. Readings: Rest of Chapter 3; AB [[[ To be announced. ]]].

2016-01-23 Nondeterminism and $\text{NP}$. Reductions, $\text{NP}$-complete problems, and the Cook-Levin theorem. Some flaming about the consequences of $\text{P} = \text{NP}$ if true. Readings: Chapter 5 [[[ pick out specific sections ]]]; AB §§2.1–2.3, 2.7. Not required, but if you are a big fan of $\text{P} \neq \text{NP}$, 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 $\text{P} = \text{NP}$ [Imp95].

2016-01-25 Various $\text{NP}$-complete problems and examples of reductions. Other nondeterministic complexity classes. Readings: [[[ To be announced. ]]]; AB §2.4–2.6.
2016-01-30 Diagonalization: Universal Turing machines, the Halting Problem, time and space hierarchy theorems. [[[ maybe Ladner's theorem ]]]. Readings: [[[ To be announced. ]]].

2016-02-01 Oracles and relativization. The Baker-Gill-Solovay theorem. Readings: [[[ To be announced. ]]].

2016-02-06 [[[ To be announced. ]]]
2016-02-08 [[[ To be announced. ]]]
2016-02-13 [[[ To be announced. ]]]
2016-02-15 [[[ To be announced. ]]]
2016-02-20 [[[ To be announced. ]]]
2016-02-22 [[[ To be announced. ]]]
2016-02-27 [[[ To be announced. ]]]
2016-03-01 [[[ To be announced. ]]]
2016-03-06 [[[ To be announced. ]]]
2016-03-08 [[[ To be announced. ]]]
2016-03-27 [[[ To be announced. ]]]
2016-03-29 [[[ To be announced. ]]]
2016-04-03 [[[ To be announced. ]]]
2016-04-05 [[[ To be announced. ]]]
2016-04-10 [[[ To be announced. ]]]
2016-04-12 [[[ To be announced. ]]]
2016-04-17 [[[ To be announced. ]]]
2016-04-19 [[[ To be announced. ]]]
2016-04-24 [[[ To be announced. ]]]
2016-04-26 [[[ To be announced. ]]]

2016-05-06 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 \footnote{[later]}), $NP$ consists of problems whose solutions can
be \textit{verified} in time polynomial in the size of the input, and the big question is whether there are any problems in \textbf{NP} that are not also in \textbf{P}. Much of the development of computational complexity theory has arisen from researchers working very hard to answer this question.

\begin{itemize}
  \item \textbf{randomization: \textbf{P} vs \textbf{BPP}}
  \item \textbf{what complexity theory is about, basic sketch of the main classes}
\end{itemize}
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:\footnote{It’s conventional to name problems in all-caps.} 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.\footnote{Since 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.} 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
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 \( \{0, 1\} \). 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 \( 1^{19} = 1111111111111111 \)); 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 \( \langle 17 \cdot 19 \rangle = 101000111 \), 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 \( x \notin L \), 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 \( x \notin L \), we say that \( M \) decides \( L \).

---

3The 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 hypothesis 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\footnote{Alternatively, the Church-Turing hypothesis 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.} The extended Church-Turing hypothesis 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 $\Gamma$, 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 hypothesis 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 even do so with only moderate slowdown. [[point to the section in AB that shows this]] 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.\footnote{This 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.}

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.
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 special delimiter characters: an example would be writing $01$ to the tape for the input $01$. The head on the first tape starts on the leftmost input symbol ($0$ in the example). 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 any delimiter characters.

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.

---

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

### 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:

- $f(n) = O(g(n))$ 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 $n \geq N$.
- $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, or equivalently 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 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

---

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.
right-hand side to make it true.

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.

---

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 $h(n) = O(n^3)$.
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 \( \left| \Sigma \right| \) possible values for each.

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.

### 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.

---

Note 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
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.

