INTRODUCTION TO THEORETICAL COMPUTER SCIENCE
Contents

Preface                 21

Mathematical Background 27

1 Introduction         55

2 Computation and Representation 71

3 Defining computation 89

4 Syntactic sugar, and computing every function 111

5 Code as data, data as code 127

6 Physical implementations of NAND programs 141

7 Loops and infinity   161

8 Equivalent models of computation 179
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Is every function computable?</td>
<td>203</td>
</tr>
<tr>
<td>10</td>
<td>Is every theorem provable?</td>
<td>219</td>
</tr>
<tr>
<td>11</td>
<td>Efficient computation</td>
<td>233</td>
</tr>
<tr>
<td>12</td>
<td>Modeling running time</td>
<td>249</td>
</tr>
<tr>
<td>13</td>
<td>NP and NP completeness</td>
<td>261</td>
</tr>
<tr>
<td>14</td>
<td>The Cook-Levin Theorem</td>
<td>273</td>
</tr>
<tr>
<td>15</td>
<td>Advanced hardness reductions (advanced lecture)</td>
<td>279</td>
</tr>
<tr>
<td>16</td>
<td>What if P equals NP?</td>
<td>281</td>
</tr>
<tr>
<td>17</td>
<td>Probability Theory 101</td>
<td>293</td>
</tr>
<tr>
<td>18</td>
<td>Probabilistic computation</td>
<td>307</td>
</tr>
<tr>
<td>19</td>
<td>Modeling randomized computation</td>
<td>317</td>
</tr>
<tr>
<td>20</td>
<td>Hardnesss vs. Randomness</td>
<td>327</td>
</tr>
<tr>
<td>21</td>
<td>Derandomization from worst-case assumptions (advanced lecture)</td>
<td>329</td>
</tr>
<tr>
<td>22</td>
<td>Cryptography</td>
<td>331</td>
</tr>
</tbody>
</table>
Contents (detailed)

Preface  21
  0.1  To the student  23
     0.1.1  Is this effort worth it?  23
  0.2  To potential instructors  24
  0.3  Acknowledgements  25

Mathematical Background  27
  0.4  A mathematician’s apology  27
  0.5  A quick overview of mathematical prerequisites  28
  0.6  Basic discrete math objects  29
     0.6.1  Sets  29
     0.6.2  Special sets  31
     0.6.3  Functions  32
     0.6.4  Graphs  33
     0.6.5  Logic operators and quantifiers.  36
     0.6.6  Quantifiers for summations and products  36
     0.6.7  Parsing formulas: bound and free variables  37
     0.6.8  Asymptotics and big-Oh notation  38
     0.6.9  Some “rules of thumbs” for big Oh notation  39
  0.7  Proofs  40
     0.7.1  Proofs and programs  41
  0.8  Extended example: graph connectivity  41
     0.8.1  Mathematical induction  43
     0.8.2  Proving the theorem by induction  43
     0.8.3  Writing down the proof  46
  0.9  Proof writing style  49
     0.9.1  Patterns in proofs  49
  0.10  Non-standard notation  51
8 Equivalent models of computation 179

8.1 Turing machines 180
8.2 Turing Machines and NAND++ programs 182
8.2.1 Simulating Turing machines with NAND++ programs 182
8.2.2 NAND++ Snapshots, Transition, and Traces 184
8.2.3 Simulating NAND++ programs with Turing machines 186
8.2.4 Advanced note: polynomial equivalence 186
8.3 "Turing Completeness" and other Computational models 187
8.3.1 RAM Machines 187
8.3.2 Imperative languages 188
8.4 Lambda calculus and functional programming languages 188
8.4.1 The “basic” λ calculus objects 190
8.4.2 How basic is “basic”? 192
8.4.3 List processing and recursion without recursion 193
8.4.4 The Y combinator 194
8.5 Other models 196
8.5.1 Parallel algorithms and cloud computing 197
8.5.2 Game of life, tiling and cellular automata 197
8.6 Our models vs standard texts 198
8.7 The Church-Turing Thesis 199
8.8 Turing completeness as a bug 200
8.9 Lecture summary 201
8.10 Exercises 201
8.11 Bibliographical notes 202
9 Is every function computable? 203
  9.1 The Halting problem 205
  9.1.1 Is the Halting problem really hard? 206
  9.1.2 Reductions 208
  9.2 Impossibility of general software verification 209
  9.2.1 Rice's Theorem 211
  9.3 Restricted vs unrestricted computational models 212
  9.3.1 Regular expressions 212
  9.3.2 Context free grammars 215
  9.4 Uncountable sets 216
  9.5 Lecture summary 217
  9.6 Exercises 218
  9.7 Bibliographical notes 218
  9.8 Further explorations 218
  9.9 Acknowledgements 218

10 Is every theorem provable? 219
  10.1 Unsolvability of Diophantine equations 219
  10.1.1 "Baby" MRDP Theorem: hardness of quantified Diophantine equations 221
  10.2 Proving the unsolvability of quantified integer statements 223
  10.2.1 Quantified mixed statements and computation traces 223
  10.2.2 "Unraveling" NAND++ programs and quantified mixed integer statements 224
  10.2.3 Completing the proof 226
  10.2.4 Reducing mixed statements to integer statements 226
  10.3 Hilbert’s Program and Gödel’s Incompleteness Theorem 228
  10.4 Lecture summary 230
  10.5 Exercises 231
  10.6 Bibliographical notes 231
  10.7 Further explorations 231
  10.8 Acknowledgements 231

11 Efficient computation 233
  11.1 Some computational problems 234
  11.1.1 Finding the shortest path in a graph 236
### 11.1 Finding the longest path in a graph
- Page 236

### 11.2 Finding the minimum cut in a graph
- Page 236

### 11.3 Finding the maximum cut in a graph
- Page 238

### 11.4 A note on convexity
- Page 238

### 11.5 The 2SAT problem
- Page 240

### 11.6 The 3SAT problem
- Page 240

### 11.7 Solving linear equations
- Page 241

### 11.8 Solving quadratic equations
- Page 241

### 11.9 The permanent (mod 2) problem
- Page 242

### 11.10 The permanent (mod 3) problem
- Page 243

### 11.11 Finding a zero-sum equilibrium
- Page 243

### 11.12 Finding a Nash equilibrium
- Page 243

### 11.13 Primality testing
- Page 244

### 11.14 Integer factoring
- Page 244

### 11.15 Our current knowledge
- Page 244

### 11.16 Lecture summary
- Page 245

### 11.17 Exercises
- Page 246

### 11.18 Bibliographical notes
- Page 246

### 11.19 Further explorations
- Page 247

### 11.20 Acknowledgements
- Page 247

### 12 Modeling running time
- Page 249

#### 12.1 NAND« vs NAND++
- Page 251

#### 12.2 Efficient universal machine: a NAND« interpreter in NAND«
- Page 252

#### 12.3 Time hierarchy theorem
- Page 253

#### 12.3.1 Note: Oh-tilde or a “Baker’s Oh”
- Page 254

#### 12.4 Simulating NAND« or NAND++ programs with NAND programs
- Page 256

#### 12.5 Simulating NAND with NAND++?
- Page 257

#### 12.5.1 Uniform vs. Nonuniform computation: A recap
- Page 258

#### 12.6 Extended Church-Turing Thesis
- Page 258

#### 12.7 Lecture summary
- Page 259

#### 12.8 Exercises
- Page 259

#### 12.9 Bibliographical notes
- Page 260

#### 12.10 Further explorations
- Page 260

#### 12.11 Acknowledgements
- Page 260

### 13 NP and NP completeness
- Page 261

#### 13.1 Decision problems
- Page 262
16.3 What does all of this imply? 285
16.4 Can P ≠ NP be neither true nor false? 287
16.5 Is P = NP “in practice”? 288
16.6 What if P ≠ NP? 289
16.7 Lecture summary 290
16.8 Exercises 290
16.9 Bibliographical notes 290
16.11 Further explorations 290
16.11 Acknowledgements 291

17  Probability Theory 101 293
17.1 Random coins 293
17.1.1 Random variables 295
17.1.2 More general sample spaces. 296
17.2 Correlations and independence 296
17.2.1 Independent random variables 298
17.2.2 Collections of independent random variables. 299
17.3 Concentration 299
17.4 Chebyshev’s Inequality 301
17.5 The Chernoff bound 302
17.6 Lecture summary 303
17.7 Exercises 303
17.8 Bibliographical notes 305
17.9 Further explorations 305
17.10 Acknowledgements 305

18  Probabilistic computation 307
18.1 Finding approximately good maximum cuts. 308
18.1.1 Amplification 309
18.1.2 What does this mean? 309
18.1.3 Solving SAT through randomization 310
18.1.4 Bipartite matching. 312
18.2 Lecture summary 315
18.3 Exercises 315
18.4 Bibliographical notes 316
18.5 Further explorations 316
18.6 Acknowledgements 316
24 Compression, coding, and information 335
24.1 Lecture summary 335
24.2 Exercises 335
24.3 Bibliographical notes 335
24.4 Further explorations 335
24.5 Acknowledgements 335

25 Space bounded computation 337
25.1 Lecture summary 337
25.2 Exercises 337
25.3 Bibliographical notes 337
25.4 Further explorations 337
25.5 Acknowledgements 337

26 Streaming, sketching, and automata 339
26.1 Lecture summary 339
26.2 Exercises 339
26.3 Bibliographical notes 339
26.4 Further explorations 339
26.5 Acknowledgements 339

27 Proofs and algorithms 341
27.1 Lecture summary 341
27.2 Exercises 341
27.3 Bibliographical notes 341
27.4 Further explorations 341
27.5 Acknowledgements 341

28 Interactive proofs (advanced lecture) 343
28.1 Lecture summary 343
28.2 Exercises 343
31.9 The “standard library” 360
31.9.1 Syntactic sugar for NAND 360
31.9.2 Encoding of integers, strings, lists. 361
31.9.3 Syntactic sugar for NAND++ / NAND* 363

Bibliography 365
Preface

“We make ourselves no promises, but we cherish the hope that the unobstructed pursuit of useless knowledge will prove to have consequences in the future as in the past” … “An institution which sets free successive generations of human souls is amply justified whether or not this graduate or that makes a so-called useful contribution to human knowledge. A poem, a symphony, a painting, a mathematical truth, a new scientific fact, all bear in themselves all the justification that universities, colleges, and institutes of research need or require”, Abraham Flexner, The Usefulness of Useless Knowledge, 1939.

These are lecture notes for an undergraduate introductory course on Theoretical Computer Science. The educational goals of this course are to convey the following:

- That computation but arises in a variety of natural and manmade systems, and not only in modern silicon-based computers.

- Similarly, beyond being an extremely important tool, computation also serves as a useful lens to describe natural, physical, mathematical and even social concepts.

- The notion of universality of many different computational models, and the related notion of the duality between code and data.

- The idea that one can precisely define a mathematical model of computation, and then use that to prove (or sometimes only conjecture) lower bounds and impossibility results.

- Some of the surprising results and discoveries in modern theoretical computer science, including the prevalence of NP completeness, the power of interaction, the power of randomness on one hand and the possibility of derandomization on the other, the ability to use hardness “for good” in cryptography, and the fascinating
possibility of quantum computing.

I hope that following this course, students would be able to recognize computation, with both its power and pitfalls, as it arises in various settings, including seemingly “static” content or “restricted” formalisms such as macros and scripts. They should be able to follow through the logic of proofs about computation, including the pervasive notion of a reduction and understanding the subtle but crucial “self referential” proofs (such as proofs involving programs that use their own code as input). Students should understand the concept that some problems are intractable, and have the ability to recognize the potential for intractability when they are faced with a new problem. While this course only touches on cryptography, students should understand the basic idea of how computational hardness can be utilized for cryptographic purposes. But more than any specific skill, this course aims to introduce students to a new way of thinking of computation as an object in its own right, and illustrate how this new way of thinking leads to far reaching insights and applications.

My aim in writing these notes is to try to convey these concepts in the simplest possible way and try to make sure that the formal notation and model help elucidate, rather than obscure, the main ideas. I also tried to take advantage of modern students’ familiarity (or at least interest!) in programming, and hence use (highly simplified) programming languages as the main model of computation, as opposed to automata or Turing machines. That said, this course does not really assume fluency with any particular programming language, but more a familiarity with the general notion of programming. We will use programming metaphors and idioms, occasionally mentioning concrete languages such as Python, C, or Lisp, but students should be able to follow these descriptions even if they are not familiar with these languages.

Proofs in this course, including the existence of a universal Turing Machine, the fact that every finite function can be computed by some circuit, the Cook-Levin theorem, and many others, are often constructive and algorithmic, in the sense that they ultimately involve transforming one program to another. While the code of these transformations (like any code) is not always easy to read, and the ideas behind the proofs can be grasped without seeing it, I do think that having access to the code, and the ability to play around with it and see how it acts on various programs, can make these theorems more concrete for the students. To that end, an accompanying website (which is still work in progress) allows executing programs in the various computational models we define, as well as see constructive
proofs of some of the theorems.

0.1 To the student

This course can be fairly challenging, mainly because it brings together a variety of ideas and techniques in the study of computation. There are quite a few technical hurdles to master, whether it is following the diagonalization argument in proving the Halting Problem is undecidable, combinatorial gadgets in NP-completeness reductions, analyzing probabilistic algorithms, or arguing about the adversary to prove security of cryptographic primitives.

The best way to engage with the material is to read these notes actively. While reading, I encourage you to stop and think about the following:

• When I state a theorem, stop and try to think of how you would prove it yourself before reading the proof in the notes. You will be amazed by how much you can understand a proof better even after only 5 minutes of attempting it yourself.

• When reading a definition, make sure that you understand what the definition means, and you can think of natural examples of objects that satisfy it and objects that don’t. Try to think of the motivation behind the definition, and there other natural ways to formalize the same concept.

• At any point in the text, try to think what are the natural questions that arise, and see whether or not they are answered in the following.

0.1.1 Is this effort worth it?

A traditional justification for such a course is that you might encounter these concepts in your career. Perhaps you will come across a hard problem and realize it is NP complete, or find a need to use what you learned about regular expressions. This might very well be true, but the main benefit of this course is not in teaching you any practical tool or technique, but rather in giving you a different way of thinking: an ability to recognize computation even when it might not be obvious that it occurs, a way to model computational tasks and questions, and to reason about them. But, regardless of any use you will derive from it, I believe this course is important because it teaches concepts that are both beautiful and fundamental.
The role that energy and matter played in the 20th century is played in the 21st by computation and information, not just as tools for our technology and economy, but also as the basic building blocks we use to understand the world. This course will give you a taste of some of the theory behind those, and hopefully spark your curiosity to study more.

0.2 To potential instructors

These lecture notes are written for my course at Harvard, but I hope that other lecturers will find them useful as well. To some extent, these notes are similar in content to “Theory of Computation” or “Great Ideas” courses such as those at CMU or MIT. There are however some differences, with the most significant being:

- I do not start with finite automata as the basic computational model, but rather with straight-line programs in an extremely simple programming language (or, equivalently, Boolean circuits). Automata are discussed later in the course in the context of space-bounded computation.

- Instead of Turing machines, I use an equivalent model obtained by extending the programming language above to include loops. I also introduce another extension of the programming language that allows pointers, and hence is essentially equivalent to the standard RAM machine model used (implicitly) in algorithms courses.

A much more minor notational difference is that rather than talking about languages (i.e., subsets $L \subseteq \{0,1\}^*$), I talk about Boolean functions (i.e., functions $f : \{0,1\}^* \rightarrow \{0,1\}$). These are of course equivalent, but the function notation extends more naturally to more general computational tasks.

Reducing the time dedicated to automata (and eliminating context free languages) allows to spend more time on topics that I believe that a modern course in the theory of computing needs to touch upon, including randomness and computation, the interaction of algorithms with society (with issues such as incentives, privacy, fairness), the basics of information theory, cryptography, and quantum computing.

My intention was to write these notes in a level of detail that will enable their use for self-study, and in particular for students to be able to read the notes before each lecture. This can help students keep
up with what is a fairly ambitious and fast-paced schedule.

### 0.3 Acknowledgements

These lecture notes are constantly evolving, and I am getting input from several people, for which I am deeply grateful. Thanks to Michele Amoretti, Jaroslaw Blasiok, Chi-Ning Chou, Juan Esteller, Ondřej Lengál, Alex Lombardi, Thomas Orton, Juan Perdomo, Salil Vadhan, Ryan Williams and Jessica Zhu for helpful feedback. I will keep adding names here as I get more comments. If you have any comments or suggestions, please do post them on the GitHub repository [https://github.com/boazbk/tcs](https://github.com/boazbk/tcs).

Thanks to Anil Ada, Venkat Guruswami, and Ryan O'Donnell for helpful tips from their experience in teaching CMU 15-251.

Thanks to Juan Esteller for his work on implementing the NAND* languages.

Thanks to David Steurer for writing the scripts (originally produce for our joint notes on the sum of squares algorithm) that I am using to produce these notes, (themselves based on several other packages, including pandoc, LaTeX, and the Tuft LaTeX and CSS packages).
Mathematical Background

“When you have mastered numbers, you will in fact no longer be reading numbers, any more than you read words when reading books. You will be reading meanings.”, W. E. B. Du Bois

In this chapter, we review some of the mathematical concepts that we will use in this course. Most of these are not very complicated, but do require some practice and exercise to get comfortable with. If you have not previously encountered some of these concepts, there are several excellent freely-available resources online for them. In particular, the CS 121 webpage contains a program for self study of all the needed notions using the lecture notes, videos, and assignments of MIT course 6.042j Mathematics for Computer science. (The MIT lecture notes are also used by Harvard CS 20.)

0.4 A mathematician’s apology

Before explaining the math background, perhaps I should explain why does this course is so “mathematically heavy”. After all, this is supposed to be a course about computation; one might think we should be talking mostly about programs, rather than more “mathematical” objects such as sets, functions, and graphs, and doing more coding on an actual computer than writing mathematical proofs with pen and paper. So, why are we doing so much math in this course? Is it just some form of hazing? Perhaps a revenge of the “math nerds” against the “hackers”?

At the end of the day, mathematics is simply a language for modelling concepts in a precise and unambiguous way. In this course, we will be mostly interested in the concept of computation. For example, we will look at questions such as “is there an efficient algorithm to find the prime factors of a given integer?”.¹ To even phrase such a question,

¹ Actually, scientists currently do not know the answer to this question, but we will see that settling it in either direction has very interesting applications touching on areas as far apart as Internet security and quantum mechanics.
we need to give a precise definition of the notion of an algorithm, and of what it means for an algorithm to be efficient. Also, if the answer to this or similar questions turns out to be negative, then this cannot be shown by simply writing and executing some code. After all, there is no empirical experiment that will prove the non existence of an algorithm. Thus, our only way to show this type of negative results is to use mathematical proofs. So you can see why our main tools in this course will be mathematical proofs and definitions.

0.5 A quick overview of mathematical prerequisites

The main notions we will use in this course are the following:

- **Proofs:** First and foremost, this course will involve a heavy dose of formal mathematical reasoning, which includes mathematical definitions, statements, and proofs.

- **Sets:** Including notation such as membership (∈), containment (⊆), and set operations such as union, intersection, set difference and Cartesian product (∪, ∩, \ and ×).

- **Functions:** Including the notions of the domain and range of a function, properties such being one-to-one or onto functions, as well as partial functions (that, unlike standard or “total” functions, are not necessarily defined on all elements of their domain).

- **Logical operations:** The operations AND, OR, and NOT (∧, ∨, ¬) and the quantifiers “exists” and “forall” (∃, ∀).

- **Tuples and strings:** The notation Σ^k and Σ^* where Σ is some finite set which is called the alphabet (quite often Σ = {0, 1}).

- **Basic combinatorics:** Notions such as \( \binom{n}{k} \) (the number of k-sized subset of a set of size n).

- **Graphs:** Undirected and directed graphs, connectivity, paths, and cycles.

- **Big Oh notation:** O, o, Ω, ω, Θ notation for analyzing asymptotics of functions.

- **Discrete probability:** Later on in this course we will use probability theory, and specifically probability over finite samples spaces such as tossing n coins. We will only use probability theory in the second half of this course, and will review it beforehand. However, probabilistic reasoning is a subtle (and extremely useful!) skill, and it’s always good to start early in acquiring it.
While I highly recommend the resources linked above, in the rest of this section we briefly review these notions. This is partially to remind the reader and reinforce material that might not be fresh in your mind, and partially to introduce our notation and conventions which might occasionally differ from those you’ve encountered before.

### 0.6 Basic discrete math objects

We now quickly review some of the mathematical objects and definitions we in this course.

#### 0.6.1 Sets

A set is an unordered collection of objects. For example, when we write $S = \{2, 4, 7\}$, we mean that $S$ denotes the set that contains the numbers 2, 4, and 7. (We use the notation “$2 \in S$” to denote that 2 is an element of $S$.) Note that the set $\{2, 4, 7\}$ and $\{7, 4, 2\}$ are identical, since they contain the same elements. Also, a set either contains an element or does not contain it -there is no notion of containing it “twice”- and so we could even write the same set $S$ as $\{2, 2, 4, 7\}$ (though that would be a little weird). The cardinality of a finite set $S$, denoted by $|S|$, is the number of distinct elements it contains. So, in the example above, $|S| = 3$. A set $S$ is a subset of a set $T$, denoted by $S \subseteq T$, if every element of $S$ is also an element of $T$. (We can also describe this by saying that $T$ is a superset of $S$.) For example, $\{2, 7\} \subseteq \{2, 4, 7\}$. The set that contains no elements is known as the empty set and it is denoted by $\emptyset$.

We can define sets by either listing all their elements or by writing down a rule that they satisfy such as

$$EVEN = \{x : x = 2y \text{ for some non-negative integer } y\}.$$  \hspace{1cm} (1)

Of course there is more than one way to write the same set, and often we will use intuitive notation listing a few examples that illustrate the rule. For example, we can also define $EVEN$ as

$$EVEN = \{0, 2, 4, \ldots\}.$$  \hspace{1cm} (2)

Note that a set can be either finite (such as the set $\{2, 4, 7\}$) or infinite (such as the set $EVEN$). Also, the elements of a set don’t have
to be numbers. We can talk about the sets such as the set \{a,e,i,o,u\} of all the vowels in the English language, or the set \{New York, Los Angeles, Chicago, Houston, Philadelphia, Phoenix, San Antonio, San Diego, Dallas\} of all cities in the U.S. with population more than one million per the 2010 census. A set can even have other sets as elements, such as the set \{\emptyset, \{1,2\}, \{2,3\}, \{1,3\}\} of all even-sized subsets of \{1,2,3\}.

**Operations on sets:** The union of two sets \(S, T\), denoted by \(S \cup T\), is the set that contains all elements that are either in \(S\) or in \(T\). The intersection of \(S\) and \(T\), denoted by \(S \cap T\), is the set of elements that are both in \(S\) and \(T\). The set difference of \(S\) and \(T\), denoted by \(S \setminus T\) (and in some texts also by \(S - T\)), is the set of elements that are in \(S\) but not in \(T\).

**Tuples, lists, strings, sequences:** A tuple is an ordered collection of items. For example \((1,5,2,1)\) is a tuple with four elements (also known as a 4-tuple or quadruple). Since order matters, this is not the same tuple as the 4-tuple \((1,1,5,2)\) or the 3-tuple \((1,5,2)\). A 2-tuple is also known as a pair. We use the terms tuples and lists interchangeably. A tuple where every element comes from some finite set \(\Sigma\) (such as \(\{0,1\}\)) is also known as a string. Analogously to sets, we denote the length of a tuple \(T\) by \(|T|\). Just like sets, we can also think of an infinite analogs of tuples, such as the ordered collection \((1,2,4,9,\ldots)\) of all perfect squares. Infinite ordered collections are known as sequences; we might sometimes use the term “infinite sequence” to emphasize this, and use “finite sequence” as a synonym for a tuple.3

**Cartesian product:** If \(S\) and \(T\) are sets, then their Cartesian product, denoted by \(S \times T\), is the set of all ordered pairs \((s,t)\) where \(s \in S\) and \(t \in T\). For example, if \(S = \{1,2,3\}\) and \(T = \{10,12\}\), then \(S \times T\) contains the 6 elements \((1,10), (2,10), (3,10), (1,12), (2,12), (3,12)\). Similarly if \(S,T,U\) are sets then \(S \times T \times U\) is the set of all ordered triples \((s,t,u)\) where \(s \in S, t \in T, \text{ and } u \in U\). More generally, for every positive integer \(n\) and sets \(S_0,\ldots,S_{n-1}\), we denote by \(S_0 \times S_1 \times \cdots \times S_{n-1}\) the set of ordered \(n\)-tuples \((s_0,\ldots,s_{n-1})\) where \(s_i \in S_i\) for every \(i \in \{0,\ldots,n-1\}\).

For every set \(S\), we denote the set \(S \times S\) by \(S^2\), \(S \times S \times S\) by \(S^3\), \(S \times S \times S \times S\) by \(S^4\), and so on and so forth.
0.6.2 Special sets

There are several sets that we will use in this course time and again, and so find it useful to introduce explicit notation for them. For starters we define

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$

(3)

to be the set of all natural numbers, i.e., non-negative integers. For any natural number \(n\), we define the set \([n]\) as \(\{0, \ldots, n - 1\} = \{k \in \mathbb{N} : k < n\}\).\(^4\)

We will also occasionally use the set \(\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}\) of (negative and non-negative) whole numbers\(^5\) as well as the set \(\mathbb{R}\) of real numbers. (This is the set that includes not just the whole numbers, but also fractional and even irrational numbers; e.g., \(\mathbb{R}\) contains numbers such as \(+0.5\), \(-\pi\), etc.) We denote by \(\mathbb{R}_+\) the set \(\{x \in \mathbb{R} : x > 0\}\) of positive real numbers. This set is sometimes also denoted as \((0, \infty)\).

**Strings:** Another set we will use time and again is

$$\{0, 1\}^n = \{(x_0, \ldots, x_{n-1}) : x_0, \ldots, x_{n-1} \in \{0, 1\}\}$$

(4)

which is the set of all \(n\)-length binary strings for some natural number \(n\). That is \(\{0, 1\}^n\) is the set of all \(n\)-tuples of zeroes and ones. This is consistent with our notation above: \(\{0, 1\}^2\) is the Cartesian product \(\{0, 1\} \times \{0, 1\}\), \(\{0, 1\}^3\) is the product \(\{0, 1\} \times \{0, 1\} \times \{0, 1\}\) and so on.

We will write the string \((x_0, x_1, \ldots, x_{n-1})\) as simply \(x_0 x_1 \cdots x_{n-1}\) and so for example

$$\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}.$$ 

(5)

For every string \(x \in \{0, 1\}^n\) and \(i \in [n]\), we write \(x_i\) for the \(i^{th}\) coordinate of \(x\). If \(x\) and \(y\) are strings, then \(xy\) denotes their concatenation. That is, if \(x \in \{0, 1\}^n\) and \(y \in \{0, 1\}^m\), then \(xy\) is equal to the string \(z \in \{0, 1\}^{n+m}\) such that for \(i \in [n]\), \(z_i = x_i\) and for \(i \in [n+1, n+m]\), \(z_i = y_{i-n}\).

We will also often talk about the set of binary strings of all lengths,
which is

\[ \{0,1\}^* = \{(x_0, \ldots, x_{n-1}) : n \in \mathbb{N}, x_0, \ldots, x_{n-1} \in \{0,1\}\}. \quad (6) \]

Another way to write this set is as

\[ \{0,1\}^* = \{0,1\}^0 \cup \{0,1\}^1 \cup \{0,1\}^2 \cup \ldots \quad (7) \]

or more concisely as

\[ \{0,1\}^* = \bigcup_{n \in \mathbb{N}} \{0,1\}^n. \quad (8) \]

The set \( \{0,1\}^* \) contains also the “string of length 0” or “the empty string”, which we will denote by “".\footnote{We follow programming languages in this notation; other texts sometimes use \( \epsilon \) or \( \lambda \) to denote the empty string. However, this doesn’t matter much since we will rarely encounter this “edge case”.}

**Generalizing the star operation:** For every set \( \Sigma \), we define

\[ \Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n. \quad (9) \]

For example, if \( \Sigma = \{a, b, c, d, \ldots, z\} \) then \( \Sigma^* \) denotes the set of all finite length strings over the alphabet a-z.

**Concatenation:** The concatenation of two strings \( x \in \Sigma^n \) and \( y \in \Sigma^m \) is the \( (n + m) \)-length string \( xy \) obtained by writing \( y \) after \( x \). That is, \( (xy)_i \) equals \( x_i \) if \( i < n \) and equals \( y_{i-n} \) if \( n \leq i < n + m \).

### 0.6.3 Functions

If \( S \) and \( T \) are sets, a function \( F \) mapping \( S \) to \( T \), denoted by \( F : S \to T \), associates with every element \( x \in S \) an element \( F(x) \in T \). Just as with sets, we can write a function either by listing the table of all the values it gives for elements in \( S \) or using a rule. For example if \( S = \{0,1,2,3,4,5,6,7,8,9\} \) and \( T = \{0,1\} \). Then the function \( F \) defined as

\[ F(x) = (x \mod 2). \]

Is the same as defining \( F(x) = (x \mod 2) \). If \( F : S \to T \) satisfies that \( F(x) \neq F(y) \) for all \( x \neq y \) then we say that \( F \) is one-to-one.

If \( F \) satisfies that for every \( y \in T \) there is some \( x \) such that \( F(x) = y \) then we say that \( F \) is onto. A one-to-one and onto function is called a bijection, and when \( S = T \) it is also known as a permutation. If \( F : S \to T \) is a bijection then for every \( y \in T \) there is a unique \( x \in S \) s.t. \( F(x) = y \). We denote this value \( x \) by \( F^{-1}(y) \). Note that \( F^{-1} \) is itself a bijection from \( T \) to \( S \) (can you see why?).
Giving a bijection between two sets is often a good way to show they have the same size. In fact, the standard mathematical definition of the notion that “$S$ and $T$ have the same cardinality” is that there exists a bijection $f: S \to T$. In particular, the cardinality of a set $S$ is defined $n$ if there is a bijection from $S$ to the set $\{0, \ldots, n-1\}$. As we will see later in this course, this is a definition that can generalizes to defining the cardinality of infinite sets.

**Partial functions**: We will sometimes be interested in partial functions from $S$ to $T$. This is a function $F$ that is not necessarily defined on every element of $S$. For example, the function $F(x) = \sqrt{x}$ is only defined on non-negative real numbers. When we want to distinguish between partial functions and standard (i.e., non-partial) functions, we will call the latter total functions. (Note that the set of partial functions is a proper superset of the set of total functions; i.e., a partial function is allowed to be defined on all its input elements.) Also, when we want to emphasize that a function $f$ from $A$ to $B$ might not be total, we will write $f : A \to_p B$. We can think of a partial function $F$ from $S$ to $T$ also as a total function from $S$ to $T \cup \{\bot\}$ where $\bot$ is some special “failure symbol”, and so instead of saying that $F$ is undefined at $x$, we can say that $F(x) = \bot$.

### 0.6.4 Graphs

*Graphs* are ubiquitous in Computer Science, and many other fields as well. They are used to model a variety of data types including social networks, road networks, deep neural nets, gene interactions, correlations between observations, and a great many more. The formal definitions of graphs are below, but if you have not encountered them before then I urge you to read up on them in one of the sources linked above. Graphs come in two basic flavors: *undirected*
It is possible, and sometimes useful, to think of an undirected graph as simply a directed graph with the special property that for every pair \( u, v \) either both the edges \( \overleftarrow{uv} \) and \( \overrightarrow{uv} \) are present or neither of them is. However, in many settings there is a significant difference between undirected and directed graphs, and so it’s typically best to think of them as separate categories.

**Definition 0.1 — Undirected graphs.** An undirected graph \( G = (V, E) \) consists of a set \( V \) of vertices and a set \( E \) of edges. Every edge is a size two subset of \( V \). We say that two vertices \( u, v \in V \) are neighbors, denoted by \( u \sim v \), if the edge \( \{u, v\} \) is in \( E \).

Given this definition, we can define several other properties of graphs and their vertices. We define degree of \( u \) to be the number of neighbors \( v \) has. A path in the graph is a tuple \( (u_0, \ldots, u_k) \in V^k \), for some \( k > 0 \) such that \( u_{i+1} \) is a neighbor of \( u_i \) for every \( i \in [k] \). A simple path is a path \( (u_0, \ldots, u_k) \) where all the \( u_i \)'s are distinct. A cycle is a path \( (u_0, \ldots, u_k) \) where \( u_0 = u_k \). We say that two vertices \( u, v \in V \) are connected if either \( u = v \) or there is a path from \( (u_0, \ldots, u_k) \) where \( u_0 = u \) and \( u_k = v \). We say that the graph \( G \) is connected if every pair of vertices in it is connected.

Here are some basic facts about undirected graphs. We give some informal arguments below, but leave the full proofs as exercises. (The proofs can also be found in most basic texts on graph theory.)

**Lemma 0.1** In any undirected graph \( G = (V, E) \), the sum of the degrees of all vertices is equal to twice the number of edges.

**Lemma 0.1** can be shown by seeing that every edge \( \{u, v\} \) contributes twice to the sum of the degrees (once for \( u \) and the second time for \( v \)).

**Lemma 0.2** The connectivity relation is transitive, in the sense that if \( u \) is connected to \( v \), and \( v \) is connected to \( w \), then \( u \) is connected to \( w \).

**Lemma 0.2** can be shown by simply attaching a path of the form \( (u, u_1, u_2, \ldots, u_{k-1}, v) \) to a path of the form \( (v, u'_1, \ldots, u'_{l-1}, w) \) to obtain the path \( (u, u_1, \ldots, u_{k-1}, v, u'_1, \ldots, u'_{l-1}, w) \) that connects \( u \) to \( w \).
Lemma 0.3 For every undirected graph $G = (V, E)$ and connected pair $u, v$, the shortest path from $u$ to $v$ is simple. In particular, for every connected pair there exists a simple path that connects them.

Lemma 0.3 can be shown by “shortcutting” any non-simple path of the form $(u, u_1, \ldots, u_{i-1}, w, u_{i+1}, \ldots, u_{j-1}, w, u_{j+1}, \ldots, u_{k-1}, v)$ where the same vertex $w$ appears in both the $i$-th and $j$-position, to obtain the shorter path $(u, u_1, \ldots, u_{i-1}, w, u_{j+1}, \ldots, u_{k-1}, v)$.

If you haven’t seen these proofs before, it is indeed a great exercise to transform the above informal exercises into fully rigorous proofs.

Definition 0.2 — Directed graphs. A directed graph $G = (V, E)$ consists of a set $V$ and a set $E \subseteq V \times V$ of ordered pairs of $V$. We denote the edge $(u, v)$ also as $\overrightarrow{uv}$. If the edge $\overrightarrow{uv}$ is present in the graph then we say that $v$ is an out-neighbor of $u$ and $u$ is an in-neighbor of $v$.

A directed graph might contain both $\overrightarrow{uv}$ and $\overrightarrow{vu}$ in which case $u$ will be both an in-neighbor and an out-neighbor of $v$ and vice versa. The in-degree of $u$ is the number of in-neighbors it has, and the out-degree of $v$ is the number of out-neighbors it has. A path in the graph is a tuple $(u_0, \ldots, u_k) \in V^k$, for some $k > 0$ such that $u_{i+1}$ is an out-neighbor of $u_i$ for every $i \in [k]$. As in the undirected case, a simple path is a path $(u_0, \ldots, u_{k-1})$ where all the $u_i$’s are distinct and a cycle is a path $(u_0, \ldots, u_k)$ where $u_0 = u_k$. One type of directed graphs we often care about is directed acyclic graphs or DAGs, which, as their name implies, are directed graphs without any cycles.

The lemmas we mentioned above have analogs for directed graphs. We again leave the proofs (which are essentially identical to their undirected analogs) as exercises for the reader:

Lemma 0.4 In any directed graph $G = (V, E)$, the sum of the in-degrees is equal to the sum of the out-degrees, which is equal to the number of edges.

Lemma 0.5 In any directed graph $G$, if there is a path from $u$ to $v$ and a path from $v$ to $w$, then there is a path from $u$ to $w$.

Lemma 0.6 For every directed graph $G = (V, E)$ and a pair $u, v$ such that there is a path from $u$ to $v$, the shortest path from $u$ to $v$ is simple.
0.6.5 Logic operators and quantifiers.

If \( P \) and \( Q \) are some statements that can be true or false, then \( P \) AND \( Q \) (denoted as \( P \land Q \)) is the statement that is true if and only if both \( P \) and \( Q \) are true, and \( P \) OR \( Q \) (denoted as \( P \lor Q \)) is the statement that is true if and only if either \( P \) or \( Q \) is true. The negation of \( P \), denoted as \( \neg P \) or \( P' \), is the statement that is true if and only if \( P \) is false.

Suppose that \( P(x) \) is a statement that depends on some parameter \( x \) (also sometimes known as an unbound variable) in the sense that for every instantiation of \( x \) with a value from some set \( S \), \( P(x) \) is either true or false. For example, \( x > 7 \) is a statement that is not a priori true or false, but does become true or false whenever we instantiate \( x \) with some real number. In such case we denote by \( \forall x \in S P(x) \) the statement that is true if and only if \( P(x) \) is true for every \( x \in S \). We denote by \( \exists x \in S P(x) \) the statement that is true if and only if there exists some \( x \in S \) such that \( P(x) \) is true.

For example, the following is a formalization of the true statement that there exists a natural number \( n \) larger than 100 that is not divisible by 3:

\[
\exists n \in \mathbb{N} (n > 100) \land (\forall k \in \mathbb{N} k + k + k \neq n).
\] (10)

0.6.6 Quantifiers for summations and products

The following shorthands for summing up or taking products of several numbers are often convenient. If \( S = \{s_0, \ldots, s_{n-1}\} \) is a finite set and \( f : S \to \mathbb{R} \) is a function, then we write \( \sum_{x \in S} f(x) \) as shorthand
for
\[ f(s_0) + f(s_1) + f(s_2) + \ldots + f(s_{n-1}) , \quad (11) \]

and \( \prod_{x \in S} f(x) \) as shorthand for
\[ f(s_0) \cdot f(s_1) \cdot f(s_2) \cdot \ldots \cdot f(s_{n-1}) . \quad (12) \]

For example, the sum of the squares of all numbers from 1 to 100 can be written as
\[ \sum_{i \in \{1, \ldots, 100\}} i^2 . \quad (13) \]

Since summing up over intervals of integers is so common, there is a special notation for it, and for every two integers \( a \leq b \), \( \sum_{i=a}^{b} f(i) \) denotes \( \sum_{i \in S} f(i) \) where \( S = \{ x \in \mathbb{Z} : a \leq x \leq b \} \). Hence we can write the sum Eq. (13) as
\[ \sum_{i=1}^{100} i^2 . \quad (14) \]

0.6.7 Parsing formulas: bound and free variables

In mathematics as in code, we often have symbolic “variables” or “parameters”. It is important to be able to understand, given some formula, whether a given variable is \textit{bound} or \textit{free} in this formula. For example, in the following statement \( n \) is free but \( a, b \) are bound by the \( \exists \) quantifier:
\[ \exists_{a,b \in \mathbb{N}} (a \neq 1) \land (a \neq n) \land (n = a \times b) \quad (15) \]

Since \( n \) is free, it can be set to any value, and the truth of the statement Eq. (16) depends on the value of \( n \). For example, if \( n = 8 \) then Eq. (16) is true, but for \( n = 11 \) it is false. (Can you see why?)

The same issue appears when parsing code. For example, in the following snippet from the C++ programming language
\[ \text{for (int i=0 ; i<n ; i=i+1) { } printf("*"); } \]
the variable \( i \) is bound to the `for` operator but the variable \( n \) is free.

The main property of bound variables is that we can change them to a different name (as long as it doesn’t conflict with another used variable) without changing the meaning of the statement. Thus for example the statement

\[
\exists x,y \in \mathbb{N} (x \neq 1) \land (x \neq n) \land (n = x \times y)
\]  

is equivalent to Eq. (16) in the sense that it is true for exactly the same set of \( n \)'s. Similarly, the code

```c
for (int j=0 ; j<n ; j=j+1) {
    printf("*");
}
```

produces the same result.

### 0.6.8 Asymptotics and big-Oh notation

It is often very cumbersome to describe precisely quantities such as running time and is also not needed, since we are typically mostly interested in the “higher order terms”. That is, we want to understand the scaling behavior of the quantity as the input variable grows. For example, as far as running time goes, the difference between an \( n^5 \)-time algorithm and an \( n^2 \)-time one is much more significant than the difference between an \( 100n^2 + 10n \) time algorithm and an \( 10n^2 \)

For this purpose, Oh notation is extremely useful as a way to “declutter” our text and focus our attention on what really matters. For example, using Oh notation, we can say that both \( 100n^2 + 10n \) and \( 10n^2 \) are simply \( \Theta(n^2) \) (which informally means “the same up to constant factors”), while \( n^2 = o(n^5) \) (which informally means that \( n^2 \) is “much smaller than” \( n^5 \)).

Generally (though still informally), if \( F, G \) are two functions mapping natural numbers to non-negative reals, then “\( F = O(G) \)” means that \( F(n) \leq G(n) \) if we don’t care about constant factors, while “\( F = o(G) \)” means that \( F \) is much smaller than \( G \), in the sense that no matter by what constant factor we multiply \( F \), if we take \( n \) to be large enough then \( G \) will be bigger (for this reason, sometimes \( F = o(G) \) is written as \( F \ll G \)). We will write \( F = \Theta(G) \) if \( F = O(G) \) and \( G = O(F) \), which one can think of as saying that \( F \) is the same as \( G \) if we don’t care about constant factors. More formally, we define Big Oh notation as follows:
Definition 0.3 — Big Oh notation. For \( F, G : \mathbb{N} \to \mathbb{R}_+ \), we define \( F = O(G) \) if there exist numbers \( a, N_0 \in \mathbb{N} \) such that \( F(n) \leq a \cdot G(n) \) for every \( n > N_0 \). We define \( F = \Omega(G) \) if \( G = O(F) \).

We write \( F = o(G) \) if for every \( \epsilon > 0 \) there is some \( N_0 \) such that \( F(n) < \epsilon G(n) \) for every \( n > N_0 \). We write \( F = \omega(G) \) if \( G = o(F) \). We write \( F = \Theta(G) \) if \( F = O(G) \) and \( G = O(F) \).

We can also use the notion of limits to define big and little oh notation. You can verify that \( F = o(G) \) (or, equivalently, \( G = \omega(F) \)) if and only if \( \lim_{n \to \infty} \frac{F(n)}{G(n)} = 0 \). Similarly, if the limit \( \lim_{n \to \infty} \frac{F(n)}{G(n)} \) exists and is a finite number then \( F = O(G) \). If you are familiar with the notion of supremum, then you can verify that \( F = O(G) \) if and only if \( \limsup_{n \to \infty} \frac{F(n)}{G(n)} < \infty \).

Using the equality sign for Oh notation is extremely common, but is somewhat of a misnomer, since a statement such as \( F = O(G) \) really means that \( F \) is in the set \( \{ G' : \exists N_c \text{ s.t. } \forall n > N G'(n) \leq cG(n) \} \). For this reason, some texts write \( F \in O(G) \) instead of \( F = O(G) \). If anything, it would have made more sense use inequalities and write \( F \leq O(G) \) and \( F \geq \Omega(G) \), reserving equality for \( F = \Theta(G) \), but by now the equality notation is quite firmly entrenched. Nevertheless, you should remember that a statement such as \( F = O(G) \) means that \( F \) is “at most” \( G \) in some rough sense when we ignore constants, and a statement such as \( F = \Omega(G) \) means that \( F \) is “at least” \( G \) in the same rough sense.

It’s often convenient to use “anonymous functions” in the context of Oh notation, and also to emphasize the input parameter to the function. For example, when we write a statement such as \( F(n) = O(n^3) \), we mean that \( F = O(G) \) where \( G \) is the function defined by \( G(n) = n^3 \). Chapter 7 in Jim Apsnes’ notes on discrete math provides a good summary of Oh notation.

0.6.9 Some “rules of thumbs” for big Oh notation

There are some simple heuristics that can help when trying to compare two functions \( F \) and \( G \):

- Multiplicative constants don’t matter in Oh notation, and so if \( F(n) = O(G(n)) \) then \( 100F(n) = O(G(n)) \).

- When adding two functions, we only care about the larger one. For example, for the purpose of Oh notation, \( n^3 + 100n^2 \) is the same
as $n^3$, and in general in any polynomial, we only care about the larger exponent.

- For every two constants $a, b > 0$, $n^a = O(n^b)$ if and only if $a \leq b$, and $n^a = o(n^b)$ if and only if $a < b$. For example, combining the two observations above, $100n^2 + 10n + 100 = o(n^3)$.

- Polynomial is always smaller than exponential: $n^a = o(2^n)$ for every two constants $a > 0$ and $\epsilon > 0$ even if $\epsilon$ is much smaller than $a$. For example, $100n^{100} = o(2^{\sqrt{n}})$.

- Similarly, logarithmic is always smaller than polynomial: $(\log n)^a$ (which we write as $\log^a n$) is $o(n^\epsilon)$ for every two constants $a, \epsilon > 0$. For example, combining the observations above, $100n^2 \log^{100} n = o(n^3)$.

In most (though not all!) cases we use $O$ notation, the constants hidden by it are not too huge and so on an intuitive level, you can think of $F = O(G)$ as saying something like $F(n) \leq 1000G(n)$ and $F = \Omega(G)$ as saying something $F(n) \geq 0.001G(n)$.

### 0.7 Proofs

Many people think of mathematical proofs as a sequence of logical deductions that starts from some axioms and ultimately arrives at a conclusion. In fact, some dictionaries define proofs that way. This is not entirely wrong, but in reality a mathematical proof of a statement $X$ is simply an argument that convinces the reader that $X$ is true beyond a shadow of a doubt. To produce such a proof you need to:

1. Understand precisely what $X$ means.

2. Convince yourself that $X$ is true.

3. Write your reasoning down in plain, precise and concise English (using formulas or notation only when they help clarity).

In many cases, Step 1 is the most important one. Understanding what a statement means is often more than halfway towards understanding why it is true. In Step 3, to convince the reader beyond a shadow of a doubt, we will often want to break down the reasoning to “basic steps”, where each basic step is simple enough to be “self evident”. The combination of all steps yields the desired statement.
0.7.1 Proofs and programs

There is a great deal of similarity between the process of writing proofs and that of writing programs, and both require a similar set of skills. Writing a program involves:

1. Understanding what is the task we want the program to achieve.

2. Convincing yourself that the task can be achieved by a computer, perhaps by planning on a whiteboard or notepad how you will break it up to simpler tasks.

3. Converting this plan into code that a compiler or interpreter can understand, by breaking up each task into a sequence of the basic operations of some programming language.

In programs as in proofs, step 1 is often the most important one. A key difference is that the reader for proofs is a human being and for programs is a compiler. Thus our emphasis is on readability and having a clear logical flow for the proof (which is not a bad idea for programs as well...). When writing a proof, you should think of your audience as an intelligent but highly skeptical and somewhat petty reader, that will “call foul” at every step that is not well justified.

0.8 Extended example: graph connectivity

To illustrate these ideas, let us consider the following example of a true theorem:

**Theorem 0.7 — Minimum edges for connected graphs.** Every connected undirected graph of \( n \) vertices has at least \( n - 1 \) edges.

We are going to take our time to understand how one would come up with a proof for Theorem 0.7, and how to write such a proof down. This will not be the shortest way to prove this theorem, but hopefully following this process will give you some general insights on reading, writing, and discovering mathematical proofs.

Before trying to prove Theorem 0.7, we need to understand what it means. Let’s start with the terms in the theorems. We defined undirected graphs and the notion of connectivity in ?? above. In particular, an undirected graph \( G = (V, E) \) is connected if for every pair \( u, v \in V \), there is a path \( (u_0, u_1, \ldots, u_k) \) such that \( u_0 = u \), \( u_k = v \), and \( \{u_i, u_{i+1}\} \in E \) for every \( i \in [k] \).
It is crucial that at this point you pause and verify that you completely understand the definition of connectivity. Indeed, you should make a habit of pausing after any statement of a theorem, even before looking at the proof, and verifying that you understand all the terms that the theorem refers to.

To prove Theorem 0.7 we need to show that there is no 2-vertex connected graph with fewer than 1 edges, 3-vertex connected graph with fewer than 2 edges, and so on and so forth. One of the best ways to prove a theorem is to first try to disprove it. By trying and failing to come up with a counterexample, we often understand why the theorem can not be false. For example, if you try to draw a 4-vertex graph with only two edges, you can see that there are basically only two choices for such a graph as depicted in Fig. 2, and in both there will remain a vertex that is not connected.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2.png}
\caption{In a four vertex graph with two edges, either both edges have a shared vertex or they don’t. In both cases the graph will not be connected.}
\end{figure}

In fact, we can see that if we have a budget of 2 edges and we choose some vertex $u$, we will not be able to connect to $u$ more than two other vertices, and similarly with a budget of 3 edges we will not be able to connect to $u$ more than three other vertices. We can keep trying to draw such examples until we convince ourselves that the theorem is probably true, at which point we want to see how we can prove it.

If you have not seen the proof of this theorem before (or don’t remember it), this would be an excellent point to pause and try to prove it yourself.

There are several ways to approach this proof, but one version is to start by proving it for small graphs, such as graphs with 2,3 or 4 edges, for which we can check all the cases, and then try to
extend the proof for larger graphs. The technical term for this proof approach is \textit{proof by induction}.

\section*{0.8.1 Mathematical induction}

\textit{Induction} is simply an application of the self-evident \textit{Modus Ponens} rule that says that if (a) \( P \) is true and (b) \( P \) implies \( Q \) then \( Q \) is true. In the setting of proofs by induction we typically have a statement \( Q(k) \) that is parameterized by some integer \( k \), and we prove that (a) \( Q(0) \) is true and (b) For every \( k > 0 \), if \( Q(0), \ldots, Q(k-1) \) are all true then \( Q(k) \) is true.\textsuperscript{9} By repeatedly applying Modus Ponens, we can deduce from (a) and (b) that \( Q(1) \) is true, and then from (a),(b) and \( Q(1) \) that \( Q(2) \) is true, and so on and so forth to obtain that \( Q(k) \) is true for every \( k \). The statement (a) is called the “base case”, while (b) is called the “inductive step”. The assumption in (b) that \( Q(i) \) holds for \( i < k \) is called the “inductive hypothesis”.

\textbf{Induction and recursion} Proofs by inductions are closely related to algorithms by recursion. In both cases we reduce solving a larger problem to solving a smaller instance of itself. In a recursive algorithm to solve some problem \( P \) on an input of length \( k \) we ask ourselves “what if someone handed me a way to solve \( P \) on instances smaller than \( k \)?”. In an inductive proof to prove a statement \( Q \) parameterized by a number \( k \), we ask ourselves “what if I already knew that \( Q(k') \) is true for \( k' < k \)?”. Both induction and recursion are crucial concepts for this course and Computer Science at large (and even other areas of inquiry, including not just mathematics but other sciences as well). Both can be initially (and even post-initially) confusing, but with time and practice they become clearer. For more on proofs by induction and recursion, you might find the following Stanford CS 103 handout, this MIT 6.00 lecture or this excerpt of the Lehman-Leighton book useful.

\section*{0.8.2 Proving the theorem by induction}

There are several ways to use induction to prove \textbf{Theorem 0.7}. We will do so by following our intuition above that with a budget of \( k \) edges, we cannot connect to a vertex more than \( k \) other vertices. That is, we will define the statement \( Q(k) \) as follows:
Note that \( Q(n - 2) \) implies our theorem, since it means that in an \( n \) vertex graph of \( n - 2 \) edges, there would be at most \( n - 1 \) vertices that are connected to \( u \), and hence in particular there would be some vertex that is not connected to \( u \). More formally, if we define, given any undirected graph \( G \) and vertex \( u \) of \( G \), the set \( C_G(u) \) to contain all vertices connected to \( u \), then the statement \( Q(k) \) is that for every undirected graph \( G = (V, E) \) with \( |E| = k \) and \( u \in V \), \(|C_G(u)| \leq k + 1\).

To prove that \( Q(k) \) is true for every \( k \) by induction, we will first prove that (a) \( Q(0) \) is true, and then prove (b) if \( Q(0), \ldots, Q(k - 1) \) are true then \( Q(k) \) is true as well. In fact, we will prove the stronger statement (b') that if \( Q(k - 1) \) is true then \( Q(k) \) is true as well. (b') is a stronger statement than (b) because it has same conclusion with a weaker assumption.) Thus, if we show both (a) and (b') then we complete the proof of Theorem 0.7.

Proving (a) (i.e., the “base case”) is actually quite easy. The statement \( Q(0) \) says that if \( G \) has zero edges, then \( |C_G(u)| = 1 \), but this is clear because in a graph with zero edges, \( u \) is only connected to itself. The heart of the proof is, as typical with induction proofs, is in proving a statement such as (b') (or even the weaker statement (b)). Since we are trying to prove an implication, we can assume the so-called “inductive hypothesis” that \( Q(k - 1) \) is true and need to prove from this assumption that \( Q(k) \) is true. So, suppose that \( G = (V, E) \) is a graph of \( k \) edges, and \( u \in V \). Since we can use induction, a natural approach would be to remove an edge \( e \in E \) from the graph to create a new graph \( G' \) of \( k - 1 \) edges. We can use the induction hypothesis to argue that \( |C_{G'}(u)| \leq k \). Now if we could only argue that removing the edge \( e \) reduced the connected component of \( u \) by at most a single vertex, then we would be done, as we could argue that \(|C_G(u)| \leq |C_{G'}(u)| + 1 \leq k + 1\).

Alas, this might not be the case. It could be that removing a single edge \( e \) will greatly reduce the size of \( C_G(u) \). For example that edge might be a “bridge” between two large connected components; such a situation is illustrated in Fig. 3. This might seem as a real stumbling...
Figure 3: Removing a single edge $e$ can greatly decrease the number of vertices that are connected to a vertex $u$.

block, and at this point we might go back to the drawing board to see if perhaps the theorem is false after all. However, if we look at various concrete examples, we see that in any concrete example, there is always a “good” choice of an edge, adding which will increase the component connected to $u$ by at most one vertex.

![Figure 4: Removing an edge $e = \{s, w\}$ where $w \in C_G(u)$ has degree one removes only $w$ from $C_G(u)$.](image)

The crucial observation is that this always holds if we choose an edge $e = \{s, w\}$ where $w \in C_G(u)$ has degree one in the graph $G$, see Fig. 4. The reason is simple. Since every path from $u$ to $w$ must pass through $s$ (which is $w$’s only neighbor), removing the edge $\{s, w\}$ merely has the effect of disconnecting $w$ from $u$, and hence $C_{G'}(u) = C_G(u) \setminus \{w\}$ and in particular $|C_{G'}(u)| = |C_G(u)| - 1$, which is exactly the condition we needed.

Now the question is whether there will always be a degree one vertex in $C_G(u) \setminus \{u\}$. Of course generally we are not guaranteed that a graph would have a degree one vertex, but we are not dealing with a general graph here but rather a graph with a small number of
edges. We can assume that $|C_G(u)| > k + 1$ (otherwise we’re done) and each vertex in $C_G(u)$ must have degree at least one (as otherwise it would not be connected to $u$). Thus, the only case where there is no vertex $w \in C_G(u) \setminus \{u\}$ of degree one, is when the degrees of all vertices in $C_G(u)$ are at least 2. But then by Lemma 0.1 the number of edges in the graph is at least $\frac{1}{2} \cdot 2 \cdot (k + 1) > k$, which contradicts our assumption that the graph $G$ has at most $k$ edges. Thus we can conclude that either $|C_G(u)| \leq k + 1$ (in which case we’re done) or there is a degree one vertex $w \neq u$ that is connected to $u$. By removing the single edge $e$ that touches $w$, we obtain a $k - 1$ edge graph $G'$ which (by the inductive hypothesis) satisfies $|C_{G'}(u)| \leq k$, and hence $|C_G(u)| = |C_{G'}(u) \cup \{w\}| \leq k + 1$. This suffices to complete an inductive proof of statement $Q(k)$.

### 0.8.3 Writing down the proof

All of the above was a discussion of how we discover the proof, and convince ourselves that the statement is true. However, once we do that, we still need to write it down. When writing the proof, we use the benefit of hindsight, and try to streamline what was a messy journey into a linear and easy-to-follow flow of logic that starts with the word “Proof:” and ends with “QED” or the symbol $\blacksquare$. All our discussions, examples and digressions can be very insightful, but we keep them outside the space delimited between these two words, where (as described by this excellent handout) “every sentence must be load bearing”. Just like we do in programming, we can break the proof into little “subroutines” or “functions” (known as lemmas or claims in math language), which will be smaller statements that help us prove the main result. However, it should always be crystal-clear to the reader in what stage we are of the proof. Just like it should always be clear to which function a line of code belongs to, it should always be clear whether an individual sentence is part of a proof of some intermediate result, or is part of the argument showing that this intermediate result implies the theorem. Sometimes we highlight this partition by noting after each occurrence of “QED” to which lemma or claim it belongs.

Let us see how the proof of Theorem 0.7 looks in this streamlined fashion. We start by repeating the theorem statement

| Theorem 0.8 — Minimum edges for connected graphs (rephrased). Every connected undirected graph of $n$ vertices has at least $n - 1$ edges. |

Proof of Theorem 0.8. The proof will follow from the following lemma:
Lemma 0.9 For every $k \in \mathbb{N}$, undirected graph $G = (V, E)$ of at most $k$ edges, and $u \in V$, the number of vertices connected to $u$ in $G$ is at most $k + 1$.

We start by showing that Lemma 0.9 implies the theorem:

Proof of Theorem 0.8 from Lemma 0.9: We will show that for undirected graph $G = (V, E)$ of $n$ vertices and at most $n - 2$ edges, there is a pair $u, v$ of vertices that are disconnected in $G$. Let $G$ be such a graph and $u$ be some vertex of $G$. By Lemma 0.9, the number of vertices connected to $u$ is at most $n - 1$, and hence (since $|V| = n$) there is a vertex $v \in V$ that is not connected to $u$, thus completing the proof. QED (Proof of Theorem 0.8 from Lemma 0.9)

We now turn to proving Lemma 0.9. Let $G = (V, E)$ be an undirected graph of $k$ vertices and $u \in V$. We define $C_G(u)$ to be the set of vertices connected to $u$. To complete the proof of Lemma 0.9, we need to prove that $|C_G(u)| \leq k + 1$. We will do so by induction on $k$.

The base case that $k = 0$ is true because a graph with zero edges, $u$ is only connected to itself.

Now suppose that Lemma 0.9 is true for $k - 1$ and we will prove it for $k$. Let $G = (V, E)$ and $u \in V$ be as above, where $|E| = k$, and suppose (towards a contradiction) that $|C_G(u)| \geq k + 2$. Let $S = C_G(u) \setminus \{u\}$. Denote by $\deg(v)$ the degree of any vertex $v$. By Lemma 0.1, $\sum_{v \in S} \deg(v) \leq \sum_{v \in V} \deg(v) = 2|E| = 2k$. Hence in particular, under our assumption that $|S| + 1 = |C_G(u)| \geq k + 2$, we get that $\frac{1}{|S|} \sum_{v \in S} \deg(v) \leq 2k/(k + 1) < 2$. In other words, the average degree of a vertex in $S$ is smaller than 2, and hence in particular there is some vertex $w \in S$ with degree smaller than 2. Since $w$ is connected to $u$, it must have degree at least one, and hence (since $w$'s degree is smaller than two) degree exactly one. In other words, $w$ has a single neighbor which we denote by $s$.

Let $G'$ be the graph obtained by removing the edge $\{s, w\}$ from $G$. Since $G'$ has at most $k - 1$ edges, by the inductive hypothesis we can assume that $|C_{G'}(u)| \leq k$. The proof of the lemma is concluded by showing the following claim:

Claim: Under the above assumptions, $|C_G(u)| \leq |C_{G'}(u)| + 1$. 
Proof of claim: The claim says that $C_G'(u)$ has at most one fewer element than $C_G(u)$. Thus it follows from the following statement (*):

$$C_G'(u) \supseteq C_G(u \setminus \{w\}).$$

To prove (*) we need to show that for every $v \neq w$ that is connected to $u$, $v \in C_G(u)$. Indeed for every such $v$, Lemma 0.3 implies that there must be some simple path $(t_0, t_1, \ldots, t_{i-1}, t_i)$ in the graph $G$ where $t_0 = u$ and $t_i = v$. But $w$ cannot belong to this path, since $w$ is different from the endpoints $u$ and $v$ of the path and can’t equal one of the intermediate points either, since it has degree one and that would make the path not simple. More formally, if $w = t_j$ for $0 < j < i$, then since $w$ has only a single neighbor $s$, it would have to hold that $w$’s neighbor $s$ satisfies $s = t_{j-1} = t_{j+1}$, contradicting the simplicity of the path. Hence the path from $u$ to $v$ is also a path in the graph $G'$, which means that $v \in C_G'(u)$, which is what we wanted to prove. QED (claim)

The claim implies Lemma 0.9 since by the inductive assumption, $|C_G'(u)| \leq k$, and hence by the claim $|C_G(u)| \leq k + 1$, which is what we wanted to prove. This concludes the proof of Lemma 0.9 and hence also of Theorem 0.8. QED (Lemma 0.9), QED (Theorem 0.8)

The proof above used the observation that if the average of some $n$ numbers $x_0, \ldots, x_{n-1}$ is at most $X$, then there must exists at least a single number $x_i \leq X$. (In this particular proof, the numbers were the degrees of vertices in $S$.) This is known as the averaging principle, and despite its simplicity, it is often extremely useful.

Reading a proof is no less of an important skill than producing one. In fact, just like understanding code, it is a highly non-trivial skill in itself. Therefore I strongly suggest that you re-read the above proof, asking yourself at every sentence whether the assumption it makes are justified, and whether this sentence truly demonstrates what it purports to achieve. Another good habit is to ask yourself when reading a proof for every variable you encounter (such as $u$, $t_i$, $G'$, etc. in the above proof) the following questions: (1) What type of variable is it? Is it a number? a graph? a vertex? a function? and (2) What do we know about it? Is it an arbitrary member of the set? Have we shown some facts about it?, and (3) What are we trying to show about it?.
0.9 Proof writing style

A mathematical proof is a piece of writing, but it is a specific genre of writing with certain conventions and preferred styles. As in any writing, practice makes perfect, and it is also important to revise your drafts for clarity.

In a proof for the statement $X$, all the text between the words "Proof:" and "QED" should be focused on establishing that $X$ is true. Digressions, examples, or ruminations should be kept outside these two words, so they do not confuse the reader. The proof should have a clear logical flow in the sense that every sentence or equation in it should have some purpose and it should be crystal-clear to the reader what this purpose is. When you write a proof, for every equation or sentence you include, ask yourself:

1. Is this sentence or equation stating that some statement is true?
2. If so, does this statement follow from the previous steps, or are we going to establish it in the next step?
3. What is the role of this sentence or equation? Is it one step towards proving the original statement, or is it a step towards proving some intermediate claim that you have stated before?
4. Finally, would the answers to questions 1-3 be clear to the reader? If not, then you should reorder, rephrase or add explanations.

Some helpful resources on mathematical writing include this handout by Lee, this handout by Hutching, as well as several of the excellent handouts in Stanford’s CS 103 class.

0.9.1 Patterns in proofs

Just like in programming, there are several common patterns of proofs that occur time and again. Here are some examples:

Proofs by contradiction: One way to prove that $X$ is true is to show that if $X$ was false then we would get a contradiction as a result. Such proofs often start with a sentence such as “Suppose, towards a contradiction, that $X$ is false” and end with deriving some contradiction (such as a violation of one of the assumptions in the theorem statement). Here is an example:

Lemma 0.10 There are no natural numbers $a, b$ such that $\sqrt{2} = \frac{a}{b}$.
Proof. Suppose, towards the sake of contradiction that this is false, and so let \( a \in \mathbb{N} \) be the smallest number such that there exists some \( b \in \mathbb{N} \) satisfying \( \sqrt{2} = \frac{a}{b} \). Squaring this equation we get that \( 2 = \frac{a^2}{b^2} \) or \( a^2 = 2b^2 \) (\(*\)). But this means that \( a^2 \) is even, and since the product of two odd numbers is odd, it means that \( a \) is even as well, or in other words, \( a = 2a' \) for some \( a' \in \mathbb{N} \). Yet plugging this into \((\ast)\) shows that \( 4a'^2 = 2b^2 \) which means \( b^2 = 2a'^2 \) is an even number as well. By the same considerations as above we get that \( b \) is even and hence \( a/2 \) and \( b/2 \) are two natural numbers satisfying \( \sqrt{2} = \frac{a}{b} \), contradicting the minimality of \( a \).

Proofs of a universal statement: Often we want to prove a statement \( X \) of the form “Every object of type \( O \) has property \( P \)” such proofs often start with a sentence such as “Let \( o \) be an object of type \( O \)” and end by showing that \( o \) has the property \( P \). Here is a simple example:

Lemma 0.11 For every natural number \( n \in \mathbb{N} \), either \( n \) or \( n + 1 \) is even.

Proof. Let \( n \in \mathbb{N} \) be some number. If \( n/2 \) is a whole number then we are done, since then \( n = 2(n/2) \) and hence it is even. Otherwise, \( n/2 + 1/2 \) is a whole number, and hence \( 2(n/2 + 1/2) = n + 1 \) is even.

Proofs of an implication: Another common case is that the statement \( X \) has the form “\( A \) implies \( B \)”. Such proofs often start with a sentence such as “Assume that \( A \)” and end with a derivation of \( B \) from \( A \). Here is a simple example:

Lemma 0.12 If \( b^2 \geq 4ac \) then there is a solution to the quadratic equation \( ax^2 + bx + c = 0 \).

Proof. Suppose that \( b^2 \geq 4ac \). Then \( d = b^2 - 4ac \) is a non-negative number and hence it has a square root \( s \). Thus \( x = \frac{-b + s}{2a} \) satisfies
\[
ax^2 + bx + c = a(-b + s)^2 / (4a^2) + b(-b + s)/(2a) + c = (b^2 - 2bs + s^2)/(4a) + (-b^2 + bs)/(2a) + c.
\]
(17)
Rearranging the terms of Eq. (17) we get
\[
s^2/(4a) + c - b^2/(4a) = (b^2 - 4ac)/(4a) + c - b^2/(4a) = 0
\]
(18)

Proofs of equivalence: If a statement has the form “\( A \) if and only if \( B \)” (often shortened as “\( A \iff B \)” then we need to prove both that \( A \) implies \( B \) and that \( B \) implies \( A \). We call the implication that \( A \)
implies B the “only if” direction, and the implication that B implies A the “if” direction.

Proofs by combining intermediate claims: When a proof is more complex, it is often helpful to break it apart into several steps. That is, to prove the statement $X$, we might first prove statements $X_1, X_2,$ and $X_3$ and then prove that $X_1 \land X_2 \land X_3$ implies $X$. Our proof of Theorem 0.7 had this form.

Proofs by case distinction: This is a special case of the above, where to prove a statement $X$ we split into several cases $C_1, \ldots, C_k,$ and prove that (a) the cases are exhaustive, in the sense that one of the cases $C_i$ must happen and (b) go one by one and prove that each one of the cases $C_i$ implies the result $X$ that we are after.

“Without loss of generality (w.l.o.g)”\textsuperscript{11}: This term can be initially quite confusing to students. It is essentially a way to shorten case distinctions such as the above. The idea is that if Case 1 is equal to Case 2 up to a change of variables or a similar transformation, then the proof of Case 1 will also imply the proof of case 2. It is always a statement that should be viewed with suspicion. Whenever you see it in a proof, ask yourself if you understand why the assumption made is truly without loss of generality, and when you use it, try to see if the use is indeed justified. Sometimes it might be easier to just repeat the proof of the second case (adding a remark that the proof is very similar to the first one).

Proofs by induction: We can think of such proofs as a variant of the above, where we have an unbounded number of intermediate claims $X_0, X_2, \ldots, X_k,$ and we prove that $X_0$ is true, as well that $X_0$ implies $X_1$, and that $X_0 \land X_1$ implies $X_2$, and so on and so forth.

The website for CMU course 15-251 contains a useful handout on potential pitfalls when making proofs by induction.

0.10 Non-standard notation

Most of the notation we discussed above is standard and is used in most mathematical texts. The main points where we diverge are:

- We index the natural numbers $\mathbb{N}$ starting with 0 (though many other texts, especially in computer science, do the same).

- We also index the set $[n]$ starting with 0, and hence define it as $\{0, \ldots, n-1\}$. In most texts it is defined as $\{1, \ldots, n\}$. Similarly, we index coordinates of our strings starting with 0, and hence a string $x \in \{0,1\}^n$ is written as $x_0x_1 \cdots x_{n-1}$.
• We use partial functions which are functions that are not necessarily defined on all inputs. When we write \( f : A \to B \) this will refer to a total function unless we say otherwise. When we want to emphasize that \( f \) can be a partial function, we will sometimes write \( f : A \to_p B \).

• As we will see later on in the course, we will mostly describe our computational problems in the terms of computing a Boolean function \( f : \{0,1\}^* \to \{0,1\} \). In contrast, most textbooks will refer to this as the task of deciding a language \( L \subseteq \{0,1\}^* \). These two viewpoints are equivalent, since for every set \( L \subseteq \{0,1\}^* \) there is a corresponding function \( f \) \( = 1_L \) such that \( f(x) = 1 \) if and only if \( x \in L \). Computing partial functions corresponds to the task known in the literature as a solving a promise problem.\(^{12}\)

• Some other notation we use is \( \lceil x \rceil \) and \( \lfloor x \rfloor \) for the “ceiling” and “floor” operators that correspond to “rounding up” or “rounding down” a number to the nearest integer. We use \( (x \mod y) \) to denote the “remainder” of \( x \) when divided by \( y \). That is, \( (x \mod y) = x - y \lfloor x/y \rfloor \).

0.11 Exercises

Exercise 0.1 — Inclusion Exclusion. Let \( A, B \) be finite sets. Prove that
\[
|A \cup B| = |A| + |B| - |A \cap B|.
\]

1. Let \( A_0, \ldots, A_{k-1} \) be finite sets. Prove that
\[
|A_0 \cup \cdots \cup A_{k-1}| \geq \sum_{i=0}^{k-1} |A_i| - \sum_{0 \leq i < j < k} |A_i \cap A_j|.
\]

2. Let \( A_0, \ldots, A_{k-1} \) be finite subsets of \( \{1, \ldots, n\} \), such that \( |A_i| = m \) for every \( i \in [k] \). Prove that if \( k > 100n \), then there exist two distinct sets \( A_i, A_j \) s.t. \( |A_i \cap A_j| \geq m^2/(10n) \).

Exercise 0.2 Prove that if \( S, T \) are finite and \( F : S \to T \) is one to one then \( |S| \leq |T| \).

Exercise 0.3 Prove that if \( S, T \) are finite and \( F : S \to T \) is onto then \( |S| \geq |T| \).

Exercise 0.4 Prove that for every finite \( S, T \), there are \((|T| + 1)^{|S|}\) partial functions from \( S \) to \( T \).

Exercise 0.5 Suppose that \( \{S_n\}_{n \in \mathbb{N}} \) is a sequence such that \( S_0 \leq 10 \) and for \( n > 1 n \leq 5S_{\lfloor n/5 \rfloor} + 2n \). Prove by induction that \( S_n \leq 100n \log n \) for every \( n \).

\(^{12}\) Because the language notation is so prevalent in textbooks, we will occasionally remind the reader of this correspondence.
Exercise 0.6  Describe the following statement in English words:
\[ \forall n \in \mathbb{N} \exists p > n \forall a, b \in \mathbb{N} (a \times b \neq p) \lor (a = 1). \]

Exercise 0.7  Prove that for every undirected graph \( G \) of 100 vertices, if every vertex has degree at most 4, then there exists a subset \( S \) of at 20 vertices such that no two vertices in \( S \) are neighbors of one another.

Exercise 0.8  Suppose that we toss three independent fair coins \( a, b, c \in \{0, 1\} \). What is the probability that the XOR of \( a, b, \) and \( c \) is equal to 1? What is the probability that the AND of these three values is equal to 1? Are these two events independent?

Exercise 0.9  For every pair of functions \( F, G \) below, determine which of the following relations holds: \( F = O(G) \), \( F = \Omega(G) \), \( F = o(G) \) or \( F = \omega(G) \).

   a. \( F(n) = n, G(n) = 100n \).
   b. \( F(n) = n, G(n) = \sqrt{n} \).
   c. \( F(n) = n, G(n) = 2^{\log(n)^2} \).
   d. \( F(n) = n, G(n) = 2^{\log n} \).

Exercise 0.10  Give an example of a pair of functions \( F, G : \mathbb{N} \to \mathbb{N} \) such that neither \( F = O(G) \) nor \( G = O(F) \) holds.

0.12 Bibliographical notes

The section heading “A Mathematician’s Apology”, refers of course to Hardy’s classic book. Even when Hardy is wrong, he is very much worth reading.

0.13 Acknowledgements
Introduction

“Computer Science is no more about computers than astronomy is about telescopes”, attributed to Edsger Dijkstra. ¹

“Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry.”, Paul Graham 2003. ²

“The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why?. . . I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block.”, Alan Cobham, 1964

¹ This quote is typically read as disparaging the importance of actual physical computers in Computer Science, but note that telescopes are absolutely essential to astronomy and are our only means of connecting theoretical speculations with actual experimental observations.

² To be fair, in the following sentence Graham says “you need to know how to calculate time and space complexity and about Turing completeness”. Apparently, NP-hardness, randomization, cryptography, and quantum computing are not essential to a hacker’s education.

The origin of much of science and medicine can be traced back to the ancient Babylonians. But perhaps their greatest contribution to humanity was the invention of the place-value number system. This is the idea that we can represent any number using a fixed number of digits, whereby the position of the digit is used to determine the corresponding value, as opposed to system such as Roman numerals, where every symbol has a fixed numerical value regardless of position. For example, the distance to the moon is 238,900 of our miles or 259,956 Roman miles. The latter quantity, expressed in standard Roman numerals is

\[
\text{MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM}
\]
Writing the distance to the sun in Roman numerals would require about 100,000 symbols: a 50 page book just containing this single number!

This means that for someone that thinks of numbers in an additive system like Roman numerals, quantities like the distance to the moon or sun are not merely large- they are unspeakable: cannot be expressed or even grasped. It’s no wonder that Eratosthenes, who was the first person to calculate the earth’s diameter (up to about ten percent error) and Hipparchus who was the first to calculate the distance to the moon, did not use a Roman-numeral type system but rather the Babylonian sexadecimal (i.e., base 60) place-value system.

The Babylonians also invented the precursors of “standard algorithms” that we were all taught in elementary school for adding and multiplying numbers. These algorithms and their variants have been of course essential to people throughout history working with abaci, papyrus, or pencil and paper, but in our computer age, do they really serve any purpose beyond torturing third graders?

To answer this question, let us try to see in what sense is the standard digit by digit multiplication algorithm “better” than the straightforward implementation of multiplication as iterated addition. Let’s start by more formally describing both algorithms:

**Naive multiplication algorithm:**
*Input:* Non-negative integers $x, y$
*Operation:*
1. Let $result \leftarrow 0$.
2. For $i = 1, \ldots, y$: set $result \leftarrow result + x$
3. Output $result$

**Standard gradeschool multiplication algorithm:**
*Input:* Non-negative integers $x, y$
*Operation:*
1. Let $n$ be number of digits of $y$, and set $result \leftarrow 0$.
2. For $i = 0, \ldots, n - 1$: set $result \leftarrow result + 10^i \times x$
Both algorithms assume that we already know how to add numbers, and the second one also assumes that we can multiply a number by a power of 10 (which is after all a simple shift) as well as multiply by a single digit (which like addition, is done by multiplying each digit and propagating carries). Now suppose that \( x \) and \( y \) are two numbers of \( n \) decimal digits each. Adding two such numbers takes at least \( n \) single digit additions (depending on how many times we need to use a “carry”), and so adding \( x \) to itself \( y \) times will take at least \( n \cdot y \) single digit additions. In contrast, the standard gradeschool algorithm reduces this problem to taking \( n \) products of \( x \) with a single digit (which require up to \( 2n \) single digit operations each, depending on carries) and then adding all of those together (total of \( n \) additions, which, again depending on carries, would cost at most \( 2n^2 \) single digit operations) for a total of at most \( 4n^2 \) single digit operations. How much faster would \( 4n^2 \) operations be than \( n \cdot y \)? and would this make any difference in a modern computer?

Let us consider the case of multiplying 64 bit or 20 digit numbers.\(^4\) That is, the task of multiplying two numbers \( x, y \) that are between \( 10^{19} \) and \( 10^{20} \). Since in this case \( n = 20 \), the standard algorithm would use at most \( 4n^2 = 1600 \) single digit operations, while repeated addition would require at least \( n \cdot y \geq 20 \cdot 10^{19} \) single digit operations. To understand the difference, consider that a human being might do a single digit operation in about 2 seconds, requiring just under an hour to complete the calculation of \( x \times y \) using the gradeschool algorithm. In contrast, even though it is more than a billion times faster, a modern PC that computes \( x \times y \) using naïve iterated addition would require about \( 10^{20} / 10^9 = 10^{11} \) seconds (which is more than three millenia!) to compute the same result.

We see that computers have not made algorithms obsolete. On the contrary, the vast increase in our ability to measure, store, and communicate data has led to a much higher demand on developing better and more sophisticated algorithms that can allow us to make better decisions based on these data. We also see that to a large extent the notion of algorithm is independent of the actual computing device that will execute it. The digit-by-digit standard algorithm is vastly better than iterated addition regardless if the technology implementing it is a silicon based chip or a third grader with pen and paper.

\(^4\) This is a common size in several programming languages; for example the long data type in the Java programming language, and (depending on architecture) the long or long long types in C.
Theoretical computer science is largely about studying the inherent properties of algorithms and computation, that are independent of current technology. We ask questions that already pondered by the Babylonians, such as “what is the best way to multiply two numbers?” as well as those that rely on cutting-edge science such as “could we use the effects of quantum entanglement to factor numbers faster” and many in between. These types of questions are the topic of this course.

In Computer Science parlance, a scheme such as the decimal (or sexadecimal) positional representation for numbers is known as a data structure, while the operations on this representations are known as algorithms. Data structures and algorithms have enabled amazing applications, but their importance goes beyond their practical utility. Structures from computer science, such as bits, strings, graphs, and even the notion of a program itself, as well as concepts such as universality and replication, have not just found (many) practical uses but contributed a new language and a new way to view the world.

1.0.1 Example: A faster way to multiply

Once you think of the standard digit-by-digit multiplication algorithm, it seems like obviously the “right” way to multiply numbers. Indeed, in 1960, the famous mathematician Andrey Kolmogorov organized a seminar at Moscow State University in which he conjectured that every algorithm for multiplying two $n$ digit numbers would require a number of basic operations that is proportional to $n^2$. Another way to say it, is that he conjectured that in any multiplication algorithm, doubling the number of digits would quadruple the number of basic operations required.

A young student named Anatoly Karatsuba was in the audience, and within a week he found an algorithm that requires only about $Cn^{1.6}$ operations for some constant $C$. Such a number becomes much smaller than $n^2$ as $n$ grows. Amazingly, Karatsuba’s algorithm is based on a faster way to multiply two digit numbers.

Suppose that $x, y \in [100] = \{0, \ldots, 99\}$ are a pair of two-digit numbers. Let’s write $\overline{x}$ for the “tens” digit of $x$, and $\overline{x}$ for the “ones” digit, so that $x = 10\overline{x} + \overline{x}$, and write similarly $y = 10\overline{y} + \overline{y}$ for $\overline{y}, \overline{y} \in [10]$. The gradeschool algorithm for multiplying $x$ and $y$ is illustrated in Fig. 1.1.

The gradeschool algorithm works by transforming the task of...
multiplying a pair of two digit number into four single-digit multiplications via the formula

\[(10\bar{x} + \bar{x}) \times (10\bar{y} + \bar{y}) = 100\bar{xy} + 10(\bar{xy} + \bar{x}) + xy \quad (1.1)\]

Karatsuba’s algorithm is based on the observation that we can express this also as

\[(10\bar{x} + \bar{x}) \times (10\bar{y} + \bar{y}) = 100\bar{xy} + 10[(\bar{x} + \bar{x})(\bar{y} + \bar{y})] - 10xy - (10 + 1)xy \quad (1.2)\]

which reduces multiplying the two-digit number \(x\) and \(y\) to computing the following three “simple” products: \(\bar{xy}\), \(\bar{x}y\) and \((\bar{x} + \bar{x})(\bar{y} + \bar{y})\).\(^7\)

Of course if all we wanted to was to multiply two digit numbers, we wouldn’t really need any clever algorithms. It turns out that we can repeatedly apply the same idea, and use them to multiply 4-digit numbers, 8-digit numbers, 16-digit numbers, and so on and so forth. If we used the gradeschool based approach then our cost for doubling the number of digits would be to quadruple the number of digits as the original input.
Karatsuba’s multiplication algorithm illustrated for multiplying $x = 10x_1 + x_0$ and $y = 10y_1 + y_0$. We compute the three orange, green and purple products $xy$, $x\bar{y}$ and $(x + \bar{x})(\bar{y} + y)$ and then add and subtract them to obtain the result.

of multiplications, which for $n = 2^\ell$ digits would result in about $4^\ell = n^2$ operations. In contrast, in Karatsuba’s approach doubling the number of digits only triples the number of operations, which means that for $n = 2^\ell$ digits we require about $3^\ell = n^\log_2 3 \approx n^{1.58}$ operations.

Specifically, we use a recursive strategy as follows:

**Karatsuba Multiplication:**

**Input:** Non negative integers $x, y$ each of at most $n$ digits

**Operation:**
1. If $n \leq 2$ then return $x \cdot y$ (using a constant number of single digit multiplications)
2. Otherwise, let $m = \lfloor n/2 \rfloor$, and write $x = 10^m x_1 + x_0$ and $y = 10^m y_1 + y_0$.
3. Use recursion to compute $A = xy$, $B = y\bar{y}$ and $C = (x + \bar{x})(\bar{y} + y)$. Note that all the numbers will have at most $m + 1$ digits.
4. Return $(10^m - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$

To understand why the output will be correct, first note that for $n > 2$, it will always hold that $m < n - 1$, and hence the recursive calls will always be for multiplying numbers with a smaller number.

$^8$ Recall that for a number $x$, $\lfloor x \rfloor$ is obtained by “rounding down” $x$ to the largest integer smaller or equal to $x$. 
of digits, and (since eventually we will get to single or double digit numbers) the algorithm will indeed terminate. Now, since \( x = 10^m \bar{x} + \bar{x} \) and \( y = 10^n \bar{y} + \bar{y} \),

\[
x \times y = 10^n \bar{x} \cdot \bar{y} + 10^m (\bar{x} \bar{y} + x y) + x y
\]  
(1.3)

But we can also write the same expression as

\[
x \times y = 10^n \bar{x} \cdot \bar{y} + 10^m \left[ (\bar{x} + \bar{x})(\bar{y} + y) - x y - \bar{x} \bar{y} \right] + x y
\]  
(1.4)

which equals to the value \((10^n - 10^m) \cdot A + 10^n \cdot B + (1 - 10^m) \cdot C\) returned by the algorithm.

The key observation is that the formula Eq. (1.4) reduces computing the product of two \( n \) digit numbers to computing three products of \( \lceil n/2 \rceil \) digit numbers (namely \( \bar{x} \bar{y}, \bar{y} y \) and \((\bar{x} + \bar{x})(\bar{y} + y)\)) as well as performing a constant number (in fact eight) additions, subtractions, and multiplications by \( 10^n \) or \( 10^{\lfloor n/2 \rfloor} \) (the latter corresponding to simple shifts). Intuitively this means that as the number of digits doubles, the cost of multiplying triples instead of quadrupling, as happens in the naive algorithm. This implies that multiplying numbers of \( n = 2^\ell \) digits costs about \( 3^\ell = n \log_2 3 \sim n^{1.585} \) operations. In a Exercise 1.3, you will formally show that the number of single digit operations that Karatsuba’s algorithm uses for multiplying \( n \) digit integers is at most \( O(n \log_2 3) \) (see also Fig. 1.3).

**Ceilings, floors, and rounding:** One of the benefits of using big Oh notation is that we can allow ourselves to be a little looser with issues such as rounding numbers etc.. For example, the natural way to describe Karatsuba’s algorithm’s running time is as following the recursive equation \( T(n) = 3T(n/2) + O(n) \) but of course if \( n \) is not even then we cannot recursively invoke the algorithm on \( n/2 \)-digit integers. Rather, the true recursion is \( T(n) = 3T(\lfloor n/2 \rfloor + 1) + O(n) \). However, this will not make much difference when we don’t worry about constant factors, since its not hard to show that \( T(n + O(1)) \leq T(n) + o(T(n)) \) for the functions we care about (which turns out to be enough to carry over the same recursion). Another way to show that this doesn’t hurt us is to note that for every number \( n \), we can find \( n' \leq 2n \) such that \( n' \) is a power of two. Thus we can always “pad” the input by adding some input bits to make sure the number of digits is a power of two, in which
Figure 1.3: Running time of Karatsuba’s algorithm vs. the Gradeschool algorithm. Figure by Marina Mele.

Figure 1.4: Karatsuba’s algorithm reduces an \( n \)-bit multiplication to three \( n/2 \)-bit multiplications, which in turn are reduced to nine \( n/4 \)-bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth \( \log_3 n \), where at the root the extra cost is \( cn \) operations, at the first level the extra cost is \( c(n/2) \) operations, and at each of the \( 3^i \) nodes of level \( i \), the extra cost is \( c(n/2^i) \). The total cost is \( cn \sum_{i=0}^{\log_3 n} (3/2)^i \leq 2cn \log_3 3^i \) by the formula for summing a geometric series.
Beyond Karatsuba’s algorithm

It turns out that the ideas of Karatsuba can be further extended to yield asymptotically faster multiplication algorithms, as was shown by Toom and Cook in the 1960s. But this was not the end of the line. In 1971, Schönhage and Strassen gave an even faster algorithm using the Fast Fourier Transform; their idea was to somehow treat integers as “signals” and do the multiplication more efficiently by moving to the Fourier domain. The latest asymptotic improvement was given by Fürer in 2007 (though it only starts beating the Schönhage-Strassen algorithm for truly astronomical numbers). And yet, despite all this progress, we still don’t know whether or not there is an $O(n)$ time algorithm for multiplying two $n$ digit numbers!

1.0.3 Advanced note: matrix multiplication

(We will have several such “advanced” notes and sections throughout these lectures notes. These may assume background that not every student has, and in any case can be safely skipped over as none of the future parts will depend on them.)

It turns out that a similar idea as Karatsuba’s can be used to speed up matrix multiplications as well. Matrices are a powerful way to represent linear equations and operations, widely used in a great many applications of scientific computing, graphics, machine learning, and many many more. One of the basic operations one can do with two matrices is to multiply them. For example, if $x = \begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}$ and $y = \begin{pmatrix} y_{0,0} & y_{0,1} \\ y_{1,0} & y_{1,1} \end{pmatrix}$ then the product of $x$ and $y$ is the matrix $\begin{pmatrix} x_{0,0} y_{0,0} + x_{0,1} y_{1,0} & x_{0,0} y_{0,1} + x_{0,1} y_{1,1} \\ x_{1,0} y_{0,0} + x_{1,1} y_{1,0} & x_{1,0} y_{0,1} + x_{1,1} y_{1,1} \end{pmatrix}$. You can see that we can compute this matrix by eight products of numbers. Now suppose that $n$ is even and $x$ and $y$ are a pair of $n \times n$ matrices which we can think of as each composed of four $(n/2) \times (n/2)$ blocks $x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}$ and $y_{0,0}, y_{0,1}, y_{1,0}, y_{1,1}$. Then the formula for the matrix product of $x$ and $y$ can be expressed in the same way as above, just replacing products $x_{a,b} y_{c,d}$ with matrix products, and addition with matrix addition. This means that we can use the formula above to give an algorithm that
doubles the dimension of the matrices at the expense of increasing the number of operation by a factor of 8, which for $n = 2^\ell$ will result in $8^\ell = n^3$ operations. In 1969 Volker Strassen noted that we can compute the product of a pair of two by two matrices using only seven products of numbers by observing that each entry of the matrix $xy$ can be computed by adding and subtracting the following seven terms: $t_1 = (x_{0,0} + x_{1,1})(y_{0,0} + y_{1,1})$, $t_2 = (x_{0,0} + x_{1,1})y_{0,0}$, $t_3 = x_{0,0}(y_{0,1} - y_{1,1})$, $t_4 = x_{1,1}(y_{0,1} - y_{0,0})$, $t_5 = (x_{0,0} + x_{0,1})y_{1,1}$, $t_6 = (x_{1,0} - x_{0,0})(y_{0,0} + y_{0,1})$, $t_7 = (x_{0,1} - x_{1,1})(y_{1,0} + y_{1,1})$. Indeed, one can verify that $xy = \left( \frac{t_1+t_4-t_5-t_7}{t_2+t_4} \frac{t_2+t_5}{t_1+t_3-t_2+t_6} \right)$. This implies an algorithm with factor 7 increased cost for doubling the dimension, which means that for $n = 2^\ell$ the cost is $7^\ell = n^{\log_2 7} \sim n^{2.807}$. A long sequence of works has since improved this algorithm, and the current record has running time about $O(n^{2.373})$. Unlike the case of integer multiplication, at the moment we don’t know of a nearly linear in the matrix size (i.e., an $O(n^{2.373})$) time algorithm for matrix multiplication. People have tried to use group representations, which can be thought of as generalizations of the Fourier transform, to obtain faster algorithms, but this effort has not yet succeeded.

1.0.4 Algorithms beyond arithmetic

The quest for better algorithms is by no means restricted to arithmetical tasks such as adding, multiplying or solving equations. Many graph algorithms, including algorithms for finding paths, matchings, spanning trees, cuts, and flows, have been discovered in the last several decades, and this is still an intensive area of research. (For example, the last few years saw many advances in algorithms for the maximum flow problem, borne out of surprising connections with electrical circuits and linear equation solvers.)

These algorithms are being used not just for the “natural” applications of routing network traffic or GPS-based navigation, but also for applications as varied as drug discovery through searching for structures in gene-interaction graphs to computing risks from correlations in financial investments.

Google was founded based on the PageRank algorithm, which is an efficient algorithm to approximate the “principal eigenvector” of (a dampened version of) the adjacency matrix of web graph. The Akamai company was founded based on a new data structure, known as consistent hashing, for a hash table where buckets are stored at different servers.

The backpropagation algorithm, that computes partial derivatives of
a neural network in $O(n)$ instead of $O(n^2)$ time, underlies many of the recent phenomenal successes of learning deep neural networks. Algorithms for solving linear equations under sparsity constraints, a concept known as compressed sensing, have been used to drastically reduce the amount and quality of data needed to analyze MRI images. This is absolutely crucial for MRI imaging of cancer tumors in children, where previously doctors needed to use anesthesia to suspend breath during the MRI exam, sometimes with dire consequences.

Even for classical questions, studied through the ages, new discoveries are still being made. For the basic task, already of importance to the Greeks, of discovering whether an integer is prime or composite, efficient probabilistic algorithms were only discovered in the 1970s, while the first deterministic polynomial-time algorithm was only found in 2002. For the related problem of actually finding the factors of a composite number, new algorithms were found in the 1980s, and (as we’ll see later in this course) discoveries in the 1990s raised the tantalizing prospect of obtaining faster algorithms through the use of quantum mechanical effects.

Despite all this progress, there are still many more questions than answers in the world of algorithms. For almost all natural problems, we do not know whether the current algorithm is the “best”, or whether a significantly better one is still waiting to be discovered. As we already saw, even for the classical problem of multiplying numbers we have not yet answered the age-old question of “is multiplication harder than addition?”.

But at least we now know the right way to ask it.

1.0.5 On the importance of negative results.

Finding better multiplication algorithms is undoubtedly a worthwhile endeavor. But why is it important to prove that such algorithms don’t exist? What useful applications could possibly arise from an impossibility result?

One motivation is pure intellectual curiosity. After all, this is a question even Archimedes could have been excited about. Another reason to study impossibility results is that they correspond to the fundamental limits of our world or in other words to laws of nature. In physics, it turns out that the impossibility of building a perpetual motion machine corresponds to the law of conservation of energy. Other laws of nature also correspond to impossibility results: the impossibility of building a heat engine beating Carnot’s bound corresponds
to the second law of thermodynamics, while the impossibility of faster-than-light information transmission is a cornerstone of special relativity. Within mathematics, while we all learned the solution for quadratic equations in high school, the impossibility of generalizing this to equations of degree five or more gave birth to group theory. In his 300 B.C. book *The Elements*, the Greek mathematician Euclid based geometry on five “axioms” or “postulates”. Ever since then people have suspected that four axioms are enough, and try to base the “parallel postulate” (roughly speaking, that every line has a unique parallel line of each distance) from the other four. It turns out that this was impossible, and the impossibility result gave rise to so called “non-Euclidean geometries”, which turn out to be crucial for the theory of general relativity.10

In an analogous way, impossibility results for computation correspond to “computational laws of nature” that tell us about the fundamental limits of any information processing apparatus, whether based on silicon, neurons, or quantum particles.11 Moreover, computer scientists have recently been finding creative approaches to apply computational limitations to achieve certain useful tasks. For example, much of modern Internet traffic is encrypted using the RSA encryption scheme, which relies on its security on the (conjectured) non existence of an efficient algorithm to perform the inverse operation for multiplication—namely, factor large integers. More recently, the Bitcoin system uses a digital analog of the “gold standard” where, instead of being based on a precious metal, minting new currency corresponds to “mining” solutions for computationally difficult problems.

1.1 Lecture summary

(The notes for every lecture will end in such a “lecture summary” section that contains a few of the “take home messages” of the lecture. It is not meant to be a comprehensive summary of all the main points covered in the lecture.)

- There can be several different algorithms to achieve the same computational task. Finding a faster algorithm can make a much bigger difference than better technology.
- Better algorithms and data structures don’t just speed up calculations, but can yield new qualitative insights.
- One of the main topics of this course is studying the question of what is the most efficient algorithm for a given problem.

10 It is fine if you have not yet encountered many of the above. I hope however it sparks your curiosity!

11 Indeed, some exciting recent research has been trying to use computational complexity to shed light on fundamental questions in physics such understanding black holes and reconciling general relativity with quantum mechanics.
• To answer such a question we need to find ways to prove lower bounds on the computational resources needed to solve certain problems. That is, show an impossibility result ruling out the existence of “too good” algorithms.