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 \(k = \lceil \lg |\Gamma| \rceil\) 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 $2^k$ counters. To reduce to just two counters, use Goedel numbering [Gö31 §V]: pick $2^k$ distinct primes $p_1, p_2, \ldots, p_{2^k}$ and encode the vector $\langle X_1, \ldots, X_{2^k} \rangle$ as $Y = \prod_{i=1}^{2^k} p_i^{X_i}$. 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 $2^k$ counters by itself makes each step cost as much as $O(k s^T)$ counter operations. Each counter operation in turn can take up to $O(p_{2^k}^T)$ steps on the two-counter machine, where $p_{2^k}$ 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 $O(k(s p_{2^k})^T)$, 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.
3.1.5 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 §3.1.3.2 that it is possible to decide the language \text{PALINDROME} = \{x \mid x = x^R\} using a Turing machine with a work tape in $O(n)$ steps. [[who do we cite for this?]] A classic lower bound shows that a one-tape machine must take $\Omega(n^2)$ steps to recognize this language. We give this argument 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 $q_1q_2\ldots q_k$ 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 \text{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.

[[yz lemma]]
[[polynomial blow-up, use crossing sequence lower bound on palindromes]]
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.\footnote{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 $\langle M \rangle$.}
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 $O(|M| + T \log |\Gamma|)$ 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 §6.2.

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 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 $\langle i \rangle \rightarrow \langle x \rangle$ where $\langle i \rangle$ and $\langle x \rangle$ 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 $\langle i \rangle$, then copy the corresponding $\langle x \rangle$ 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 $\langle i \rangle \rightarrow \langle x \rangle$ 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 hypothesis 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 hypothesis

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 hypothesis**, which says that

**Claim 3.3.1.** Any language that can be decided by a physically realizable computing device $M$ in time $T(n)$ can be decided by a Turing machine in time $O(T(n)^k)$ for some fixed $k$ 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 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.

---

9If this seems implausible, substitute one of the Zoo computers.
Chapter 4

Time and space complexity classes

A **complexity class** is a set of languages that are similarly hard in some sense. For example, the class \( \text{P} \) 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 \( \text{TIME}(f(n)) \) 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 \( \text{SPACE}(f(n)) \) 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 \( \text{TIME}(f(n)) \) and \( \text{SPACE}(f(n)) \) may depend on the specific details of the computational model we are using. For example, it is know 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 hypothesis 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.

\[ \text{[[[ EXPTIME, NP? ]]]} \]
Chapter 5

Nonterminism and NP

Historically, a **nondeterministic Turing machine** was defined as having the somewhat magical ability to split into two copies, each given a different bit telling it what to do next. These copies could 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 **hit** 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 \( x \not\in L \), 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 \( x \not\in L \) 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 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 \( w \).

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 \textit{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 \( \text{P} = \text{NP} \).

### 5.1 Examples of problems in 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 \textsc{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 \textsc{Independent Set} of size \( k \)? Have \( \exists w \) guess the list of nodes
CHAPTER 5. NONTERMINISM AND NP

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 \( \text{NP} \)-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 \( L \leq_{\text{P}} L' \).

The idea behind writing this 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 \( \text{P} \) 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 \( f \).

A language \( L \) is \( \text{NP} \)-hard if \( L' \leq_{\text{P}} L \) for any language \( L' \) in \( \text{NP} \). A language \( L \) is \( \text{NP} \)-complete if it is both in \( \text{NP} \) and \( \text{NP} \)-hard. The \( \text{NP} \)-complete languages are the hardest languages in \( \text{NP} \), in the sense that if we can recognize any \( \text{NP} \)-complete language in polynomial time, then \( \text{P} = \text{NP} \).

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

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

Alternatively, if we believe that \( \text{P} \neq \text{NP} \), then this immediately tells us that any \( \text{NP} \)-complete language (more generally, any \( \text{NP} \)-hard language) will not have a polynomial-time algorithm. Even if we aren’t sure if \( \text{P} \neq \text{NP} \), we still know that we have no examples so far of a polynomial-time algorithm.
for any problem that is \( NP \)-hard. So proving that a particular problem is \( NP \)-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 \( NP \)-complete problem. The **Cook-Levin theorem** gives one such problem, called **3SAT**. This is the problem of testing whether there exists a satisfying assignment to a Boolean formula in **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 \( NP \) into such a 3CNF formula.