1.1.1 Roadmap to the rest of this course

Often, when we try to solve a computational problem, whether it is solving a system of linear equations, finding the top eigenvector of a matrix, or trying to rank Internet search results, it is enough to use the “I know it when I see it” standard for describing algorithms. As long as we find some way to solve the problem, we are happy and don’t care so much about formal descriptions of the algorithm. But when we want to answer a question such as “does there exist an algorithm to solve the problem P?” we need to be much more precise.

In particular, we will need to (1) define exactly what does it mean to solve P, and (2) define exactly what is an algorithm. Even (1) can sometimes be non-trivial but (2) is particularly challenging; it is not at all clear how (and even whether) we can encompass all potential ways to design algorithms. We will consider several simple models of computation, and argue that, despite their simplicity, they do capture all “reasonable” approaches for computing, including all those that are currently used in modern computing devices.

Once we have these formal models of computation, we can try to obtain impossibility results for computational tasks, showing that some problems can not be solved (or perhaps can not be solved within the resources of our universe). Archimedes once said that given a fulcrum and a long enough lever, he could move the world. We will see how reductions allow us to leverage one hardness result into a slew of a great many others, illuminating the boundaries between the computable and uncomputable (or tractable and intractable) problems.

Later in this course we will go back to examining our models of computation, and see how resources such as randomness or quantum entanglement could potentially change the power of our model. In the context of probabilistic algorithms, we will see a glimpse of how randomness has become an indispensable tool for understanding computation, information, and communication.

We will also see how computational difficulty can be an asset rather than a hindrance, and be used for the “derandomization” of
probabilistic algorithms. The same ideas also show up in cryptography, which has undergone not just a technological but also an intellectual revolution in the last few decades, much of it building on the foundations that we explore in this course.

Theoretical Computer Science is a vast topic, branching out and touching upon many scientific and engineering disciplines. This course only provides a very partial (and biased) sample of this area. More than anything, I hope I will manage to “infect” you with at least some of my love for this field, which is inspired and enriched by the connection to practice, but which I find to be deep and beautiful regardless of applications.

1.2 Exercises

**Exercise 1.1** Rank the significance of the following inventions in speeding up multiplication of large (that is 100 digit or more) numbers. That is, use “back of the envelope” estimates to order them in terms of the speedup factor they offered over the previous state of affairs.

- a. Discovery of the gradeschool style digit by digit algorithm (improving upon repeated addition)
- b. Discovery of Karatsuba’s algorithm (improving upon the digit by digit algorithm)
- c. Invention of modern electronic computers (improving upon calculations with pen and paper)

**Exercise 1.2** The 1977 Apple II personal computer had a processor speed of 1.023 MHz or about $10^6$ operations per seconds. At the time of this writing the world’s fastest supercomputer performs 93 “petaflops” ($10^{15}$ floating point operations per second) or about $10^{18}$ basic steps per second. For each one of the following running times (as a function of the input length $n$), compute for both computers how large an input they could handle in a week of computation, if they run an algorithm that has this running time:

- a. $n$ operations.
- b. $n^2$ operations.
- c. $n \log n$ operations.
- d. $2^n$ operations.
- e. $n!$ operations.
Exercise 1.3 — Analysis of Karatsuba’s Algorithm. a. Suppose that $T_1, T_2, T_3, \ldots$ is a sequence of numbers such that $T_2 \leq 10$ and for every $n$, $T_n \leq 3T_{\lfloor n/2 \rfloor} + Cn$. Prove that $T_n \leq 10Cn^{\log_2 3}$ for every $n$.\(^\dagger\)

b. Prove that the number of single digit operations that Karatsuba’s algorithm takes to multiply two $n$ digit numbers is at most $1000n^{\log_2 3}$.

Exercise 1.4 Implement in the programming language of your choice functions Gradeschool_multiply($x, y$) and Karatsuba_multiply($x, y$) that take two arrays of digits $x$ and $y$ and return an array representing the product of $x$ and $y$ (where $x$ is identified with the number $x[0]+10\times x[1]+100\times x[2]+\ldots$ etc..) using the gradeschool algorithm and the Karatsuba algorithm respectively. At what number of digits does the Karatsuba algorithm beat the gradeschool one?

1.3 Bibliographical notes

For an overview of what we’ll see in this course, you could do far worse than read Bernard Chazelle’s wonderful essay on the Algorithm as an Idiom of modern science.

1.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

• The Fourier transform, the Fast Fourier transform algorithm and how to use it multiply polynomials and integers. This lecture of Jeff Erickson (taken from his collection of notes) is a very good starting point. See also this MIT lecture and this popular article.

• Fast matrix multiplication algorithms, and the approach of obtaining exponent two via group representations.

• The proofs of some of the classical impossibility results in mathematics we mentioned, including the impossibility of proving Euclid’s fifth postulate from the other four, impossibility of trisecting an angle with a straightedge and compass and the impossibility of solving a quintic equation via radicals. A geometric proof of the impossibility of angle trisection (one of the three geometric problems of antiquity, going back to the ancient greeks) is given in
this blog post of Tao. This book of Mario Livio covers some of the background and ideas behind these impossibility results.

1.5 Acknowledgements
Computation and Representation

“The alphabet was a great invention, which enabled men to store and to learn with little effort what others had learned the hard way—that is, to learn from books rather than from direct, possibly painful, contact with the real world.”, B.F. Skinner

“I found that every number, which may be expressed from one to ten, surpasses the preceding by one unit: afterwards the ten is doubled or tripled . . . until a hundred; then the hundred is doubled and tripled in the same manner as the units and the tens . . . and so forth to the utmost limit of numeration.”, Muhammad ibn Mūsā al-Khwārizmī, 820, translation by Fredric Rosen, 1831.

To a first approximation, computation can be thought of as a process that maps an input to an output.

![Diagram](image)

**Figure 2.1:** Our basic notion of computation is some process that maps an input to an output.
When discussing computation, it is important to separate the question of **what** is the task we need to perform (i.e., the specification) from the question of **how** we achieve this task (i.e., the implementation). For example, as we’ve seen, there is more than one way to achieve the computational task of computing the product of two integers.

In this lecture we focus on the **what** part, namely defining computational tasks. For starters, we need to define the inputs and outputs. A priori this seems nontrivial, since computation today is applied to a huge variety of objects. We do not compute merely on numbers, but also on texts, images, videos, connection graphs of social networks, MRI scans, gene data, and even other programs. We will represent all these objects as **strings of zeroes and ones**, that is objects such as 0011101 or 1011 or any other finite list of 1’s and 0’s.

![Figure 2.2](image)

**Figure 2.2:** We represent numbers, texts, images, networks and many other objects using strings of zeroes and ones. Writing the zeroes and ones themselves in green font over a black background is optional.

Today, we are so used to the notion of digital representation that we are not surprised by the existence of such an encoding. But it is a deep insight with significant implications. Many animals can convey a particular fear or desire, but what’s unique about humans is *language*: we use a finite collection of basic symbols to describe a potentially unlimited range of experiences. Language allows transmission of information over both time and space, and enables societies that span a great many people and accumulate a body of shared knowledge over time.

Over the last several decades, we’ve seen a revolution in what we are able to represent and convey in digital form. We can capture
experiences with almost perfect fidelity, and disseminate it essentially instantaneously to an unlimited audience. What’s more, once information is in digital form, we can compute over it, and gain insights from data that were not accessible in prior times. At the heart of this revolution is this simple but profound observation that we can represent an unbounded variety of objects using a finite set of symbols (and in fact using only the two symbols 0 and 1).¹

In later lectures, we will often fall back on taking this representation for granted, and hence write something like “program P takes x as input” when x might be a number, a vector, a graph, or any other objects, when we really mean that P takes as input the representation of x as a binary string. However, in this lecture, let us dwell a little bit on how such representations can be devised.

2.1 Examples of binary representations

In many instances, choosing the “right” string representation for a piece of data is highly nontrivial, and finding the “best” one (e.g., most compact, best fidelity, most efficiently manipulable, robust to errors, most informative features, etc.) is the object of intense research. But for now, let us start by describing some simple representations for various natural objects.

2.1.1 Representing natural numbers

Perhaps the simplest object we want to represent is a natural number. That is, a member x of the set N = {0,1,2,3,...}. We can represent a number x ∈ N as a string using the binary basis. Specifically, every natural number x can be written in a unique way as \( x = x_02^0 + x_12^1 + \cdots + x_{n-1}2^{n-1} \) (or \( \sum_{i=0}^{n-1} x_i2^i \) for short) where \( x_0, \ldots, x_{n-1} \) are zero/one and \( n \) is the smallest number such that \( 2^n > x \) (and hence \( x_{n-1} = 1 \) for every nonzero x). We can then represent x as the string \( (x_0, x_1, \ldots, x_{n-1}) \).² For example, the number 35 is represented as the string \( (1,1,0,0,1) \).³

We can think of a representation as consisting of encoding and decoding functions. In the case of the binary representation for integers, the encoding function \( E : \mathbb{N} \rightarrow \{0,1\}^* \) maps a natural number to the string representing it, and the decoding function \( D : \{0,1\}^* \rightarrow \mathbb{N} \) maps a string into the number it represents (i.e., \( D(x_0, \ldots, x_{n-1}) = 2^0x_0 + 2^1x_1 + \ldots + 2^{n-1}x_{n-1} \) for every \( x_0, \ldots, x_{n-1} \in \{0,1\} \)). For the representation to be well defined, we need every natural number to

¹ There is nothing “holy” about using zero and one as the basic symbols, and we can (indeed sometimes people do) use any other finite set of two or more symbols as the fundamental “alphabet”. We use zero and one in this course mainly because it simplifies notation.

² We can represent the number zero either as some string that contains only zeroes, or as the empty string. The choice will not make any difference for us.

³ Typically when people write down the binary representation, they would print the string x in reverse order, with the least significant digit as the rightmost one. Representing the number x as \( (x_{n-1}, x_{n-2}, \ldots, x_0) \) will of course work just as well (in some contexts this is known as “Big Endian” vs. “Little Endian” representation). We chose the particular representation above for the sake of simplicity, so the the i-th bit corresponds to \( 2^i \), but such low level choices will not make a difference in this course.
be represented by some string, where two distinct numbers must have distinct representations. This corresponds to requiring the encoding function to be one-to-one, and the decoding function to be onto.

If you don’t remember the definitions of one-to-one, onto, total and partial functions, now would be an excellent time to review them. Make sure you understand why the function E described above is one-to-one, and the function D is onto.

2.1.2 Representing (potentially negative) integers

Now that we can represent natural numbers, we can represent the full set of integers (i.e., members of the set \( \mathbb{Z} = \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\} \)) by adding one more bit that represents the sign. So, the string \((\sigma, x_0, \ldots, x_{n-1}) \in \{0, 1\}^{n+1}\) will represent the number

\[
(-1)^\sigma \left[x_02^0 + \cdots + x_{n-1}2^n\right]
\]

(2.1)

For this representation, the decoding function is not one-to-one: the two strings 1 and 0 both represent the number zero (since they can be thought of as representing −0 and +0 respectively, can you see why?). The decoding function is also only a partial function, since there is no number that is represented by the empty string. But this is still a fine representation, since the encoding function is the one-to-one total function \(E : \mathbb{Z} \to \{0, 1\}^*\) which maps an integer of the form \(a \times k\), where \(a \in \{\pm 1\}\) and \(k \in \mathbb{N}\) to the bit 
\((-1)^a\) concatenated with the binary representation of \(k\). That is, every integer can be represented as a string, and two distinct integers have distinct representations.

Interpretation and context: Given a string \(x \in \{0, 1\}^*\), how do we know if it’s “supposed” to represent a (nonnegative) natural number or a (potentially negative) integer? For that matter, even if we know \(x\) is “supposed” to be an integer, how do we know what representation scheme it uses? The short answer is that we don’t necessarily know this information, unless it is supplied from the context.\(^4\) We can treat the same string \(x\) as representing a natural number, an integer, a piece of text, an image, or a green gremlin. Whenever we say a sentence such as “let \(n\) be the number represented by the string \(x\)”, we will assume that we are fixing some canonical representation scheme such as the ones above. The choice of the particular representation

\(^4\) In programming language, the compiler or interpreter determines the representation of the sequence of bits corresponding to a variable’s type.
scheme will almost never matter, except that we want to make sure to stick with the same one for consistency.

2.1.3 Representing rational numbers

We can represent a rational number of the form $a/b$ by representing the two numbers $a$ and $b$ (again, this is not a unique representation but this is fine). However, simply concatenating the representations of $a$ and $b$ will not work. For example, recall that we represent 4 as $(0,1)$ and 35 as $(1,1,0,0,0,1)$, but the concatenation $(0,1,1,0,0,0,1)$ of these strings is also the concatenation of the representation $(0,1,1)$ of 6 and the representation $(1,0,0,0,1)$ of 17. Hence, if we used such simple concatenation then we would not be able to tell if the string $(0,1,1,0,0,0,1)$ is supposed to represent 4/35 or 6/17.

The way to tackle this is to find a general representation for pairs of numbers. If we were using a pen and paper, we would simply use a separator such as the semicolon symbol to represent, for example, the pair consisting of the numbers represented by $(0,1)$ and $(1,1,0,0,0,1)$ as the length-9 string $s_{01;110001}$. By adding a little redundancy, we can do just that in the digital domain. The idea is that we will map the three element set $\Sigma = \{0,1,;\}$ to the four element set $\{0,1\}^2$ via the one-to-one map that takes 0 to 00, 1 to 11 and ; to 01. In particular, if apply this map to every symbol of the length-9 string $s$ above, we get the length 18 binary string 001101111100000011. More generally, the above encoding yields a one-to-one map $E$ from strings over the alphabet $\Sigma$ to binary string, such that for every $s \in \Sigma^*$, $|E(s)| = 2|s|$.

Using this, we get get a one to one map $E' : (\{0,1\}^*) \times (\{0,1\}^*) \rightarrow \{0,1\}^*$ mapping pairs of binary strings into a single binary string. Given every pair $(a,b)$ of binary strings, we will first map it in a one-to-one way to a string $s \in \Sigma^*$ using ; as a separator, and then map $s$ to a single (longer) binary string using the encoding $E$. The same idea can be used to represent triples, quadruples, and generally all tuples of strings as a single string (can you see why?).

2.1.4 Representing real numbers

The set of real numbers $\mathbb{R}$ contains all numbers including positive, negative, and fractional, as well as irrational numbers such as $\pi$ or $e$. Every real number can be approximated by a rational number, and so up to a small error we can represent every real number $x$ by a
rational number $a/b$ that is very close to $x$. This is a fine representation though a more common choice to represent real numbers is the floating point representation, where we represent $x$ by the pair $(a, b)$ of integers of some prescribed sizes (determined by the desired accuracy) such that $a2^{-b}$ is closest to $x$. The reader might be (rightly) worried about this issue of approximation, but in many (though not all) computational applications, one can make the accuracy tight enough so that this does not affect the final result.\footnote{This is called “floating point” because we can think of the number $a$ as specifying a sequence of binary digits, and $b$ as describing the location of the “binary point” within this sequence. This internal representation is the reason why, for example, in Python typing $0.1 + 0.2$ will result in $0.30000000000000004$ and not $0.3$, see here, here and here for more. A floating point error has been implicated in the explosion of the Ariane 5 rocket, a bug that cost more than 370 million dollars, and the failure of a U.S. Patriot missile to intercept an Iraqi Scud missile, costing 28 lives. Floating point is often problematic in financial applications as well.} Also, some error in representing real numbers is unavoidable: there is no exact representation of real numbers as strings; see Exercise 2.8.\footnote{The reason for this inherent error is that the set of real numbers is uncountable as we will see later in this course.}

2.2 Beyond numbers

We can of course represent objects other than numbers as binary strings. Let us give a general definition for representation:

\begin{definition}[String representation]
Let $O$ be some set. A representation scheme for $O$ consists of a pair $(E, D)$ where $E : O \rightarrow \{0, 1\}^*$ is a total one-to-one function, $D : \{0, 1\}^* \rightarrow \mathcal{P} O$ is a (possibly partial) function, and such that $D$ and $E$ satisfy that $D(E(o)) = o$ for every $o \in O$. $E$ is known as the encoding function and $D$ is known as the decoding function.
\end{definition}

Note that the condition $D(E(o)) = o$ for every $o \in O$ implies that $D$ is onto. It turns out that to construct a representation scheme we only need to find an encoding function. That is, every one-to-one encoding function has a corresponding decoding function, as shown in the following lemma:

\begin{lemma}
Suppose that $E : O \rightarrow \{0, 1\}^*$ is one-to-one. Then there exists a function $D : \{0, 1\}^* \rightarrow O$ such that $D(E(o)) = o$ for every $o \in O$.
\end{lemma}

\begin{proof}
Let $o_0$ be some arbitrary element of $O$. For every $x \in \{0, 1\}^*$, there exists either zero or a single $o \in O$ such that $E(o) = x$ (otherwise $E$ would not be one-to-one). We will define $D(x)$ to equal $o_0$ in the first case and this single object $o$ in the second case. By definition $D(E(o)) = o$ for every $o \in O$. 
\end{proof}

Note that, while in general we allowed the decoding function to be partial, this proof shows that we can always obtain a total decoding function if we need to. This observation can sometimes be useful.
2.2.1 Prefix free encoding

In our discussion of the representation of rational numbers, we used the “hack” of encoding the alphabet \{0,1,\} to represent tuples of strings as a single string. This turns out to be a special case of the general paradigm of prefix free encoding. An encoding function \( E: \mathcal{O} \rightarrow \{0,1\}^* \) is prefix free if there are no two objects \( o \neq o' \) such that the representation \( E(o) \) is a prefix of the representation \( E(o') \). The definition of prefix is as you would expect: a length \( n \) string \( x \) is a prefix of a length \( n' \geq n \) string \( x' \) if \( x_i = x'_i \) for every \( 1 \leq i \leq n \). Given a representation scheme for \( \mathcal{O} \) with a prefix-free encoding map, we can use simple concatenation to encode tuples of objects in \( \mathcal{O} \):

**Theorem 2.2 — Prefix free implies tuple encoding.** Suppose that \( (E,D) \) is a representation scheme for \( \mathcal{O} \) and \( E \) is prefix free. Then there exists a representation scheme \( (E',D') \) for \( \mathcal{O}^* \) such that for every \( (o_0, \ldots, o_{k-1}) \in \mathcal{O}^* \), \( E'(o_0, \ldots, o_{k-1}) = E(o_0)E(o_1) \cdots E(o_{k-1}) \).

![Figure 2.3](image)

The idea behind the proof is simple. Suppose that for example we want to decode a triple \( (o_0, o_1, o_2) \) from its representation \( x = E'(o_0, o_1, o_2) = E(o_0)E(o_1)E(o_2) \). We will do so by first finding the first prefix \( x_0 \) of \( x \) such is a representation of some object. Then we will decode this object, remove \( x_0 \) from \( x \) to obtain a new string \( x' \), and continue onwards to find the first prefix \( x_1 \) of \( x' \) and so on and so forth (see Exercise 2.5). The prefix-freeness property of \( E \) will ensure that \( x_0 \) will in fact be \( E(o_0) \), \( x_1 \) will be \( E(o_1) \) etc. We now show the formal proof.

**Proof of Theorem 2.2.** By Lemma 2.1, to prove the theorem it suffices show that \( E' \) is one-to-one. Suppose, towards the sake of
contradiction that there exist two distinct tuples \((o_0, \ldots, o_{k-1})\) and \((o'_0, \ldots, o'_{k'-1})\) such that

\[
E'(o_0, \ldots, o_{k-1}) = E'(o'_0, \ldots, o'_{k'-1}), \tag{2.2}
\]

and denote this string by \(x\).

We denote \(x_i = E(o_i)\) and \(x'_i = E'(o_i)\). By our assumption and the definition of \(E'\), \(x_0x_1 \cdots x_k = x'_0x'_1 \cdots x'_{k'}\).

Let’s make the assumption A that there is some index \(i \in \{\min \{k, k'\}\}\) such that \(o_i \neq o'_i\). Now, let \(i\) be the first such index, and hence \(o_j = o'_j\) for all \(j < i\) but \(o_i \neq o'_i\), and so (since \(E\) is one-to-one) also \(x_i \neq x'_i\). That means that if we write \(p = x_0 \cdots x_{j-1}\), then by Eq. (2.2) the first \(|p| + |x_i|\) bits of the string \(x\) need to equal \(px_i\) and the first \(|p| + |x'_i|\) bits of \(x\) need to equal \(px'_i\). If \(|x_i| = |x'_i|\), then the only way this can happen is if \(px_i = px'_i\), which implies \(x_i = x'_i\), in contradiction to the fact that \(E\) is one-to-one. Otherwise, without loss of generality \(|x_i| > |x'_i|\) and so \(px'_i\) must be a prefix of \(px_i\), but this contradicts the prefix-freeness of \(E\). The only remaining case is when Assumption A is false. Since the tuples are different, if \(o_i = o'_i\) for every \(i \in \{\min \{k, k'\}\}\), it must mean that \(k \neq k'\).

Without loss of generality, assume that \(k < k'\) (again, check that this is justified!). But then, since \(o_i = o'_i\) for all \(i \in [k]\), it holds that \(x_0 \cdots x_{k-1} = x_0 \cdots x_{k-1} x'_k \cdots x'_{k'}\). But this can only happen if \(x'_i\) for \(i \geq k\) is the empty string, while a prefix-free encoding can never encode an object as the empty string (can you see why?). ■

2.2.2 Making representations prefix free

Some natural representations are prefix free. For example, every fixed output length representation (i.e., one-to-one function \(E : O \rightarrow \{0, 1\}^n\)) is automatically prefix free, since a string \(x\) can only be a prefix of an equal-length \(x'\) if \(x\) and \(x'\) are identical. Moreover, the approach we used for representing rational numbers can be used to show the following:

**Lemma 2.3** Let \(E : O \rightarrow \{0, 1\}^*\) be a one-to-one function. Then there is a one-to-one prefix-free encoding \(E'\) such that \(|E'(o)| \leq 2|o|\) for every \(o \in O\).

In fact, we can even obtain a more efficient transformation where \(|E'(o)| \leq |o| + O(\log |o|)\). We leave proving this as an exercise (see Exercise 2.6).
2.2.3 Representing letters and text

We can represent a letter or symbol by a string, and then if this representation is prefix free, we can represent a sequence of symbols by simply concatenating the representation of each symbol. One such representation is the ASCII that represents 128 letters and symbols as strings of 7 bits. Since it is a fixed-length representation it is automatically prefix free (can you see why?). Unicode is a representation of (at the time of this writing) about 128,000 symbols into numbers (known as code points) between 0 and 1,114,111. There are several types of prefix-free representations of the code points, a popular one being UTF-8 that encodes every codepoint into a string of length between 8 and 32.

2.2.4 Representing vectors, matrices, images, graphs

Once we can represent numbers, and lists of numbers, then we can obviously represent vectors (which are just lists of numbers). Similarly, we can represent lists of lists and so in particular matrices. To represent an image, we can represent the color at each pixel by a list of three numbers corresponding to the intensity of Red, Green and Blue. Thus an image of \( n \) pixels would be represented of a list of \( n \) such length-three lists. A video can be represented as a list of images.*

A graph on \( n \) vertices can be represented as an \( n \times n \) matrix whose \((i,j)\)th entry is equal to 1 if the edge \((i,j)\) is present and is equal to 0 otherwise. (We can also represent a graph using the so-called “adjacency list” representation, though the difference between these two representations will almost never matter for this course.)

2.2.5 Representing lists

If we have a way of represent objects from a set \( \mathcal{O} \) as binary strings, then we can represent lists of these objects ny applying a prefix-free transformation. Moreover, we can use a trick similar to the above to handle nested lists. The idea is that if we have some representation \( E: \mathcal{O} \to \{0,1\}^* \), then we can represent nested lists of items from \( \mathcal{O} \) using strings over the five element alphabet \( \Sigma = \{0,1[,]\,\} \). For example, if \( o_1 \) is represented by \( \theta011 \), \( o_2 \) is represented by \( 10011 \), and \( o_3 \) is represented by \( \theta011 \), then we can represent the nested list \((o_1, (o_2, o_3))\) as the string “[0011, [1011, \theta0111]]” over the alphabet \( \Sigma \). By encoding every element of \( \Sigma \) itself as a three-bit string, we can...
transform any representation for objects \( O \) into a representation that allows to represent (potentially nested) lists of these objects.

### 2.2.6 Notation

We will typically identify an object with its representation as a string. For example, if \( F : \{0, 1\}^* \to \{0, 1\}^* \) is some function that maps strings to strings and \( x \) is an integer, we might make statements such as “\( F(x) + 1 \) is prime” to mean that if we represent \( x \) as a string \( \bar{x} \) and let \( y = F(\bar{x}) \), then the integer \( y \) represented by the string \( \bar{y} \) satisfies that \( y + 1 \) is prime. (You can see how this convention of identifying objects with their representation can save us a lot of cumbersome formalism.) Similarly, if \( x, y \) are some objects and \( F \) is a function that takes strings as inputs, then by \( F(x, y) \) we will mean the result of applying \( F \) to the representation of the order pair \( (x, y) \). We will use the same notation to invoke functions on \( k \)-tuples of objects for every \( k \).

This convention of identifying an object with its representation as a string is one that we humans follow all the time. For example, when people say a statement such as “17 is a prime number”, what they really mean is that the integer whose decimal representation is the string “17”, is prime.

### 2.3 Defining computational tasks

Abstractly, a computational process is some process that takes an input which is a string of bits, and produces an output which is a string of bits. This transformation of input to output can be done using a modern computer, a person following instructions, the evolution of some natural system, or any other means.

![Figure 2.4: A computational process](image-url)
In future lectures, we will turn to mathematically defining computational process, but, as we discussed above for now we want to focus on computational tasks; i.e., focus on the specification and not the implementation. Again, at an abstract level, a computational task can specify any relation that the output needs to have with the input. But for most of this course, we will focus on the simplest and most common task of computing a function. Here are some examples:

- Given (a representation) of two integers \( x, y \), compute the product \( x \times y \). Using our representation above, this corresponds to computing a function from \( \{0, 1\}^* \) to \( \{0, 1\}^* \). We’ve seen that there is more than one way to solve this computational task, and in fact, we still don’t know the best algorithm for this problem.

- Given (a representation of) an integer \( z \), compute its factorization; i.e., the list of primes \( p_1 \leq \cdots \leq p_k \) such that \( z = p_1 \cdots p_k \). This again corresponds to computing a function from \( \{0, 1\}^* \) to \( \{0, 1\}^* \). The gaps in our knowledge of the complexity of this problem are even longer.

- Given (a representation of) a graph \( G \) and two vertices \( s \) and \( t \), compute the length of the shortest path in \( G \) between \( s \) and \( t \), or do the same for the longest path (with no repeated vertices) between \( s \) and \( t \). Both these tasks correspond to computing a function from \( \{0, 1\}^* \) to \( \{0, 1\}^* \), though it turns out that there is a huge difference in their computational difficulty.

- Given the code of a Python program, is there an input that would force it into an infinite loop. This corresponds to computing a partial function from \( \{0, 1\}^* \) to \( \{0, 1\} \); though it is easy to make it into a total function by mapping every string into the trivial Python program that stops without doing anything. We will see that we do understand the computational status of this problem, but the answer is quite surprising.

- Given (a representation of) an image \( I \), decide if \( I \) is a photo of a cat or a dog. This correspond to computing some (partial) function from \( \{0, 1\}^* \) to \( \{0, 1\} \).

An important special case of computational tasks corresponds to computing Boolean functions, whose output is a single bit \( \{0, 1\} \). Computing such functions corresponds to answering a YES/NO question, and hence this task is also known as a decision problem. Given any function \( F : \{0, 1\}^* \to \{0, 1\} \) and \( x \in \{0, 1\}^* \), the task of computing \( F(x) \) corresponds to the task of deciding whether or not \( x \in L \) where \( L = \{ x : F(x) = 1 \} \) is known as the language that corresponds to the function \( F \).\(^{11} \) Hence many texts refer to such as

\(^{11} \) The language terminology is due to historical connections between the theory of computation and formal linguistics as developed by Noam Chomsky.
computational task as \textit{deciding a language}.

For every particular function \( F \), there can be several possible \textit{algorithms} to compute \( F \). We will be interested in questions such as:

- For a given function \( F \), can it be the case that \textit{there is no algorithm} to compute \( F \)?

- If there is an algorithm, what is the best one? Could it be that \( F \) is “effectively uncomputable” in the sense that every algorithm for computing \( F \) requires a prohibitively large amount of resources?

- If we can’t answer this question, can we show equivalence between different functions \( F \) and \( F' \) in the sense that either they are both easy (i.e., have fast algorithms) or they are both hard?

- Can a function being hard to compute ever be a \textit{good thing}? Can we use it for applications in areas such as cryptography?

In order to do that, we will need to mathematically define the notion of an \textit{algorithm}, which is what we’ll do in the next lecture.

\section*{2.3.1 Advanced note: beyond computing functions}

Functions capture quite a lot of computational tasks, but one can consider more general settings as well. For starters, we can and will talk about \textit{partial} functions, that are not defined on all inputs. When computing a partial function, we only need to worry about the inputs on which the function is defined. Another way to say it is that we can design an algorithm for a partial function \( F \) under the assumption that someone “promised” us that all inputs \( x \) would be such that \( F(x) \) is defined (as otherwise we don’t care about the result). Hence such tasks are also known as \textit{promise problems}.

Another generalization is to consider \textit{relations} that may have more than one possible admissible output. For example, consider the task of finding any solution for a given set of equation. A relation \( R \) maps a string \( x \in \{0,1\}^* \) into a \textit{set of strings} \( R(x) \) (for example, \( x \) might describe a set of equations, in which case \( R(x) \) would correspond to the set of all solutions to \( x \)). We can also identify a relation \( R \) with the set of pairs of strings \((x,y)\) where \( y \in R(x) \). A computational process solves a relation if for every \( x \in \{0,1\}^* \), it outputs some string \( y \in R(x) \).

Later on in this course we will consider even more general tasks, including \textit{interactive} tasks, such as finding good strategy in a game, tasks defined using probabilistic notions, and others. However,
for much of this course we will focus on the task of computing a function, and often even a Boolean function, that has only a single bit of output. It turns out that a great deal of the theory of computation can be studied in the context of this task, and the insights learned are applicable in the more general settings.

2.4 Lecture summary

• We can represent essentially every object we want to compute on using binary strings.

• A representation scheme for a set of objects \( \mathcal{O} \) is a one-to-one map from \( \mathcal{O} \) to \( \{0, 1\}^* \).

• A basic computational task is the task of computing a function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \). This encompasses not just arithmetical computations such as multiplication, factoring, etc. but a great many other tasks arising in areas as diverse as scientific computing, artificial intelligence, image processing, data mining and many many more.

• We will study the question of finding (or at least giving bounds on) what is the best algorithm for computing \( F \) for various interesting functions \( F \).

2.5 Exercises

Exercise 2.1 Which one of these objects can be represented by a binary string?

a. An integer \( x \)

b. An undirected graph \( G \).

c. A directed graph \( H \)

d. All of the above.

Exercise 2.2 — Multiplying in different representation. Recall that the grade-school algorithm for multiplying two numbers requires \( O(n^2) \) operations. Suppose that instead of using decimal representation, we use one of the following representations \( R(x) \) to represent a number \( x \) between 0 and \( 10^n - 1 \). For which one of these representations you can still multiply the numbers in \( O(n^2) \) operations?
a. The standard binary representation: \( B(x) = (x_0, \ldots, x_k) \) where \( x = \sum_{i=0}^{k} x_i 2^i \) and \( k \) is the largest number s.t. \( x \geq 2^k \).

b. The reverse binary representation: \( B(x) = (x_k, \ldots, x_0) \) where \( x_i \) is defined as above for \( i = 0, \ldots, k - 1 \).

c. Binary coded decimal representation: \( B(x) = (y_0, \ldots, y_{n-1}) \) where \( y_i \in \{0, 1\}^4 \) represents the \( i^{th} \) decimal digit of \( x \) mapping 0 to 0000, 1 to 0001, 2 to 0010, etc. (i.e. 9 maps to 1001)

d. All of the above.

**Exercise 2.3** Suppose that \( R : \mathbb{N} \to \{0, 1\}^* \) corresponds to representing a number \( x \) as a string of \( x \) 1’s, (e.g., \( R(4) = 1111, R(7) = 1111111, \) etc.). If \( x, y \) are numbers between 0 and \( 10^n - 1 \), can we still multiply \( x \) and \( y \) using \( O(n^2) \) operations if we are given them in the representation \( R(\cdot) \)?

**Exercise 2.4** Recall that if \( F \) is a one-to-one and onto function mapping elements of a finite set \( U \) into a finite set \( V \) then the sizes of \( U \) and \( V \) are the same. Let \( B : \mathbb{N} \to \{0, 1\}^* \) be the function such that for every \( x \in \mathbb{N} \), \( B(x) \) is the binary representation of \( x \).

a. Prove that \( x < 2^k \) if and only if \( |B(x)| \leq k \).

b. Use a. to compute the size of the set \( \{y \in \{0, 1\}^* : |y| \leq k\} \) where \( |y| \) denotes the length of the string \( y \).

c. Use a. and b. to prove that \( 2^k - 1 = 1 + 2 + 4 + \cdots + 2^{k-1} \).

**Exercise 2.5** — Prefix-free encoding of tuples. Suppose that \( F : \mathbb{N} \to \{0, 1\}^* \) is some (not necessarily prefix free) representation of the objects in the set \( O \), and \( G : \mathbb{N} \to \{0, 1\}^* \) is a prefix-free representation of the natural numbers. Define \( F'(o) = G(|F(o)|)F(o) \) (i.e., the concatenation of the representation of the length \( F(o) \) and \( F(o) \)).

a. Prove that \( F' \) is a prefix-free representation of \( O \).

b. Show that we can transform any representation to a prefix-free one by a modification that takes a \( k \) bit string into a string of
Exercise 2.7 — Kraft’s Inequality. Suppose that $S \subseteq \{0,1\}^n$ is some finite prefix-free set.

a. For every $k \leq n$ and length-$k$ string $x \in S$, let $L(x) \subseteq \{0,1\}^n$ denote all the length-$n$ strings whose first $k$ bits are $x_0, \ldots, x_{k-1}$. Prove that (1) $|L(x)| = 2^{n-|x|}$ and (2) if $x \neq x'$ then $L(x)$ is disjoint from $L(x')$.

b. Prove that $\sum_{x \in S} 2^{-|x|} \leq 1$.

c. Prove that there is no prefix-free encoding of strings with less than logarithmic overhead. That is, prove that there is no function $PF : \{0,1\}^* \to \{0,1\}^*$ s.t. $|PF(x)| \leq |x| + 0.9 \log |x|$ for every $x \in \{0,1\}^*$ and such that the set $\{PF(x) : x \in \{0,1\}^*\}$ is prefix-free.

Exercise 2.8 — No lossless representation of reals (challenge). In this exercise we will prove that there is no “lossless” representation of real numbers as strings. That is, that there is no one-to-one function $F$ mapping the real numbers $\mathbb{R}$ to the set of finite strings $\{0,1\}^*$.

a. Suppose, towards the sake of contradiction, that there exists such a function $F : \mathbb{R} \to \{0,1\}^*$. Prove that there exists an onto function $G : \{0,1\}^* \to \mathbb{R}$.

b. Prove that there is an onto function $G' : \mathbb{N} \to \{0,1\}^*$. Conclude that if there is an onto function $G : \{0,1\}^* \to \mathbb{R}$ then there exists an onto function $H : \mathbb{N} \to \mathbb{R}$.

c. For any real number $x$ and $i > 0$, define $D(x,i) \in \{0,\ldots,9\}$ to be the $i^{th}$ decimal digit following the decimal point of $x$. That is, $D(x,i)$ is the remainder when we divide $[10x]$ by 10. For example $D(1/4,1) = 2$, $D(1/4,2) = 5$ and $D(1/4,3) = 0$ for every $i > 2$. Similarly, $D(1/3,i) = 3$ for every $i$. Prove that if $x$ is between 0 and 1 then $x = \sum_{i=1}^{\infty} 10^{-i}D(x,i)$.

d. Let $S$ be the set of all functions from $\mathbb{N}$ to $\{0,1\}$ that maps a number $x$ to the function $i \mapsto D(x,i)$ (mod 2) is onto.

e. Prove that there is no onto map from $\mathbb{N}$ to $S$.

f. Combine a-e to get a contradiction to the assumption that there is one-to-one map from $\mathbb{R}$ to $\{0,1\}^*$.

Hint: Think recursively how to represent the length of the string.

**Note:** If you are not familiar with infinite series, start by showing that this is true for the case where $x$ has finite decimal expansion, namely that there is some $n$ such that $D(x,i) = 0$ for all $i > n$. Formally, what you need to prove for the infinite case is that for every $\epsilon > 0$, there is some $n$ such that $|x - \sum_{i=1}^{n} 10^{-i}D(x,i)| < \epsilon$.

Hint: Show that for every function $f : \mathbb{N} \to \{0,1\}$, the number $x = \sum_{i=0}^{\infty} 10^{-i-1}f(i)$ satisfies $D(x) = f$.

Hint: Suppose that there was such a map $O$, the we can define the function $f \in S$ such that $f(i) = 1 - O(i)$ and show that it is not in the image of $O$.

TODO: can we have a proof that doesn’t need people to know limits?
2.6 Bibliographical notes

The idea that we should separate the definition or specification of a function from its implementation or computation might seem “obvious”, but it took some time for mathematicians to arrive at this viewpoint. Historically, a function \( F \) was identified by rules or formulas showing how to derive the output from the input. As we discuss in greater depth in our lecture on uncomputability, in the 1800’s this somewhat informal notion of a function started “breaking at the seams” and eventually mathematicians arrived at the more rigorous definition of a function as an arbitrary assignment of input to outputs. While many functions may be described (or computed) by one or more formulas, today we do not consider that to be an essential property of functions, and also allow functions that do not correspond to any “nice” formula.

2.7 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

- **Succinct** data structures. These are representations that map objects from some set \( O \) into strings of length not much larger than the minimum of \( \log_2 |O| \) but still enable fast access to certain queries, see for example this paper.

- We’ve mentioned that all representations of the real numbers are inherently approximate. Thus an important endeavor is to understand what guarantees we can offer on the approximation quality of the output of an algorithm, as a function of the approximation quality of the inputs. This is known as the question of numerical stability.

- The linear algebraic view of graphs. The adjacency matrix representation of graphs is not merely a convenient way to map a graph into a binary string, but it turns out that many natural notions and operations on matrices are useful for graphs as well. (For example, Google’s PageRank algorithm relies on this viewpoint.) The notes of this course are an excellent source for this area, known as spectral graph theory. We might discuss this view much later in this course when we talk about random walks.
2.8 Acknowledgements
Defining computation

“There is no reason why mental as well as bodily labor should not be economized by the aid of machinery”, Charles Babbage, 1852

“If, unwarned by my example, any man shall undertake and shall succeed in constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results.”, Charles Babbage, 1864

“To understand a program you must become both the machine and the program.”, Alan Perlis, 1982

People have been computing for thousands of years, with aids that include not just pen and paper, but also abacus, slide rulers, various mechanical devices, and modern electronic computers. A priori, the notion of computation seems to be tied to the particular mechanism that you use. You might think that the “best” algorithm for multiplying numbers will differ if you implement it in Python on a modern laptop than if you use pen and paper. However, as we saw in the introduction, an algorithm that is asymptotically better would eventually beat a worse one regardless of the underlying technology. This gives us hope for a technology independent way of defining computation, which is what we will do in this lecture.
Figure 3.1: Calculating wheels by Charles Babbage. Image taken from the Mark I 'operating manual'.

Figure 3.2: A 1944 Popular Mechanics article on the Harvard Mark I computer.
3.1 Defining computation

The name “algorithm” is derived from the Latin transliteration of Muhammad ibn Musa al-Khwarizmi, who was a Persian scholar during the 9th century whose books introduced the western world to the decimal positional numeral system, as well as the solutions of linear and quadratic equations (see Fig. 3.4). Still his description of the algorithms were rather informal by today’s standards. Rather than use “variables” such as $x, y$, he used concrete numbers such as 10 and 39, and trusted the reader to be able to extrapolate from these examples.¹

Indeed, extrapolation from examples is still the way most of us first learn algorithms such as addition and multiplication, see Fig. 3.3.

For concreteness we will sometimes include code of actual programming languages in these notes. However, these will be simple enough to be understandable even by people that are not familiar with these languages.

Here is how al-Khwarizmi described how to solve an equation of the form $x^2 + bx = c$:²

```python
def solve_eq(b, c):
    # return solution of $x^2 + bx = c$ using Al Khwarizmi's instructions
    val1 = b/2.0  # halve the number of the roots
    val2 = val1*val1  # this you multiply by itself
    val3 = val2 + c  # Add this to thirty-nine (c)
    val4 = math.sqrt(val3)  # take the root of this
    val5 = val4 - val1  # subtract from it half the number of roots
```

¹ Translation from “The Algebra of Ben-Musa”, Fredric Rosen, 1831.

² For concreteness we will sometimes include code of actual programming languages in these notes. However, these will be simple enough to be understandable even by people that are not familiar with these languages.
Figure 3.3: An explanation for children of the two digit addition algorithm

Figure 3.4: Text pages from Algebra manuscript with geometrical solutions to two quadratic equations. Shelfmark: MS. Huntingdon 214 fol. 004v-005r
3.2 The NAND Programming language

We can try to use a modern programming language such as Python or C for our formal model of computation, but it would be quite hard to reason about, given that the Python language reference has more than 100 pages. Thus we will define computation using an extremely simple “programming language”: one that has only a single operation. This raises the question of whether this language is rich enough to capture the power of modern computing systems. We will see that (to a first approximation), the answer to this question is Yes.

We start by defining a programming language that can only compute finite functions. That is, functions $F$ that map $\{0, 1\}^n$ to $\{0, 1\}^m$ for some natural numbers $m, n$. Later we will discuss how to extend the language to allow for a single program that can compute a function of every length, but the finite case is already quite interesting and will give us a simple setting for exploring some of the salient features of computing.

The NAND programming language has no loops, functions, or if statements. It has only a single operation: NAND. That is, every line in a NAND program has the form:

```
foo := bar NAND baz
```

where $foo$, $bar$, $baz$ are variable names. When this line is executed, the variable $foo$ is assigned the negation of the logical AND of (i.e., the NAND operation applied to) the values of the two variables $bar$ and $baz$.

All variables in the NAND programming language are Boolean: can take values that are either zero or one. Variables such as $x_{22}$ or $y_{18}$ (that is, of the form $x_\langle i \rangle$ or $y_\langle i \rangle$ where $i$ is a natural number) have a special meaning. The variables beginning with $x_\_$ are input variables and those beginning with $y_\_$ are output variables. Thus for example the following four line NAND program takes an input of two bits and outputs a single bit:

```
u := x_0 NAND x_1
v := x_0 NAND u
w := x_1 NAND u
y_0 := v NAND w
```

\[ The \textit{terms} foo \textit{and} bar \textit{are often used} to describe generic variable names in the context of programming, and we will follow this convention throughout the course. See the appendix and the website \url{http://nandpl.org} for a full specification of the NAND programming language. \]

\[ The \textit{logical AND} of two bits $x, x' \in \{0, 1\}$ is equal to 1 if $x = x' = 1$ and is equal to 0 otherwise. Thus its negation satisfies NAND(0, 0) = NAND(0, 1) = NAND(1, 0) = 1, while NAND(1, 1) = 0. If a variable hasn’t been assigned a value, then its default value is zero. \]

\[ In these lecture notes, we use the convention that when we write $\langle e \rangle$ then we mean the numerical value of this expression. So for example if $k = 10$ then we can write $x_\langle k + 7 \rangle$ to mean $x_{17}$. This is just for the notes: in the NAND programming language itself the indices have to be absolute numerical constants. \]
At present the web interface is not yet implemented, and you can run NAND program using an OCaml interpreter that you can download from that website. The implementation is in a fluid state and so the text below might not exactly match the output of the interpreter.

Can you guess what function from \(\{0,1\}^2\) to \(\{0,1\}\) this program computes? It might be a good idea for you to pause here and try to figure this out.

To find the function that this program computes, we can run it on all the four possible two bit inputs: 00, 01, 10, and 11.

For example, let us consider the execution of this program on the input 00, keeping track of the values of the variables as the program runs line by line. On the website http://nandpl.org we can run NAND programs in a “debug” mode, which will produce an execution trace of the program. When we run the program above on the input 01, we get the following trace:

```
Executing step 1: "u := x_0 NAND x_1" x_0 = 0, x_1 = 1, u is assigned 1,
Executing step 2: "v := x_0 NAND u" x_0 = 0, u = 1, v is assigned 1,
Executing step 3: "w := x_1 NAND u" x_1 = 1, u = 1, w is assigned 0,
Executing step 4: "y_0 := v NAND w" v = 1, w = 0, y_0 is assigned 1,
Output is y_0=1
```

On the other hand if we execute this program on the input 11, then we get the following execution trace:

```
Executing step 1: "u := x_0 NAND x_1" x_0 = 1, x_1 = 1, u is assigned 0,
Executing step 2: "v := x_0 NAND u" x_0 = 1, u = 0, v is assigned 1,
Executing step 3: "w := x_1 NAND u" x_1 = 1, u = 0, w is assigned 1,
Executing step 4: "y_0 := v NAND w" v = 1, w = 1, y_0 is assigned 0,
Output is y_0=0
```

You can verify that on input 10 the program will also output 1, while on input 00 it will output zero. Hence the output of this program on every input is summarized in the following table:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

In other words, this program computes the exclusive or (also known as XOR) function.
### Adding one-bit numbers

Now that we can compute XOR, let us try something just a little more ambitious: adding a pair of one-bit numbers. That is, we want to compute the function $ADD_1 : \{0,1\}^2 \to \{0,1\}^2$ such that $ADD(x_0,x_1)$ is the binary representation of the addition of the two numbers $x_0$ and $x_1$. Since the sum of two 0/1 values is a number in $\{0,1,2\}$, the output of the function $ADD_1$ is of length two bits.

If we write the sum $x_0 + x_1$ as $y_02^0 + y_12^1$ then the table of values for $ADD_1$ is the following:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

One can see that $y_0$ will be the XOR of $x_0$ and $x_1$ and $y_1$ will be the AND of $x_0$ and $x_1$. Thus we can compute one bit variable addition using the following program:

```plaintext
// Add two single-bit numbers
u := x_0 NAND x_1
v := x_0 NAND u
w := x_1 NAND u
y_0 := v NAND w
y_1 := u NAND u
```

If we run this program on the input $(1,1)$ we get the execution trace:

-Executing step 1: "u := x_0 NAND x_1" $x_0 = 1$, $x_1 = 1$, $u$ is assigned 0,
-Executing step 2: "v := x_0 NAND u" $x_0 = 1$, $u = 0$, $v$ is assigned 1,
-Executing step 3: "w := x_1 NAND u" $x_1 = 1$, $u = 0$, $w$ is assigned 1,
As mentioned in the appendix, we require that all output variables are assigned a value, and that the largest index used in an \( L \) line NAND program is smaller than \( L \), and so all functions in \( \text{SIZE}(L) \) have at most \( L \) inputs and \( L \) outputs.

Executing step 4: "\( y_0 := v \text{ NAND } w \)" \( v = 1, w = 1 \), \( y_0 \) is assigned 0,

Executing step 5: "\( y_1 := u \text{ NAND } u \)" \( u = 0, u = 0 \), \( y_1 \) is assigned 1,

Output is \( y_0 = 0, y_1 = 1 \)

and so you can see that the output \((0, 1)\) is indeed the binary encoding of \( 1 + 1 = 2 \).

3.2.2 Formal definitions

We define the notion of computing a function by a NAND program in the natural way:

\[ \text{Definition 3.1 — Computing a function.} \]
\[ \text{The number of inputs in a NAND program } P \text{ is the largest number } n \text{ such that } P \text{ contains a variable of the form } x_{n-1}, \text{ and the number of outputs is the largest number } m \text{ such that } P \text{ contains a variable of the form } y_{m-1}. \]

Let \( F : \{0, 1\}^n \rightarrow \{0, 1\}^m \). A NAND program \( P \) with \( n \) inputs and \( m \) outputs computes \( F \) if for every \( x \in \{0, 1\}^n \), whenever \( P \) is executed with the \( x_{\langle i \rangle} \) variable initialized to \( x_i \) for all \( i \in [n] \), at the end of the execution the variable \( y_{\langle j \rangle} \) will equal \( y_j \) for all \( j \in [m] \) where \( y = F(x) \).

For every \( L \in \mathbb{N} \), we define \( \text{SIZE}(L) \) to be the set of all functions that are computable by a NAND program of at most \( L \) lines.

\[ \text{Please pause here and verify why Definition 3.1 does indeed capture the natural notion of computing a function by a NAND program.} \]

Let \( \text{XOR}_n : \{0, 1\}^n \rightarrow \{0, 1\} \) be the function that maps \( x \in \{0, 1\}^n \) to \( \sum_{i=0}^n x_i \pmod{2} \). The NAND program we presented above yields a proof of the following theorem

\[ \text{Theorem 3.1 — Computing XOR.} \quad \text{XOR}_2 \in \text{SIZE}(4) \]

Similarly, the addition program we presented shows that \( \text{ADD}_1 \in \text{SIZE}(5) \).
3.3 Composing functions

Computing the XOR or addition of two bits is all well and good, but still seems a long way off from even the algorithms we all learned in elementary school, let alone *World of Warcraft*. We will get to computing more interesting functions, but for starters let us prove the following simple extension of Theorem 3.1.

**Theorem 3.2** — Computing four bit parity. \( \text{XOR}_4 \in \text{SIZE}(12) \)

We can prove Theorem 3.2 by explicitly writing down a 12 line program. But writing NAND programs by hand can get real old real fast. So, we will prove more general results about composing functions:

**Theorem 3.3** — Sequential composition of functions. If \( F : \{0,1\}^n \to \{0,1\}^m \) is a function in \( \text{SIZE}(L) \) and \( G : \{0,1\}^m \to \{0,1\}^k \) is a function in \( \text{SIZE}(L') \) then \( G \circ F \) is a function in \( \text{SIZE}(L + L') \), where \( G \circ F : \{0,1\}^n \to \{0,1\}^k \) is the function that maps \( x \in \{0,1\}^n \) to \( G(F(x)) \).

**Theorem 3.4** — Parallel composition of functions. If \( F : \{0,1\}^n \to \{0,1\}^m \) is a function in \( \text{SIZE}(L) \) and \( G : \{0,1\}^{n'} \to \{0,1\}^{m'} \) is a function in \( \text{SIZE}(L) \) then \( F \oplus G \) is a function in \( \text{SIZE}(L + L') \), where \( F \oplus G : \{0,1\}^{n+n'} \to \{0,1\}^{m+m'} \) is the function that maps \( x \in \{0,1\}^{n+n'} \) to \( F(x_0, \ldots, x_{n-1})G(x_n, \ldots, x_{n+n'-1}) \).

Before proving Theorem 3.3 and Theorem 3.4, note that they do imply Theorem 3.2. Indeed, it’s easy to verify that for every \( x \in \{0,1\}^4 \),

\[
\text{XOR}_4(x) = \sum_{i=0}^{3} x_i (\ mod \ 3) = ((x_0 + x_1 \mod 2) + (x_2 + x_3 \mod 2) \mod 2) = \text{XOR}_2(\text{XOR}_2(x_0, x_1) \text{XOR}_2(x_2, x_3))
\]

and hence

\[
\text{XOR}_4 = \text{XOR}_2 \circ (\text{XOR}_2 \oplus \text{XOR}_2). \tag{3.1}
\]

Since \( \text{XOR}_2 \) is in \( \text{SIZE}(4) \), it follows that \( \text{XOR}_4 \in \text{SIZE}(4 + (4 + 4)) = \text{SIZE}(12) \).
Using the same idea we can prove the following more general result:

**Theorem 3.5** — *Computing parity via NAND circuits.* For every $n > 1$, $\text{XOR}_n \in \text{SIZE}(10n)$

We leave proving Theorem 3.5 as Exercise 4.5.

### 3.4 Representing programs as graphs

We can prove Theorem 3.3 and Theorem 3.4 by directly arguing that we can “copy and paste” the code for $F$ and $G$ to achieve a program that computes $F \circ G$ and $F \oplus G$ respectively. However, we will use a more general approach, first giving a more “mathematical” representation for NAND programs as *graphs*, and then using this representation to prove these two theorems.

#### Definition 3.2 — NAND circuit

A *NAND circuit* with $n$ inputs and $m$ outputs is a labeled directed acyclic graph (DAG) in which every vertex has in-degree at most two. We require that there are $n$ vertices with in-degree zero, known as *input variables*, that are labeled with $x_{\langle i \rangle}$ for $i \in [n]$. Every vertex apart from the input variables is known as a *gate*. We require that there are $m$ vertices of out-degree zero, denoted as the *output gates*, and that are labeled with $y_{\langle j \rangle}$ for $j \in [m]$. While not all vertices are labeled, no two vertices get the same label. We denote the circuit as $C = (V, E, L)$ where $V, E$ are the vertices and edges of the circuit, and $L : V \to \gamma S$ is the (partial) one-to-one labeling function that maps vertices into the set $S = \{ x_0, \ldots, x_{\langle n - 1 \rangle}, y_0, \ldots, y_{\langle m - 1 \rangle} \}$. The *size* of a circuit $C$, denoted by $|C|$, is the number of gates that it contains.

The definition of NAND circuits is not ultimately that complicated, but may take a second or third read to fully parse. It might help to look at Fig. 3.5, which describes the NAND circuit that corresponds to the 4-line NAND program we presented above for the $\text{XOR}_2$ function.
A NAND circuit corresponds to computation in the following way. To compute some output on an input $x \in \{0, 1\}^n$, we start by assigning to the input vertex labeled with $x_{\langle i \rangle}$ the value $x_i$, and then proceed by assigning for every gate $v$ the value that is the NAND of the values assigned to its in-neighbors (if it has less than two in-neighbors, we replace the value of the missing neighbors by zero). The output $y \in \{0, 1\}^m$ corresponds to the value assigned to the output gates, with $y_j$ equal to the value assigned to the value assigned to the gate labeled $y_{\langle j \rangle}$ for every $j \in [m]$. Formally, this is defined as follows:

**Definition 3.3 — Computing a function by a NAND circuit.** Let $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ and let $C = (V, E, L)$ be a NAND circuit with $n$ inputs and $m$ outputs. We say that $C$ computes $F$ if there is a map $Z : V \rightarrow \{0, 1\}$, such that for every $x \in \{0, 1\}^n$, if $y = F(x)$ then:

* For every $i \in [n]$, if $v$ is labeled with $x_{\langle i \rangle}$ then $Z(v) = x_i$.

* For every $j \in [m]$, if $v$ is labeled with $y_{\langle j \rangle}$ then $Z(v) = y_j$.

* For every gate $v$ with in-neighbors $u, w$, if $a = Z(u)$ and $b = Z(w)$, $Z(v) = \text{NAND}(a, b)$. (If $v$ has fewer than two neighbors
then we replace either $b$ or both $a$ and $b$ with zero in the condition above.)

You should make sure you understand why Definition 3.3 captures the informal description above. This might require reading the definition a second or third time, but would be crucial for the rest of this course.

The following theorem says that these two notions of computing a function are actually equivalent: we can transform a NAND program into a NAND circuit computing the same function, and vice versa.

**Theorem 3.6 — Equivalence of circuits and straightline programs.** For every $F : \{0, 1\}^n \to \{0, 1\}^m$ and $S \in \mathbb{N}$, $F$ can be computed by an $S$-line NAND program if and only if $F$ can be computed by an $n$-input $m$-output NAND circuit of $S$ gates.

The idea behind the proof is simple. Just like we did to the XOR program, if we have a NAND program $P$ of $S$ lines, $n$ inputs, and $m$ outputs, we can transform it into a NAND circuit with $n$ inputs and $m$ gates, where each gate corresponds to a line in the program $P$. If line $\ell$ involves the NAND of two variables assigned to in lines $\ell'$ and $\ell''$, then we will have edges to the gate corresponding to $\ell$ from the gates corresponding to $\ell'$, $\ell''$. In the other direction, we can transform a NAND circuit $C$ of $n$ inputs, $m$ outputs and $S$ gates to an $S$-line program by essentially inverting this process. For every gate in the program, we will have a line in the program which assigns to a variable the NAND of the variables corresponding to the in-neighbors of this gate. If the gate is an output gate labeled with $y_{\langle j \rangle}$ then the corresponding line will assign the value to the variable $y_{\langle j \rangle}$. Otherwise we will assign the value to a fresh “workspace” variable. We now show the formal proof.

**Proof.** We start with the “only if” direction. That is, we show how to transform a NAND program to a circuit. Suppose that $P$ is an $S$ line program that computes $F$. We will build a NAND circuit $C = (V, E, L)$ that computes $F$ as follows. The vertex set $V$ will have the $n + S$ elements $\{ (0, 0), \ldots, (0, n-1), (1, 0), \ldots, (1, S-1) \}$. That is, it will have $n$ vertices of the form $(0, i)$ for $i \in [n]$ (corresponding to the $n$ inputs), and $S$ vertices of the form $(1, \ell)$ (corresponding to the lines in the program). For every line $\ell$ in the program $P$ of the form $foo := bar \ \text{NAND} \ \text{baz}$, we put edges in the graph of the form
We label the vertices of the form \((0, i)\) with \(x_i\) for every \(i \in [n]\). For every \(j \in [m]\), let \(\ell\) be the last line in which the variable \(z_j\) is assigned a value,\(^{10}\) and label the vertex \((1, \ell)\) with \(y_j\). Note that the vertices of the form \((0, i)\) have in-degree zero, and all edges of the form \((1, \ell') (1, \ell)\) satisfy \(\ell > \ell'\). Hence this graph is a DAG, as in any cycle there would have to be at least one edge going from a vertex of the form \((1, \ell)\) to a vertex of the form \((1, \ell')\) for \(\ell' < \ell\) (can you see why?). Also, since we don’t allow a variable of the form \(y_j\) on the right-hand side of a NAND operation, the output vertices have out-degree zero.

To complete the proof of the “only if” direction, we need to show that the circuit \(C\) we constructed computes the same function \(F\) as the program \(P\) we were given. Indeed, let \(x \in \{0, 1\}^n\) and \(y = F(x)\). For every \(\ell\), let \(z_\ell\) be the value that is assigned by the \(\ell\)-th line in the execution of \(P\) on input \(x\). Now, as per Definition 3.3, define the map \(Z: V \rightarrow \{0, 1\}\) as follows: \(Z((0, i)) = x_i\) for \(i \in [n]\) and \(Z((1, \ell)) = z_\ell\) for every \(\ell \in [S]\). Then, by our construction of the circuit, the map satisfies the condition that for vertex \(v\) with in-neighbors \(u\) and \(w\), the value \(Z(v)\) is the NAND of \(Z(u)\) and \(Z(w)\) (replacing missing neighbors with the value 0), and hence in particular for every \(j \in [m]\), the value assigned in the last line that touches \(y_j\) equals \(y_j\). Thus the circuit \(C\) does compute the same function \(F\).

For the “if” direction, we need to transform an \(S\)-gate circuit \(C = (V, E, L)\) that computes \(F: \{0, 1\}^n \rightarrow \{0, 1\}^m\) into an \(S\)-line NAND program \(P\) that computes the same function. We start by doing a topological sort of the graph \(C\). That is we sort the vertex set \(V\) as \(\{v_0, \ldots, v_{n+S-1}\}\) such that if \(v_i < v_j\) then \(v_i v_j \in E\). Such a sorting can be found for every DAG. Moreover, because the input vertices of \(C\) are “sources” (have in-degree zero), we can ensure they are placed first in this sorting and moreover for every \(i \in [n]\), \(v_i\) is the input vertex labeled with \(x_i\).

Now for \(\ell = 0, 1, \ldots, n + S - 1\) we will define a variable \(\text{var}(\ell)\) in our resulting program as follows: If \(\ell < n\) then \(\text{var}(\ell) = x_i\). If \(v_\ell\) is an output gate labeled with \(y_j\) then \(\text{var}(\ell) = y_j\). Otherwise \(\text{var}(\ell)\) will be a temporary workspace variable \(\text{temp}_\ell(\ell - n)\).

Our program \(P\) will have \(S\) lines, where for every \(k \in [S]\), if the in-neighbors of \(v_{n+k}\) are \(v_i\) and \(v_j\) then the \(k\)-th line in the program...
will be \( \text{var}(n + k) := \text{var}(i) \text{ NAND } \text{var}(j) \). If \( v_k \) has fewer than two in-neighbors then we replace the corresponding variable with the variable \( \text{zero} \) (which is never set to any value and hence retains its default value of 0).

To complete the proof of the “if” direction we need to show that the program \( P \) we constructed computes the function \( F \) as the circuit \( C \) we were given. Indeed, let \( x \in \{0, 1\}^n \) and \( y = F(x) \). Since \( C \) computes \( F \), there is a map \( Z : V \rightarrow \{0, 1\} \) as per Definition 3.3. We claim that if we run the program \( P \) on input \( x \), then for every \( k \in [S] \) the value assigned by the \( k \)-th line corresponds to \( Z(v_{n+k}) \). Indeed by construction the value assigned in the \( k \)-th line corresponds to the NAND of the value assigned to the in-neighbors of \( v_{n+k} \). Hence in particular if \( v_{n+k} \) is the output gate labeled \( y_j \) then this value will equal \( y_j \), meaning that on input \( x \) our program will output \( y = F(x) \).  

\[ \blacksquare \]

### 3.5 Composition from graphs

Given Theorem 3.6, we can prove Theorem 3.3 and Theorem 3.4 by showing how to transform circuits for \( F \) and \( G \) into circuits for \( F \circ G \) and \( F \oplus G \). This is what we do now:

**Theorem 3.7 — Sequential composition, circuit version.** If \( C, D \) are NAND circuits such that \( C \) computes \( F : \{0, 1\}^n \rightarrow \{0, 1\}^m \) and \( D \) computes \( G : \{0, 1\}^m \rightarrow \{0, 1\}^k \) then there is a circuit \( E \) of size \( |C| + |D| \) computing the function \( G \circ F : \{0, 1\}^n \rightarrow \{0, 1\}^k \).

**Proof.** Let \( C \) be the \( n \)-input \( m \)-output circuit computing \( F \) and \( D \) be the \( m \)-input \( k \)-output circuit computing \( G \). The circuit to compute \( G \circ F \) is illustrated in Fig. 3.6. We simply “stack” \( D \) after \( C \), by obtaining a combined circuit with \( n \) inputs and \( |C| + |D| \) gates. The gates of \( C \) remain the same, except that we identify the output gates of \( C \) with the input gates of \( D \). That is, for every edge that connected the \( i \)-th input of \( D \) to a gate \( v \) of \( D \), we now connect to \( v \) the output gate of \( C \) corresponding to \( y_\langle i \rangle \) instead. After doing so, we remove the output labels from \( C \) and keep only the outputs of \( D \). For every input \( x \), if we execute the composed circuits on \( x \) (i.e., compute a map \( Z \) from the vertices to \( \{0, 1\} \) as per Definition 3.3), then the output gates of \( C \) will get the values corresponding to \( F(x) \) and hence the output gates of \( D \) will have the value \( G(F(x)) \).  

\[ \blacksquare \]
Figure 3.6: Given a circuit $C$ computing $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ and a circuit $D$ computing $G : \{0, 1\}^m \rightarrow \{0, 1\}^k$, we obtain a circuit $E$ computing $G \circ F$ by identifying the inputs of $D$ with the outputs of $C$. That is, the resulting circuit consists of the gates of both $C$ and $D$, where we replace every in-neighbor of $D$ that was an input gate with the corresponding output gate of $C$.

Theorem 3.8 — Parallel composition, circuit versions. If $C$, $D$ are NAND circuits such that $C$ computes $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ and $D$ computes $G : \{0, 1\}^m \rightarrow \{0, 1\}^k$ then there is a circuit $E$ of size $|C| + |D|$ computing the function $G \oplus F : \{0, 1\}^{n+n'} \rightarrow \{0, 1\}^{m+m'}$.

Proof. If $C$, $D$ are circuits that compute $F$, $G$ then we can transform them to a circuit $E$ that computes $F \oplus G$ as in Fig. 3.7. The circuit $E$ simply consists of two disjoint copies of the circuits $C$ and $D$, where we modify the labelling of the inputs of $D$ from $x_0, \ldots, x_n - 1$ to $x_n, \ldots, x_n + n' - 1$ and the labelling of the outputs of $D$ from $y_0, \ldots, y_m' - 1$ to $y_m, \ldots, y_m + m' - 1$. By the fact that $C$ and $D$ compute $F$ and $G$ respectively, we see that $E$ computes the function $F \oplus G : \{0, 1\}^{n+n'} \rightarrow \{0, 1\}^{m+m'}$ that on input $x \in \{0, 1\}^{n+n'}$ outputs $F(x_0, \ldots, x_{n-1})G(x_n, \ldots, x_{n+n'-1})$. ■

While we proved Theorem 3.3 and Theorem 3.4 using the circuit formalism, it is also possible to directly give syntactic transformations of the code of programs computing $F$ and $G$ to programs comput-
Given a circuit $C$ computing $F : \{0, 1\}^n \to \{0, 1\}^m$ and a circuit $D$ computing $G : \{0, 1\}^{n'} \to \{0, 1\}^{m'}$, we obtain a circuit $E$ computing $F \oplus G$ by simply putting the circuits “side by side”, and renaming the labels of the inputs and outputs of $D$ to $x_{.n}, x_{.n} + n' - 1$ and $y_{.m}, y_{.m} + m' - 1$.

It is a good exercise for you to pause here and see that you know how to give such a transformation. Try to think how you would write a program (in the programming language of your choice) that given two strings $C$ and $D$ that contain the code of NAND programs for computing $F$ and $G$, would output a string $E$ that contains that code of a NAND program for $G \circ F$ (or $F \oplus G$).

### 3.5.1 Example: Adding two-bit numbers

Using composition, we can show how to add two bit numbers. That is, the function $ADD_2 : \{0, 1\}^4 \to \{0, 1\}^3$ that takes two numbers $x, x'$ each between 0 and 3 (each represented with two bits using the binary representation) and outputs their sum, which is a number between 0 and 6 that can be represented using three bits. The grade-school algorithm gives us a way to compute $ADD_2$ using $ADD_1$.

That is, we can add each digit using $ADD_1$ and then take care of the carry. That is, if the two input numbers have the form $x_0 + 2x_1$ and $x_2 + 2x_3$, then the output number $y_0 + y_12 + y_32^2$ can be computed via the following “pseudocode” (see also Fig. 3.8)

```plaintext
y_.0,c_.1 := ADD_1(x_.0,x_.2) // add least significant digits
z_.1,c_.2 := ADD_1(x_.1,x_.3) // add second digits
y_.1,c'_.2 := ADD_1(z_.1,c_.1) // second output is sum + carry
y_.2 := c_.2 OR c'_.2 // top digit is 1 if one of the top carries is 1
```
To transform this pseudocode into an actual program or circuit, we can use Theorem 3.3 and Theorem 3.4. That is, we first compute $(y_0, c_1, z_1, c_2) = ADD_1 \oplus ADD_1(x_0, x_2, x_1, x_3)$, which we can do in 10 lines via Theorem 3.4, then apply $ADD_1$ to $(z_1, c_1)$, and finally use the fact that $OR(a, b) = NAND(NOT(a), NOT(b))$ and $NOT(a) = NAND(a, a)$ to compute $c_2 \lor c'_2$ via three lines of NAND. The resulting code is the following:

```
// Add a pair of two-bit numbers
// Input: (x_0, x_1) and (x_2, x_3)
// Output: (y_0, y_1, y_2) representing the sum
// x_0 + 2x_1 + x_2 + 2x_3

// Operation:
// 1) y_0, c_1 := ADD_1(x_0, x_2):
// add the least significant digits
// c_1 is the "carry"
    u := x_0 NAND x_2
    v := x_0 NAND u
    w := x_2 NAND u
    y_0 := v NAND w
    c_1 := u NAND u

// 2) z_1, z_1 := ADD_1(x_1, x_3):
// add second digits
    u := x_1 NAND x_3
```
v := x_1 NAND u \\
w := x_3 NAND u \\
z_1 := v NAND w \\
z'_1 := u NAND u \\
// 3) Take care of carry: \\
// 3a) y_1 := XOR(z_1, c_1) \\
u := z_1 NAND c_1 \\
v := z_1 NAND u \\
w := c_1 NAND u \\
y_1 := v NAND w \\
// 3b) y_2 := z'_1 OR (z_1 AND c_1) \\
// = NAND(NOT(z'_1), NAND(z_1, c_1)) \\
u := z'_1 NAND z'_1 \\
v := z_1 NAND c_1 \\
y_2 := u NAND v \\

For example, the computation of the deep fact that 2 + 3 = 5 corresponds to running this program on the inputs (0, 1, 1, 1) which will result in the following trace:

Executing step 1: "u := x_0 NAND x_2" x_0 = 0, x_2 = 1, u is assigned 1, 
Executing step 2: "v := x_0 NAND u" x_0 = 0, u = 1, v is assigned 1, 
Executing step 3: "w := x_2 NAND u" x_2 = 1, u = 1, w is assigned 0, 
Executing step 4: "y_0 := v NAND w" v = 1, w = 0, y_0 is assigned 1, 
Executing step 5: "c_1 := u NAND u" u = 1, u = 1, c_1 is assigned 0, 
Executing step 6: "u := x_1 NAND x_3" x_1 = 1, x_3 = 1, u is assigned 0, 
Executing step 7: "v := x_1 NAND u" x_1 = 1, u = 0, v is assigned 1, 
Executing step 8: "w := x_3 NAND u" x_3 = 1, u = 0, w is assigned 1, 
Executing step 9: "z_1 := v NAND w" v = 1, w = 1, z_1 is assigned 0, 
Executing step 10: "z'_1 := u NAND u" u = 0, u = 0, z'_1 is assigned 1, 
Executing step 11: "u := z_1 NAND c_1" z_1 = 0, c_1 = 0, u is assigned 1, 
Executing step 12: "v := z_1 NAND u" z_1 = 0, u = 1, v is assigned 1, 
Executing step 13: "w := c_1 NAND u" c_1 = 0, u = 1, w is
assigned 1,
Executing step 14: "y_1 := v NAND w" v = 1, w = 1, y_1 is assigned 0,
Executing step 15: "u := z'_1 NAND z'_1" z'_1 = 1, u is assigned 0,
Executing step 16: "v := z_1 NAND c_1" z_1 = 0, c_1 = 0, v is assigned 1,
Executing step 17: "y_2 := u NAND v" u = 0, v = 1, y_2 is assigned 1,
Output is y_0=1, y_1=0, y_2=1

3.5.2 Composition in NAND programs

We can generalize the above examples to handle not just sequential and parallel but all forms of composition. That is, if we have an s line program P that computes the function F, and a t line program P’ that can compute the function G using k calls to a “black box” for computing F, then we can obtain a t + ks line program P’’ to compute G (without any “magic boxes”) by replacing every call to F in P’ with a copy of P (while appropriately renaming the variables). Similarly, in the circuit formulation, we can transform an s-gate circuit C for computing G, and a t-gate “augmented circuit” C’ that can compute F using k “magic gates” that compute G instead of NAND, into a standard s + tk gate circuit C’’ that computes F, by replacing each gate with a copy of C.

Figure 3.9: We can compose a program P that computes F with a program P’ that computes G by making calls to F, to obtain a program P’’ that computes G without any calls.
3.6 Lecture summary

- We can define the notion of computing a function via a simplified “programming language”, where computing a function $F$ in $T$ steps would correspond to having a $T$-line NAND program that computes $F$.

- An equivalent formulation is that a function is computable by a NAND program if it can be computed by a NAND circuit.

3.7 Exercises

Exercise 3.1 Which of the following statements is false? a. There is a NAND program to add two 4-bit numbers that has at most 100 lines.

b. Every NAND program to add two 4-bit numbers has at most 100 lines.

c. Every NAND program to add two 4-bit numbers has least 5 lines.

Exercise 3.2 Write a NAND program that adds two 3-bit numbers.

Exercise 3.3 Prove Theorem 3.5.\(^{11}\)

3.8 Bibliographical notes

The exact notion of “NAND programs” we use is nonstandard, but these are equivalent to standard models in the literature such as straightline programs and Boolean circuits.

An historical review of calculating machines can be found in Chapter I of the 1946 “operating manual” for the Harvard Mark I computer, written by Lieutenant Grace Murray Hopper and the staff of the Harvard Computation Laboratory.

3.9 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

(to be completed)

\(^{11}\) Hint: Prove by induction that for every $n > 1$ which is a power of two, $\text{XOR}_n \in \text{SIZE}(4(n - 1))$. Then use this to prove the result for every $n$. 

3.10 Acknowledgements
4

Syntactic sugar, and computing every function

“Syntactic sugar causes cancer of the semicolon.”, Alan Perlis, 1982.

The NAND programming language is pretty much as “bare bones” as programming languages come. After all, it only has a single operation. But, it turns out we can implement some “added features” on top of it. That is, we can show how we can implement those features using the underlying mechanisms of the language.

Let’s start with a simple example. One of the most basic operations a programming language has is to assign the value of one variable into another. And yet in NAND, we cannot even do that, as we only allow assignments of the result of a NAND operation. Yet, it is possible to “pretend” that we have such an assignment operation, by transforming code such as

\[
\text{foo} := \text{bar}
\]

into the valid NAND code:

\[
\text{notbar} := \text{bar NAND bar}
\]
\[
\text{foo} := \text{notbar NAND notbar}
\]

Thus in describing NAND programs we can (and will) allow ourselves to use the variable assignment operation, with the understanding that in actual programs we will replace every line of the first form with the two lines of the second form. In programming language parlance this is known as “syntactic sugar”, since we are not changing the definition of the language, but merely introducing some convenient notational shortcuts.¹ We will use several such “syntactic sugar” constructs to make our descriptions of NAND programs shorter and simpler. However, these descriptions are merely

¹ This concept is also known as “macros” or “meta-programming” and is sometimes implemented via a preprocessor or macro language in a programming language or a text editor.
shorthand for the equivalent standard or “sugar free” NAND program that is obtained after removing the use of all these constructs. In particular, when we say that a function $F$ has an $s$-line NAND program, we mean a standard NAND program, that does not use any syntactic sugar. The website http://www.nandpl.org contains an online “unsweetener” that can take a NAND program that uses these features and modifies it to an equivalent program that does not use them.

### 4.1 Some useful syntactic sugar

In this section, we will list some additional examples of “syntactic sugar” transformations. Going over all these examples can be somewhat tedious, but we do it for two reasons:

1. To convince you that despite its seeming simplicity and limitations, the NAND programming language is actually quite powerful and can capture many of the fancy programming constructs such as if statements and function definitions that exists in more fashionable languages.

2. So you can realize how lucky you are to be taking a theory of computation course and not a compilers course. :)

#### 4.1.1 Constants

We can create variables zero and one that are have the values 0 and 1 respectively by adding the lines

```plaintext
notx_0 := x_0 NAND x_0
one := x_0 NAND notx_0
zero := one NAND one
```

Note that since for every $x \in \{0, 1\}$, $NAND(x, x) = 1$, the variable one will get the value 1 regardless of the value of $x_0$, and the variable zero will get the value $NAND(1, 1) = 0$.\footnote{We could have saved a couple of lines using the convention that uninitialized variables default to 0, but it’s always nice to be explicit.} Hence we can replace code such as $a := 0$ with $a := one NAND one$ and similarly $b := 1$ will be replaced with $b := zero NAND zero$.

#### 4.1.2 Conditional statements

Another sorely missing feature in NAND is a conditional statement. We would have liked to be able to write something like
if (a) {
    ...
    some code here
    ...
}

To ensure that there is code that will only be executed when the variable a is equal to 1. We can do so by replacing every variable var that is assigned a value in the code by a variable tempvar and then execute it normally. After it is executed, we assign to every such variable the value $\text{MUX}(a, \text{var}, \text{tempvar})$ where $\text{MUX} : \{0,1\}^3 \rightarrow \{0,1\}$ is the multiplexer function that on input $(a, b, c)$ outputs $b$ if $a = 0$ and $c$ if $a = 1$. This function has a 4-line NAND program:

\[
\begin{align*}
\text{nx}_2 & := \text{x}_2 \text{ NAND } \text{x}_2 \\
\text{u} & := \text{x}_0 \text{ NAND } \text{nx}_2 \\
\text{v} & := \text{x}_1 \text{ NAND } \text{x}_2 \\
\text{y}_0 & := \text{u} \text{ NAND } \text{v}
\end{align*}
\]

We leave it as Exercise 4.2 to verify that this program does indeed compute the $\text{MUX}$ function.

4.1.3 Functions / Macros

Another staple of almost any programming language is the ability to execute functions. However, we can achieve the same effect as (non recursive) functions using “copy pasting”. That is, we can replace code such as

```
def a,b := Func(c,d) {
    function_code
}
```

```
e,f := Func(g,h)
```

with

```
def a,b := Func(c,d) {
    function_code'
}
```

```
e,f := Func(g,h)
```

where `function_code'` is obtained by replacing all occurrences of a with e,f with b, c with g, d with h. When doing that we will need to ensure that all other variables appearing in `function_code'` don’t interfere with other variables by replacing every instance of a variable foo with upfoo where up is some unique prefix.
4.1.4 Example:

Using these features, we can express the code of the ADD\textsubscript{2} function we saw last lecture as

```plaintext
def c := \text{AND}(a,b) \{ 
    notc := a \text{ NAND} b 
    c := notc \text{ NAND} c 
\}
def c := \text{XOR}(a,b) \{ 
    u := a \text{ NAND} b 
    v := a \text{ NAND} u 
    w := b \text{ NAND} u 
    c := v \text{ NAND} w 
\}
y_0 := \text{XOR}(x_0,x_2) \quad \text{// add first digit} 
c_1 := \text{AND}(x_0,x_2) 
z_1 := \text{XOR}(x_1,x_2) \quad \text{// add second digit} 
c_2 := \text{AND}(x_1,x_2) 
y_1 := \text{XOR}(c_1,y_2) \quad \text{// add carry from before} 
c'_{-2} := \text{AND}(c_1,y_2) 
y_{-2} := \text{XOR}(c'_{-2},x_2) 
```

4.1.5 More indices

As stated, the NAND programming language only allows for “one dimensional arrays”, in the sense that we can use variables such as `foo_7` or `foo_29` but not `foo_5,15`. However we can easily embed two dimensional arrays in one-dimensional ones using a one-to-one function \(PAIR : \mathbb{N}^2 \to \mathbb{N}\). (For example, we can use \(PAIR(x,y) = 2^x3^y\), but there are also more efficient embeddings, see Exercise 4.1.) Hence we can replace any variable of the form `foo_{i,j}` with `foo_{PAIR(i,j)}`, and similarly for three dimensional arrays.

4.1.6 Non-Boolean variables, lists and integers

While the basic variables in NAND++ are Boolean (only have 0 or 1), we can easily extend this to other objects using encodings. For example, we can encode the alphabet \{a,b,c,d,e,f\} using three bits as 000, 001, 010, 011, 100, 101. Hence, given such an encoding, we could use the code

```plaintext
foo := "b"
```
would be a shorthand for the program

\begin{verbatim}
foo_0 := 0
foo_1 := 0
foo_2 := 1
\end{verbatim}

Using our notion of multi-indexed arrays, we can also use code such as

\begin{verbatim}
foo := "be"
\end{verbatim}

as a shorthand for

\begin{verbatim}
foo_{0,0} := 0
foo_{0,1} := 0
foo_{0,2} := 1
foo_{1,0} := 1
foo_{1,1} := 0
foo_{1,2} := 0
\end{verbatim}

which can then in turn be mapped to standard NAND code using a one-to-one embedding \( \text{pair} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) as above.

4.1.7 Storing integers

We can also handle non-finite alphabets, such as integers, by using some prefix-free encoding and encoding the integer in an array. For example, to store non-negative integers, we can use the convention that 01 stands for 0, 11 stands for 1, and 00 is the end marker. To store integers that could be potentially negative we can use the convention 10 in the first coordinate stands for the negative sign. So, code such as

\begin{verbatim}
foo := 5 // (1,0,1) in binary
\end{verbatim}

will be shorthand for

\begin{verbatim}
foo_0 := 1
foo_1 := 1
foo_2 := 0
foo_3 := 1
foo_4 := 1
foo_5 := 1
foo_6 := 0
foo_7 := 0
\end{verbatim}

\footnote{TODO: possibly add an exercise using this with the alphabet including [],., to encode lists.}

\footnote{This is just an arbitrary choice made for concreteness, and one can choose other representations. In particular, as discussed before, if the integers are known to have a fixed size, then there is no need for additional encoding to make them prefix-free.}
while

foo := -5

will be the same as

foo_0 := 1
foo_1 := 0
foo_2 := 1
foo_3 := 1
foo_4 := 0
foo_5 := 1
foo_6 := 1
foo_7 := 1
foo_8 := 0
foo_9 := 0

Using multidimensional arrays, we can use arrays of integers and hence replace code such as

foo := [12, 7, 19, 33]

with the equivalent NAND expressions.

For integer valued variables, we can use the standard algorithms of addition, multiplication, comparisons and so on. to write code such as

j := k + l
if (m*n>k) {
    code...
}

which then gets translated into standard NAND++ program by copy pasting these algorithms.

4.2 Adding and multiplying n bit numbers

We have seen how to add one and two bit numbers. We can use the gradeschool algorithm to show that NAND programs can add n-bit numbers for every n:

**Theorem 4.1 — Addition using NAND programs.** For every n, let \( ADD_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{n+1} \) be the function that, given \( x, x' \in \{0,1\}^n \) computes the representation of the sum of the numbers that x and x' represent. Then there is a NAND program that computes the
function \( ADD_n \). Moreover, the number of lines in this program is smaller than 100n.

**Proof.** To prove this theorem we repeatedly appeal to the notion of composition, and to the “gradeschool” algorithm for addition. To add the numbers \((x_0, \ldots, x_{n-1})\) and \((x_n, \ldots, x_{2n-1})\), we set \( c_0 = 0 \) and do the following for \( i = 0, \ldots, n-1 \):

* Compute \( z_i = \text{XOR}(x_i, x_{n+i}) \) (add the two corresponding digits)

* Compute \( y_i = \text{XOR}(z_i, c_i) \) (add in the carry to get the final digit)

* Compute \( c_{i+1} = \text{ATLEASTTWO}(x_i, x_{n+i}, c_i) \) where \( \text{ATLEASTTWO} : \{0,1\}^3 \to \{0,1\} \) is the function that maps \((a, b, c)\) to 1 if \( a + b + c \geq 2 \). (The new carry is 1 if and only if at least two of the values \( x_i, x_{n+i}, y_i \) were equal to 1.) The most significant digit \( y_n \) of the output will of course be the last carry \( c_n \).

To transform this algorithm to a NAND program we just need to plug in the program for XOR, and use the observation (see Exercise 4.3) that

\[
\text{ATLEASTTWO}(a, b, c) = (a \land b) \lor (a \land c) \lor (b \land c)
\]

\[
= \text{NAND}(\text{NOT} (\text{NAND}(\text{NAND}(a, b), \text{NAND}(a, c))), \text{NAND}(b, c))
\]

\[
(4.1)
\]

We leave accounting for the number of lines, and verifying that it is smaller than 100n, as an exercise to the reader. \( \square \)
4.2.1 Multiplying numbers

Once we have addition, we can use the gradeschool algorithm to obtain multiplication as well, thus obtaining the following theorem:

**Theorem 4.2 — Multiplication NAND programs.** For every $n$, let $MULT_n : \{0,1\}^{2n} \to \{0,1\}^{2n}$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the product of the numbers that $x$ and $x'$ represent. Then there is a NAND program that computes the function $MULT_n$. Moreover, the number of lines in this program is smaller than $1000n^2$.

We omit the proof, though in Exercise 4.7 we ask you to supply a “constructive proof” in the form of a program (in your favorite programming language) that on input a number $n$, outputs the code of a NAND program of at most $1000n^2$ lines that computes the $MULT_n$ function. In fact, we can use Karatsuba’s algorithm to show that there is a NAND program of $O(n \log_2 3)$ lines to compute $MULT_n$ (and one can even get further asymptotic improvements using the newer algorithms).

4.3 Functions beyond arithmetic

We have seen that NAND programs can add and multiply numbers. But can they compute other type of functions, that have nothing to do with arithmetic? Here is one example:

**Definition 4.1 — Lookup function.** For every $k$, the lookup function $LOOKUP_k : \{0,1\}^{2^k+k} \to \{0,1\}$ is defined as follows: For every $x \in \{0,1\}^{2^k}$ and $i \in \{0,1\}^k$,

$$LOOKUP_k(x,i) = x_i$$ (4.2)

where $x_i$ denotes the $i^{th}$ entry of $x$, using the binary representation to identify $i$ with a number in $\{0,\ldots,2^k-1\}$.

The function $LOOKUP_1 : \{0,1\}^3 \to \{0,1\}$ maps $(x_0, x_1, i) \in \{0,1\}^3$ to $x_i$. It is actually the same as the $MUX$ function we have seen.
above, that has a 4 line NAND program. However, can we compute higher levels of $LOOKUP$? This turns out to be the case:

**Theorem 4.3 — Lookup function.** For every $k$, there is a NAND program that computes the function $LOOKUP_k : \{0,1\}^{2^k + k} \to \{0,1\}$. Moreover, the number of lines in this program is at most $4 \cdot 2^k$.

### 4.3.1 Constructing a NAND program for $LOOKUP$

We now prove Theorem 4.3. We will do so by induction. That is, we show how to use a NAND program for computing $LOOKUP_k$ to compute $LOOKUP_{k+1}$. Let us first see how we do this for $LOOKUP_2$.

Given input $x = (x_0, x_1, x_2, x_3)$ and an index $i = (i_0, i_1)$, if the most significant bit $i_1$ of the index is 0 then $LOOKUP_2(x, i)$ will equal $x_0$ if $i_0 = 0$ and equal $x_1$ if $i_0 = 1$. Similarly, if the most significant bit $i_1$ is 1 then $LOOKUP_2(x, i)$ will equal $x_2$ if $i_0 = 0$ and will equal $x_3$ if $i_0 = 1$. Another way to say this is that

$$LOOKUP_2(x_0, x_1, x_2, x_3, i_0, i_1) = LOOKUP_1(LOOKUP_1(x_0, x_1, i_0), LOOKUP_1(x_2, x_3, i_0), i_1)$$

(4.3)

That is, we can compute $LOOKUP_2$ using three invocations of $LOOKUP_1$. The “pseudocode” for this program will be

\[
\begin{align*}
  z_0 & := LOOKUP_1(x_0, x_1, x_2, x_3) \\
  z_1 & := LOOKUP_1(x_4, x_5, x_6, x_7) \\
  y_0 & := LOOKUP_1(z_0, z_1, x_4, x_5, x_6, x_7)
\end{align*}
\]

(Note that since we call this function with $(x_0, x_1, x_2, x_3, i_0, i_1)$, the inputs $x_4$ and $x_5$ correspond to $i_0$ and $i_1$.) We can obtain an actual “sugar free” NAND program of at most 12 lines by replacing the calls to $LOOKUP_1$ by an appropriate copy of the program above.

We can generalize this to compute $LOOKUP_3$ using two invocations of $LOOKUP_2$ and one invocation of $LOOKUP_1$. That is, given input $x = (x_0, \ldots, x_7)$ and $i = (i_0, i_1, i_2)$ for $LOOKUP_3$, if the most significant bit of the index $i_2$ is 0, then the output of $LOOKUP_3$ will equal $LOOKUP_2(x_0, x_1, x_2, x_3, i_0, i_1)$, while if this index $i_2$ is 1 then the output will be $LOOKUP_2(x_4, x_5, x_6, x_7, i_0, i_1)$, meaning that the following pseudocode can compute $LOOKUP_3$:

\[
\begin{align*}
  z_0 & := LOOKUP_2(x_0, x_1, x_2, x_3, x_8, x_9) \\
  z_1 & := LOOKUP_2(x_4, x_5, x_6, x_7, x_8, x_9) \\
  y_0 & := LOOKUP_1(z_0, z_1, x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)
\end{align*}
\]

where again we can replace the calls to $LOOKUP_2$ and $LOOKUP_1$ by invocations of the process above.
Formally, we can prove the following lemma:

**Lemma 4.4 — Lookup recursion.** For every \( k \geq 2 \), \( \text{LOOKUP}_k(x_0, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1}) \)

is equal to

\[
\text{LOOKUP}_1(\text{LOOKUP}_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-2}), \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-2}), i_{k-1})
\]  

(4.4)

**Proof.** If the most significant bit \( i_{k-1} \) of \( i \) is zero, then the index \( i \) is in \( \{0, \ldots, 2^{k-1} - 1\} \) and hence we can perform the lookup on the “first half” of \( x \) and the result of \( \text{LOOKUP}_k(x, i) \) will be the same as 

\[
a = \text{LOOKUP}_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-1}).
\]

On the other hand, if this most significant bit \( i_{k-1} \) is equal to 1, then the index is in \( \{2^{k-1}, \ldots, 2^k - 1\} \), in which case the result of \( \text{LOOKUP}_k(x, i) \) is the same as 

\[
b = \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1}).
\]

Thus we can compute \( \text{LOOKUP}_k(x, i) \) by first computing \( a \) and \( b \) and then outputting \( \text{LOOKUP}_1(a, b, i_{k-1}) \).

Lemma 4.4 directly implies **Theorem 4.3.** We prove by induction on \( k \) that there is a NAND program of at most \( 4 \cdot 2^k \) lines for \( \text{LOOKUP}_k \). For \( k = 1 \) this follows by the four line program for \( \text{LOOKUP}_1 \) we’ve seen before. For \( k > 1 \), we use the following pseudocode

\[
a = \text{LOOKUP}_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-1})
\]

\[
b = \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})
\]

\[
y_{0} = \text{LOOKUP}_1(a, b, i_{k-1})
\]

If we let \( L(k) \) be the number of lines required for \( \text{LOOKUP}_k \), then the above shows that

\[
L(k) \leq 2L(k-1) + 4.
\]

(4.5)

We will prove by induction that \( L(k) \leq 4(2^k - 1) \). This is true for \( k = 1 \) by our construction. For \( k > 1 \), using the inductive hypothesis and Eq. (4.5), we get that

\[
L(k) \leq 2 \cdot 4 \cdot (2^{k-1} - 1) + 4 = 4 \cdot 2^k - 8 + 4 = 4(2^k - 1)
\]

(4.6)

completing the proof of **Theorem 4.3.**

### 4.4 Computing every function

At this point we know the following facts about NAND programs:

1. They can compute at least some non trivial functions.
2. Coming up with NAND programs for various functions is a very tedious task.

Thus I would not blame the reader if they were not particularly looking forward to a long sequence of examples of functions that can be computed by NAND programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND programs can compute every finite function:

Theorem 4.5 — Universality of NAND. For every $n, m$ and function $F : \{0,1\}^n \to \{0,1\}^m$, there is a NAND program that computes the function $F$. Moreover, there is such a program with at most $O(m2^n)$ lines.

The implicit constant in the $O(\cdot)$ notation can be shown to be at most 10. We also note that the bound of Theorem 4.5 can be improved to $O(m2^n/n)$, see Remark 4.4.1.

4.4.1 Proof of NAND’s Universality

To prove Theorem 4.5, we need to give a NAND program for every possible function. We will restrict our attention to the case of Boolean functions (i.e., $m = 1$). In Exercise 4.9 you will show how to extend the proof for all values of $m$. A function $F : \{0,1\}^n \to \{0,1\}$ can be specified by a table of its values for each one of the $2^n$ inputs. Here is for example one particular function $G : \{0,1\}^4 \to \{0,1\}$.

We can see that for every $x \in \{0,1\}^4$, $G(x) = LOOKUP_4(1100100100001111, x)$. Therefore the following is NAND “pseudocode” to compute $G$:

\[
\begin{align*}
G0000 & := 1 \\
G0001 & := 1 \\
G0010 & := 0 \\
G0011 & := 0 \\
G0100 & := 1 \\
G0101 & := 0 \\
G0110 & := 0 \\
G0111 & := 1 \\
G1000 & := 0 \\
G1001 & := 0 \\
G1010 & := 0 \\
G1011 & := 0 \\
G1100 & := 1 \\
G1101 & := 1 \\
G1110 & := 1 \\
\end{align*}
\]

\*In case you are curious, this is the function that computes the digits of π in the binary basis.
<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>

G_{1111} := 1
y_0 := LOOKUP(G_{0000}, G_{0001}, G_{0010}, G_{0011}, G_{0100}, G_{0101}, G_{0110}, G_{0111},
G_{1000}, G_{1001}, G_{1010}, G_{1011}, G_{1100}, G_{1101}, G_{1110}, G_{1111}, x_0, x_1,
x_2, x_3)

Recall that we can translate this pseudocode into an actual NAND program by adding three lines to define variables zero and one that are initialized to 0 and 1 respectively, and then replacing a statement such as G_{xxx} := 0 with G_{xxx} := one NAND one and a statement such as G_{xxx} := 1 with G_{xxx} := zero NAND zero. The call to LOOKUP will be replaced by the NAND program that computes LOOKUP\_n, but we will replace the variables i_0, ..., i_3 in this program with x_0, ..., x_3 and the variables x_0, ..., x_15 with G_{000}, ..., G_{1111}.

There was nothing about the above reasoning that was particular to this program. Given every function F : \{0, 1\}^n \rightarrow \{0, 1\}, we can write a NAND program that does the following:

1. Initialize \(2^n\) variables of the form \(F_0\)...\(F_{11}...1\) so that for every \(z \in \{0, 1\}^n\), the variable corresponding to \(z\) is assigned the value \(F(z)\).

2. Compute LOOKUP\_n on the \(2^n\) variables initialized in the previous step, with the index variable being the input variables \(x_{\langle 0 \rangle}, ..., x_{\langle 2^n - 1 \rangle}\). That is, just like in the pseudocode for \(G\) above,
we use \( y_0 := \text{LOOKUP}(F00...00,F00...01,...,F11...1,x_1,...,x_{2^n-1}) \)

The total number of lines in the program will be \( 2^n \) plus the \( 4 \cdot 2^n \) lines that we pay for computing \( \text{LOOKUP}_n \). This completes the proof of Theorem 4.5.