[[[ state and prove it ]]]

5.4 More \( NP \)-complete problems

[[[ GRAPH 3-COLORABILITY, INDEPENDENT SET, CLIQUE, 1-OF-3 SAT, maybe HAMILTONIAN PATH ]]]
Chapter 6

Diagonalization

[[[ mention Cantor and/or Gödel? ]]]

6.1 Uncomputability

[[[ Halting problem, maybe Rice’s Theorem ]]]

6.2 Hierarchy theorems

[[[ time and space hierarchy theorems ]]]
Chapter 7

Relativatization

[[[ Baker-Gill-Solovay ]]]
Chapter 8

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. ]]]
Chapter 9

L vs NL

[space complexity in general, Savitch’s theorem, NL-complete problems, Immerman-Szelepcsényi and the logspace hierarchy]
Chapter 10

PSPACE

[[[ TQBF ]]]
Chapter 11

The polynomial-time hierarchy

[[[ ATMs, $\Sigma^P_k$ and $\Pi^P_k$, more descriptive complexity? ]]]
Chapter 12

Randomized classes

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

Circuits

[[[ circuit models, Karp-Lipton, P/poly and Adleman’s theorem, AC0 and NC1, AC0 lower bounds, Barrington’s theorem ]]]
Chapter 14

Natural proofs

[[[ Razborov-Rudich ]]]
Chapter 15

Toda’s Theorem

[[[ do it ]]]
Chapter 16

Interactive proofs
Chapter 17

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 L\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.

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 \( f(n) = o(h(n)) \).

A.1.4 Vector balancing

The input to VECTOR BALANCING consists of \( n \) 0–1 vectors \( v_1, \ldots, v_n \)
of dimension \( m \) each. A positive instance is a sequence of vectors for which
there exist \( \pm 1 \) values \( a_i \) such that \( \sum_{i=1}^{n} a_i v_i = 0 \).

For example, the instance \( \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle \) is a positive instance \( (a_1 =
\begin{equation}
a_2 = -a_3 \end{equation} \); but \( (1, 0) \), \( (1, 1) \) is a negative instance, because that 1 in the
second coordinate is going nowhere no matter what we do.

1. Show that VECTOR BALANCING is in \( \text{NP} \).

2. Show that VECTOR BALANCING is in fact \( \text{NP} \)-complete, by giving
a polynomial-time many-one reduction from some known \( \text{NP} \)-hard
problem.

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

[[[ To be announced. ]]]
A.3 Assignment 3: due Wednesday, 2017-03-01 at 23:00

[[[ To be announced. ]]]

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

[[[ To be announced. ]]]

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


Index

NP-complete, 26
NP-hard, 26
3SAT, 27
alphabet, 4 5
binary, 4
branch, 24
Church-Turing hypothesis, 5 21
extended, 21 23
class
complexity, 4
CNF, 27
complex systems, 1
complexity
time, 7
complexity class, 4 4 22
complexity theory, 1
computational complexity theory, 1
configuration, 6
conjunctive normal form, 27
constructible
space-, 22
time-, 22
Cook-Levin theorem, 27
course staff, viii
decide, 4
decision problem, 3
extended Church-Turing hypothesis, 5 21 23
feasible, 23
finite-state control, 5
finite-state transducer, 40
halt, 6
halting state, 6
heads, 5
hit, 24
hypothesis
Church-Turing, 21
extended, 21
independence
representation, 4
instructor, viii
language, 4
machine
Turing, 5
universal, 18
many-one reduction
polynomial-time, 26
nondeterministic polynomial time, 25
nondeterministic Turing machine, 24
polynomial time, 22
nondeterministic, 25
polynomial-time many-one reduction, 26
problem
decision, 3
INDEX

RAM, 19
random access machine, 19
reduction
    polynomial-time many-one, 26
representation independence, 4

sequence
    crossing, 17
size, 7
space complexity, 7
space-constructible, 22
staff, viii
state, 6
    halting, 6
supertask, 5
tapes, 5
time
    polynomial
        nondeterministic, 25
time complexity, 7
time-constructible, 22
TM, 5
transducer
    finite-state, 40
transition function, 6
Turing machine, 5
    nondeterministic, 24
    universal, 18
unary, 4
Undergraduate Learning Assistant, viii
universal, 18
universal Turing machine, 18
witness, 24