The NAND programming language website allows you to construct a NAND program for an arbitrary function.

---

**Advanced note: improving by a factor of \( n \)** By being a little more careful, we can improve the bound of Theorem 4.5 and show that every function \( F : \{0,1\}^n \rightarrow \{0,1\}^m \) can be computed by a NAND program of at most \( O(m2^n/n) \) lines. As before, it is enough to prove the case that \( m = 1 \).

The idea is to use the technique known as memoization. Let \( k = \log(n - 2 \log n) \) (the reasoning behind this choice will become clear later on). For every \( a \in \{0,1\}^{n-k} \) we define \( F_a : \{0,1\}^k \rightarrow \{0,1\} \) to be the function that maps \( w_0,...,w_{k-1} \) to \( F(a_0,...,a_{n-k-1},w_0,...,w_{k-1}) \). On input \( x = x_0,...,x_{n-1} \), we can compute \( F(x) \) as follows: First we compute a \( 2^{n-k} \) long string \( P \) whose \( a^\text{th} \) entry (identifying \( \{0,1\}^{n-k} \) with \( [2^{n-k}] \)) equals \( F_a(x_0,...,x_{n-k-1}) \). One can verify that \( F(x) = \text{LOOKUP}_{n-k}^P(P,x_0,...,x_{n-k}) \). Since we can compute \( \text{LOOKUP}_{n-k}^P \) using \( O(2^{n-k}) \) lines, if we can compute the string \( P \) (i.e., compute variables \( P(0),...,P(2^{n-k}-1) \)) using \( T \) lines, then we can compute \( F \) in \( O(2^{n-k}) + T \) lines. The trivial way to compute the string \( P \) would be to use \( O(2^k) \) lines to compute for every \( a \) the map \( x_0,...,x_{k-1} \mapsto F_a(x_0,...,x_{k-1}) \) as in the proof of Theorem 4.5. Since there are \( 2^{n-k} \) \( a \)'s, that would be a total cost of \( O(2^{n-k} \cdot 2^k) = O(2^n) \) which would not improve at all on the bound of Theorem 4.5. However, a more careful observation shows that we are making some redundant computations. After all, there are only \( 2^k \) distinct functions mapping \( k \) bits to one bit. If \( a \) and \( a' \) satisfy that \( F_a = F_{a'} \), then we don’t need to spend \( 2^k \) lines computing both \( F_a \) and \( F_{a'} \) but rather can only compute the variable \( P(\langle a \rangle) \) and then copy \( P(\langle a \rangle) \) to \( P(\langle a' \rangle) \) using \( O(1) \) lines. Since we have \( 2^k \) unique functions, we can bound the total cost to compute \( P \) by \( O(2^k 2^k) + O(2^{n-k}) \). Now it just becomes a matter of calculation. By our choice of \( k \), \( 2^k = n - 2 \log n \) and hence \( 2^k = n^2/2n \). Since \( n/2 \leq 2^k \leq n \), we can bound the total cost of computing \( F(x) \) (including also the additional \( O(2^{n-k}) \) cost of computing \( \text{LOOKUP}_{n-k}^P \)) by \( O(2n^2/n) + O(2^n/n) \), which is what we wanted to prove.
Discussion: In retrospect, it is perhaps not surprising that every finite function can be computed with a NAND program. A finite function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ can be represented by simply the list of its outputs for each one of the $2^n$ input values. So it makes sense that we could write a NAND program of similar size to compute it. What is more interesting is that some functions, such as addition and multiplication, have a much more efficient representation: one that only requires $O(n^2)$ or even smaller number of lines.

4.5 The class $\text{SIZE}_{n,m}(T)$

For every $n, m, T \in \mathbb{N}$, we denote by $\text{SIZE}_{n,m}(T)$, the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$ that can be computed by NAND programs of at most $T$ lines. Theorem 4.5 shows that $\text{SIZE}_{n,m}(4m2^n)$ is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$. The results we’ve seen before can be phrased as showing that $\text{ADD}_T \in \text{SIZE}_{2n,n+1}(100n)$ and $\text{MULT}_T \in \text{SIZE}_{2n,2n}(10000n \log_2 3)$.

4.6 Lecture summary

- We can define the notion of computing a function via a simplified “programming language”, where computing a function $F$ in $T$ steps would correspond to having a $T$-line NAND program that computes $F$.
- While the NAND programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
- Every function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ can be computed by a NAND program of at most $O(m2^n)$ lines (and in fact at most $O(m2^n/n)$ lines).
- Sometimes (or maybe always?) we can translate an efficient algorithm to compute $F$ into a NAND program that computes $F$ with a number of lines comparable to the number of steps in this algorithm.
4.7 Exercises

**Exercise 4.1 — Pairing.**
1. Prove that the map \( F(x, y) = 2^x 3^y \) is a one-to-one map from \( \mathbb{N}^2 \) to \( \mathbb{N} \).

2. Show that there is a one-to-one map \( F : \mathbb{N}^2 \to \mathbb{N} \) such that for every \( x, y \), \( F(x, y) \leq 100 \cdot \max\{x, y\}^2 \).

3. For every \( k \), show that there is a one-to-one map \( F : \mathbb{N}^k \to \mathbb{N} \) such that for every \( x_0, \ldots, x_{k-1} \), \( F(x_0, \ldots, x_{k-1}) \leq 100 \cdot \max\{x_0, \ldots, x_{k-1}\}^k \).

**Exercise 4.2 — Computing MUX.** Prove that the NAND program below computes the function \( MUX \) (or \( LOOKUP \)) where \( MUX(a, b, c) \) equals \( a \) if \( c = 0 \) and equals \( b \) if \( c = 1 \):

\[
\begin{align*}
\text{nx}_2 & := x_2 \text{ NAND } x_2 \\
\text{u} & := x_0 \text{ NAND } \text{nx}_2 \\
\text{v} & := x_1 \text{ NAND } x_2 \\
y_0 & := \text{u} \text{ NAND } v
\end{align*}
\]

**Exercise 4.3 — At least two.** Give a NAND program of at most 6 lines to compute \( \text{AT LEAST TWO} : \{0,1\}^3 \to \{0,1\} \) where \( \text{AT LEAST TWO}(a, b, c) = 1 \) iff \( a + b + c \geq 2 \).

**Exercise 4.4 — Conditional statements.** In this exercise we will show that even though the NAND programming language does not have an \( \text{if .. then .. else ..} \) statement, we can still implement it. Suppose that there is an \( s \)-line NAND program to compute \( F : \{0,1\}^n \to \{0,1\} \) and an \( s' \)-line NAND program to compute \( F' : \{0,1\}^n \to \{0,1\} \). Prove that there is a program of at most \( s + s' + 10 \) lines to compute the function \( G : \{0,1\}^{n+1} \to \{0,1\} \) where \( G(x_0, \ldots, x_{n-1}, x_n) \) equals \( F(x_0, \ldots, x_{n-1}) \) if \( x_n = 0 \) and equals \( F'(x_0, \ldots, x_{n-1}) \) otherwise.

**Exercise 4.5** Prove Theorem 3.5.8

**Exercise 4.6 — Addition.** Write a program using your favorite programming language that on input an integer \( n \), outputs a NAND program that computes \( ADD_n \). Can you ensure that the program it outputs for \( ADD_n \) has fewer than \( 10n \) lines?

**Exercise 4.7 — Multiplication.** Write a program using your favorite programming language that on input an integer \( n \), outputs a NAND program that computes \( MULT_n \). Can you ensure that the program it outputs for \( MULT_n \) has fewer than \( \text{1000} \cdot n^2 \) lines?

**Exercise 4.8 — Efficient multiplication (challenge).** Write a program using your favorite programming language that on input an integer \( n \),

---

\(^8\) **Hint:** Prove by induction that for every \( n > 1 \) which is a power of two, \( \text{XOR}_n \in \text{SIZE}(4(n-1)) \). Then use this to prove the result for every \( n \).
outputs a NAND program that computes $\text{MULT}_n$ and has at most $10000n^{1.9}$ lines. What is the smallest number of lines you can use to multiply two 64 bit numbers? 

Exercise 4.9 — Multibit function. Prove that

a. If there is an $s$-line NAND program to compute $F : \{0,1\}^n \rightarrow \{0,1\}$ and an $s'$-line NAND program to compute $F' : \{0,1\}^n \rightarrow \{0,1\}$ then there is an $s + s'$-line program to compute the function $G : \{0,1\}^n \rightarrow \{0,1\}^2$ such that $G(x) = (F(x), F'(x))$.

b. For every function $F : \{0,1\}^n \rightarrow \{0,1\}^m$, there is a NAND program of at most $10^m \cdot 2^n$ lines that computes $F$.

4.8 Bibliographical notes

4.9 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

(to be completed)

4.10 Acknowledgements
5

Code as data, data as code

“The term code script is, of course, too narrow. The chromosomal structures are at the same time instrumental in bringing about the development they foreshadow. They are law-code and executive power - or, to use another simile, they are architect’s plan and builder’s craft - in one.”, Erwin Schrödinger, 1944.

“The importance of the universal machine is clear. We do not need to have an infinity of different machines doing different jobs. . . . The engineering problem of producing various machines for various jobs is replaced by the office work of ‘programming’ the universal machine”, Alan Turing, 1948

A NAND program can be thought of as simply a sequence of symbols, each of which can be encoded with zeros and ones using (for example) the ASCII standard. Thus we can represent every NAND program as a binary string. This statement seems obvious but it is actually quite profound. It means that we can treat a NAND program both as instructions to carrying computation and also as data that could potentially be input to other computations.

This correspondence between code and data is one of the most fundamental aspects of computing. It underlies the notion of general purpose computers, that are not pre-wired to compute only one task, and it is also the basis of our hope for obtaining general artificial intelligence. This concept finds immense use in all areas of computing, from scripting languages to machine learning, but it is fair to say that we haven’t yet fully mastered it. Indeed many security exploits involve cases such as “buffer overflows” when attackers manage to
inject code where the system expected only “passive” data. The idea of code as data reaches beyond the realm of electronic computers. For example, DNA can be thought of as both a program and data (in the words of Schrödinger, who wrote before DNA’s discovery a book that inspired Watson and Crick, it is both “architect’s plan and builder’s craft”).

![XKCD Cartoon](image)

**Figure 5.1**: As illustrated in this xkcd cartoon, many exploits, including buffer overflow, SQL injections, and more, utilize the blurry line between “active programs” and “static strings”.

### 5.1 A NAND interpreter in NAND

One of the most interesting consequences of the fact that we can represent programs as strings is the following theorem:

**Theorem 5.1 — Bounded Universality of NAND programs.** For every \( s, n, m \in \mathbb{N} \) there is a NAND program that computes the function

\[
\text{EVAL}_{s,n,m}: \{0,1\}^{s+n} \rightarrow \{0,1\}^m
\]

(5.1)

defined as follows: For every string \((P,x)\) where \(P \in \{0,1\}^s\) and \(x \in \{0,1\}^n\), if \(P\) describes a NAND program with \(n\) input bits and \(m\) outputs bits, then \(\text{EVAL}_{s,n,m}(x)\) is the output of this program on input \(x\). \(^1\)

Of course to fully specify \(\text{EVAL}_{s,n,m}\), we need to fix a precise representation scheme for NAND programs as binary strings. We can simply use the ASCII representation, though later we will use a somewhat more convenient representation. But regardless of the choice of representation, **Theorem 5.1** is an immediate corollary of the fact that every finite function, and so in particular the function \(\text{EVAL}_{s,n,m}\) above, can be computed by some NAND program.

**Theorem 5.1** can be thought of as providing a “NAND interpreter in NAND”. That is, for a particular size bound, we give a single NAND program that can evaluate all NAND programs of that size. We call the NAND program \(U_{s,n,m}\) that computes \(\text{EVAL}_{s,n,m}\) a bounded

\(^1\) If \(P\) does not describe a program then we don’t care what \(\text{EVAL}_{s,n,m}(P,x)\) is, but for concreteness we will set it to be \(0^m\).
universal program. “Universal” stands for the fact that this is a single program that can evaluate arbitrary code, where “bounded” stands for the fact that $U_{s,n,m}$ only evaluates programs of bounded size. Of course this limitation is inherent for the NAND programming language where an $N$-line program can never compute a function with more than $N$ inputs. (We will later on introduce the concept of loops, that allows to escape this limitation.)

It turns out that we don’t even need to pay that much of an overhead for universality.

**Theorem 5.2 — Efficient bounded universality of NAND programs.** For every $s, n, m \in \mathbb{N}$ there is a NAND program of at most $O(s \log s)$ lines that computes the function $\text{EVAL}_{s,n,m} : \{0,1\}^{s+n} \rightarrow \{0,1\}^m$ defined above.

We will prove a weaker version of Theorem 5.2, that will use a large number of $O(s^3 \log s)$ lines instead of $O(s \log s)$ as stated in the theorem. We will sketch how we can improve this proof and get the $O(s \log s)$ bound in the next lecture. Unlike Theorem 5.1, Theorem 5.2 is not a trivial corollary of the fact that every function can be computed, and takes much more effort to prove. It requires us to present a concrete NAND program for the $\text{EVAL}_{s,n,m}$ function. We will do so in several stages.

First, we will spell out precisely how to represent NAND programs as strings. We can prove Theorem 5.2 using the ASCII representation, but a “cleaner” representation will be more convenient for us. Then, we will show how we can write a program to compute $\text{EVAL}_{s,n,m}$ in Python. Finally, we will show how we can transform this Python program into a NAND program.

### 5.2 Concrete representation for NAND programs

Every line in a NAND program has the form

```plaintext
foo := bar NAND baz
```

Since the actual labels for the variables are meaningless (except for designating if they are input, output or “workspace” variables), we can encode them as numbers from 0 to $t - 1$, where $t$ is a bound on the largest index and the number of distinct variables used in the program. (We can set $t = 3s$, where $s$ is the number of lines in the program, since every line in the program involves at most three distinct variables.) We encode a variable of the form $\text{foo}_i$ as the
Figure 5.2: In the Harvard Mark I computer, a program was represented as a list of triples of numbers, which were then encoded by perforating holes in a control card.

A pair of natural numbers \((a, i)\) where \(a\) is the number corresponding to the label \(\text{foo}\). Thus we encode a line of the form

\[
\text{foo}_{-(i)} := \text{bar}_{-(j)} \text{ NAND } \text{baz}_{-(k)}
\]

as a six-tuple \((a, i, b, j, c, k)\) where \(a, b, c\) are the numbers corresponding to \(\text{foo}, \text{bar}, \text{baz}\) respectively. We encode a NAND program of \(s\) lines as simply a list of \(s\) six-tuples.³ For example, the XOR program:

\[
\begin{align*}
\text{u}_{-0} & := x_{-0} \text{ NAND } x_{-1} \\
\text{v}_{-0} & := x_{-0} \text{ NAND } \text{u}_{-0} \\
\text{w}_{-0} & := x_{-1} \text{ NAND } \text{u}_{-0} \\
\text{y}_{-0} & := v_{-0} \text{ NAND } \text{w}_{-0}
\end{align*}
\]

is represented by the following list of four six-tuples:

\[
\begin{align*}
[2, 0, 0, 0, 0, 1], \\
[3, 0, 0, 0, 2, 0], \\
[4, 0, 0, 1, 2, 0], \\
[1, 0, 3, 0, 4, 0]
\end{align*}
\]

³If a variable \(\text{foo}\) appears in the program without an index, then we assume that this is the same as \(\text{foo}_{-0}\).

Note that even if we renamed \(u, v\) and \(w\) to \(\text{foo}, \text{bar}\) and \(\text{blah}\) then the representation of the program will remain the same (which is fine, since it does not change its semantics).
It is very easy to transform a string containing the program code to a list-of-tuples representation; for example, it can be done in 15 lines of Python.

To evaluate a NAND program $P$ given in this representation, on an input $x$, we will do the following:

- We create an array $\text{avars}$ of $ts$ integers. The value of the variable with label $a$ and index $i$ will be stored in the $t \cdot i + a$ location of this array.
- We initialize the value of the input variables. We set 0 to be the index corresponding to the label $x$, and so to initialize the value of the $x_{(i)}$ variables we set the $(t \cdot i)\text{th}$ coordinate of $\text{avars}$ to $x_i$ for every $i \in [n]$.
- For every line $(a, i, b, j, c, k)$ in the program, we read from $\text{avars}$ the values $x, y$ of the variables $(b, j)$ and $(c, k)$ respectively, and then set the value of the variable $(a, i)$ to $\text{NAND}(x, y) = 1 - x \cdot y$. That is, we set $\text{avars}[i \cdot t + a] = 1 - \text{avars}[j \cdot t + b] \cdot \text{avars}[k \cdot t + c]$.
- We set 1 to be the index corresponding to the label $y$ and so the output is the value of the variables $(1, 0), \ldots, (0, m - 1)$ which are equal to $\text{avars}[0 \cdot t + 1], \ldots, \text{avars}[(m - 1) \cdot t + 1]$.

The following is a Python function `EVAL` that on input $n, m, P, x$ where $P$ is a list of six-tuples and $x$ is list of 0/1 values, returns the result of the execution of the NAND program represented by $P$ on $x$.

```python
# Evaluates an n-input, m-output NAND program P on input x
# P is given in the list of tuples representation
def EVAL(n, m, P, x):
    s = len(P)  # no. of lines in the program
    t = 3 * len(P)  # maximum no. of unique labels
    avars = [0] * (t * s)  # initialize array to 0
    for i in range(n):  # initialize inputs to x
        avars[i * t] = x[i]

    for (a, i, b, j, c, k) in P:  # evaluate every line of program
        avars[i * t + a] = 1 - avars[j * t + b] * avars[k * t + c]

    # return y_0...y_(m-1) which is
    # avars[1], avars[t+1], ..., avars[(m-1)*t+1]
    return [avars[i * t + 1] for i in range(m)]
```

TODO: Perhaps comment that we will not use the indices 2 and 3 as to maintain compatibility with the representation of NAND++ that will be introduced later on.

To keep things simple, we will not worry about the case that $P$ does not represent a valid program of $n$ inputs and $m$ outputs.
For example, if we run

\[
\text{EVAL}(2, 1,
[\{2, 0, 0, 0, 0, 1\},
[3, 0, 0, 0, 2, 0],
[4, 0, 0, 1, 2, 0],
[1, 0, 3, 0, 4, 0]\},
[0, 1])
\]

then this corresponds to running our XOR program on the input \((0, 1)\) and hence the resulting output is \([1]\).

Accessing an element of the array \(\text{avars}\) at a given index takes a constant number of basic operations.\(^7\) Hence (since \(n, m \leq s\)), apart from the initialization phase that costs \(O(ts) = O(s^2)\) steps, the program above will use \(O(s)\) basic operations.

## 5.3 A NAND interpreter in NAND

To prove Theorem 5.2 it is of course not enough to give a Python program. We need to transform the code above to a NAND program that will compute the function \(\text{EVAL}_{n,m,s}\) that takes a (representation of) \(s\)-line NAND program \(P\) computing a function on \(n\) inputs and with \(m\) outputs, and an input \(w \in \{0, 1\}^n\), and outputs \(P(w)\).

The total number of distinct labels and indices in an \(s\) line program is at most \(t = 3s\). Hence we can think of the six-tuple representation of such a program as simply a sequence of \(ts\) numbers between 0 and \(t - 1\), each of which can be represented by \(\ell = \lceil \log t \rceil\) bits, meaning that such a program \(P\) can be identified with a string in \(\{0, 1\}^{6s\ell}\).

The NAND program takes an input \((P, w)\) of length \(6s\ell + n\) and needs to output the result of applying \(P\) to \(w\). It will follow very closely the Python implementation above:

- We define variables \(\text{avars}_0, \ldots, \text{avars}_{(t \cdot s)}\).
- Every line of the program is represented by a string of length \(6\ell\), which we can think of as having the form \((a, i, b, j, c, k)\) with \(a, i, b, j, c, k\) being numbers in \(\{0, \ldots, t - 1\}\) represented using the binary representation as \(\ell\)-bit strings.
- We will go over every line of the original program and execute it just as the Python program does: we will retrieve the values corresponding to \((b, j)\) and \((c, k)\) in \(\text{avars}\), and then update \(\text{avars}\) in the location corresponding to \((a, i)\) to the NAND of these values.

\(^7\) Python does not distinguish between lists and arrays, but allows constant time random access to an indexed elements to both of them. One could argue that if we allowed programs of truly unbounded length (e.g., larger than \(2^{64}\)) then the price would not be constant but logarithmic in the length of the array/lists, but the difference between \(O(1)\) and \(O(\log s)\) will not be important for our discussions.
Since we have algorithms for addition and multiplication, we can transform a pair \((a, i)\) to the index \(u = t \cdot i + a\) in \(O(t^2)\) time. Hence to evaluate such a line, we need to be able to compute the following functions:

- \text{GETVAL}, which maps a string \(\text{vars} \in \{0,1\}^N\) and an index \(u \in \{0, \ldots, N-1\}\) (which we also think of as a string in \(\{0,1\}^{\log N}\)), outputs \(\text{var}_u\).

- \text{SETVAL}, which maps a string \(\text{vars} \in \{0,1\}^N\), an index \(u \in \{0, \ldots, N-1\}\) (which as above can also be considered as a string in \(\{0,1\}^{\log N}\)), and a value \(v \in \{0,1\}\), returns \(\text{vars}' \in \{0,1\}^N\) such that \(\text{vars}'\) is the same as \(\text{vars}\) except that \(\text{vars}'_u = v\).

By rounding up \(st\) to the nearest power of 2 (which at most doubles its size), we can assume the array \(\text{avars}\) is of length \(2^k\) for \(k = \lceil \log(st) \rceil\). What is left is to show that we can implement both \text{GETVAL} and \text{SETVAL} in \(O(2^k)\) lines.

1. A moment of reflection shows that \text{GETVAL} is simply the \text{LOOKUP}_k function which we know can be implemented with \(O(2^k) = O(st)\) lines.

2. The function \text{SETVAL} is also very simple. For every index \(u'\), the \(u'^{th}\) output of \text{SETVAL} is equal to \(\text{vars}_{u'}\) if \(u \neq u'\) and is equal to \(v\) otherwise. So, to compute \text{SETVAL} we need to compute for every \(u'\) the function \(\text{EQUAL}_{u'} : \{0,1\}^{\log N} \to \{0,1\}\) such that \(\text{EQUAL}_{u'}(u') = 1\) and \(\text{EQUAL}_{u'}(u) = 0\) for every \(u \neq u'\). That is \(\text{EQUAL}_{u'}(u)\) corresponds to the AND of the \(\log N\) conditions \(\text{ui} = u'_i\) for \(i \in \{0, \ldots, \log N\}\). It is not hard to verify that we can compute \(\text{EQUAL}_{u'}\) in \(O(\log N)\) lines. Since \(\text{SETVAL}(\text{vars}, u, v)_{u'} = \text{LOOKUP}_1(\text{vars}_{u'}, v, \text{EQUAL}_{u'}(u))\), we get that \text{SETVAL} can be computed using \(O(N \log N)\) lines.

The total cost to compute \text{EVAL} will be \(s\) times the cost to compute two \text{GETVAL’s} and one \text{SETVAL}, which will come up to \(O(s(st) \log(st)) = O(s^3 \log s)\). This completes the proof of Theorem 5.2.

If you want to see the actual resulting code, the website \text{http://nandpl.org} contains (or will eventually contain..) the full implementation of this NAND program where you can also play with it by feeding it various other programs as inputs. The NAND program above is less efficient that its Python counterpart, since NAND does not offer arrays with efficient random access, and hence the \text{LOOKUP} operation on an array of \(N\) bits takes \(\Omega(N)\) lines in NAND even though it takes \(O(1)\) steps (or maybe \(O(\log N)\) steps,
More concretely, if the Python program takes \( T(n) \) operations on inputs of length at most \( n \) then we can find a NAND program of \( O(T(n) \log T(n)) \) lines that agrees with the Python program on inputs of length \( n \).

ARM stands for "Advanced RISC Machine" where RISC in turn stands for "Reduced instruction set computer."

The reverse direction of compiling NAND to C code, is much easier. We show code for a \texttt{NAND2C} function in the appendix.

There are even simpler machine languages, such as the LEG architecture for which a backend for the LLVM compiler was implemented (and hence can be the target of compiling any of large and growing list of languages that this compiler supports). Other examples include the TinyRAM architecture (motivated by interactive proof systems that we will discuss much later in this course) and the teaching-oriented Ridiculously Simple Computer architecture.

Going one by one over the instruction sets of such computers and translating them to NAND snippets is no fun, but it is a feasible thing to do. In fact, ultimately this is very similar to the transformation that takes place in converting our high level code to actual silicon gates that (as we will see in the next lecture) are not so different from the operations of a NAND program.

Indeed, tools such as MyHDL that transform "Python to Silicon" can be used to convert a Python program to a NAND program.

To prove Theorem 5.2 we essentially translated every line of the Python program for \texttt{EVAL} into an equivalent NAND snippet. It turns out that none of our reasoning was specific to the particular function \texttt{EVAL}. It is possible to translate every Python program into an equivalent NAND program of comparable efficiency. Actually doing so requires taking care of many details and is beyond the scope of this course, but let me convince you why you should believe it is possible in principle. We can use CPython (the reference implementation for Python), to evaluate every Python program using a C program. We can combine this with a C compiler to transform a Python program to various flavors of "machine language."

So, to transform a Python program into an equivalent NAND program, it is enough to show how to transform a machine language program into an equivalent NAND program. One minimalistic (and hence convenient) family of machine languages is known as the ARM architecture which powers a great many mobile devices including essentially all Android devices.

There are even simpler machine languages, such as the LEG architecture for which a backend for the LLVM compiler was implemented (and hence can be the target of compiling any of large and growing list of languages that this compiler supports). Other examples include the TinyRAM architecture (motivated by interactive proof systems that we will discuss much later in this course) and the teaching-oriented Ridiculously Simple Computer architecture.

Going one by one over the instruction sets of such computers and translating them to NAND snippets is no fun, but it is a feasible thing to do. In fact, ultimately this is very similar to the transformation that takes place in converting our high level code to actual silicon gates that (as we will see in the next lecture) are not so different from the operations of a NAND program.

Indeed, tools such as MyHDL that transform "Python to Silicon" can be used to convert a Python program to a NAND program.

\* More concretely, if the Python program takes \( T(n) \) operations on inputs of length at most \( n \) then we can find a NAND program of \( O(T(n) \log T(n)) \) lines that agrees with the Python program on inputs of length \( n \).

\* ARM stands for "Advanced RISC Machine" where RISC in turn stands for "Reduced instruction set computer."

\* The reverse direction of compiling NAND to C code, is much easier. We show code for a \texttt{NAND2C} function in the appendix.
The NAND programming language is just a teaching tool, and by no means do I suggest that writing NAND programs, or compilers to NAND, is a practical, useful, or even enjoyable activity. What I do want is to make sure you understand why it can be done, and to have the confidence that if your life (or at least your grade in this course) depended on it, then you would be able to do this. Understanding how programs in high level languages such as Python are eventually transformed into concrete low-level representation such as NAND is fundamental to computer science.

The astute reader might notice that the above paragraphs only outlined why it should be possible to find for every particular Python-computable function $F$, a particular comparably efficient NAND program $P$ that computes $F$. But this still seems to fall short of our goal of writing a “Python interpreter in NAND” which would mean that for every parameter $n$, we come up with a single NAND program $\text{UNIV}_n$ such that given a description of a Python program $P$, a particular input $x$, and a bound $T$ on the number of operations (where the length of $P$, $x$ and the magnitude of $T$ are all at most $n$) would return the result of executing $P$ on $x$ for at most $T$ steps. After all, the transformation above would transform every Python program into a different NAND program, but would not yield “one NAND program to rule them all” that can evaluate every Python program up to some given complexity. However, it turns out that it is enough to show such a transformation for a single Python program. The reason is that we can write a Python interpreter in Python: a Python program $U$ that takes a bit string, interprets it as Python code, and then runs that code. Hence, we only need to show a NAND program $U^*$ that computes the same function as the particular Python program $U$, and this will give us a way to evaluate all Python programs.

What we are seeing time and again is the notion of universality or self reference of computation, which is the sense that all reasonably rich models of computation are expressive enough that they can “simulate themselves”. The importance of this phenomena to both the theory and practice of computing, as well as far beyond it, including the foundations of mathematics and basic questions in science, cannot be overstated.

5.5 Counting programs, and lower bounds on the size of NAND programs

One of the consequences of our representation is the following:
**Theorem 5.3 — Counting programs.** There are at most $2^{O(s \log s)}$ functions computed by $s$-line NAND programs.

Moreover, the implicit constant in the $O(\cdot)$ notation in Theorem 5.3 is at most 10. Using the notation introduced in the last lecture, another way to state Theorem 5.3, is that for every $n, m, s$, $|SIZE_{n,m}(s)| \leq 2^{10s \log s}$. The idea of the proof is that because every such program can be represented by a binary string of at most $10s \log s$ bits, the number of functions they compute cannot be larger than the number of such strings. Let us now show the formal proof.

**Proof.** Every NAND program with $s$ lines has at most $s$ inputs, $s$ outputs, and $s$ workspace variables. Hence it can be represented by $s$ six-tuples of numbers in $\{0, \ldots, 3s-1\}$. If two programs compute distinct functions then they have distinct representations.

Let $T_s$ be the set of lists of at most $6s$ numbers between $\{0, \ldots, 3s-1\}$. Note that $|T_s| = \sum_{t=1}^{6s} (3s)^t \leq (6s)(3s)^{6s} \leq 2^{10s \log s}$. Let $F_s$ be the set of functions from $\{0, 1\}^* \to \{0, 1\}^*$ that can be computed by $s$-line NAND programs. We can define a one-to-one map $R : F_s \to T_s$ by setting for every function $F \in F_s$ the value $R(F)$ to be the representation of the shortest NAND program that computes the function $F$ (breaking ties arbitrarily). Note that $R(F)$ is indeed in $T_s$ since for every $F \in F_s$, the shortest program that computes it will have at most $s$ lines. Since $F \neq F'$ means that the programs $R(F)$ and $R(F')$ compute different functions, and hence have distinct representations, $R$ is a one-to-one function implying that $|F_s| \leq |T_s| \leq 2^{10s \log s}$.

**Note:** We can also establish Theorem 5.3 directly from the ASCII representation of the source code. Since an $s$-line NAND program has at most $3s$ distinct variables, we can change all the workspace variables of such a program to have the form `work_⟨i⟩` for $i$ between 0 and $3s-1$ without changing the function that it computes. This means that after removing comments and extra whitespaces, every line of such a program (which will have the form `var := var' NAND var"` for variable identifiers which will be either `x_###`, `y_###` or `work_###` where `###` is some number smaller than $3s$) will require at most, say, $20 + 3 \log_{10}(3s) \leq O(\log s)$ characters. Since each one of those characters can be encoded using seven bits in the ASCII representation, we see that the number of functions computed by $s$-line NAND programs is at most $2^{O(s \log s)}$.

A function mapping $\{0, 1\}^2$ to $\{0, 1\}$ can be identified with the
table of its four values on the inputs 00, 01, 10, 11; a function mapping \( \{0, 1\}^3 \) to \( \{0, 1\} \) can be identified with the table of its eight values on the inputs 000, 001, 010, 100, 101, 110, 111. More generally, every function \( F : \{0, 1\}^n \rightarrow \{0, 1\} \) can be identified with the table of its \( 2^n \) values on the inputs \( \{0, 1\}^n \). Hence the number of functions mapping \( \{0, 1\}^n \) to \( \{0, 1\} \) is equal to the number of such tables which (since we can choose either 0 or 1 for every row) is exactly \( 2^{2^n} \). This has the following interesting corollary:

**Theorem 5.4 — Counting argument lower bound.** There is a function \( F : \{0, 1\}^n \rightarrow \{0, 1\} \) such that the shortest NAND program to compute \( F \) requires \( 2^n/(100n) \) lines.

*Proof.* Suppose, towards the sake of contradiction, that every function \( F : \{0, 1\}^n \rightarrow \{0, 1\} \) can be computed by a NAND program of at most \( s = 2^n/(100n) \) lines. Then the by Theorem 5.3 the total number of such functions would be at most \( 2^{10n \log s} \leq 2^{10n \log s} 2^n/(100n) \). Since \( \log s = n - \log(100n) \leq n \) this means that the total number of such functions would be at most \( 2^{2n}/10 \), contradicting the fact that there are \( 2^{2^n} \) of them. \( \square \)

We have seen before that *every* function mapping \( \{0, 1\}^n \) to \( \{0, 1\} \) can be computed by an \( O(2^n/n) \) line program. We now see that this is tight in the sense that some functions do require such an astronomical number of lines to compute. In fact, as we explore in the exercises below, this is the case for *most* functions. Hence functions that can be computed in a small number of lines (such as addition, multiplication, finding short paths in graphs, or even the \( \text{EVAL} \) function) are the exception, rather than the rule.

### 5.6 Lecture summary

- We can think of programs both as describing a *process*, as well as simply a list of symbols that can be considered as *data* that can be fed as input to other programs.
- We can write a NAND program that evaluates arbitrary NAND programs. Moreover, the efficiency loss in doing so is not too large.
- We can even write a NAND program that evaluates programs in other programming languages such as Python, C, Lisp, Java, Go, etc.
138 Introduction to Theoretical Computer Science

Figure 5.3: All functions mapping $n$ bits to $m$ bits can be computed by NAND programs of $O(n2^n/n)$ lines, but most functions cannot be computed using much smaller programs. However there are many important exceptions which are functions such as addition, multiplication, program evaluation, and many others, that can be computed in polynomial time with a small exponent.

5.7 Exercises

Exercise 5.1 Which one of the following statements is false:

a. There is an $O(s^3)$ line NAND program that given as input program $P$ of $s$ lines in the list-of-tuples representation computes the output of $P$ when all its input are equal to 1.

b. There is an $O(s^3)$ line NAND program that given as input program $P$ of $s$ characters encoded as a string of $7s$ bits using the ASCII encoding, computes the output of $P$ when all its input are equal to 1.

c. There is an $O(\sqrt{s})$ line NAND program that given as input program $P$ of $s$ lines in the list-of-tuples representation computes the output of $P$ when all its input are equal to 1.

Exercise 5.2 — Equals function. For every $k$, show that there is an $O(k)$ line NAND program that computes the function $EQUALS_k : \{0,1\}^{2k} \rightarrow \{0,1\}$ where $EQUALS(x,x') = 1$ if and only if $x = x'$.

Exercise 5.3 — Improved evaluation. Show that there is an $O(s^2 \log s)$-line NAND program to evaluate $EVAL_{n,m,s}$.

Exercise 5.4 — Random functions are hard (challenge). Suppose $n > 1000$ and that we choose a function $F : \{0,1\}^n \rightarrow \{0,1\}$ at random,

11 Hint: Show that the array avar will always have at most $O(s)$ nonzero values, and use that to give a more compact representation for it.
choosing for every \( x \in \{0, 1\}^n \) the value \( F(x) \) to be the result of tossing an independent unbiased coin. Prove that the probability that there is a \( 2^n / (1000n) \) line program that computes \( F \) is at most \( 2^{-100} \).

**Exercise 5.5 — Circuit hierarchy theorem (challenge).** Prove that there is a constant \( c \) such that for every \( n \), there is some function \( F : \{0, 1\}^n \to \{0, 1\} \) s.t. (1) \( F \) can be computed by a NAND program of at most \( cn^5 \) lines, but (2) \( F \) can not be computed by a NAND program of at most \( n^4 / c \) lines.

---

### 5.8 Bibliographical notes

**5.9 Further explorations**

Some topics related to this lecture that might be accessible to advanced students include:

- Lower bounds. While we’ve seen the “most” functions mapping \( n \) bits to one bit require NAND programs of exponential size \( \Omega(2^n/n) \), we actually do not know of any explicit function for which we can prove that it requires, say, at least \( n^{100} \) or even \( 100n \) size. At the moment, strongest such lower bound we know is that there are quite simple and explicit \( n \)-variable functions that require at least \( (5 - o(1))n \) lines to compute, see this paper of Iwama et al as well as this more recent work of Kulikov et al. Proving lower bounds for restricted models of straightline programs (more often described as circuits) is an extremely interesting research area, for which Jukna’s book provides very good introduction and overview.

---

**5.10 Acknowledgements**
Physical implementations of NAND programs

“In existing digital computing devices various mechanical or electrical devices have been used as elements: Wheels, which can be locked . . . which on moving from one position to another transmit electric pulses that may cause other similar wheels to move; single or combined telegraph relays, actuated by an electromagnet and opening or closing electric circuits; combinations of these two elements;—and finally there exists the plausible and tempting possibility of using vacuum tubes”, John von Neumann, first draft of a report on the EDVAC, 1945

We have defined NAND programs as a model for computation, but is this model only a mathematical abstraction, or is it connected in some way to physical reality? For example, if a function $F : \{0, 1\}^n \to \{0, 1\}$ can be computed by a NAND program of $s$ lines, is it possible, given an actual input $x \in \{0, 1\}^n$, to compute $F(x)$ in the real world using an amount of resources that is roughly proportional to $s$?

In some sense, we already know that the answer to this question is Yes. We have seen a Python program that can evaluate NAND programs, and so if we have a NAND program $P$, we can use any computer with Python installed on it to evaluate $P$ on inputs of our choice. But do we really need modern computers and programming languages to run NAND programs? And can we understand more directly how we can map such programs to actual physical processes that produce an output from an input? This is the content of this lecture.

We will also talk about the following “dual” question. Suppose we have some way to compute a function $F : \{0, 1\}^n \to \{0, 1\}$ using
roughly an $s$ amount of “physical resources” such as material, energy, time, etc.. Does this mean that there is also a NAND program to compute $F$ using a number of lines that is not much bigger than $s$? This might seem like a wishful fantasy, but we will see that the answer to this question might be (up to some important caveats) essentially Yes as well.

6.1 Physical implementation of computing devices.

Computation is an abstract notion, that is distinct from its physical implementations. While most modern computing devices are obtained by mapping logical gates to semi-conductor based transistors, over history people have computed using a huge variety of mechanisms, including mechanical systems, gas and liquid (known as fluidics), biological and chemical processes, and even living creatures (e.g., see Fig. 6.1 or this video for how crabs or slime mold can be used to do computations).

In this lecture we review some of these implementations, both so you can get an appreciation of how it is possible to directly translate NAND programs to the physical world, without going through the entire stack of architecture, operating systems, compilers, etc... as well as to emphasize that silicon-based processors are by no means the only way to perform computation. Indeed, as we will see much later in this course, a very exciting recent line of works involves using different media for computation that would allow us to take advantage of quantum mechanical effects to enable different types of algorithms.

6.2 Transistors and physical logic gates

A transistor can be thought of as an electric circuit with two inputs, known as source and gate and an output, known as the sink. The gate controls whether current flows from the source to the sink. In a standard transistor, if the gate is “ON” then current can flow from the source to the sink and if it is “OFF” then it can’t. In a complimentary transistor this is reversed: if the gate is “OFF” then current can flow from the source to the sink and if it is “ON” then it can’t.

There are several ways to implement the logic of a transistor. For example, we can use faucets to implement it using water pressure (e.g. Fig. 6.2).¹ However, the standard implementation uses electrical current. One of the original implementations used vacuum tubes.

¹ This might seem as curiosity but there is a field known as fluidics concerned with implementing logical operations using liquids or gasses. Some of the motivations include operating in extreme environmental conditions such as in space or a battlefield, where standard electronic equipment would not survive.
Figure 6.1: Crab-based logic gates from the paper “Robust soldier-crab ball gate” by Gunji, Nishiyama and Adamatzky. This is an example of an AND gate that relies on the tendency of two swarms of crabs arriving from different directions to combine to a single swarm that continues in the average of the directions.

Figure 6.2: We can implement the logic of transistors using water. The water pressure from the gate closes or opens a faucet between the source and the sink.
As its name implies, a vacuum tube is a tube containing nothing (i.e., vacuum) and where a priori electrons could freely flow from source (a wire) to the sink (a plate). However, there is a gate (a grid) between the two, where modulating its voltage can block the flow of electrons.

Early vacuum tubes were roughly the size of lightbulbs (and looked very much like them too). In the 1950’s they were supplanted by transistors, which implement the same logic using semiconductors which are materials that normally do not conduct electricity but whose conductivity can be modified and controlled by inserting impurities (“doping”) and an external electric field (this is known as the field effect). In the 1960’s computers were started to be implemented using integrated circuits which enabled much greater density. In 1965, Gordon Moore predicted that the number of transistors per circuit would double every year (see Fig. 6.3), and that this would lead to “such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment”. Since then, (adjusted versions of) this so-called “Moore’s law” has been running strong, though exponential growth cannot be sustained forever, and some physical limitations are already becoming apparent.

Figure 6.3: The number of transistors per integrated circuits from 1959 till 1965 and a prediction that exponential growth will continue at least another decade. Figure taken from “Cramming More Components onto Integrated Circuits”, Gordon Moore, 1965
Figure 6.4: Gordon Moore’s cartoon “predicting” the implications of radically improving transistor density.

Figure 6.5: The exponential growth in computing power over the last 120 years. Graph by Steve Jurvetson, extending a prior graph of Ray Kurzweil.
6.3 Gates and circuits

We can use transistors to implement a NAND gate, which would be a system with two input wires $x, y$ and one output wire $z$, such that if we identify high voltage with “1” and low voltage with “0”, then the wire $z$ will equal to “1” if and only if the NAND of the values of the wires $x$ and $y$ is 1 (see Fig. 6.6).

![Implementing a NAND gate using transistors.](image)

More generally, we can use transistors to implement the model of Boolean circuits. We list the formal definition below, but let us start with the informal one:

Let $B$ be some set of functions (known as “gates”) from $\{0, 1\}^k$ to $\{0, 1\}$. A Boolean circuit with the basis $B$ is obtained by connecting “gates” which compute functions in $B$ together by “wires” where each gate has $k$ wires going into it and one wire going out of it. We have $n$ special wires known as the “input wires” and $m$ special wires known as the “output wires”. To compute a function $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ using a circuit, we feed the bits of $x$ to the $n$ input wires, and then each gate computes the corresponding function, and we “read off” the output $y \in \{0, 1\}^m$ from the $m$ output wires.
The number $k$ is known as the \textit{arity} of the basis $B$. We think of $k$ as a small number (such as $k = 2$ or $k = 3$) and so the idea behind a Boolean circuit is that we can compute complex functions by combining together the simple components which are the functions in $B$. It turns out that NAND programs correspond to circuits where the basis is the single function $\text{NAND} : \{0,1\}^2 \rightarrow \{0,1\}$. However, as we’ve seen, we can simulate any $k$-arity basis $B$ using NAND gates with a blowup of at most a $4 \cdot 2^k$ factor in the number of gates. So, as long as we think of $k$ as small, the choice of basis does not make much difference.

6.4 Boolean circuits: a formal definition

We now define Boolean circuits more formally using the notion of labeled directed acyclic graphs (DAGs).

\textbf{Definition 6.1 — Boolean circuits.} Let $k$ be some number and $B$ be a subset of the functions from $\{0,1\}^k \rightarrow \{0,1\}$. For every $n, m \in \mathbb{N}$, an $n$ input, $m$ output Boolean circuit with $B$-gates is a directed acyclic graph (DAG) $G$ over the vertex set $[s] = \{0, 1, \ldots, s - 1\}$ where every vertex is labeled with either a function $f \in B$ or a number $i \in \{0, \ldots, \max\{m, n\} - 1\}$ such that:

* Every source vertex (vertex without incoming edges) is labeled with a number between 0 and $n - 1$.
* Every sink vertex (vertex without outgoing edges) has only a single incoming edge and is labeled with a number between 0 and $m - 1$. There should be exactly $m$ sink vertices and every one of them gets a unique label.
* Every other vertex has exactly $k$ incoming edges and is labeled with a function $f \in B$.

6.4.1 Evaluating a function using circuits

An $n$-input $m$-output circuit $C$ computes a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ as follows. For every input $x \in \{0,1\}^n$, we inductively define the value of every vertex based on its incoming edges:

- For a source vertex $v$ labeled with an integer $i \in \{0, \ldots, n - 1\}$, we define the value $\text{val}(v)$ of $v$ to be $x_i$.
- For a vertex $v$ that’s neither a sink nor a source and is labeled with
Figure 6.7: A Boolean circuit to compute the XOR function with NAND gates. In red are the corresponding variable names in the NAND program.

\[ f \in B, \text{ if its incoming neighbors are vertices } v_1, \ldots, v_k \text{ (sorted in order) then we let } \text{val}(v) = f(\text{val}(v_1), \ldots, \text{val}(v_k)). \]

- Sink vertices get the same value of their sole incoming neighbor.

The output of the circuit on input \( x \) is the string \( y \in \{0, 1\}^m \) such that for every \( i \in \{0, \ldots, m - 1\} \), \( y_i \) is the value of the sink vertex labeled with \( i \). We say that the circuit \( C \) computes the function \( F \) if for every \( x \in \{0, 1\}^n \), the output of the circuit \( C \) on input \( x \) is equal to \( F(x) \).

### 6.4.2 Circuits and NAND programs

Boolean circuits with the basis \( B \) consisting of the single function \( \text{NAND} : \{0, 1\}^2 \to \{0, 1\} \) that maps \( x, y \in \{0, 1\} \) to \( 1 - xy \) directly correspond to NAND programs. For example the program

\[
\begin{align*}
    u & := x_0 \ \text{NAND} \ x_1 \\
    v & := x_0 \ \text{NAND} \ u \\
    w & := x_1 \ \text{NAND} \ u \\
    y_0 & := v \ \text{NAND} \ w
\end{align*}
\]

corresponds to the circuit of Fig. 6.7. Every line in the program will correspond to a gate (i.e., non sink and non-source vertex) in the graph, where the input variables \( x_<(i) \) and output variables \( y_<(i) \) correspond to the sources and the sink vertices. This is stated in the following theorem:
**Theorem 6.1** — NAND programs are equivalent to NAND circuits. For every function \( F : \{0,1\}^n \rightarrow \{0,1\}^m \), if we let \( S(f) \) denote the smallest number of lines in a NAND program that computes \( F \) and \( S'(f) \) denote the smallest number of vertices in a Boolean circuit with the basis \( B = \{NAND\} \) then

\[
S(f) \leq S'(f) \leq S(f) + n + m + 10 \tag{6.1}
\]

To prove **Theorem 6.1** we need to show two statements. The first statement is that given an \( s' \)-vertex circuit \( C \), we can find an \( s' \) line NAND program that computes the same function as \( C \). The second statement is that given an \( s \)-line NAND program \( P \) with \( n \) inputs and \( m \) outputs, we can find a circuit of at most \( s + n + m + 10 \) vertices that computes the same function as \( P \). Both of these can be proven using the above correspondence, but we leave verifying the details as Exercise 6.1.

We have seen that every function \( f : \{0,1\}^k \rightarrow \{0,1\} \) has a NAND program with at most \( 4 \cdot 2^k \) lines, and hence **Theorem 6.1** implies the following theorem (see Exercise 6.2):

**Theorem 6.2** — NAND programs simulate all circuits. For every function \( F : \{0,1\}^n \rightarrow \{0,1\}^m \) and \( B \) a subset of the functions from \( \{0,1\}^k \) to \( \{0,1\} \), if we let \( S(f) \) denote the smallest number of lines in a NAND program that computes \( F \) and \( S_B(f) \) denote the smallest number of vertices in a Boolean circuit with the basis \( B \), then

\[
S(f) \leq (4 \cdot 2^k)S_B(f) \tag{6.2}
\]

One can ask whether there is an equivalence here as well. However, this is not the case. For example if the set \( B \) only consists of constant functions, then clearly a circuit whose gates are in \( B \) cannot compute any non-constant function. A slightly less boring example is if \( B \) is the \( \land \) (i.e. AND) function (as opposed to the NAND function). One can show that such a circuit will always output 0 on the all zero inputs, and hence it can never compute the simple negation function \( \neg : \{0,1\} \rightarrow \{0,1\} \) such that \( \neg(x) = 1 - x \).

We say that a subset \( B \) of functions from \( k \) bits to a single bit is a **universal basis** if there is a “\( B \)-circuit” (i.e., circuit all whose gates are labeled with functions in \( B \)) that computes the NAND function. **Exercise 6.3** asks you to explore some examples of universal and non-universal bases.
6.5 Neural networks

One particular basis we can use are threshold gates. Given any vector \( w = (w_0, \ldots, w_{k-1}) \) of integers and some integer \( t \) (some or all of whom could be negative), the threshold function corresponding to \( w, t \) is the function \( T_{w, t} : \{0, 1\}^k \to \{0, 1\} \) that maps \( w \) to 1 if and only if \( \sum_{i=0}^{k-1} w_i x_i \geq t \).

The NAND function is of course itself a threshold gate, since \( \text{NAND}(x, y) = 1 \) if and only if \( -x - y \geq -1 \). Threshold gates can be thought of as an approximation for neuron cells that make up the core of human and animal brains. To a first approximation, a neuron has \( k \) inputs and a single output and the neurons “fires” or “turns on” its output when those signals pass some threshold. Hence circuits with threshold gates are sometimes known as neural networks. Unlike the cases above, when we considered \( k \) to be a small constant, in such neural networks we often do not put any bound on the number of inputs. However, since any threshold function can be computed by a NAND program of \( \text{poly}(k) \) lines (see Exercise 6.5), the power of NAND programs and neural networks is not much different.

6.6 Biological computing

Computation can be based on biological or chemical systems. For example the lac operon produces the enzymes needed to digest lactose only if the conditions \( x \land (\neg y) \) hold where \( x \) is “lactose is present” and \( y \) is “glucose is present”. Researchers have managed to create transistors, and from them the NAND function and other logic gates, based on DNA molecules (see also Fig. 6.8). One motivation for DNA computing is to achieve increased parallelism or storage density; another is to create “smart biological agents” that could perhaps be injected into bodies, replicate themselves, and fix or kill cells that were damaged by a disease such as cancer. Computing in biological systems is not restricted of course to DNA. Even larger systems such as flocks of birds can be considered as computational processes.

6.7 Cellular automata and the game of life

As we will discuss later, cellular automata such as Conway’s “Game of Life” can be used to simulate computation gates, see Fig. 6.9.
Figure 6.8: Performance of DNA-based logic gates. Figure taken from paper of Bonnet et al, Science, 2013.

Figure 6.9: An AND gate using a “Game of Life” configuration. Figure taken from Jean-Philippe Rennard’s paper.
6.8 Circuit evaluation algorithm

A Boolean circuit is a labeled graph, and hence we can use the adjacency list representation to represent an s-vertex circuit over an arity-k basis $B$ by $s$ elements of $B$ (that can be identified with numbers in $|B|$) and $s$ lists of $k$ numbers in $[s]$. Hence we can represent such a circuit by a string of length $O(s \log |B| + s \log s)$. We can define $\text{CIRCEVAL}_{B,s,n,m}$ to be the function that takes as input a pair $(C, x)$ where $C$ is string describing an $s$-size $n$-input $m$-output circuit over $B$, and $x \in \{0, 1\}^n$, and returns the evaluation of $C$ over $n$.

Theorem 6.2 implies that every circuit $C$ of $s$ gates over a $k$-ary basis $B$ can be transformed into a NAND program of $O(s \cdot 2^k)$ lines, and hence we can combine this transformation with last lecture’s evaluation procedure for NAND programs to conclude that $\text{CIRCEVAL}$ can be evaluated in time $O(2^k s^3 \text{poly}(\log s))$.

6.8.1 Advanced note: evaluating circuits in quasilinear time.

We can improve the evaluation procedure, and evaluate $s$-size constant arity circuits (or NAND programs) in time $O(s \text{ polylog}(s))$.\(^4\)

6.9 The physical extended Church-Turing thesis

We’ve seen that NAND gates can be implemented using very different systems in the physical world. What about the reverse direction? Can NAND programs simulate any physical computer?

We can take a leap of faith and stipulate that NAND programs do actually encapsulate every computation that we can think of. Such a statement (in the realm of infinite functions, which we’ll encounter in a couple of lectures) is typically attributed to Alonzo Church and Alan Turing, and in that context is known as the Church Turing Thesis. As we will discuss in future lectures, the Church-Turing Thesis is not a mathematical theorem or conjecture. Rather, like theories in physics, the Church-Turing Thesis is about mathematically modelling the real world. In the context of finite functions, we can make the following informal hypothesis or prediction:

If a function $F : \{0, 1\}^n \to \{0, 1\}^m$ can be computed in the physical world using $s$ amount of “physical resources” then it can be computed by a NAND program of roughly $s$ lines.
We call this hypothesis the "Physical Extended Church-Turing Thesis" or PECTT for short.

There is no single universally-agreed-upon formalization of "s physical resources", but we can approximate this notion by considering the size of any physical computing device and the time it takes to compute the output. That is we can stipulate that any function that can be computed by a device of volume $V$ and time $t$, must be computable by a NAND program that has at most $\alpha(Vt)^{\beta}$ lines for some constants $\alpha, \beta$. The exact values for $\alpha, \beta$ are not so clear, but it is generally accepted that if $F : \{0, 1\}^n \rightarrow \{0, 1\}$ is an exponentially hard function, in the sense that it has not NAND program of fewer than, say, $2^{n/2}$ lines, then a demonstration of a physical device that can compute $F$ for moderate input lengths (e.g., $n = 500$) would be a violation of the PECTT.

**Advanced note: making things concrete:** We can attempt at a more exact phrasing of the PECTT as follows. Suppose that $Z$ is a physical system that accepts $n$ binary stimuli and has a binary output, and can be enclosed in a sphere of volume $V$. We say that the system $Z$ computes a function $F : \{0, 1\}^n \rightarrow \{0, 1\}$ within $t$ seconds if whenever we set the stimuli to some value $x \in \{0, 1\}^n$, if we measure the output after $t$ seconds. We can phrase the PECTT as stipulating that whenever there exists such a system $Z$ computes $F$ within $t$ seconds, there exists a NAND program that computes $F$ of at most $\alpha(Vt)^2$ lines, where $\alpha$ is some normalization constant. In particular, suppose that $F : \{0, 1\}^n \rightarrow \{0, 1\}$ is a function that requires $2^n/(100n) > 2^{0.8n}$ lines for any NAND program (we have seen that such functions exist in the last lecture). Then the PECTT would imply that either the volume or the time of a system that computes $F$ will have to be at least $2^{0.2n}/\sqrt{\alpha}$. To fully make it concrete, we need to decide on the units for measuring time and volume, and the normalization constant $\alpha$. One conservative choice is to assume that we could squeeze computation to the absolute physical limits (which are many orders of magnitude beyond current technology). This corresponds to setting $\alpha = 1$ and using the Planck units for volume and time. The Planck length $\ell_P$ (which is, roughly speaking, the shortest distance that can be theoretically be measured) is roughly $2^{-120}$ meters. The Planck time $t_P$ (which is the time it takes for
We can also consider variants where we use surface area instead of volume, or use a different power than 2. However, none of these choices makes a qualitative difference to the discussion below.

There are of course several hurdles to refuting the PECTT in this way, one of which is that we can't actually test the system on all possible inputs. However, it turns out we can get around this issue using notions such as interactive proofs and program checking that we will see later in this course. Another, perhaps more salient problem, is that while we know many hard functions exist, at the moment there is no single explicit function $F : \{0, 1\}^n \rightarrow \{0, 1\}$ for which we can prove an $\omega(n)$ lower bound on the number of lines that a NAND program needs to compute it.

6.9.1 Attempts at refuting the PECTT

One of the admirable traits of mankind is the refusal to accept limitations. In the best case this is manifested by people achieving longstanding “impossible” challenges such as heavier-than-air flight, putting a person on the moon, circumnavigating the globe, or even resolving Fermat’s Last Theorem. In the worst-case it is manifested by people continually following the footsteps of previous failures to try to do proven-impossible tasks such as build a perpetual motion machine, trisect an angle with a compass and straightedge, or refute Bell’s inequality. The Physical Extended Church Turing thesis (in its various forms) has attracted both types of people. Here are some physical devices that have been speculated to achieve computational tasks that cannot be done by not-too-large NAND programs:

**Spaghetti sort:** One of the first lower bounds that Computer Science students encounter is that sorting $n$ numbers requires making $\Omega(n \log n)$ comparisons. The “spaghetti sort” is a description of a proposed “mechanical computer” that would do this faster. The idea is that to sort $n$ numbers $x_1, \ldots, x_n$, we could cut $n$ spaghetti noodles into lengths $x_1, \ldots, x_n$, and then if we simply hold them together in our hand and bring them down to a flat surface, they will emerge in sorted order. There are a great many reasons why this is not truly a challenge to the PECTTT hypothesis, and I will not ruin the reader’s fun in finding them out by her or himself.

**Soap bubbles:** One function $F : \{0, 1\}^n \rightarrow \{0, 1\}$ that is conjectured to require a large number of NAND lines to solve is the...
Euclidean Steiner Tree problem. This is the problem where one is given \( m \) points in the plane \((x_1, y_1), \ldots, (x_m, y_m)\) (say with integer coordinates ranging from 1 till \( m \), and hence the list can be represented as a string of \( n = O(m \log m) \) size) and some number \( K \). The goal is to figure out whether it is possible to connect all the points by line segments of total length at most \( K \). This function is conjectured to be hard because it is \textit{NP complete} - a concept that we'll encounter later in this course - and it is in fact reasonable to conjecture that as \( m \) grows, the number of NAND lines required to compute this function grows \textit{exponentially} in \( m \), meaning that the PECTT would predict that if \( m \) is sufficiently large (such as a few hundreds or so) then no physical device could compute \( F \). Yet, some people claimed that there is in fact a very simple physical device that could solve this problem, that can be constructed using some wooden pegs and soap. The idea is that if we take two glass plates, and put \( m \) wooden pegs between them in the locations \((x_1, y_1), \ldots, (x_m, y_m)\) then bubbles will form whose edges touch those pegs in the way that will minimize the total energy which turns out to be a function of the total length of the line segments. The problem with this device of course is that nature, just like people, often gets stuck in “local optima”. That is, the resulting configuration will not be one that achieves the absolute minimum of the total energy but rather one that can’t be improved with local changes. Aaronson has carried out actual experiments (see Fig. 6.10), and saw that while this device often is successful for three or four pegs, it starts yielding suboptimal results once the number of pegs grows beyond that.

- **DNA computing.** People have suggested using the properties of DNA to do hard computational problems. The main advantage of DNA is the ability to potentially encode a lot of information in relatively small physical space, as well as operate on this information in a highly parallel manner. At the time of this writing, it was demonstrated that one can use DNA to store about \( 10^{16} \) bits of information in a region of radius about millimeter, as opposed to about \( 10^{10} \) bits with the best known hard disk technology. This does not posit a real challenge to the PECTT but does suggest that one should be conservative about the choice of constant and not assume that current hard disk + silicon technologies are the absolute best best possible.\footnote{We were extremely conservative in the suggested parameters for the PECTT, having assumed that as many as \( \ell_p^2 10^{-6} \sim 10^{63} \) bits could potentially be stored in a millimeter radius region.}

- **Continuous/real computers.** The physical world is often described using continuous quantities such as time and space, and people have suggested that analog devices might have direct access to computing with real-valued quantities and would be inherently
more powerful than discrete models such as NAND machines. Whether the “true” physical world is continuous or discrete is an open question. In fact, we do not even know how to precisely phrase this question, let alone answer it. Yet, regardless of the answer, it seems clear that the effort to measure a continuous quantity grows with the level of accuracy desired, and so there is no “free lunch” or way to bypass the PECTT using such machines (see also this paper). Related to that are proposals known as “hypercomputing” or “Zeno’s computers” which attempt to use the continuity of time by doing the first operation in one second, the second one in half a second, the third operation in a quarter second and so on. These fail for a similar reason to the one guaranteeing that Achilles will eventually catch the tortoise despite the original Zeno’s paradox.

- **Relativity computer and time travel.** The formulation above assumed the notion of time, but under the theory of relativity time is in the eye of the observer. One approach to solve hard problems is to leave the computer to run for a lot of time from his perspective, but to ensure that this is actually a short while from our perspective. One approach to do so is for the user to start the computer and then go for a quick jog at close to the speed of light before checking on its status. Depending on how fast one goes, few seconds from the point of view of the user might correspond to centuries in computer time (it might even finish updating its Windows operating system!). Of course the catch here is that the
energy required from the user is proportional to how close one needs to get to the speed of light. A more interesting proposal is to use time travel via \textit{closed timelike curves} (CTCs). In this case we could run an arbitrarily long computation by doing some calculations, remembering the current state, and the travelling back in time to continue where we left off. Indeed, if CTCs exist then we’d probably have to revise the PECTT (though in this case I will simply travel back in time and edit these notes, so I can claim I never conjectured it in the first place…) 

- **Humans.** Another computing system that has been proposed as a counterexample to the PECTT is a 3 pound computer of about 0.1m radius, namely the human brain. Humans can walk around, talk, feel, and do others things that are not commonly done by NAND programs, but can they compute partial functions that NAND programs cannot? There are certainly computational tasks that \textit{at the moment} humans do better than computers (e.g., play some video games, at the moment), but based on our current understanding of the brain, humans (or other animals) have no \textit{inherent} computational advantage over computers. The brain has about $10^{11}$ neurons, each operating in a speed of about 1000 operations per seconds. Hence a rough first approximation is that a NAND program of about $10^{14}$ lines could simulate one second of a brain’s activity.\footnote{This is a very rough approximation that could be wrong to a few orders of magnitude in either direction. For one, there are other structures in the brain apart from neurons that one might need to simulate, hence requiring higher overhead. On the other hand, it is by no mean clear that we need to fully clone the brain in order to achieve the same computational tasks that it does.} Note that the fact that such a NAND program (likely) exists does not mean it is easy to find it. After all, constructing this program took evolution billions of years. Much of the recent efforts in artificial intelligence research is focused on finding programs that replicate some of the brain’s capabilities and they take massive computational effort to discover, these programs often turn out to be much smaller than the pessimistic estimates above. For example, at the time of this writing, Google’s \textit{neural network for machine translation} has about $10^4$ nodes (and can be simulated by a NAND program of comparable size). Philosophers, priests and many others have since time immemorial argued that there is something about humans that cannot be captured by mechanical devices such as computers; whether or not that is the case, the evidence is thin that humans can perform computational tasks that are inherently impossible to achieve by computers of similar complexity.\footnote{There are some well known scientists that have \textit{advocated} that humans have inherent computational advantages over computers. See also \textit{this}.} 

- **Quantum computation.** The most compelling attack on the Physical Extended Church Turing Thesis comes from the notion of \textit{quantum computing}. The idea was initiated by the observation that systems with strong quantum effects are very hard to simulate on a computer. Turning this observation on its head, people have
proposed using such systems to perform computations that we do not know how to do otherwise. We will discuss quantum computing in much more detail later in this course. Modeling it will essentially involve extending the NAND programming language to the “QNAND” programming language that has one more (very special) operation. However, the main take away is that while quantum computing does suggest we need to amend the PECTT, it does not require a complete revision of our worldview. Indeed, almost all of the content of this course remains the same whether the underlying computational model is the “classical” model of NAND programs or the quantum model of QNAND programs (also known as quantum circuits).

6.10 Lecture summary

- NAND gates can be implemented by a variety of physical means.
- NAND programs are equivalent (up to constants) to Boolean circuits using any finite universal basis.
- By a leap of faith, we could hypothesize that the number of lines in the smallest NAND program for a function $F$ captures roughly the amount of physical resources required to compute $F$. This statement is known as the Physical Extended Church-Turing Thesis (PECTT).
- NAND programs capture a surprisingly wide array of computational models. The strongest currently known challenge to the PECTT comes from the potential for using quantum mechanical effects to speed-up computation, a model known as quantum computers.

6.11 Exercises


Exercise 6.2 — Simulating all circuits with NAND programs. Prove Theorem 6.2.

Exercise 6.3 — Universal basis. For every one of the following sets, either prove that it is a universal basis or prove that it is not. 1. $B = \{\wedge, \vee, \neg\}$. (To make all of them be function on two inputs, define $\neg(x, y) = \bar{x}$.)
2. \( B = \{\wedge, \vee\} \).

3. \( B = \{\oplus, 0, 1\} \) where \( \oplus : \{0, 1\}^2 \to \{0, 1\} \) is the XOR function and 0 and 1 are the constant functions that output 0 and 1.

4. \( B = \{\text{LOOKUP}_1, 0, 1\} \) where 0 and 1 are the constant functions as above and \( \text{LOOKUP}_1 : \{0, 1\}^3 \to \{0, 1\} \) satisfies \( \text{LOOKUP}_1(a, b, c) = a \) if \( c = 0 \) and equals \( b \) if \( c = 1 \).

Exercise 6.4 — Bound on universal basis size (challenge). Prove that for every subset \( B \) of the functions from \( \{0, 1\}^k \) to \( \{0, 1\} \), if \( B \) is universal then there is a \( B \)-circuit of at most \( O(2^k) \) gates to compute the NAND function.\(^10\)

Exercise 6.5 — Threshold using NANDs. Prove that for every \( w, t \), the function \( T_{w,t} \) can be computed by a NAND program of at most \( O(k^3) \) lines.\(^11\)

### 6.12 Bibliographical notes

Scott Aaronson’s blog post on how information is physical is a good discussion on issues related to the physical extended Church-Turing Physics. Aaronson’s survey on NP complete problems and physical reality is also a great source for some of these issues, though might be easier to read after we reach the lectures on NP and NP completeness.

### 6.13 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

- The notion of the fundamental limits for information and their interplay with physics, is still not well understood.

### 6.14 Acknowledgements
7

Loops and infinity

“We thus see that when $n = 1$, nine operation-cards are used; that when $n = 2$, fourteen Operation-cards are used; and that when $n > 2$, twenty-five operation-cards are used; but that no more are needed, however great $n$ may be; and not only this, but that these same twenty-five cards suffice for the successive computation of all the numbers”, Ada Augusta, countess of Lovelace, 1843

“It is found in practice that (Turing machines) can do anything that could be described as ‘rule of thumb’ or ‘purely mechanical’… (Indeed,) it is now agreed amongst logicians that ‘calculable by means of (a Turing Machine)’ is the correct accurate rendering of such phrases.”, Alan Turing, 1948

“All problems in computer science can be solved by another level of indirection”, attributed to David Wheeler.

The NAND programming language has one very significant drawback: a finite NAND program $P$ can only compute a finite function $F$, and in particular the number of inputs of $F$ is always smaller than the number of lines of $P$. This does not capture our intuitive notion of an algorithm as a single recipe to compute a potentially infinite function. For example, the standard elementary school multiplication algorithm is a single algorithm that multiplies numbers of all lengths, but yet we cannot express this algorithm as a single NAND program, but rather need a different NAND program for every input length.
Let us consider the case of the simple \textit{parity} function \(\text{PARITY} : \{0,1\}^* \to \{0,1\}\) such that \(\text{PARITY}(x)\) equals 1 iff the number of 1's in \(x\) is odd cannot be computed by a NAND program. Rather, for every \(n\), we can compute \(\text{PARITY}_n\) (the restriction of \(\text{PARITY}\) to \(\{0,1\}^n\)) using a different NAND program. For example, here is the NAND program to compute \(\text{PARITY}_5\):

\[
\begin{align*}
  u &:= x_0 \ \text{NAND} \ x_1 \\
  v &:= x_0 \ \text{NAND} \ u \\
  w &:= x_1 \ \text{NAND} \ u \\
  s &:= v \ \text{NAND} \ w \\
  u &:= s \ \text{NAND} \ x_2 \\
  v &:= s \ \text{NAND} \ u \\
  w &:= x_2 \ \text{NAND} \ u \\
  s &:= v \ \text{NAND} \ w \\
  u &:= s \ \text{NAND} \ x_3 \\
  v &:= s \ \text{NAND} \ u \\
  w &:= x_3 \ \text{NAND} \ u \\
  s &:= v \ \text{NAND} \ w \\
  u &:= s \ \text{NAND} \ x_4 \\
  v &:= s \ \text{NAND} \ u \\
  w &:= x_4 \ \text{NAND} \ u \\
  y_0 &:= v \ \text{NAND} \ w
\end{align*}
\]

This is rather repetitive, and more importantly, does not capture the fact that there is a \textit{single} algorithm to compute the parity on all inputs. Typical programming language use the notion of \textit{loops} to express such an algorithm, and so we would rather write something like:

\[
\begin{align*}
  \text{# s is the "running parity", initialized to 0} \\
\text{while i < length(x):} \\
  &\quad u := x_i \ \text{NAND} \ s \\
  &\quad v := s \ \text{NAND} \ u \\
  &\quad w := x_i \ \text{NAND} \ u \\
  &\quad s := v \ \text{NAND} \ w \\
  &\quad i++ \\
  &\quad ns := s \ \text{NAND} \ s \\
  &\quad y_0 := ns \ \text{NAND} \ ns
\end{align*}
\]

We will now discuss how we can extend the NAND programming language so that it can capture this kind of a construct.
7.1 The NAND++ Programming language

Keeping to our minimalist form, we will not add a while keyword to the NAND programming language. But we will extend this language in a way that allows for executing loops and accessing arrays of arbitrary length.

The main new ingredients are the following:

- We add a special integer valued variable i, and allow expressions of the form foo_i (for every variable identifier foo) which are evaluated to equal foo_(i) where (i) denotes the current value of the variable i. As usual, i is initially assigned the value 0.

- We add a special variable loop with the following semantics: when the program ends, if loop is equal to one, then execution goes back to the first line and the variable i is either incremented or decremented by 1. In the first iteration of the loop, i is incremented, in the second iteration, it is decremented, then in the next two iterations i is incremented, and in the next two after that it is decremented, and so on. That is, the variable i takes the following sequence of values:

\[
0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \ldots
\]

- Because the input to NAND++ programs can have variable length, we also add a special read-only array validx such that validx_(n) is equal to 1 if and only if the n is smaller than the length of the input.

- Like NAND programs, the output of a NAND++ program is the string y_0, \ldots, y_(k) where k is the largest integer such that y_(k) was assigned a value.

See the appendix for a more formal specification of the NAND++ programming language, and the website http://nandpl.org for an implementation.

Here is the NAND++ program to compute parity of arbitrary length: (It is a good idea for you to see why this program does indeed compute the parity)

```
# compute sum x_i (mod 2)
# s = running parity
# seen_i = 1 if this index has been seen before

# Do val := (NOT seen_i) AND x_i
```
When we invoke this program on the input 010, we get the following execution trace:

... (complete this here)
End of iteration 0, loop = 1, continuing to iteration 1
... 
End of iteration 2, loop = 0, halting program

We say that a NAND program completed its $r$-th round when the index variable $i$ completed the sequence:

$$0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \ldots, 0, 1, \ldots, r - 1, \ldots, 0 \quad (7.2)$$

This happens when the program completed

$$1 + 2 + 4 + 6 + \cdots + 2r = r^2 + r + 1 \quad (7.3)$$

iterations of its main loop. (The last equality is obtained by applying the formula for the sum of an algebraic progression.) This means that if we keep a “program counter” $pc$ that is initially set to 0 and increases by one at the end of any iteration, then the “round” $r$ is the largest integer such that $r(r + 1) \leq pc$, which equals $\lfloor \sqrt{pc + 1/4} - 1/2 \rfloor$.

Thus the value of $i$ in the iteration with counter $pc$ equals:

$$\text{index}(pc) = \begin{cases} 
    pc - r(r + 1) & pc \leq (r + 1)^2 \\
    (r + 1)(r + 2) - pc & \text{otherwise}
\end{cases} \quad (7.4)$$
where \( r = \lfloor \sqrt{pc + 1/4} - 1/2 \rfloor \). (We ask you to prove this in Exercise 7.1.)

**7.1.1 Remark: Inner loops via syntactic sugar**

While NAND+ only has a single “outer loop”, we can use conditionals to implement inner loops as well. That is, we can replace code such as

preloop_code
while (a) {
  loop_code
}
postloop_code

by

// finishedpreloop is initialized to 0
// finishedloop is initialized to 0
if NOT(finishedpreloop) {
  code1
  finishedpreloop := 1
}
if NOT(finishedloop) {
  if (a) {
    code2
  }
  if NOT(a) {
    finishedloop := 1
  }
}
if (finishedloop) {
  postloop_code
}

(Applying the standard syntactic sugar transformations to convert the conditionals into NAND code.) We can apply this transformation repeatedly to convert programs with multiple loops, and even nested loops, into a standard NAND++ program.

**7.2 Uniformity and NAND vs NAND++**

While NAND++ adds an extra operation over NAND, it is not exactly accurate to say that NAND++ programs are “more powerful” than
NAND programs. NAND programs, having no loops, are simply not applicable for computing functions with more inputs than they have lines. The key difference between NAND and NAND++ is that NAND++ allows us to express the fact that the algorithm for computing parities of length-100 strings is really the same one as the algorithm for computing parities of length-5 strings (or similarly the fact that the algorithm for adding $n$-bit numbers is the same for every $n$, etc.). That is, one can think of the NAND++ program for general parity as the “seed” out of which we can grow NAND programs for length 10, length 100, or length 1000 parities as needed. This notion of a single algorithm that can compute functions of all input lengths is known as uniformity of computation and hence we think of NAND++ as uniform model of computation, as opposed to NAND which is a nonuniform model, where we have to specify a different program for every input length.

Looking ahead, we will see that this uniformity leads to another crucial difference between NAND++ and NAND programs. NAND++ programs can have inputs and outputs that are longer than the description of the program and in particular we can have a NAND++ program that “self replicates” in the sense that it can print its own code.

This notion of “self replication”, and the related notion of “self reference” is crucial to many aspects of computation, as well of course to life itself, whether in the form of digital or biological programs.

**Advanced remark:** This notion of a NAND++ program as a “seed” that can grow a different NAND program for every input length is one that we will come back to later on in this course, when we consider bounding the time complexity of computation. As we will see, we can think of a NAND++ program $P$ that computes some function $F$ in $T(n)$ steps on input length $n$, as a two phase process. For any input $x \in \{0,1\}^*$, the program $P$ can be thought of as first producing a $T(|x|)$-line NAND program $P'$, and then executing this program $P'$ on $x$. This might not be easy to see at this point, but will become clearer in a few lectures when we tackle the issue of efficiency in computation.

### 7.2.1 Infinite loops and computing a function

There is another important difference between NAND and NAND++ programs: looking at a NAND program, we can always tell how many inputs and how many outputs it has (by looking at the number
of $x_-$ and $y_-$ variables) and are guaranteed that if we invoke it on any input then some output will be produced.

In contrast, given any particular NAND++ program $P$, we cannot determine a priori the length of the output. In fact, we don’t even know if an output would be produced at all! For example, the following NAND++ program would go into an infinite loop if the first bit of the input is zero:

\begin{lstlisting}
\textbf{loop} := x_0 \textbf{NAND} x_0
\end{lstlisting}

For a NAND++ program $P$ and string $x \in \{0,1\}^*$, if $P$ produces an output when executed with input $x$ then we denote this output by $P(x)$. If $P$ does not produce an output on $x$ then we say that $P(x)$ is undefined and denote this as $P(x) = \bot$.

**Definition 7.1 — Computing a function.** We say that a NAND++ program $P$ computes a function $F : \{0,1\}^* \rightarrow \{0,1\}^*$ if $P(x) = F(x)$ for every $x \in \{0,1\}^*$.

If $F$ is a partial function then we say that $P$ computes $F$ if $P(x) = F(x)$ for every $x$ on which $F$ is defined.

### 7.3 The NAND« programming language

Even the program to compute parities in NAND++ is somewhat tedious, and hence we will now define a seemingly more powerful programming language: NAND«. NAND« has some additional operators, but as we will see, it can ultimately be implemented by applying certain “syntactic sugar” constructs on top of NAND++. Nonetheless, NAND« will still serve (especially later in the course) as a useful computational model.\(^3\) There are two key differences between NAND« and NAND:

1. The NAND« programming language works with integer valued as opposed to binary variables.

2. NAND« allows indirection in the sense of accessing the $bar$-th location of an array $foo$. Specifically, since we use integer valued variables, we can assign the value of $bar$ to the special index $i$ and then use $foo_i$.

We will allow the following operations on variables:\(^4\)

- $foo := bar$ or $i := bar$ (assignment)
- $foo := bar + baz$ (addition)

\(^3\) If you have encountered computability or computational complexity before, we can already “let you in on the secret”. NAND++ is equivalent to the model known as single tape oblivious Turing machines, while NAND« is (essentially) equivalent to the model known as RAM machines. For the purposes of the current lecture, these two models are indistinguishable (due to a notion known as “Turing completeness”) but the difference between them can matter if one is interested in a fine enough resolution of computational efficiency.

\(^4\) Below $foo$, $bar$ and $baz$ are indexed or non-indexed variable identifiers (e.g., they can have the form $\text{blah}$ or $\text{blah.12}$ or $\text{blah.i}$), as usual, we identify an indexed identifier $\text{blah}$ with $\text{blah.0}$. Except for the assignment, where $i$ can be on the lefthand side, the special index variable $i$ cannot be involved in these operations.
• \texttt{foo := bar - baz} (subtraction)
• \texttt{foo := bar \gg baz} (right shift: \texttt{idx := \lfloor foo2^{-bar} \rfloor})
• \texttt{foo := bar \ll baz} (left shift: \texttt{idx := \lfloor foo2^{bar} \rfloor})
• \texttt{foo := bar \% baz} (modular reduction)
• \texttt{foo := bar * baz} (multiplication)
• \texttt{foo := bar / baz} (integer division: \texttt{idx := \lfloor foo/baz \rfloor})
• \texttt{foo := bar \& baz} (bitwise AND)
• \texttt{foo := bar \^ baz} (bitwise XOR)
• \texttt{foo := bar > baz} (greater than)
• \texttt{foo := bar < baz} (smaller than)
• \texttt{foo := bar == baz} (equality)

The semantics of these operations are as expected except that we maintain the invariant that all variables always take values between 0 and the current value of the program counter (i.e., number of iterations of the program that have been completed). If an operation would result in assigning to a variable \texttt{foo} a number that is smaller than 0, then we assign 0 to \texttt{foo}, and if it assigns to \texttt{foo} a number that is larger than the program counter, then we assign the value of the program counter to \texttt{foo}. Just like C, we interpret any nonzero value as “true” or 1, and hence \texttt{foo := bar NAND baz} will assign to \texttt{foo} the value 0 if both \texttt{bar} and \texttt{baz} are not zero, and 1 otherwise.

Apart from those operations, NAND« is identical to NAND++. For consistency, we still treat the variable \texttt{i} as special, in the sense that we only allow it to be used as an index, even though the other variables contain integers as well, and so we don’t allow variables such as \texttt{foo_bar} though we can simulate it by first writing \texttt{i := bar} and then \texttt{foo_i}. We also maintain the invariant that at the beginning of each iteration, the value of \texttt{i} is set to the same value that it would have in a NAND++ program (i.e., the function of the program counter stated in Exercise 7.1), though this can be of course overwritten by explicitly assigning a value to \texttt{i}. Once again, see the appendix for a more formal specification of NAND«.

### Computing on integers
Most of the time we will be interested in applying NAND« programs on bits, and hence we will assume that both inputs and outputs are bits. We can enforce the latter condition by not allowing \texttt{y_} variables to be on the lefthand
side of any operation other than NAND. However, the same model can be used to talk about functions that map tuples of integers to tuples of integers, and so we may very occasionally abuse notation and talk about NAND\(\ast\) programs that compute on integers.

### 7.3.1 Simulating NAND\(\ast\) in NAND++

The most important fact we need to know about NAND\(\ast\) is that it can be implemented by mere “syntactic sugar” and hence does not give us more computational power than NAND++, as stated in the following theorem:

**Theorem 7.1** — NAND++ and NAND\(\ast\) are equivalent. For every (partial) function \(F : \{0,1\}^* \to \{0,1\}^*\), \(F\) is computable by a NAND++ program if and only if \(F\) is computable by a NAND\(\ast\) program.

The rest of this section is devoted to outlining the proof of Theorem 7.1. The “only if” direction of the theorem is immediate. After all, every NAND++ program \(P\) is in particular also a NAND\(\ast\) program, and hence if \(F\) is computable by a NAND++ program then it is also computable by a NAND\(\ast\) program. To show the “if” direction, we need to show how we can implement all the operations of NAND\(\ast\) in NAND++.

Note that it’s quite easy to store integers as bit-arrays, and so we can also simulate an array of integers using a two-dimensional array of bits (which we have seen how to embed in the standard single-dimensional arrays supplied by NAND++). That is, if in NAND\(\ast\) the variable \(\text{foo}_{\langle i \rangle}\) corresponded to an integer, then we can simulate this in NAND++ by having \(\text{foo}_{\text{PAIR}(i,j)}\) correspond to the \(j\)-th bit in the representation of the integer \(\text{foo}_{\langle i \rangle}\) where \(\text{PAIR} : \mathbb{N}^2 \to \mathbb{N}\) is some easily computable one-to-one embedding of \(\mathbb{N}^2\) in \(\mathbb{N}\).

We can in principle use the standard algorithms for addition, multiplication, division, etc., to perform the arithmetic operations on these arrays. The key difficulty is in actually controlling the index variable \(i\), which in NAND++ moves obliviously according to the set schedule 0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \ldots. To achieve control of \(i\) we use the well known observation that a bus is just like a taxi if you are willing to wait long enough. That is, instead of moving \(i\) to where we want, we wait until it eventually gets there on its own.

One useful observation is that in a NAND++ program we can
know whether the index is increasing or decreasing using the Hansel and Gretel technique of “breadcrumbs”. We create an array \texttt{atstart} such that \texttt{atstart.0} equals 1 but \texttt{atstart.\langle j \rangle} equals 0 for all \texttt{j > 0}, and an array \texttt{breadcrumb} where we set \texttt{breadcrumb.1} to 1 in every iteration. Then we can setup a variable \texttt{indexincreasing} and set it to 1 when we reach the zero index (i.e., when \texttt{atstart.i} is equal to 1) and set it to 0 when we reach the end point (i.e., when we see an index for which \texttt{breadcrumb.i} is 0 and hence we have reached it for the first time). We can also maintain an array \texttt{arridx} that contains 0 in all positions except the current value of \textit{i}. Now we can simulate incrementing and decrementing \textit{i} by one as follows. If we want to increment \textit{i} and \texttt{indexincreasing} then we simply wait one step. Otherwise (if \texttt{indexincreasing} is 0) then we enter into a special state in which we do nothing until we reach again the point when \texttt{arridx.i} is 1 and \texttt{indexincreasing} is equal to 1. Decrementing \textit{i} is done in the analogous way.

Once we can increment and decrement \textit{i}, we can use this, together with the notion of inner loops, to perform all the operations needed on the representations of integers as bits. We can also simulate an operation such as \texttt{i := foo} by creating a temporary array that contains 0 except for a single 1 in the location corresponding to the integer represented by \texttt{foo} and waiting until we reach the point where \texttt{foo.i} equals 1.

We omit the full details of the proofs. However, the webpage nandpl.org contains an OCaml program that transform a NAND\textsuperscript{-} program into an equivalent NAND++ program.

### 7.3.2 NAND++ normal form

In many programming language, we can make syntactic transformation on programs that do not change their operations, but might make them “cleaner” or “easier to understand” in some way. For example, we could declare variables in the beginning of every function even if this is not required by the programming language. For NAND++, it will also be sometimes useful for us that a programs have a “nice form”, which we can ensure by making some syntactic transformations. Specifically we will make the following definition:

\begin{definition}[NAND++ normal form] We say that a NAND++ program \textit{P} is in normal form if it satisfies the following properties:
1. \textit{P} has a variable \texttt{indexincreasing} with the code to ensure that
\end{definition}
indexincreasing is 1 whenever in the next iteration the value of \( i \) increases and indexincreasing is 0 otherwise.

2. \( P \) has a variable zero with the code to ensure that zero_i is equal to 1 if and only if \( i \) is zero.

3. There are no absolute numerical indices in \( P \). All variables either have the form foo_i or bar: no blah_17. Moreover, every variable identifier that appears with the index \( i \) never appears without an index and vice versa.

4. There are no two lines in \( P \) that assign a value to the same variable.

5. \( P \) has a variable halted which the only line that refers to it is the last line of the program which has the form

\[
\text{halted} := \text{loop NAND loop.}
\]

6. All assignments in \( P \) to the \( y \) variables and \( \text{loop} \) are “guarded” by halted which means that any such assignment has the form that the value of \( y_i \) or \( \text{loop} \) is unchanged if halted equals 1.

It might not be clear at this point why we care about the conditions of Definition 7.2, but we will see later in this course that they can help make certain proofs easier. The following theorem shows that we can ensure these conditions at a small cost:

**Theorem 7.2 — NAND++ normal form.** For every NAND++ program \( P \) there is a NAND++ program \( P' \) of normal form that computes the same function as \( P \). Moreover, the number of lines in \( P' \) is at most \( c \) times the number of lines in \( P \), where \( c \leq 10 \) is some absolute constant. Furthermore, for every input \( x \), the number of iterations that \( P' \) takes on input \( x \) is at most a constant additive number than the number of iterations that \( P \) takes on the same input.

**Proof.** We only sketch the proof since it’s not so insightful. We will go over the conditions one by one. For 1 and 2 we discussed above how to add code to a NAND++ program that ensures these conditions, so we can focus on the remaining ones:

3. We can replace a variable of the form bar_17 with some unique number name such as barseventeen. We can add code to test if \( i \) is one of the constantly many indices that appeared as absolute numerical instances, and if so then replace variable such as bar_i with barindexvalue.

4. If two lines \( i < j \) assign a value to the same (indexed or unin-
dexed) variable foo, then we can replace all occurrences of foo in lines $i, i+1, \ldots, j-1$ with tempfoo where temp is some unique prefix.

5. We can ensure this by simply adding that line of code, and replacing any use of halted with uphalted where up stands for some unique prefix.

6. This can be ensured by replacing each such assignment with a constant number of lines ensuring this if condition. That is, we replace an assignment of the form $y_i := \text{foo NAND bar}$ or $\text{loop} := \text{foo NAND bar}$ with the code if NOT(halted) $y_i := \text{foo NAND bar}$ or if NOT(halted) $\text{loop} := \text{foo NAND bar}$, and then use the standard “de-sugaring” transformation to remove the syntactic sugar for if. ■

7.4 Example

Here is a program that computes the function $\text{PALINDROME} : \{0,1\}^* \rightarrow \{0,1\}$ that outputs 1 on $x$ if and only if $x_i = x_{|x|-i}$ for every $i \in \{0, \ldots, |x|-1\}$. This program uses NAND« with the syntactic sugar we described before, but as discussed above, we can transform it into a NAND++ program.

// A sample NAND<< program that computes the language of palindromes
// By Juan Esteller

def a := NOT(b) {
    a := NOT(b) NAND b
}
o := NOT(z)
two := o + o
if(NOT(seen_0)) {
    cur := z
    seen_0 := o
}
i := cur
if(validx_i) {
    cur := cur + o
    loop := o
}
if(NOT(validx_i)) {
    computedlength := o
}
if(computedlength) {

if (justentered) {
    justentered := o
    iter := z
}
i := iter
left := x_i
i := (cur - iter) - o
right := x_i
if (NOT (left == right)) {
    loop := z
    y_0 := z
}
halflength := cur / two
if (NOT (iter < halflength)) {
    y_0 := o
    loop := z
}
iter := iter + o
}

7.5 Universality: A NAND++ interpreter in NAND++

Like a NAND program, a NAND++ or a NAND« program is ultimately a sequence of symbols and hence can obviously be represented as a binary string. We will spell out the exact details of representation later, but as usual, the details are not so important (e.g., we can use the ASCII encoding of the source code). What is crucial is that we can use such representation to evaluate any program. That is, we prove the following theorem:

**Theorem 7.3 — Universality of NAND++.** There is a NAND++ program that computes the partial function $EVAL : \{0, 1\}^* \rightarrow \{0, 1\}^*$ defined as follows:

$$EVAL(P, x) = P(x)$$

(7.5)

for strings $P, x$ such that $P$ is a valid representation of a NAND++ program which produces an output on $x$.

This is a stronger notion than the universality we proved for NAND, in the sense that we show a single universal NAND++ program $U$ that can evaluate all NAND programs, including those that have more lines than the lines in $U$. In particular, $U$ can even be used to evaluate itself! This notion of self reference will appear time and
again in this course, and as we will see, leads to several counterintuitive phenomena in computing.

Because we can easily transform a NAND\textsc{\textasciitilde} program into a NAND++ program, this means that even the seemingly “weaker” NAND++ programming language is powerful enough to simulate NAND\textsc{\textasciitilde} programs. Indeed, as we already alluded to before, NAND++ is powerful enough to simulate also all other standard programming languages such as Python, C, Lisp, etc..

### 7.5.1 Representing NAND++ programs as string

Before we prove Theorem 12.2 formal, we need to make its statement precise by specifying a representation scheme for NAND++ programs. As mentioned above, simply representing the program as a string using ASCII or UTF-8 encoding will work just fine, but we will use a somewhat more convenient and concrete representation, which is the natural generalization of the “list of tuples” representation for NAND programs. We will assume that all variables are of the form $\text{foo}_{##}$ where ## is some number or the index i. If a variable foo does not have an index then we add the index zero to it. We represent an instruction of the form

$$\text{foo}_{\langle j \rangle} := \text{bar}_{\langle k \rangle} \text{ NAND } \text{baz}_{\langle \ell \rangle}$$

as a 6 tuple $(a, j, b, k, c, \ell)$ where $a, b, c$ are numbers corresponding to the labels foo, bar, and baz respectively, and $j, k, \ell$ are the corresponding indices. We let $L$ be the number of lines in the program, and set the index to be $L + 1$ if instead of a number the variable is indexed by the special variable $i$. (There is no risk of conflict since we did not allow numerical indices larger than the number of lines in the program.) We will set the identifiers of $x, y, \text{validx}$ and $\text{loop}$ to 0, 1, 2, 3 respectively. Therefore the representation of the parity program

tmp_1 := \text{seen}_i \text{ NAND } \text{seen}_i \\
tmp_2 := x_i \text{ NAND } \text{tmp}_1 \\
val := \text{tmp}_2 \text{ NAND } \text{tmp}_2 \\
ns := s \text{ NAND } s \\
y_0 := ns \text{ NAND } ns \\
u := val \text{ NAND } s \\
v := s \text{ NAND } u \\
w := val \text{ NAND } u \\
s := v \text{ NAND } w \\
\text{seen}_i := z \text{ NAND } z
stop := validx_i NAND validx_i
loop := stop NAND stop

will be
[[4, 1, 5, 61, 5, 61],
 [4, 2, 0, 61, 4, 1],
 [6, 0, 4, 2, 4, 2],
 [7, 0, 8, 0, 8, 0],
 [1, 0, 7, 0, 7, 0],
 [9, 0, 6, 0, 8, 0],
 [10, 0, 8, 0, 9, 0],
 [11, 0, 6, 0, 9, 0],
 [8, 0, 10, 0, 11, 0],
 [5, 61, 12, 0, 12, 0],
 [13, 0, 2, 61, 2, 61],
 [3, 0, 13, 0, 13, 0]]

**Binary encoding:** The above is a way to represent any NAND++ program as a list of numbers. We can of course encode such a list as a binary string in a number of ways. For concreteness, since all the numbers involved are between 0 and $L + 1$ (where $L$ is the number of lines), we can simply use a string of length $6\lceil\log(L + 1)\rceil$ to represent them, starting with the prefix $0^{L+1}$ to encode $L$. For convenience we will assume that any string that is not formatted in this way encodes the single line program $y_0 := x_0 \text{ NAND } x_0$. This way we can assume that every string $P \in \{0, 1\}^*$ represents some program.

### 7.5.2 A NAND++ interpreter in NAND«

Here is the “pseudocode”/“sugar added” version of an interpreter for NAND++ programs (given in the list of 6 tuples representation) in NAND«. We assume below that the input is given as integers $x_0, \ldots, x_{(6 \cdot \text{lines} - 1)}$ where lines is the number of lines in the program. We also assume that NumberVariables gives some upper bound on the total number of distinct non-indexed identifiers used in the program (we can also simply use lines as this bound).

```plaintext
simloop := 3
totalvars := NumberVariables(x)
maxlines := Length(x) / 6
currenti := 0
currentround := 0
increasing := 1
pc := 0
```
while (true) {
    line := 0
    foo := x_{6*line + 0}
    fooidx := x_{6*line + 1}
    bar := x_{6*line + 2}
    baridx := x_{6*line + 3}
    baz := x_{6*line + 4}
    bazidx := x_{6*line + 5}
    if (fooidx == maxlines) {
        fooidx := currenti
    }
    ...
    // similar for baridx, bazidx

    vars_{totalvars*fooidx+foo} := vars_{totalvars*baridx+bar}
} NAND vars_{totalvars+bazidx+baz}

line++

if line==maxlines {
    if not avars[simloop] {
        break
    }
    pc := pc+1
    if (increasing) {
        i := i + 1
    } else {
        i := i - 1
    }
    if i>r {
        increasing := 0
        r := r+1
    }
    if i==0 {
        increasing := 1
    }
}

// keep track in loop above of largest m that y_{m-1} was assigned a value
// add code to move vars[0*totalvars+1]...vars[(m-1)*totalvars+1] to y_0..y_{m-1}
}

Since we can transform every NAND\times program to a NAND\times\times one,
we can also implement this interpreter in NAND++.

### 7.5.3 A Python interpreter in NAND++

At this point you probably can guess that it is possible to write an interpreter for languages such as C or Python in NAND<< and hence in NAND++ as well. After all, with NAND++ / NAND<< we have access to an unbounded array of memory, which we can use to simulate memory allocation and access, and can do all the basic computation steps offered by modern CPUs. Writing such an interpreter is nobody’s idea of a fun afternoon, but the fact it can be done gives credence to the belief that NAND++ is a good model for general-purpose computing.

### 7.6 Lecture summary

- NAND++ programs introduce the notion of loops, and allow us to capture a single algorithm that can evaluate functions of any length.

- NAND<< programs include more operations, including the ability to use indirection to obtain random access to memory, but they are computationally equivalent to NAND++ program.

- We can translate many (all?) standard algorithms into NAND<< and hence NAND++ programs.

- There is a universal NAND++ program $U$ such that on input a description of a NAND++ program $P$ and some input $x$, $U(P,x)$ halts and outputs $P(x)$ if (and only if) $P$ halts on input $x$.

### 7.7 Exercises

**Exercise 7.1 — Compute index.** Suppose that $pc$ is the “program counter” of a NAND++ program, in the sense that $pc$ is initialized to zero, and is incremented by one each time the program finishes an iteration and goes back to the first line. Prove that the value of the variable $i$ is equal to $pc - r(r + 1)$ if $pc \leq (r + 1)^2$ and equals $(r + 2)(r + 1) - pc$ otherwise, where $r = \lfloor \sqrt{pc + 1} - 1/2 \rfloor$.
7.8 Bibliographical notes

The notion of “NAND++ programs” we use is nonstandard but (as we will see) they are equivalent to standard models used in the literature. Specifically, NAND++ programs are closely related (though not identical) to oblivious one-tape Turing machines, while NAND« programs are essentially the same as RAM machines. As we’ve seen in these lectures, in a qualitative sense these two models are also equivalent to one another, though the distinctions between them matter if one cares (as is typically the case in algorithms research) about polynomial factors in the running time.

7.9 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

7.10 Acknowledgements
Equivalent models of computation

"Computing is normally done by writing certain symbols on paper. We may suppose that this paper is divided into squares like a child’s arithmetic book. The behavior of the [human] computer at any moment is determined by the symbols which he is observing, and of his ‘state of mind’ at that moment… We may suppose that in a simple operation not more than one symbol is altered."

"We compare a man in the process of computing… to a machine which is only capable of a finite number of configurations… The machine is supplied with a ‘tape’ (the analogue of paper)… divided into sections (called ‘squares’) each capable of bearing a ‘symbol’", Alan Turing, 1936

*" What is the difference between a Turing machine and the modern computer? It’s the same as that between Hillary’s ascent of Everest and the establishment of a Hilton hotel on its peak."*, Alan Perlis, 1982.

We have defined the notion of computing a function based on the rather esoteric NAND++ programming language. In this lecture we justify this choice by showing that the definition of computable functions will remain the same under a wide variety of computational models. In fact, a widely believed claim known as the **Church-Turing Thesis** holds that every “reasonable” definition of computable function is equivalent to ours. We will discuss the Church-Turing Thesis and the potential definitions of “reasonable” at the end of this lecture.
8.1 Turing machines

The “granddaddy” of all models of computation is the Turing Machine, which is the standard model of computation in most textbook.¹ Turing machines were defined in 1936 by Alan Turing in an attempt to formally capture all the functions that can be computed by human “computers” that follow a well-defined set of rules, such as the standard algorithms for addition or multiplication.

Figures 8.1: Until the advent of electronic computers, the word “computer” was used to describe a person, often female, that performed calculations. These human computers were absolutely essential to many achievements including mapping the stars, breaking the Enigma cipher, and the NASA space mission. Two recent books about these “computers” and their important contributions are The Glass Universe (from which this photo is taken) and Hidden Figures.

Turing thought of such a person as having access to as much “scratch paper” as they need. For simplicity we can think of this scratch paper as a one dimensional piece of graph paper (commonly known as “work tape”), where in each box or “cell” of the tape holds a single symbol from some finite alphabet (e.g., one digit or letter). At any point in time, the person can read and write a single box of the paper, and based on the contents can update his/her finite mental state, and/or move to the box immediately to the left or right.

Thus, Turing modeled such a computation by a “machine” that maintains one of \( k \) states, and at each point can read and write a single symbol from some alphabet \( \Sigma \) (containing \( \{0, 1\} \)) from its “work tape”. To perform computation using this machine, we write

¹ This definitional choice does not make much difference since, as we will show, NAND+/NAND− programs are equivalent to Turing machines in their computing power.
the input $x \in \{0, 1\}^n$ on the tape, and the goal of the machine is to ensure that at the end of the computation, the value $F(x)$ will be written there. Specifically, a computation of a Turing Machine $M$ with $k$ states and alphabet $\Sigma$ on input $x \in \{0, 1\}^*$ proceeds as follows:

- Initially the machine is at state 0 (known as the “starting state”) and the tape is initialized to $\triangleright, x_0, \ldots, x_{n-1}, \emptyset, \emptyset, \ldots$.
- The location $i$ to which the machine points to is set to 0.
- At each step, the machine reads the symbol $\sigma = T[i]$ that is in the $i$th location of the tape, and based on this symbol and its state $s$ decides on:
  - What symbol $\sigma'$ to write on the tape
  - Whether to move Left (i.e., $i \leftarrow i - 1$) or Right (i.e., $i \leftarrow i + 1$)
  - What is going to be the new state $s \in [k]$
- When the machine reaches the state $s = k - 1$ (known as the “halting state”) then it halts. The output of the machine is obtained by reading off the tape from location 1 onwards, stopping at the first point where the symbol is not 0 or 1.

![Figure 8.2](image.png)

**Figure 8.2**: A Turing machine has access to an *tape* of unbounded length. At each point in the execution, the machine can read/write a single symbol of the tape, and based on that decide whether to move left, right or halt.

The formal definition of Turing machines is as follows:

**Definition 8.1 — Turing Machine.** A (one tape) *Turing machine* with $k$ states and alphabet $\Sigma \supseteq \{0, 1, \triangleright, \emptyset\}$ is a function $M : [k] \times \Sigma \rightarrow \Sigma \times [k] \times \{L, R\}$. We say that the Turing machine $M$ *computes* a (partial) function $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$ if for every $x \in \{0, 1\}^*$ on which $F$ is defined, the result of the following process is $F(x)$:

- Initialize the array $T$ of symbols in $\Sigma$ as follows: $T[0] = \triangleright$,
\[ T[i] = x_i \text{ for } i = 1, \ldots, |x| \]

- Let \( s = 1 \) and \( i = 0 \) and repeat the following while \( s \neq k - 1 \):
  1. Let \( \sigma = T[i] \). If \( T[i] \) is not defined then let \( \sigma = \emptyset \)
  2. Let \( (s', \sigma', D) = M(s, \sigma) \)
  3. Modify \( T[i] \) to equal \( \sigma' \)
  4. If \( D = L \) and \( i > 0 \) then set \( i \leftarrow i - 1 \). If \( D = R \) then set \( i \leftarrow i + 1 \).
  5. Set \( s \leftarrow s' \)

- Let \( n \) be the first index larger than 0 such that \( T[i] \notin \{0, 1\} \). We define the output of the process to be \( T[1], \ldots, T[n - 1] \). The number of steps that the Turing Machine \( M \) takes on input \( x \) is the number of times that the while loop above is executed. (If the process never stops then we say that the machine did not halt on \( x \).)

8.2 Turing Machines and NAND++ programs

As mentioned, Turing machines turn out to be equivalent to NAND++ programs:

**Theorem 8.1 — Turing machines and NAND++ programs.** For every \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \), \( F \) is computable by a NAND++ program if and only if there is a Turing Machine \( M \) that computes \( F \).

8.2.1 Simulating Turing machines with NAND++ programs

We now prove the “if” direction of Theorem 8.1, namely we show that given a Turing machine \( M \), we can find a NAND++ program \( P_M \) such that for every input \( x \), if \( M \) halts on input \( x \) with output \( y \) then \( P_M(x) = y \). Because NAND« and NAND++ are equivalent in power, it is sufficient to construct a NAND« program that has this property. Moreover, we can take advantage of the syntactic sugar transformations we have seen before for NAND«, including conditionals, loops, and function calls.

If \( M : [k] \times \Sigma \rightarrow [k] \times \{L, R\} \) then there is a finite length NAND program \( \text{Compute}_M \) that computes the finite function \( M \) (representing the finite sets \( [k], \Sigma, \{L, R\} \) appropriately by bits). The NAND« program simulating \( M \) will be the following:
// tape is an array with the alphabet Sigma
// we use ">" for the start-tape marker and "." for the empty cell
// in the syntactic sugar below, we assume some binary encoding of the alphabet.
tape_0 := ">"
k := 0
while (validx_j) { // copy input to tape
tape_j := x_k
k++
j++
}
tape_j := "<"
state := 0
head := 0 // location of the head
while NOT EQUAL(state,k-1) { // not ending state
state', dir := ComputeM(tape_head,state)
if EQUAL(dir,'L') AND NOT(EQUAL(tape_head,">")) {
    head--
}
if EQUAL(dir,'R') {
    head++
}
state' := state
}

// copy output to y variables
j := 1
s := 0
while EQUAL(tape_j,0) OR EQUAL(tape_j,1) {
y_s := tape_j
j++
s++
}

In addition to the standard syntactic sugar, we assumed above we can make function calls to the function EQUAL that checks equality of two symbols as well as the finite function ComputeM that corresponds to the transition function of the Turing machine.
8.2.2 NAND++ Snapshots, Transition, and Traces

To show the other direction of Theorem 8.1 (namely simulate a NAND++ program with a Turing machine), we will introduce some observations about NAND++ programs that will come in very handy in other settings as well. We can characterize a computation of a NAND++ program by keeping track of its state, namely the setting of all its variables (including the input and output variables), and the current state of its computation. A snapshot of a NAND++ program $P$ is simply the state of $P$ at a given step in the computation. To simulate a NAND++ program by a Turing machine (or any other method), it is sufficient to use the machine to compute, given a snapshot $\sigma_i$ of $P$ at step $i$, the snapshot $\sigma_{i+1}$ at step $i+1$. To make this notion precise, we need to formally define snapshots, which is what we do next.

For convenience, we will assume that a NAND++ program $P$ is in normal form, as per Definition 7.2 from the previous lecture. Recall that this means that $P$ satisfies a long list of properties, but the most crucial for our purposes are the following:

- $P$ has a variable $\text{indexincreasing}$ with the code to ensure that $\text{indexincreasing}$ is 1 whenever in the next iteration the value of $i$ increases and $\text{indexincreasing}$ is 0 otherwise.

- There are no absolute numerical indices in $P$. All variables either have the form $\text{foo}_i$ or $\text{bar}$: no $\text{blah}_{17}$. Moreover, every variable identifier that appears with the index $i$ never appears without an index and vice versa.

As mentioned above, a snapshot of $P$ will be a string that encodes the entire state of $P$. If the largest value that the index $i$ achieved was $T$, then, we would need $O(T)$ bits for this encoding, so we can encode for every indexed variable $\text{foo}$ the value of $\text{foo}_{\langle i \rangle}$ for every $i$. Thus our snapshot will be a string $S$ of $O(T)$ length, that we think of as being divided to $T$ constant-sized blocks, corresponding to each value of the counter. We now specify this encoding precisely:

A normal-form program $P$ has two types of variables: unindexed and indexed. If $P$ has $\ell$ lines, $a$ variables of the former type, and $b$ of the latter, and we let $T$ be one plus the largest value $i$ such that a variable $\text{foo}_\langle i \rangle$ has been assigned a value, then we can represent the current status of $P$ by a string $S \in \{0, 1\}^{T(1+\lceil \log \ell \rceil +a+b)}$ as follows: $S = (S^1, \ldots, S^T)$ where $S^i \in \{0, 1\}^{a+b+1}$ is a string such that:

- The first bit of $S^i$ (i.e., $S^i_0$) equals 1 if the current value of $i$ is $i$ and equals 0 otherwise.

\[\text{That is, when the computation begins then } T = n \text{ where } n \text{ is the input length, and later } T \text{ is the maximum of } n \text{ and one plus the largest value that the counter } i \text{ reaches.}\]
• The next \(\lceil \log \ell \rceil\) bits are the binary encoding of the index line that the program is just going to execute.

• The next \(a\) bits are either all zeroes (if \(i\) is not the current value of \(i\)) or are equal to the values of all the unindexed variables (in alphabetical order, except that the special variables \(\text{loop}, \text{halted}, \text{and idxincreasing}\) are first).

• The next \(b\) bits correspond to the value of \(\text{foo}_i\) for the \(b\) indexed variables \(\text{foo}\) (in alphabetical order, except that the special variables \(x, y, \text{validx}, \text{zero}\) are indexed first).

If \(S(t)\) denotes the snapshot of computation at step \(t\), then to compute \(S(t+1)\) from \(S(t)\) we need to do the following:

• Find the unique block \(i\) such that \(S(t)_i = 1\).

• Let \(c\) be the index of the current line to be executed. If \(c \neq 0\), then just execute this line: if the line is \(\text{foo} := \text{bar} \quad \text{NAND} \quad \text{baz}\) then compute the NAND of \(\text{bar}\) and \(\text{baz}\) based on the content of \(S_i\) and update the value of \(\text{foo}\) to this new value. (Here \(\text{foo}, \text{bar}\) and \(\text{baz}\) are either unindexed or indexed by \(i\).) Then let \(c = c + 1\mod \ell\) and update \(S_i\) accordingly. Thus \(S(t+1)\) is identical to \(S(t)\) in all but the \(i\)-th block, where the difference is a single bit in the variable, and the fact that the line counter is incremented.

• If \(c = 0\) then if \(\text{halted}\) is equal to 1 we do nothing. Otherwise we first let \(i'\) be \(i+1\) or \(i-1\), based on the value of \(\text{idxincreasing}\) which we can read off \(S(t)_i\). We thus set the first bit of \(S_i\) to be zero, set the first bit of \(S_i'\) to one, and copy the values of the unindexed variables from the \(i\)-th block to the \(i'\)-th block, zeroing them out in the \(i\)-th block. We then execute line zero in the program in the \(i'\)-th block as above. In this case \(S(t+1)\) is identical to \(S(t)\) in all but the two adjacent blocks \(i\) and \(i'\). If \(i'\) does not exist in \(S(t+1)\) then we append an additional \(1 + \lceil \log \ell \rceil + a + b\) bits to it.

**Definition 8.2 — Next step function.** Let \(P\) be a normal-form NAND++ program. We define \(\text{NEXT}_P : \{0,1\}^* \rightarrow \{0,1\}^*\) to be the partial function such that for every input \(x \in \{0,1\}^*\) and \(t \in \mathbb{N}\), if \(S \in \{0,1\}^*\) is a string encoding a valid snapshot of the computation of \(P\) on input \(x\) after \(t\) step, then \(\text{NEXT}_P(S) = S'\) where \(S'\) is the string encoding the snapshot of the computation of \(P\) on input \(x\) after \(t + 1\) steps. (If \(S\) corresponds to a halting state, in which \(\text{halted}\) has value 1, then we define \(\text{NEXT}_P(S) = S\).)
state of the program at step 0 on input $x$, where all variables except for the $x_i$ variables are equal to zero, and then compute $\text{NEXT}_P$ repeatedly until we reach a state where $\text{halted}$ equals 1, in which case we can read off the output from the $y_i$ variables.

### 8.2.3 Simulating NAND++ programs with Turing machines

Using the above, we can finish Theorem 8.1 by showing that for every normal-form $P$, there is a Turing machine $M$ that computes the function $\text{NEXT}_P$. Let $B = 1 + \lceil \log \ell \rceil + a + b$ be the block size of the snapshots of $P$. Note that $B$ is a constant independent of the input size, and in particular our machine $M$ will have about $2^B$ states, which we can think of as a local memory of $7B$ bits. The machine will do the following:

- Scan the tape in jumps of $B$ bits until we find a block $i$ that begins with 1.
- Copy to memory the contents of the blocks $i-1, i, i+1$ to the first $3B$ bits local memory (requires $3B$ bits).
- Compute the updated value of these three blocks based on the logic above (this is a finite function from $\{0, 1\}^{3B}$ to bits $\{0, 1\}^{3B}$) and write this to the next $3B$ bits of memory. the state accordingly.
- Write to the new contents of the $i-1, i, i+1$ block from memory.

Writing down the full description of $M$ from the above “pseudocode” is straightforward, even if somewhat painful, exercise, and hence this completes the proof of Theorem 8.1.

### 8.2.4 Advanced note: polynomial equivalence

If we examine the proof of Theorem 8.1 then we can see that the equivalence between NAND++ programs and NAND« programs is up to polynomial overhead in the number of steps. That is, for every NAND++ program $P$, there is a Turing machine $M$ and a constant $c$ such that for every $x \in \{0, 1\}^*$, if on input $x$, $P$ halts within $T$ steps and outputs $y$, then on the same input $x$, $M$ halts within $c \cdot T^c$ steps with the same output $y$. Similarly, for every Turing machine $M$, there is a NAND++ program $P$ and a constant $d$ such that for every $x \in \{0, 1\}^*$, if on input $x$, $M$ halts within $T$ steps and outputs $y$, then on the same input $x$, $P$ outputs within $d \cdot T^d$ steps with the same output $y$.\(^5\)

\(^5\) TODO: check if the overhead is really what I say it is.


8.3 “Turing Completeness” and other Computational models

A computational model \( M \) is Turing complete if every partial function \( F : \{0, 1\}^* \to \{0, 1\}^* \) that is computable by a Turing Machine is also computable in \( M \). A model is Turing equivalent if the other direction holds as well; that is, for every partial function \( F : \{0, 1\}^* \to \{0, 1\}^* \), \( F \) is computable by a Turing machine if and only if it is computable in \( M \). Another way to state Theorem 8.1 is that NAND++ is Turing equivalent. Since we can simulate any NAND\texttt{«} program by a NAND++ program (and vice versa), NAND\texttt{«} is Turing equivalent as well. It turns out that there are many other Turing-equivalent models of computation. We now discuss a few examples.

8.3.1 RAM Machines

The Word RAM model is a computational model that is arguably more similar to real-world computers than Turing machines or NAND++ programs. In this model the memory is an array of unbounded size where each cell can store a single word, which we think of as a string in \( \{0, 1\}^w \) and also as a number in \([2^w]\). The parameter \( w \) is known as the word size and is chosen as some function of the input length \( n \). A typical choice is that \( w = c \log n \) for some constant \( c \). This is sometimes known as the “transdichotomous RAM model”. In this model there are a constant number of registers \( r_1, \ldots, r_k \) that also contain a single word. The operations in this model include loops, arithmetic on registers, and reading and writing from a memory location addressed by the register. See this lecture for a more precise definition of this model.

We will not show all the details but it is not hard to show that the word RAM model is equivalent to NAND\texttt{«} programs. Every NAND\texttt{«} program can be easily simulated by a RAM machine as long as \( w \) is larger than the logarithm of its running time. In the other direction, a NAND\texttt{«} program can simulate a RAM machine, using its variables as the registers. The only significant difference between NAND\texttt{«} and the word-RAM model is that in NAND\texttt{«}, since we have a constant number of variables, we can think of our memory as an array where in each location there are only \( t = O(1) \) bits stored (one bit for each distinct unindexed variable identifier used in the program). Thus we will need a factor of \( O(w) \) multiplicative overhead to simulate the word-RAM model with an array storing \( w \)-length words.
8.3.2 Imperative languages

As we discussed before, any function computed by a standard programming language such as C, Java, Python, Pascal, Fortran etc. can be computed by a NAND++ program. Indeed, a compiler for such languages translates programs into machine languages that are quite similar to NAND++ programs or RAM machines. We can also translate NAND++ programs to such programming languages. Thus all these programming languages are Turing equivalent.\(^6\)

8.4 Lambda calculus and functional programming languages

The \(\lambda\) calculus is another way to define computable functions. It was proposed by Alonzo Church in the 1930’s around the same time as Alan Turing’s proposal of the Turing Machine. Interestingly, while Turing Machines are not used for practical computation, the \(\lambda\) calculus has inspired functional programming languages such as LISP, ML and Haskell, and so indirectly, the development of many other programming languages as well.

**The \(\lambda\) operator.** At the core of the \(\lambda\) calculus is a way to define “anonymous” functions. For example, instead of defining the squaring function as

\[
square(x) = x \cdot x
\]  \hspace{1cm} (8.1)

we write it as

\[
\lambda x.x \cdot x
\]  \hspace{1cm} (8.2)

\(\lambda\) Some programming language have hardwired fixed (even if extremely large) bounds on the amount of memory they can access. We ignore such issues in this discussion and assume access to some storage device without a fixed upper bound on its capacity.
Generally, an expression of the form

$$\lambda x. e$$  \hspace{1cm} (8.3)

corresponds to the function that maps any expression $z$ into the expression $e[x \to z]$ which is obtained by replacing every occurrence of $x$ in $e$ with $z$.\(^7\)

**Currying.** The expression $e$ can itself involve $\lambda$, and so for example the function

$$\lambda x. (\lambda y. x + y)$$  \hspace{1cm} (8.4)

maps $x$ to the function $y \mapsto x + y$.

In particular, if we invoke this function on $a$ and then invoke the result on $b$ then we get $a + b$. We can use this approach, to achieve the effect of functions with more than one input and so we will use the shorthand $\lambda x, y. e$ for $\lambda x. (\lambda y. e)$.\(^8\)

Figure 8.4: In the “currying” transformation, we can create the effect of a two parameter function $f(x, y)$ with the $\lambda$ expression $\lambda x. (\lambda y. f(x, y))$ which on input $x$ outputs a one-parameter function $f_x$ that has $x$ “hardwired” into it and such that $f_x(y) = f(x, y)$. This can be illustrated by a circuit diagram; see Chelsea Voss’s site.

**Precedence and parenthesis.** The above is a complete description of the $\lambda$ calculus. However, to avoid clutter, we will allow to drop parenthesis for function invocation, and so if $f$ is a $\lambda$ expression and $z$ is some other expression, then we can write $f z$ instead of $f(z)$ for the expression corresponding to invoking $f$ on $z$.\(^9\) That is, if $f$ has the form $\lambda x. e$ then $f z$ is the same as $f(z)$, which corresponds to the expression $e[x \to z]$ (i.e., the expression obtained by invoking $f$ on $z$ via replacing all copies of the $x$ parameter with $z$).

\(^7\) More accurately, we replace every expression of $x$ that is bounded by the $\lambda$ operator. For example, if we have the $\lambda$ expression $\lambda x. (\lambda x. x + 1)(x)$ and invoke it on the number $7$ then we get $(\lambda x. x + 1)(7) = 8$ and not the nonsensical expression $(\lambda 7.7 + 1)(7)$. To avoid such annoyances, we can always ensure that every instance of $\lambda \text{var}.e$ uses a unique variable identifier $\text{var}$.

\(^8\) This technique of simulating multiple-input functions with single-input functions is known as Currying and is named after the logician Haskell Curry.

\(^9\) When using identifiers with multiple letters for $\lambda$ expressions, we’ll separate them with spaces or commas.
We can still use parenthesis for grouping and so \( f(gh) \) means invoking \( g \) on \( h \) and then invoking \( f \) on the result, while \((fg)h\) means invoking \( f \) on \( g \) and then considering the result as a function which then is invoked on \( h \). We will associate from left to right and so identify \( fgh \) with \((fg)h\). For example, if \( f = \lambda x. (\lambda y. x + y) \) then \( fzw = (fz)w = z + w. \)

**Functions as first-class citizens.** The key property of the \( \lambda \) calculus (and functional languages in general) is that functions are “first-class citizens” in the sense that they can be used as parameters and return values of other functions. Thus, we can invoke one \( \lambda \) expression on another. For example if \( \text{DOUBLE} \) is the \( \lambda \) expression \( \lambda f. (\lambda x. f(x)) \), then for every function \( f \), \( \text{DOUBLE} f \) corresponds to the function that invokes \( f \) twice on \( x \) (i.e., first computes \( fx \) and then invokes \( f \) on the result). In particular, if \( f = \lambda y.y + 1 \) then \( \text{DOUBLE} f = \lambda x.x + 2. \)

**(Lack of) types.** Unlike most programming languages out there, the pure \( \lambda \)-calculus doesn’t have the notion of types. Every object in the \( \lambda \) calculus can also be thought of as a \( \lambda \) expression and hence as a function that takes one input and returns one output. All functions take one input and return one output, and if you feed a function an input of a form it didn’t expect, it still evaluates the \( \lambda \) expression via “search and replace”, replacing all instances of its parameter with copies of the input expression you fed it.

### 8.4.1 The “basic” \( \lambda \) calculus objects

To calculate, it seems we need some basic objects such as 0 and 1, and so we will consider the following set of “basic” objects and operations:

- **Boolean constants:** 0 and 1. We also have the \( \text{IF}(\text{cond}, a, b) \) functions that outputs \( a \) if \( \text{cond} = 1 \) and \( b \) otherwise. Using \( \text{IF} \) we can also compute logical operations such as \( \text{AND}, \text{OR}, \text{NOT}, \text{NAND} \) etc.: can you see why?

- **The empty string:** The value \( \text{NIL} \) and the function \( \text{ISNIL}(x) \) that returns 1 iff \( x \) is \( \text{NIL}. \)

- **Strings/lists:** The function \( \text{PAIR}(x, y) \) that creates a pair from \( x \) and \( y \). We will also have the function \( \text{HEAD} \) and \( \text{TAIL} \) to extract the first and second member of the pair. We can now create the list \( x, y, z \) by \( \text{PAIR}(x, \text{PAIR}(y, \text{PAIR}(z, \text{NIL}))) \), see Fig. 8.5. A string is of course simply a list of bits.
• **List operations:** The functions $MAP, REDUCE, FILTER$. Given a list $L = (x_0, \ldots, x_{n-1})$ and a function $f$, $MAP(L, f)$ applies $f$ on every member of the list to obtain $L = (f(x_0), \ldots, f(x_{n-1}))$. The function $FILTER(L, f)$ returns the list of $x_i$’s such that $f(x_i) = 1$, and $REDUCE(L, f)$ “combines” the list by outputting

$$f(x_0, f(x_1, \cdots f(x_{n-3}, f(x_{n-2}, x_{n-1})) \cdots) \quad (8.5)$$

For example $REDUCE(L, +)$ would output the sum of all the elements of the list $L$. See Fig. 8.6 for an illustration of these three operations.

---

**Figure 8.5:** A list $(x_0, x_1, x_2)$ in the $\lambda$ calculus is constructed from the tail up, building the pair $(x_2, NIL)$, then the pair $(x_1, (x_2, NIL))$ and finally the pair $(x_0, (x_1, (x_2, NIL)))$.

That is, a list is a pair where the first element of the pair is the first element of the list and the second element is the rest of the list. The figure on the left renders this “pairs inside pairs” construction, though it is often easier to think of a list as a “chain”, as in the figure on the right, where the second element of each pair is thought of as a link, pointer or reference to the remainder of the list.

---

**Figure 8.6:** Illustration of the $MAP, FILTER$ and $REDUCE$ operations.

Together these operations more or less amount to the Lisp/Scheme programming language.\(^{10}\)

Given that, it is perhaps not surprising that we can simulate NAND++ programs using the $\lambda$-calculus plus these basic elements, hence showing the following theorem:

\(^{10}\) In Lisp, the $PAIR, HEAD$ and $TAIL$ functions are traditionally called cons, car and cdr.
Theorem 8.2 — Lambda calculus and NAND++. For every function 
\( F : \{0,1\}^* \to \{0,1\}^* \), \( F \) is computable in the \( \lambda \) calculus with 
the above basic operations if and only if it is computable by a 
NAND++ program.

Proof. The “only if” direction is simple. As mentioned above, evaluat-
ing \( \lambda \) expressions basically amounts to “search and replace”. It is also 
a fairly straightforward programming exercise to implement all the 
above basic operations in an imperative language such as Python or 
C, and using the same ideas we can do so in NAND« as well, which 
we can then transform to a NAND++ program.

For the “if” direction, it suffices to show that for every normal-
form NAND++ program \( P \), we can compute the next-snapshot 
function \( \text{NEXT}_P : \{0,1\}^* \to \{0,1\}^* \) using the above operations. 
It turns out not to be so hard. A snapshot of \( P \) is a string of length 
\( TB \) where \( B \) is the (constant sized) block size, and so we can think 
of it as a list \( S = (S_1, \ldots ,S^T) \) of \( T \) lists of bits, each of length \( B \). 
Extracting from this list the \( B \) sized string corresponding to the 
block \( S_i \) where \( S_i[0] = 1 \) can be done via a single \( \text{REDUCE} \) operations. 
Using this we can tell if this is an operation where \( i \) stays the same, 
increases, or decreases. If it stays the same then we can compute 
\( \text{NEXT}_P \) via a \( \text{MAP} \) operation, using the function that on input \( C \in \{0,1\}^8 \), keeps \( C \) the same if \( C_0 = 0 \) and otherwise updates it to 
the value in its next step. If it increases, then we can update it by a 
\( \text{REDUCE} \) operation, with the function that on input a block \( C \) and a 
list \( S \), we output \( \text{PAIR}(C, L) \) unless \( C_0 = 1 \) in which case we output 
\( \text{PAIR}(C', \text{PAIR}(C'', \text{TAIL}(S))) \) where \( (C', C'') \) are the new values of 
the blocks \( i \) and \( i + 1 \). The case for decreasing \( i \) is analogous. ■

8.4.2 How basic is “basic”?

While the collection of “basic” functions above is smaller than what’s 
provided by most Lisp dialects, coming from NAND++ it still seems 
a little “bloated”. Can we make do with less? In other words, can we 
find a subset of these basic operations that can implement the rest?

This is a good point to pause and think how you 
would implement these operations yourself. For ex-
ample, start by thinking how you could implement 
\( \text{MAP} \) using \( \text{REDUCE} \), and then try to continue 
and minimize things further, trying to implement 
\( \text{REDUCE} \) with from \( 0,1,\text{IF, PAIR, HEAD, TAIL} \).
It turns out that there is in fact a proper subset of these basic operations that can be used to implement the rest. That subset is the empty set. That is, we can implement all the operations above using the λ formalism only, even without using 0’s and 1’s. It’s λ’s all the way down! The idea is that we encode 0 and 1 themselves as λ expressions, and build things up from there. This notion is known as Church encoding, as was originated by Church in his effort to show that the λ calculus can be a basis for all computation.

We now outline how this can be done:

• We define 0 to be the function that on two inputs $x, y$ outputs $y$, and 1 to be the function that on two inputs $x, y$ outputs $x$. Of course we use Currying to achieve the effect of two inputs and hence $0 = \lambda x.\lambda y.y$ and $1 = \lambda x.\lambda y.x$.\(^{11}\)

• The above implementation makes the IF function trivial:
  \[ IF(\text{cond}, a, b) \text{ is simply } \text{cond}, a, b \text{ since } 0ab = b \text{ and } 1ab = a. \]
  (We can write \(IF = \lambda x.x\) to achieve \(IF\text{cond}ab = \text{cond}ab\).)

• To encode a pair $(x, y)$ we will produce a function $f_{x,y}$ that has $x$ and $y$ “in its belly” and such that $f_{x,y}g = gxy$ for every function $g$. That is, we write \(PAIR = \lambda x, y.\lambda g.\lambda x, y.gxy\). Note that now we can extract the first element of a pair $p$ by writing $p1$ and the second element by writing $p0$, and so \(\text{HEAD} = \lambda p.p1\) and \(\text{TAIL} = \lambda p.p0\).

• We define \(NIL\) to be the function that ignores its input and always outputs 1. That is, \(NIL = \lambda x.1\). The ISNIL function checks, given an input $p$, whether we get 1 if we apply $p$ to the function $0_{x,y}$ that ignores both its inputs and always outputs 0. For every valid pair $p0_{x,y} = 0$ while $NIL0_{x,y} = 1$. Formally, \(\text{ISNIL} = \lambda p.\lambda x, y.0\).

8.4.3 List processing and recursion without recursion

Now we come to the big hurdle, which is how to implement MAP, FILTER, and REDUCE in the λ calculus. It turns out that we can build MAP and FILTER from REDUCE. For example MAP($L, f$) is the same as REDUCE($L, g$) where $g$ is the operation that on input $x$ and $y$, outputs $f(x)$ if $y$ is NIL and otherwise outputs $PAIR(f(x), y)$. (I leave checking this as a (recommended!) exercise for you, the reader.) So, it all boils down to implementing REDUCE. We can define REDUCE($L, g$) recursively, by setting REDUCE($NIL, g$) = NIL.
and stipulating that given a non-empty list \( L \), which we can think of as a pair \((\text{head}, \text{rest})\), \( \text{REDUCE}(L, g) = g(\text{head}, \text{REDUCE}(\text{rest}, g)) \). Thus, we might try to write a \( \lambda \) expression for \( \text{REDUCE} \) as follows

\[
\text{REDUCE} = \lambda L, g. \text{IF}(\text{ISNIL}(L), \text{NIL}, g\text{HEAD}(L)\text{REDUCE}(\text{TAIL}(L), g)) .
\]  

(8.6)

The only fly in this ointment is that the \( \lambda \) calculus does not have the notion of recursion, and so this is an invalid definition. This seems like a very serious hurdle: if we don’t have loops, and don’t have recursion, how are we ever going to be able to compute a function like \( \text{REDUCE} \)?

The idea is to use the “self referential” properties of the \( \lambda \) calculus. Since we are able to work with \( \lambda \) expressions, we can possibly inside \( \text{REDUCE} \) compute a \( \lambda \) expression that amounts to running \( \text{REDUCE} \) itself. This is very much like the common exercise of a program that prints its own code. For example, suppose that you have some programming language with an \( \text{eval} \) operation that given a string code and an input \( x \), evaluates its own code. Then, if you have a program \( P \) that can print its own code, you can use \( \text{eval} \) as an alternative to recursion: instead of using a recursive call, the program will compute its own code and store it in a variable \( \text{str} \) and then use \( \text{eval} \). You might find this confusing. I definitely find this confusing. But hopefully the following will make things a little more concrete.

12

8.4.4 The \( Y \) combinator

The solution is to think of a recursion as a sort of “differential equation” on functions. For example, suppose that all our lists contain either 0 or 1 and consider \( \text{REDUCE}(L, \text{XOR}) \) which simply computes the parity of the list elements. The ideas below will clearly generalize for implementing \( \text{REDUCE} \) with any other function, and in fact for implementing recursive functions in general. We can define the parity function \( \text{par} \) recursively as

\[
\text{par}(x_0, \ldots, x_n) = \begin{cases} 
0 & |x| = 0 \\
x_0 \oplus \text{par}(x_1, \ldots, x_n) & \text{otherwise}
\end{cases}
\]  

(8.7)

where \( \oplus \) denotes the XOR operator.

Our key insight would be to recast Eq. (8.13) not as a definition of

° TODO: add a direct example how to implement \( \text{REDUCE} \) with \( \text{XOR} \) without using the \( Y \) combinator. Hopefully it can be done in a way that makes things more intuitive.
the parity function but rather as an equation on it. That is, we can think of Eq. (8.13) as stating that

\[ par = \text{PAREQ}(par) \]  

(8.8)

where PAREQ is a non-recursive function that takes a function \( p \) as input, and returns the function \( q \) defined as

\[
q(x_0,\ldots,x_n) = \begin{cases} 
0 & \text{if } |x| = 0 \\
x_0 \oplus p(x_1,\ldots,x_n) & \text{otherwise}
\end{cases} 
\]

(8.9)

In fact, it’s not hard to see that satisfying Eq. (8.8) is equivalent to satisfying Eq. (8.13), and hence \( par \) is the unique function that satisfies the condition Eq. (8.8). This means that to find a function \( par \) computing parity, all we need is a “magical function” \( \text{SOLVE} \) that given a function PAREQ finds “fixed point” of PAREQ: a function \( p \) such that \( \text{PAREQ}(p) = p \). Given such a “magical function”, we could give a non-recursive definition for \( par \) by writing \( par = \text{SOLVE}(\text{PAREQ}) \).

It turns out that we can find such a “magical function” \( \text{SOLVE} \) in the \( \lambda \)-calculus, and this is known as the \( Y \) combinator.

**Theorem 8.3 — Y combinator.** Let

\[
Y = \lambda f. (\lambda x. f(xx))(\lambda y. f(yy))
\]

(8.10)

then for every \( \lambda \) expression \( F \), if we let \( h = YF \) then \( h = Fh \).

**Proof.** Indeed, for every \( \lambda \) expression \( F \) of the form \( \lambda t.e \), we can see that

\[
YF = (\lambda x. F(xx))(\lambda y. F(yy))
\]

(8.11)

But this is the same as applying \( F \) to \( gg \) where \( g = \lambda y. F(y,y) \), or in other words

\[
YF = F ((\lambda y. F(y,y))(\lambda y. F(y,y)))
\]

(8.12)

but by a change of variables the RHS is the same as \( F(YF) \).

Using the \( Y \) combinator we can implement recursion in the \( \lambda \)-calculus, and hence loops. This can be used to complete the “if” direction of ??.
For example, to compute parity we first give a recursive definition of parity using the \( \lambda \)\-calculus as

\[
par L = \text{IF}(\text{ISNIL}(L), 0, \text{XORHEAD}(L)\text{par}(\text{TAIL}(L))) \quad (8.13)
\]

We then avoid the recursion by converting Eq. (8.13) to the operator \( PAREQ \) defined as

\[
PAREQ = \lambda p.\lambda L.\text{IF}(\text{ISNIL}(L), 0, \text{XORHEAD}(L)p(\text{TAIL}(L))) \quad (8.14)
\]

and then we can define \( \text{par} \) as \( YPAREQ \) since this will be the unique solution to \( p = PAREQp \).

**Infinite loops in \( \lambda \)-expressions.** The fact that \( \lambda \)-expressions can simulate NAND++ programs means that, like them, it can also enter into an infinite loop. For example, consider the \( \lambda \) expression

\[
(\lambda x.xxx)(\lambda x.xxx)
\]

If we try to evaluate it then the first step is to invoke the lefthand function on the righthand one and then obtain

\[
(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)
\]

To evaluate this, the next step would be to apply the second term on the third term,\(^{13}\) which would result in

\[
(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)
\]

We can see that continuing in this way we get longer and longer expressions, and this process never concludes.

### 8.5 Other models

There is a great variety of models that are computationally equivalent to Turing machines (and hence to NAND++/NAND\(_k\) program). Chapter 8 of the book *The Nature of Computation* is wonderful resource for some of those models. We briefly mention a few examples.
8.5.1 Parallel algorithms and cloud computing

The models of computation we considered so far are inherently sequential, but these days much computation happens in parallel, whether using multi-core processors or in massively parallel distributed computation in data centers or over the Internet. Parallel computing is important in practice, but it does not really make much difference for the question of what can and can’t be computed. After all, if a computation can be performed using \( m \) machines in \( t \) time, then it can be computed by a single machine in time \( mt \).

8.5.2 Game of life, tiling and cellular automata

Many physical systems can be described as consisting of a large number of elementary components that interact with one another. One way to model such systems is using cellular automata. This is a system that consists of a large number (or even infinite) cells. Each cell only has a constant number of possible states. At each time step, a cell updates to a new state by applying some simple rule to the state of itself and its neighbors.

A canonical example of a cellular automaton is Conway’s Game of Life. In this automata the cells are arranged in an infinite two dimensional grid. Each cell has only two states: “dead” (which we can encode as 0) or “alive” (which we can encode as 1). The next state of a cell depends on its previous state and the states of its 8 vertical, horizontal and vertical neighbors. A dead cell becomes alive only if exactly three of its neighbors are alive. A live cell continues to live if it has two or three live neighbors. Even though the number of cells is potentially infinite, we can have a finite encoding for the state by only keeping track of the live cells. If we initialize the system in a configuration with a finite number of live cells, then the number of live cells will stay finite in all future steps.

We can think of such a system as encoding a computation by starting it in some initial configuration, and then defining some halting condition (e.g., we halt if the cell at position \((0,0)\) becomes dead) and some way to define an output (e.g., we output the state of the cell at position \((1,1)\)). Clearly, given any starting configuration \(x\), we can simulate the game of life starting from \(x\) using a NAND\(^{-}\) (or NAND\(\text{++}\)) program, and hence every “Game-of-Life computable” function is computable by a NAND\(^{-}\) program. Surprisingly, it turns out that the other direction is true as well: as simple as its rules seem, we can simulate a NAND\(\text{++}\) program using the game of life (see
Fig. 8.7). The Wikipedia page for the Game of Life contain some beautiful figures and animations of configurations that produce some very interesting evolutions. See also the book The Nature of Computation. It turns out that even one dimensional cellular automata can be Turing complete, see Fig. 8.8.

![Figure 8.7: A Game-of-Life configuration simulating a Turing Machine. Figure by Paul Rendell.](image)

8.6 Our models vs standard texts

We can summarize the models we use versus those used in other texts in the following table:

<table>
<thead>
<tr>
<th>Model</th>
<th>These notes</th>
<th>Other texts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonuniform</td>
<td>NAND programs</td>
<td>Boolean circuits, straightline programs</td>
</tr>
<tr>
<td>Uniform (random access)</td>
<td>NAND+ programs</td>
<td>RAM machines</td>
</tr>
<tr>
<td>Uniform (sequential access)</td>
<td>NAND++ programs</td>
<td>Oblivious one-tape Turing machines</td>
</tr>
</tbody>
</table>

Later on in this course we will study memory bounded computation. It turns out that NAND++ programs with a constant amount of memory are equivalent to the model of finite automata (the adjectives “deterministic” or “nondeterministic” are sometimes added as well, this model is also known as finite state machines) which in turns
captures the notion of regular languages (those that can be described by regular expressions).

8.7 The Church-Turing Thesis

We have defined functions to be computable if they can be computed by a NAND++ program, and we’ve seen that the definition would remain the same if we replaced NAND++ programs by Python programs, Turing machines, λ calculus, cellular automata, and many other computational models. The Church-Turing thesis is that this is the only sensible definition of “computable” functions. Unlike the “Physical Extended Church Turing Thesis” (PECTT) which we saw before, the Church Turing thesis does not make a concrete physical prediction that can be experimentally tested, but it certainly motivates predictions such as the PECTT. One can think of the Church-Turing Thesis as either advocating a definitional choice, making some prediction about all potential computing devices, or suggesting some laws of nature that constrain the natural world. In Scott Aaronson’s words, “whatever it is, the Church-Turing thesis can only be regarded as extremely successful”. No candidate computing device (including
quantum computers, and also much less reasonable models such as the hypothetical “closed time curve” computers we mentioned before) has so far mounted a serious challenge to the Church Turing thesis. These devices might potentially make some computations more efficient, but they do not change the difference between what is finitely computable and what is not.\textsuperscript{14}
tonomous Organization” or The DAO. The idea was to create a smart contract that would create an autonomously run decentralized venture capital fund, without human managers, were shareholders could decide on investment opportunities. The DAO was the biggest crowdfunding success in history and at its height was worth 150 million dollars, which was more than ten percent of the total Ethereum market. Investing in the DAO (or entering any other “smart contract”) amounts to providing your funds to be run by a computer program. i.e., “code is law”, or to use the words the DAO described itself: “The DAO is borne from immutable, unstoppable, and irrefutable computer code”. Unfortunately, it turns out that (as we’ll see in the next lecture) understanding the behavior of Turing-complete computer programs is quite a hard thing to do. A hacker (or perhaps, some would say, a savvy investor) was able to fashion an input that would cause the DAO code to essentially enter into an infinite recursive loop in which it continuously transferred funds into their account, thereby cleaning out about 60 million dollars out of the DAO. While this transaction was “legal” in the sense that it complied with the code of the smart contract, it was obviously not what the humans who wrote this code had in mind. There was a lot of debate in the Ethereum community how to handle this, including some partially successful “Robin Hood” attempts to use the same loophole to drain the DAO funds into a secure account. Eventually it turned out that the code is mutable, stoppable, and refutable after all, and the Ethereum community decided to do a “hard fork” (also known as a “bailout”) to revert history to before this transaction. Some elements of the community strongly opposed this decision, and so an alternative currency called Ethereum Classic was created that preserved the original history.

8.9 Lecture summary

- While we defined computable functions using NAND++ programs, we could just as well have done so using many other models, including not just NAND++ but also Turing machines, RAM machines, the λ-calculus and many other models.

- Very simple models turn out to be “Turing complete” in the sense that they can simulate arbitrarily complex computation.

8.10 Exercises

TODO: Add an exercise showing that NAND++ programs where the integers are represented using the unary basis are equivalent up to polylog terms with multi-tape Turing machines.
Exercise 8.1 — **lambda calculus requires three variables.** Prove that for every \( \lambda \)-expression \( e \) with no free variables there is an equivalent \( \lambda \)-expression \( f \) using only the variables \( x, y, z \).\(^{16} \)

**Hint:** You can reduce the number of variables a function takes by “pairing them up”. That is, define a \( \lambda \) expression \( \text{PAIR} \) such that for every \( x, y \), \( \text{PAIR}xy \) is some function \( f \) such that \( f0 = x \) and \( f1 = y \). Then use \( \text{PAIR} \) to iteratively reduce the number of variables used.

7 TODO: Recommend Chapter 7 in the nature of computation

8.11 Bibliographical notes

8.12 Further explorations

Some topics related to this lecture that might be accessible to advanced students include:

- Tao has **proposed** showing the Turing completeness of fluid dynamics (a “water computer”) as a way of settling the question of the behavior of the Navier-Stokes equations, see this [popular article](#).

8.13 Acknowledgements
Is every function computable?

“A function of a variable quantity is an analytic expression composed in any way whatsoever of the variable quantity and numbers or constant quantities.”, Leonhard Euler, 1748.

We saw that NAND programs can compute every finite function. A natural guess is that NAND++ programs could compute every infinite function. However, this turns out to be false, even for functions with 0/1 output. That is, there exists a function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ that is uncomputable! This is actually quite surprising, if you think about it. Our intuitive notion of a “function” (and the notion most scholars had until the 20th century) is that a function $f$ defines some implicit or explicit way of computing the output $f(x)$ from the input $x$. The notion of an “uncomputable function” thus seems to be a contradiction in terms, but yet the following theorem shows that such creatures do exist:

\begin{theorem}
Uncomputable functions. There exists a function $F^* : \{0, 1\}^* \rightarrow \{0, 1\}$ that is not computable by any NAND++ program.
\end{theorem}

Proof. The proof is illustrated in Fig. 9.1. We start by defining the following function $G : \{0, 1\}^* \rightarrow \{0, 1\}$:

For every string $x \in \{0, 1\}^*$, if $x$ satisfies (1) $x$ is a valid representation of a NAND++ program $P_x$ and (2) when the program $P_x$ is executed on the input $x$ it halts and produces an output, then we define $G(x)$ as the first bit of this output. Otherwise (i.e., if $x$ is not a valid representation of a program, or the program $P_x$ never halts on $x$) we define $G(x) = 0$. We define $F^*(x) := 1 - G(x)$.

\*In the 1800’s, with the invention of the Fourier series and with the systematic study of continuity and differentiability, people have starting looking at more general kinds of functions, but the modern definition of a function as an arbitrary mapping was not yet universally accepted. For example, in 1899 Poincare wrote “we have seen a mass of bizarre functions which appear to be forced to resemble as little as possible honest functions which serve some purpose. . . . they are invented on purpose to show that our ancestor’s reasoning was at fault, and we shall never get anything more than that out of them.”
We claim that there is no NAND++ program that computes $F^*$. Indeed, suppose, towards the sake of contradiction, that there was some program $P$ that computed $F^*$, and let $x$ be the binary string that represents the program $P$. Then on input $x$, the program $P$ outputs $F^*(x)$. But by definition, the program should also output $1 - F^*(x)$, hence yielding a contradiction.

![Figure 9.1](image)

**Figure 9.1:** We construct an uncomputable function by defining for every two strings $x, y$ the value $1 - P_y(x)$ which equals to 0 if the program described by $y$ outputs 1 on $x$, and equals to 1 otherwise. We then define $F^*(x)$ to be the “diagonal” of this table, namely $F^*(x) = 1 - P_x(x)$ for every $x$. The function $F^*$ is uncomputable, because if it was computable by some program whose string description is $x^*$ then we would get that $P_{x^*}(x^*) = F(x^*) = 1 - P_x(x^*)$.

The proof of Theorem 9.1 is short but subtle, and it crucially uses the dual view of a program as both instructions for computation, as well as a string that can be an input for this computation. I suggest that you pause here and go back to read it again and think about it - this is a proof that is worth reading at least twice if not three or four times. It is not often the case that a few lines of mathematical reasoning establish a deeply profound fact - that there are problems we simply cannot solve and the “firm conviction” that Hilbert alluded to above is simply false. The type of argument used to prove Theorem 9.1 is known as diagonalization since it can be described as defining a function based on the diagonal entries of a table as in Fig. 9.1.
9.1 The Halting problem

Theorem 9.1 shows that there is some function that cannot be computed. But is this function the equivalent of the “tree that falls in the forest with no one hearing it”? That is, perhaps it is a function that no one actually wants to compute.

It turns out that there are natural uncomputable functions:

**Theorem 9.2 — Uncomputability of Halting function.** Let $HALT : \{0,1\}^* \rightarrow \{0,1\}$ be the function such that $HALT(P, x) = 1$ if the program $P$ halts on input $x$ and equals to 0 if it does not. Then $HALT$ is not computable.

Before turning to prove Theorem 9.2, we note that $HALT$ is a very natural function to want to compute. For example, one can think of $HALT$ as a special case of the task of managing an “App store”. That is, given the code of some application, the gatekeeper for the store needs to decide if this code is safe enough to allow in the store or not. At a minimum, it seems that we should verify that the code would not go into an infinite loop.

![Figure 9.2](image-url)  
**Figure 9.2:** We prove that $HALT$ is uncomputable using a reduction to the uncomputability of the function $F'$. We assume that we had an algorithm that computes $HALT$ and use that to obtain an algorithm that computes $F'$. 

Proof. The proof is by a reduction to Theorem 9.1, as illustrated in Fig. 9.2. That is, we will assume, towards a contradiction, that there is NAND++ program $P^*$ that can compute the $HALT$ function, and use that to derive that there is some NAND++ program $Q^*$ that computes the function $F^*$ defined above, contradicting Theorem 9.1.

Indeed, suppose that $P^*$ was a NAND++ program that computes $HALT$. Then we can write a NAND++ program $Q^*$ that does the following on input $x \in \{0, 1\}^*$:

1. Compute $z = P^*(x, x)$
2. If $z = 0$ then output 1.
3. Otherwise, if $z = 1$ then let $y$ be the first bit of $EVAL(x, x)$ (i.e., evaluate the program described by $x$ on the input $x$). If $y = 1$ then output 0. Otherwise output 1.

Claim: For every $x \in \{0, 1\}^*$, if $P^*(x, x) = HALT(x, x)$ then the program $Q^*(x) = F^*(x)$ where $F^*$ is the function from the proof of Theorem 9.1.

Note that the claim immediately implies that our assumption that $P^*$ computes $HALT$ contradicts Theorem 9.1, where we proved that the function $F^*$ is uncomputable. Hence the claim is sufficient to prove the theorem.

Proof of claim: Let $x$ be any string. If the program described by $x$ halts on input $x$ and its first output bit is 1 then $F^*(x) = 0$ and the output $Q^*(x)$ will also equal 0 since $z = HALT(x, x) = 1$, and hence in step 3 the program $Q^*$ will run in a finite number of steps (since the program described by $x$ halts on $x$), obtain the value $y = 1$ and output 0.

Otherwise, there are two cases. Either the program described by $x$ does not halt on $x$, in which case $z = 0$ and $Q^*(x) = 1 = F^*(x)$. Or the program halts but its first output bit is not 1. In this case $z = 1$ but the value $y$ computed by $Q^*(x)$ is not 1 and so $Q^*(x) = 1 = F^*(x)$.

\[ \square \]

9.1.1 Is the Halting problem really hard?

Many people’s first instinct when they see the proof of Theorem 9.2 is to not believe it. That is, most people do believe the mathematical statement, but intuitively it doesn’t seem that the Halting problem is really that hard. After all, being uncomputable only means that
HALT cannot be computed by a NAND++ program. But programmers seem to solve HALT all the time by informally or formally arguing that their programs halt. While it does occasionally happen that a program unexpectedly enters an infinite loop, is there really no way to solve the halting problem? Some people argue that they can, if they think hard enough, determine whether any concrete program that they are given will halt or not. Some have even argued that humans in general have the ability to do that, and hence humans have inherently superior intelligence to computers or anything else modeled by NAND++ programs (aka Turing machines).²

The best answer we have so far is that there truly is no way to solve HALT, whether using Macs, PCs, quantum computers, humans, or any other combination of mechanical and biological devices. Indeed this assertion is the content of the Church-Turing Thesis. This of course does not mean that for every possible program P, it is hard to decide if P enter an infinite loop. Some programs don’t even have loops at all (and hence trivially halt), and there are many other far less trivial examples of programs that we can certify to never enter an infinite loop (or programs that we know for sure that will enter such a loop). However, there is no general procedure that would determine for an arbitrary program P whether it halts or not. Moreover, there are some very simple programs for which it not known whether they halt or not. For example, the following Python program will halt if and only if Goldbach’s conjecture is false:

```python
def isprime(p):
    return all(p % i for i in range(2,p-1))

def Goldbach(n):
    return any((isprime(p) and isprime(n-p))
               for p in range(2,n-1))

n = 4
while True:
    if not Goldbach(n): break
    n+= 2
```

Given that Goldbach’s Conjecture has been open since 1742, it is unclear that humans have any magical ability to say whether this (or other similar programs) will halt or not.

² This argument has also been connected to the issues of consciousness and free will. I am not completely sure of its relevance but perhaps the reasoning is that humans have the ability to solve the halting problem but they exercise their free will and consciousness by choosing not to do so.
9.1.2 Reductions

The Halting problem turns out to be a linchpin of uncomputability, in the sense that Theorem 9.2 has been used to show the uncomputability of a great many interesting functions. We will see several examples in such results in the lecture and the exercises, but there are many more such results in the literature (see Fig. 9.4).

The idea behind such uncomputability results is conceptually simple but can at first be quite confusing. If we know that \(HALT\) is uncomputable, and we want to show that some other function \(BLAH\) is uncomputable, then we can do so via a contrapositive argument (i.e., proof by contradiction). That is, we show that if we had a NAND++ program that computes \(BLAH\) then we could have a NAND++ program that computes \(HALT\). (Indeed, this is exactly how we showed that \(HALT\) itself is uncomputable, by reducing to the uncomputability of the function \(F^*\) from Theorem 9.1.)

For example, to prove that \(BLAH\) is uncomputable, we could show that there is a computable function \(R : \{0,1\}^* \rightarrow \{0,1\}^*\) such that for every \(x \in \{0,1\}^*\), \(HALT(x) = BLAH(R(x))\). Such a function is known as a reduction. The confusing part about reductions is that we are assuming something we believe is false (that \(BLAH\) has an algorithm) to derive something that we know is false (that \(HALT\) has an algorithm). For this reason Michael Sipser described such results as having the form “If pigs could whistle then horses could fly”.

At the end of the day reduction-based proofs are just like other proofs by contradiction, but the fact that they involve hypothetical algorithms that don’t really exist tends to make such proofs quite
confusing. The one silver lining is that at the end of the day the
notion of reductions is mathematically quite simple, and so it’s not
that bad even if you have to go back to first principles every time you
need to remember what is the direction that a reduction should go
in. (If this discussion itself is confusing, feel free to ignore it; it might
become clearer after you see an example of a reduction such as the
proof of Theorem 9.5.)

Figure 9.4: Some of the functions that have been proven uncomputable. An arrow
from problem X to problem Y means that the proof that Y is uncomputable follows
by reducing computing X to computing Y. Black arrows correspond to proofs that are
shown in this and the next lecture while pink arrows correspond to proofs that are
known but not shown here. There are many other functions that have been shown
uncomputable via a reduction from the Halting function HALT.

9.2 Impossibility of general software verification

The uncomputability of the Halting problem turns out to be a special
case of a much more general phenomenon. Namely, that we cannot
certify semantic properties of general purpose programs. “Semantic prop-
erties” mean properties of the function that the program computes,
as opposed to properties that depend on the particular syntax. For
example, we can easily check whether or not a given C program
contains no comments, or whether all function names begin with an
upper case letter. As we’ve seen, we cannot check whether a given
program enters into an infinite loop or not. But we could still hope
to check some other properties of the program, for example verifying
that if it does halt then it will conform with some specification. Alas, this turns out to be not the case.

We start by proving a simple generalization of the Halting problem:

**Theorem 9.3 — Halting without input.** Let HALTONZERO : \{0, 1\} \* → \{0, 1\} be the function that on input \( P \in \{0, 1\}^\* \), maps \( P \) to 1 if and only if the NAND++ program represented by \( P \) halts when supplied the single bit 0 as input. Then HALTONZERO is uncomputable.

The proof of Theorem 9.3 is below, but before reading it you might want to pause for a couple of minutes and think how you would prove it yourself. This is an excellent way to get some initial comfort with the notion of proofs by reduction.

**Proof.** The proof is by reduction to HALT. Suppose, towards the sake of contradiction, that there was an algorithm \( A \) such that \( A(P') = \text{HALTONZERO}(P) \) for every \( P' \in \{0, 1\}^\* \). Then, we will construct an algorithm \( B \) that solves HALT. On input a program \( P \) and some input \( x \), the algorithm \( B \) will construct a program \( P' \) such that \( P'(0) = P(x) \) for every \( x \), and then feed this to \( A \), returning \( A(P') \). Constructing such a program is very simple: add \( |x| \) lines in the beginning of \( P' \) of the form \( \text{myx}_{\langle i \rangle} := \langle x_i \rangle \) for every \( i < |x| \). Then add the program \( P \), but change any reference to \( x_{\langle i \rangle} \) to \( \text{myx}_{\langle i \rangle} \). Clearly, regardless of its input, \( P' \) always behaves like \( P \) on input \( x \). In particular \( P' \) will halt on the input 0 if and only if \( P \) halts on the input \( x \). Thus if \( A(P') = \text{HALTONZERO}(P') \) for every \( P' \) then \( B(P, x) = \text{HALT}(P, x) \) for every \( P, x \), contradicting the uncomputability of HALT. \( \blacksquare \)

Once we show the uncomputability of HALTONZERO we can extend to various other natural functions:

**Theorem 9.4 — Computing all zero function.** Let ZEROFUNC : \{0, 1\} \* → \{0, 1\} be the function that on input \( P \in \{0, 1\}^\* \), maps \( P \) to 1 if and only if the NAND++ program represented by \( P \) outputs 0 on every input \( x \in \{0, 1\}^\* \). Then ZEROFUNC is uncomputable.

**Proof.** The proof is by reduction to HALTONZERO. Suppose, towards the sake of contradiction, that there was an algorithm \( A \) such that \( A(P') = \text{ZEROFUNC}(P') \) for every \( P' \in \{0, 1\}^\* \). Then we will construct an algorithm \( B \) that solves HALTONZERO. Given a pro-
gram $P$, Algorithm $B$ will construct the following program $P'$: on input $x \in \{0,1\}^*$, $P'$ will first run $P(0)$, and then output 0.

Now if $P$ halts on 0 then $P'(x) = 0$ for every $x$, but if $P$ does not halt on 0 then $P'$ will never halt on every input and in particular will not compute ZEROFUNC. Hence, $ZEROFUNC(P') = 1$ if and only if $HALTONZERO(P) = 1$. Thus if we define algorithm $B$ as $B(P) = A(P')$ (where a program $P$ is mapped to $P'$ as above) then we see that if $A$ computes $ZEROFUNC$ then $B$ computes $HALTONZERO$, contradicting Theorem 9.3.

We can simply prove the following:

**Theorem 9.5 — Uncomputability of verifying parity.** The following function is uncomputable

$$COMPUTES-PARITY(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{otherwise} \end{cases}$$

(9.1)

We leave the proof of Theorem 9.5 as an exercise.

### 9.2.1 Rice’s Theorem

Theorem 9.5 can be generalized far beyond the parity function and in fact it rules out verifying any type of semantic specification on programs. Define a **semantic specification** on programs to be some property that does not depend on the code of the program but just on the function that the program computes. More formally, for a subset $S$ of the functions from $\{0,1\}^*$ to $\{0,1\}^*$, define the function $COMPUTES-S : \{0,1\}^* \rightarrow \{0,1\}$ as follows: if $P \in \{0,1\}^*$ is a description of a NAND++ program that computes some $F \in S$ then $COMPUTES-S(P) = 1$ and otherwise $COMPUTES-S(P) = 0$. The following theorem shows that $COMPUTES-S$ is either trivial or non-computable:

**Theorem 9.6 — Rice’s Theorem (slightly restricted version).** Say that $S$ as above is **trivial** if either there is no computable function $F \in S$ (and hence $COMPUTES-S$ is identically 0) or every computable function belongs to $S$ (and hence $COMPUTES-S$ is identically 1). If $S$ is not trivial then $COMPUTES-S$ is uncomputable.

The proof of Theorem 9.6 follows by generalizing the proof of Theorem 9.5 and we leave it to the reader as **Exercise 9.2**.
9.3 Restricted vs unrestricted computational models

We have mentioned before that many natural computational models turn out to be equivalent to one another, in the sense that we can transform a “program” of that other model (such as a λ expression, or a game-of-life configurations) into a NAND++ program. This equivalence implies that we can translate the uncomputability of the Halting problem for NAND++ programs into uncomputability for Halting in other models. For example:

**Theorem 9.7 — Turing Machine Halting.** Let \( TMHALT : \{0,1\}^* \rightarrow \{0,1\} \) be the function that on input strings \( M \in \{0,1\}^* \) and \( x \in \{0,1\}^* \) outputs 1 if the Turing machine described by \( M \) halts on the input \( x \) and outputs 0 otherwise. Then \( TMHALT \) is uncomputable.

**Proof.** We have seen that for every NAND++ program \( P \) there is an equivalent Turing machine \( M(P) \) such that for every \( x \), \( M(P) \) halts on \( x \) if and only \( P \) halts on \( x \) (and moreover if they both halt, they produce the same output). The transformation of \( P \) to \( M(P) \) was completely algorithmic and hence it can be thought of as a computable function \( M : \{0,1\}^* \rightarrow \{0,1\}^* \). We see that \( HALT(P, x) = TMHALT(M(P), x) \) and hence if we assume (towards the sake of a contradiction) that \( TMHALT \) is computable then we get that \( HALT \) is computable, hence contradicting Theorem 9.2. ■

The same proof carries over to other computational models such as the λ calculus, two dimensional (or even one-dimensional) automata etc. Hence for example, there is no algorithm to decide if a λ expression evaluates the identity function, and no algorithm to decide whether an initial configuration of the game of life will result in eventually coloring the cell \((0,0)\) black or not.

The uncomputability of halting and other semantic specification problems motivates coming up with restricted computational models that are (a) powerful enough to capture a set of functions useful for certain applications but (b) weak enough that we can still solve semantic specification problems on them. Here are some examples:

9.3.1 Regular expressions

A regular expression over some alphabet \( \Sigma \) is obtained by combining elements of \( \Sigma \) with the operations | (corresponding to or) and * (corresponding to repetition zero or more times). For example, the
following regular expression over the alphabet \( \{0, 1\} \) corresponds to the set of all even length strings \( x \in \{0, 1\}^* \) such that \( x_{2i} = x_{2i+1} \) for every \( i \in \{0, \ldots, |x|/2 - 1\} \):

\[
(0|1)^* 
\]

(9.2)

You have probably come across regular expressions in the context of searching for a file, doing search-and-replace in an editor, or text manipulations in programming languages or tools such as grep.\(^4\)

Formally, regular expressions are defined by the following recursive definition:\(^5\)

**Definition 9.1 — Regular expression.** A regular expression \( \text{exp} \) over an alphabet \( \Sigma \) is a string over \( \Sigma \cup \{ \text{"",""}, \text{",""}, \text{""}*"" \} \) that has one of the following forms: 1. \( \text{exp} = \sigma \) where \( \sigma \in \Sigma \)

2. \( \text{exp} = (\text{exp}'\text{exp}'') \) where \( \text{exp}', \text{exp}'' \) are regular expressions.

3. \( \text{exp} = (\text{exp}')(\text{exp}'') \) where \( \text{exp}', \text{exp}'' \) are regular expressions.

4. \( \text{exp} = (\text{exp}')* \) where \( \text{exp}' \) is a regular expression. \(^6\)

Every regular expression \( \text{exp} \) computes a function \( \Phi_{\text{exp}} : \Sigma^* \rightarrow \{0, 1\} \) defined as follows: 1. If \( \text{exp} = \sigma \) then \( \Phi_{\text{exp}}(x) = 1 \) iff \( x = \sigma \)

2. If \( \text{exp} = (\text{exp}'\text{exp}'') \) then \( \Phi_{\text{exp}}(x) = \Phi_{\text{exp}'}(x) \lor \Phi_{\text{exp}''}(x) \) where \( \lor \) is the OR operator.

3. If \( \text{exp} = (\text{exp}')(\text{exp}'') \) then \( \Phi_{\text{exp}}(x) = 1 \) iff there is some \( x', x'' \in \Sigma^* \) such that \( x \) is the concatenation of \( x' \) and \( x'' \) and \( \Phi_{\text{exp}'}(x') = \Phi_{\text{exp}''}(x'') = 1 \).

4. If \( \text{exp} = (\text{exp}')* \) then \( \Phi_{\text{exp}}(x) = 1 \) iff there are is \( k \in \mathbb{N} \) and some \( x_0, \ldots, x_{k-1} \in \Sigma^* \) such that \( x \) is the concatenation \( x_0 \cdots x_{k-1} \) and \( \Phi_{\text{exp}}(x_i) = 1 \) for every \( i \in [k] \).

\(^4\) Standard implementations of regular expressions typically include more "syntactic sugar" such as using short-hands such as \([a \rightarrow d] \) for \([a|b|c|d] \) and others, but they can all be implemented using the operators \(| \) and \(*\).

\(^5\) Just like recursive functions, we can define a concept recursively. Indeed, a definition of some class \( C \) of objects can be thought of as defining a function that maps an object \( o \) to either 0 or 1 depending on whether \( o \in C \). Thus we can think of the definition as defining a recursive function that maps a string \( \text{exp} \) over \( \Sigma \cup \{ \text{"",""}, \text{",""}, \text{""}*"" \} \) to 0 or 1 depending on whether \( \text{exp} \) describes a valid regular expression.

\(^6\) Many texts also allow regular expressions that accept no strings or only the empty string. In the interest of simplicity, we drop these "edge cases" from our definition, though it does not matter much.

By **Definition 9.1**, regular expressions can be thought of as a "programming language" that defines functions \( \text{exp} : \Sigma^* \rightarrow \{0, 1\} \). But it turns out that the "halting problem" for these functions is easy: they always halt.
**Theorem 9.8 — Regular expression always halt.** For every set $\Sigma$ and $exp \in (\Sigma \cup \{"","\",","\|","\ast\})^*$, if $exp$ is a valid regular expression over $\Sigma$ then $\Phi_{exp}$ is a total function from $\Sigma^*$ to $\{0, 1\}$. Moreover, there is an always halting NAND++ program $P_{exp}$ that computes $\Phi_{exp}$.

Proof. Definition 9.1 gives a way of recursively computing $\Phi_{exp}$. The key observation is that in our recursive definition of regular expressions, whenever $exp$ is made up of one or two expressions $exp', exp''$ then these two regular expressions are smaller than $exp$, and eventually (when they have size 1) then they must correspond to the non-recursive case of a single alphabet symbol.

Therefore, we can prove the theorem by induction over the length $m$ of $exp$. For $m = 1$, $exp$ is a single alphabet symbol and the function $\Phi_{exp}$ is trivial. In the general case, for $m = |exp|$ we assume by the induction hypothesis that we have proven the theorem for $|exp| = 1, \ldots, m - 1$. Then by the definition of regular expressions, $exp$ is made up of one or two sub-expressions $exp', exp''$ of length smaller than $m$, and hence by the induction hypothesis we assume that $\Phi_{exp'}$ and $\Phi_{exp''}$ are total computable functions. But then we can follow the definition for the cases of concatenation, union, or the star operator to compute $\Phi_{exp}$ using $\Phi_{exp'}$ and $\Phi_{exp''}$. ■

The proof of Theorem 9.8 gives a recursive algorithm to evaluate whether a given string matches or not a regular expression. However, it turns out that there is a much more efficient algorithm to match regular expressions, based on their connection to finite automata. We will discuss this other algorithm, and this connection, later in this course.

The fact that functions computed by regular expressions always halt is one of the reasons why they are so useful. When you make a regular expression search, you are guaranteed that you will get a result. This is why operating systems, for example, restrict you for searching a file via regular expressions and don’t allow searching by specifying an arbitrary function via a general-purpose programming language.

But this always-halting property comes at a cost. Regular expressions cannot compute every function that is computable by NAND++ programs. In fact there are some very simple (and useful!) functions that they cannot compute, such as the following:
Theorem 9.9 — Matching parenthesis. Let $\Sigma = \{\text{"\'',\''}\} \text{ and } MATCHPAREN : \Sigma^* \rightarrow \{0, 1\}$ be the function that given a string of parenthesis, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over $\Sigma$ that computes MATCHPAREN.

Proof. TO BE COMPLETED ■

9.3.2 Context free grammars.

Another example of uncomputable functions arises from the notions of grammars. The idea of a grammar is best illustrated by an example. Consider the set of all valid arithmetical expressions involving natural numbers, and the operators $+,-,\times,\div$. We can describe this set recursively as follows:

\[
\text{digit} := 0|1|2|3|4|5|6|7|8|9 \quad (9.3)
\]

\[
\text{number} := \text{digit|digitnumber} \quad (9.4)
\]

\[
\text{expression} := \text{number}|(\text{expression} + \text{expression})|(\text{expression} - \text{expression})|(\text{expression} \times \text{expression})|(\text{expression} \div \text{expression}) \quad (9.5)
\]

A valid expression is any string in $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, (,), +, -, \times, \div\}^*$ that can be obtained by repeatedly applying some of these rules to the initial symbol expression until we remain a string that does not contain number,digit or expression but only symbols in $\Sigma$ (and hence cannot be reduced further).

More generally, a grammar over an alphabet $\Sigma$ consists of a set of pairs of strings $(\alpha, \beta)$ where $\alpha, \beta \in (\Sigma \cup N)^*$ and $N$ is some finite set disjoint from $\Sigma$ containing a special symbol we call start. $\Sigma$ is known as the set of terminals and $N$ as the set of non terminals.

A grammar defines a subset $L \subseteq \Sigma^*$ (known as a language) as follows: $x$ is in $\Sigma$ if and only if there exists some finite sequence of rules such that if we start from the string start and each time replace a substring of the current string $\alpha$ with the corresponding righthand side of the rule $\beta$, then we eventually end up with $x$. 

\footnote{TODO: maybe add here more example of context-sensitive grammar and then a proof that grammars are undecidable if there is in fact a simple proof of this (maybe based on lambda calculus?).}
9.4 Uncountable sets

The diagonalization argument is useful outside the realm of computation. One of its most striking applications (one that preceded and inspired the results of Gödel and Turing discussed in this lecture and the next) is Cantor’s proof that there is a hierarchy of types of infinity. What can something like that mean?

We know that there are infinitely many natural numbers, and that there are also infinitely many even natural numbers. There seem to be “more” natural numbers than natural even numbers, but Cantor said that in fact the size of these two sets is the same. The reason is that there is a one-to-one map from the natural numbers \( \mathbb{N} \) to the set \( \text{EVEN} \) of even natural numbers, namely the map \( f(x) = 2x \).

Similarly, while we may think that there set \( \mathbb{N}^2 \) of pairs of natural numbers is larger than the set of natural numbers, it turns out that we can consider them as having the same size as well, since there is a one-to-one map \( \text{PAIR} : \mathbb{N}^2 \rightarrow \mathbb{N} \) (there are many ways to define such a map including \( \text{PAIR}(x, y) = 2^x3^y \); we’ve use such a map in embedding two-dimensional arrays in one-dimensional arrays).

Cantor defined two sets \( A \) and \( B \) to have the same cardinality if there is a one-to-one map \( f : A \rightarrow B \) and a one-to-one map \( g : B \rightarrow A \). One can verify that if \( A \) and \( B \) are finite then they have the same cardinality under this definition if and only if they have the same number of elements (can you see why?). Cantor denoted the cardinality of the set \( \mathbb{N} \) of natural numbers by \( \aleph_0 \) and sets with such cardinality are called countable.

A natural question is whether there is an uncountable infinite set. That is, a infinite set \( S \) that is bigger than the natural numbers, in the sense that there is no one-to-one function from \( S \) to \( \mathbb{N} \). It turns out that the answer is yes:

**Theorem 9.10 — Cantor’s Theorem.** The set \( \mathbb{R} \) of real numbers is uncountable.

**Proof of ??**. To prove the theorem we need to show that there is no one-to-one map from \( \mathbb{R} \) to \( \mathbb{N} \). In fact, we will show the stronger statement that there is no one-one-map from \( S \) to \( \mathbb{N} \) where \( S \) is some subset of \( \mathbb{R} \) (and even of the interval \( [0, 1] \)). We have already shown this, using different language, in Exercise 2.8. We will now repeat an outline of the argument.

We let \( S \) be the set of numbers in \( [0, 1] \) whose decimal expansion is \( x = 0.x_1x_2x_3 \cdots \) where \( x_i \in \{0, 1\} \). Note that every \( x \in S \) has
a unique decimal expansion involving only the numbers 0 and 1. (By restricting to $S$ we avoid technicalities such as the equivalence of $0.0999999\ldots$ and $0.1$.) Suppose for the sake of contradiction that there is a one-to-one map $f : S \to \mathbb{N}$. Then there is an onto map $g : \mathbb{N} \to S$. We can use diagonalization to obtain a contradiction by constructing a number $x^* \in S$ such that $g(n) \neq x^*$ for every $n \in \mathbb{N}$. Write the decimal expansion of $x^* = 0.x_1^* x_2^* x_3^* \ldots$. We will let $x_n^*$ equal 1 if the $n^{th}$ digit following the decimal point of $g(n)$ equals 0 and will define $x_n^* = 0$ otherwise. Now we can see that the $n^{th}$ digit of $g(n)$ differs from $x_n^*$ and hence $g(n) \neq x^*$ for every $n \in \mathbb{N}$, thus contradicting our assumption that $g$ is onto and completing the proof.

Cantor denoted the cardinality of the real numbers by $\mathfrak{c}$. It turns out that there are sets with even larger cardinality than $\mathfrak{c}$, and sets with even greater cardinality as well. These levels of infinity is sometimes known as transfinite ordinals. Cantor conjectured that there is no set with cardinality strictly between $\aleph_0$ and $\mathfrak{c}$. This is known as the Continuum Hypothesis and was presented by Hilbert in 1900 as the first in his list of the most important open questions in mathematics. In a twist of fate, using techniques originating from the works Gödel and Turing, Paul Cohen showed in 1963 that the Continuum Hypothesis is independent of the axioms of set theory, which means that neither it nor its negation is provable from these axioms and hence in some sense can be considered as “neither true nor false”.

9.5 Lecture summary

• Unlike the finite case, there are actually functions that are inherently uncomputable in the sense that they cannot be computed by any NAND++ program.

• These include not only some “degenerate” or “esoteric” functions but also functions that people have deeply cared about and conjectured that could be computed.

• If the Church-Turing thesis holds then a function $F$ that is uncomputable according to our definition cannot be computed by any finite means.
9.6 Exercises

Exercise 9.1 — Halting problem. Give an alternative, more direct, proof for the uncomputability of the Halting problem. Let us define
\[ H : \{0,1\}^* \rightarrow \{0,1\} \]
to be the function such that \( H(P) = 1 \) if, when we interpret \( P \) as a program, then \( H(P) \) equals 1 if \( P(P) \) halts (i.e., invoke \( P \) on its own string representation) and \( H(P) \) equals 0 otherwise. Prove that there no program \( P^* \) that computes \( H \), by building from such a supposed \( P^* \) a program \( Q \) such that, under the assumption that \( P^* \) computes \( H \), \( Q(Q) \) halts if and only if it does not halt.\(^8\)

Exercise 9.2 — Rice’s Theorem (slightly restricted form). 1. Generalize the result that COMPUTES-PARITY is uncomputable as follows. Prove that for every computable function \( F : \{0,1\}^* \rightarrow \{0,1\}^* \), the function COMPUTES-\( F \) which on input a NAND++ program \( P \), outputs 1 iff it holds that \( P(x) = F(x) \) for every \( x \), is uncomputable.

2. Generalize this even further to show that for every nontrivial (neither empty nor the entire set) subset \( S \) of the set \( \mathbb{R} \) of the computable functions from \( \{0,1\}^* \) to \( \{0,1\}^* \), the function COMPUTES-\( S \) that outputs 1 on a program \( P \) if and only if \( P \) computes some function in \( S \), is uncomputable.\(^9\)

\(^8\) Hint: See Christopher Strachey’s letter in the biographical notes.

\(^9\) Hint: Pick some \( F \in S \) and for every Turing machine \( Q \) and input \( x \), construct a machine \( P_{Q,x} \) that either computes \( F \) or computes nothing, based on whether \( Q \) halts on \( x \).

9.7 Bibliographical notes

9.8 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

9.9 Acknowledgements

\(^{10}\) TODO: Add letter of Christopher Strachey to the editor of The Computer Journal. Explain right order of historical achievements. Talk about intuitionistic, logicist, and formalist approaches for the foundations of mathematics. Perhaps analogy to veganism. State the full Rice’s Theorem and say that it follows from the same proof as in the exercise.
10

Is every theorem provable?

"Take any definite unsolved problem, such as... the existence of an infinite number of prime numbers of the form \(2^n + 1\). However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes..."

"...This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus.\”, David Hilbert, 1900.

10.1 Unsolvability of Diophantine equations

The problems of the previous lecture, while natural and important, still intimately involved NAND++ programs or other computing mechanisms in their definitions. One could perhaps hope that as long as we steer clear of functions whose inputs are themselves programs, we can avoid the “curse of uncomputability”. Alas, we have no such luck.

Many of the functions people wanted to compute over the years involved solving equations. These have a much longer history than mechanical computers. The Babilonians already knew how to solve some quadratic equations in 2000BC, and the formula for all quadratics appears in the Bakshali Manuscript that was composed in India around the 3rd century. During the Renaissance, Italian mathematicians discovered generalization of these formulas for cubic and
quartic (degrees 3 and 4) equations. Many of the greatest minds of the 17th and 18th century, including Euler, Lagrange, Leibnitz and Gauss worked on the problem of finding such a formula for quintic equations to no avail, until in the 19th century Ruffini, Abel and Galois showed that no such formula exists, along the way giving birth to group theory.

However, the fact that there is no closed-form formula does not mean we can not solve such equations. People have been solving higher degree equations numerically for ages. The Chinese manuscript Jiuzhang Suanshu from the first century mentions such approaches. Solving polynomial equations is by no means restricted only to ancient history or to students’ homeworks. The gradient descent method is the workhorse powering many of the machine learning tools that have revolutionized Computer Science over the last several years.

But there are some equations that we simply do not know how to solve by any means. For example, it took more than 200 years until people succeeded in proving that the equation \( a^{11} + b^{11} = c^{11} \) has no solution in integers.\(^1\) The notorious difficulty of so called Diophantine equations (i.e., finding integer roots of a polynomial) motivated the mathematician David Hilbert in 1900 to include the question of finding a general procedure for solving such equations in his famous list of twenty-three open problems for mathematics of the 20th century. I don’t think Hilbert doubted that such a procedure exists. After all, the whole history of mathematics up to this point involved the discovery of ever more powerful methods, and even impossibility results such as the inability to trisect an angle with a straightedge and compass, or the non-existence of an algebraic formula for quintic equations, merely pointed out to the need to use more general methods.

Alas, this turned out not to be the case for Diophantine equations: in 1970, Yuri Matiyasevich, building on a decades long line of work by Martin Davis, Hilary Putnam and Julia Robinson, showed that there is simply no method to solve such equations in general:

\[\textbf{Theorem 10.1 — MRDP Theorem.}\]

Let \( \text{SOLVE} : \{0,1\}^* \to \{0,1\}^* \) be the function that takes as input a multivariate polynomial with integer coefficients \( P : \mathbb{R}^k \to \mathbb{R} \) for \( k \leq 100 \) and outputs either \((x_1,\ldots,x_k) \in \mathbb{N}^k \) s.t. \( P(x_1,\ldots,x_k) = 0 \) or the string \textit{no solution} if no \( P \) does not have non-negative integer roots.\(^2\) Then \( \text{SOLVE} \) is uncomputable. Moreover, this holds even for the easier function \( \text{HASSOL} : \{0,1\}^* \to \{0,1\} \) that given such a polynomial \( P \) outputs

\(^1\) This is a special case of what’s known as “Fermat’s Last Theorem” which states that \( a^n + b^n = c^n \) has no solution in integers for \( n > 2 \). This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case \( n = 11 \) (along with all other so called “regular prime exponents”) was established by Kummer in 1850.

\(^2\) This is a special case of what’s known as “Fermat’s Last Theorem” which states that \( a^n + b^n = c^n \) has no solution in integers for \( n > 2 \). This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case \( n = 11 \) (along with all other so called “regular prime exponents”) was established by Kummer in 1850.
is every theorem provable?

As usual, we assume some standard way to express numbers and text as binary strings.

221

As usual, we assume some standard way to express numbers and text as binary strings.

2

As usual, we assume some standard way to express numbers and text as binary strings.

3

Recall that \( \exists \) denotes the existential quantifier; that is, a statement of the form \( \exists x \phi(x) \) is true if there is some assignment for \( x \) that makes the Boolean function \( \phi(x) \) true. The dual quantifier is the universal quantifier, denoted by \( \forall \), where a statement \( \forall x \phi(x) \) is true if every assignment for \( x \) makes the Boolean function \( \phi(x) \) true. Logical statements where all variables are bound to some quantifier (and hence have no parameters) can be either true or false, but determining which is the case is sometimes highly nontrivial. If you could use a review of quantifiers, section 3.6 of the text by Lehman, Leighton and Meyer is an excellent source for this material.

10.1.1 “Baby” MRDP Theorem: hardness of quantified Diophantine equations

Computing the function \( HASSOL \) is equivalent to determining the truth of a logical statement of the following form:

\[
\exists x_1, \ldots, x_k \in \mathbb{N} \text{ s.t. } P(x_1, \ldots, x_k) = 0.
\]

(10.1)

Theorem 10.1 states that there is no NAND++ program that can determine the truth of every statements of the form Eq. (10.1). The proof is highly involved and we will not see it here. Rather we will prove the following weaker result that there is no NAND++ program that can the truth of more general statements that mix together the existential (\( \exists \)) and universal (\( \forall \)) quantifiers. The reason this result is weaker than Theorem 10.1 is because deciding the truth of more general statements (that involve both quantifier) is a potentially harder problem than only existential statements, and so it is potentially easier to prove that this problem is uncomputable. (If you find the last sentence confusing, it is worthwhile to reread it until you are sure you follow its logic; we are so used to trying to find solution for problems that it can be quite confusing to follow the arguments showing that problems are uncomputable.)

Definition 10.1 — Quantified integer statements. A quantified integer statement is a well-formed statement with no unbound variables involving integers, variables, the operators \( >, <, \times, +, -, = \), the logical operations \( \neg \) (NOT), \( \land \) (AND), and \( \lor \) (OR), as well as quantifiers of the form \( \exists x \in \mathbb{N} \) and \( \forall y \in \mathbb{N} \) where \( x, y \) are variable names.

Definition 10.1 is interesting in its own right and not just as a “toy version” of Theorem 10.1. We often care deeply about determining the truth of quantified integer statements. For example, the statement

\[
1 \text{ if there are } x_1, \ldots, x_k \in \mathbb{N} \text{ s.t. } P(x_1, \ldots, x_k) = 0 \text{ and 0 otherwise.}
\]
that Fermat’s Last Theorem is true for \( n = 3 \) can be phrased as the quantified integer statement

\[
-\exists a \in \mathbb{N} \exists b \in \mathbb{N} \exists c \in \mathbb{N} a \times a + b \times b = c \times c \times c. \tag{10.2}
\]

The *twin prime hypothesis*, that states that there is an infinite number of numbers \( p \) such that both \( p \) and \( p + 2 \) are primes can be phrased as the quantified integer statement

\[
\forall n \in \mathbb{N} \exists p \in \mathbb{N} (p > n) \land \text{PRIME}(p) \land \text{PRIME}(p + 2) \tag{10.3}
\]

where we replace an instance of \( \text{PRIME}(q) \) with the statement \((q > 1) \land \forall a \in \mathbb{N} \forall b \in \mathbb{N} (a = 1) \lor (a = q) \lor - (a \times b = q)\).

The claim (mentioned in Hilbert’s quote above) that there are infinitely many *Fermat primes* of the form \( p = 2^n + 1 \) can be phrased as follows:

\[
\forall n \in \mathbb{N} \exists p \in \mathbb{N} (p > n) \land \text{PRIME}(p) \land \\
(\forall k \in \mathbb{N} (k = 2) \lor -\text{PRIME}(k) \lor -\text{DIVIDES}(k, p - 1)) \tag{10.4}
\]

where \( \text{DIVIDES}(a, b) \) is the statement \( \exists c \in \mathbb{N} b \times c = a \). In English, this corresponds to the claim that for every \( n \) there is some \( p > n \) such that all of \( p - 1 \)’s prime factors are equal to 2.

Much of number theory is concerned with determining the truth of quantified integer statements. Since our experience has been that, given enough time (which could sometimes be several centuries) humanity has managed to do so for the statements that she cared enough about, one could (as Hilbert did) hope that eventually we will discover a *general procedure* to determine the truth of such statements. The following theorem shows that this is not the case:

**Theorem 10.2 — Uncomputability of quantified integer statements.** Let \( \text{QIS} : \{0, 1\}^* \rightarrow \{0, 1\} \) be the function that given a (string representation of) a quantified integer statement outputs 1 if it is true and 0 if it is false. Then \( \text{QIS} \) is uncomputable.

Note that Theorem 10.2 is an immediate corollary of Theorem 10.1. Indeed, if you can compute \( \text{QIS} \) then you can compute \( \text{HASSOL} \) and hence if you can’t compute \( \text{HASSOL} \) then you can’t compute \( \text{QIS} \) either. But Theorem 10.2 is easier (though not trivial) to prove, and we will provide the proof in the following section.

\(^4\) Since a quantified integer statement is simply a sequence of symbols, we can easily represent it as a string. We will assume that *every* string represents some quantified integer statement, by mapping strings that do not correspond to such a statement to an arbitrary statement such as \( \exists x \in \mathbb{N} x = 1 \).
10.2 Proving the unsolvability of quantified integer statements.

In this section we will prove Theorem 10.2. The proof will, as usual, go by reduction from the Halting problem, but we will do so in two steps:

1. We will first use a reduction from the Halting problem to show that a deciding quantified mixed statements is uncomputable. Unquantified mixed statements involve both strings and integers.

2. We will then reduce the problem of quantified mixed statements to quantifier integer statements.

10.2.1 Quantified mixed statements and computation traces

As mentioned above, before proving Theorem 10.2, we will give an easier result showing the uncomputability of deciding the truth of an even more general class of statements- one that involves not just integer-valued variables but also string-valued ones.

**Definition 10.2 — Quantified mixed statements.** A quantified mixed statement is a well-formed statement with no unbound variables involving integers, variables, the operators $>,<\times,+,−,=$, the logical operations $\neg$ (NOT), $\land$ (AND), and $\lor$ (OR), as well as quantifiers of the form $\exists x \in \mathbb{N}, \exists a \in \{0,1\}^*, \forall y \in \mathbb{N}, \forall b \in \{0,1\}^*$, where $x, y, a, b$ are variable names. These also include the operator $|a|$ which returns the length of a string valued variable $a$, as well as the operator $a_i$ where $a$ is a string-valued variable and $i$ is an integer valued expression which is true if $i$ is smaller than the length of $a$ and the $i^{th}$ coordinate of $a$ is 1, and is false otherwise.

For example, the true statement that for every string $a$ there is a string $b$ that correspond to $a$ in reverse order can be phrased as the following quantified mixed statement

$$\forall a \in \{0,1\}^*, \exists b \in \{0,1\}^* (\forall i \in \mathbb{N} (|a| > |b|) \lor (a_i \land b_{|a|-i}) \lor (\neg a_i \land \neg b_{|a|-i})) .$$

(10.5)

Quantified mixed statements are a more general than quantified integer statements, and so the following theorem is potentially easier to prove than Theorem 10.2:
**Theorem 10.3 — Uncomputability of quantified mixed statements.** Let
\[ Q_{MS} : \{0,1\}^* \rightarrow \{0,1\} \]
be the function that given a (string representation of) a quantified mixed statement outputs 1 if it is true and 0 if it is false. Then \( Q_{MS} \) is uncomputable.

10.2.2 “Unraveling” NAND++ programs and quantified mixed integer statements

We will first prove Theorem 10.3 and then use it to prove Theorem 10.2. The proof is again by reduction to HALT (see Fig. 10.1). That is, we do so by giving a program that transforms any NAND++ program \( P \) and input \( x \) into a quantified mixed statement \( \phi_{P,x} \) such that \( \phi_{P,x} \) is true if and only if \( P \) halts on input \( x \). This will complete the proof, since it will imply that if \( Q_{MS} \) is computable then so is the HALT problem, which we have already shown is uncomputable.

![Figure 10.1](image-url)

Figure 10.1: We prove that \( Q_{MS} \) is uncomputable by giving a reduction that maps every pair \((P,x)\) into a quantified mixed statements \( \phi_{P,x} \) that is true if and only if \( P \) halts on \( x \).

The idea behind the construction of the statement \( \phi_{P,x} \) is the following. The statement will be true if and only if there exists a string \( L \in \{0,1\}^* \) which corresponds to an execution trace that proves that \( P \) halts on input \( x \). At a high level, the crucial insight is that unlike when we actually run the computation, to verify the correctness of a execution trace we only need to verify local consistency between pairs of lines.
We have seen execution traces before, but now we define them more precisely. Informally, an execution trace of a program $P$ on an input $x$ is a string that represents a “log” of all the lines executed and variables assigned in the course of the execution. For example, if we execute the parity program

```plaintext
tmp1 := seen_i NAND seen_i
tmp2 := x_i NAND notseen_i
val := tmp2 NAND notseen_i
ns := s NAND s
y_0 := ns NAND ns
u := val NAND s
v := s NAND u
w := val NAND u
s := v NAND w
seen_i := zero NAND zero
stop := validx_i NAND validx_i
loop := stop NAND stop
```

on the input $01$, the trace will be

... add trace here...

More formally, given a NAND++ program $P$ and an input $x \in \{0, 1\}^n$, if $P$ takes $T$ steps to halt on $x$, then the execution trace of $P$ on $x$ will be a string $L \in \{0, 1\}^{T+n}$ such that $(L_0, \ldots, L_{n-1}) = x$ and for every $i \in \{n, \ldots, n+T-1\}$, $L_i$ correspond to the value that is assigned to a variable in the $(i-n)^{th}$ step of $P$’s execution on $x$. Note that such an execution trace $L \in \{0, 1\}^{n+T}$ satisfies that for every $i \in \{n, \ldots, n+T-1\}$, $L_i$ is the NAND of $L_j$ and $L_k$ where $j$ and $k$ are the last lines in which the two variables referred to in the corresponding line are assigned a value.

We can compute $j$ and $k$ as an arithmetic function of $i$ as follows:

- If $P$ has $L$ lines then line $\ell$ of $P$ that is executed in the $j^{th}$ step is $j(\mod L)$ and the value of the program counter $pc$ is $\lfloor j/L \rfloor$.
- The value of the index variable $i$ at this point is $INDEX(pc)$ where $INDEX$ is the explicit function that we computed in Lecture 6.
- The variables that a referred to in the $\ell$-th line are computed by a constant size function since there is only a constant number of lines in $P$. 
10.2.3 Completing the proof

The idea of the reduction is that given a NAND++ program \( P \) and an input \( x \), we can come up with a mixed quantifier statement \( \Psi_{P,x}(L) \) such that for every \( L \in \{0,1\}^* \), \( \Psi_{P,x}(L) \) is true if and only if \( L \) is a consistent execution trace of \( P \) on input \( x \) that ends in a halting state (with the loop variable set to 0). The full details are rather tedious, but the crucial points are the following:

- We can come up with a quantified integer statement \( INDEX(t,i) \) that will be true if and only if the value of \( i \) when the program executes step \( t \) equals \( i \).

- We can come up with quantified integer statements \( PREV_1(t,s) \) and \( PREV_2(t,r) \) that will satisfy the following. If at step \( t \) the operation invoked is \( \text{foo} := \text{bar NAND baz} \) then \( PREV_1(t,s) \) is true if and only if \( s \) is the last step before \( t \) in which \( \text{bar} \) was written to and \( PREV_2(t,r) \) is true if and only if \( r \) is the last step before \( t \) in which \( \text{baz} \) was written to. Note that these statements will themselves use \( INDEX \) because if \( \text{bar} \) and/or \( \text{baz} \) are indexed by \( i \) then part of the condition for \( PREV_1(t,s) \) and \( PREV_2(t,r) \) will be to ensure that \( INDEX(t) = INDEX(s) \) and/or \( INDEX(t) = INDEX(r) \).

- We can come up with a quantified integer statement \( LOOP(t) \) that will be true if and only if the variable written to at step \( t \) in the execution is equal to \( \text{loop} \).

Given a construction of such a formula \( \Psi_{P,x}(L) \) we can see that \( HALT(P,x) = 1 \) if and only if the following quantified mixed statement \( \varphi_{P,x} \) is true

\[
\varphi_{P,x} = \exists L \in \{0,1\}^* . \Psi_{P,x}(L)
\]  

(10.6)

and hence we can write \( HALT(P,x) = QMS(\varphi_{P,x}) \). Since we can compute from \( P,x \) the statement \( \varphi_{P,x} \), we see that if \( QMS \) is computable then so would have been \( HALT \), yielding a proof by contradiction of Theorem 10.3.

10.2.4 Reducing mixed statements to integer statements

We now show how to prove Theorem 10.2 using Theorem 10.3. The idea is again a proof by reduction. We will show a transformation of every quantifier mixed statement \( \varphi \) into a quantified integer statement \( \xi \) that does not use string-valued variables such that \( \varphi \) is true if and only if \( \xi \) is true.
To remove string-valued variables from a statement, we encode them by integers. We will show that we can encode a string $x \in \{0,1\}^*$ by a pair of numbers $(X, n) \in \mathbb{N}$ s.t.

- $n = |x|$
- There is a quantified integer statement $\text{INDEX}(X, i)$ that for every $i < n$, will be true if $x_i = 1$ and will be false otherwise.

This will mean that we can replace a quantifier such as $\forall_{x \in \{0,1\}^*}$ with $\forall_X \forall_n \in \mathbb{N}$ (and similarly replace existential quantifiers over strings). We can later replace all calls to $|x|$ by $n$ and all calls to $x_i$ by $\text{INDEX}(X, i)$. Hence an encoding of the form above yields a proof of Theorem 10.2, since we can use it to map every mixed quantified statement $\varphi$ to quantified integer statement $\xi$ such that $\text{QMS}(\varphi) = \text{QIS}(\xi)$. Hence if $\text{QIS}$ was computable then $\text{QMS}$ would be computable as well, leading to a contradiction.

To achieve our encoding we use the following technical result:

**Lemma 10.4 — Constructible prime sequence.** There is a sequence of prime numbers $p_0 < p_1 < p_3 < \cdots$ such that there is a quantified integer statement $\text{PINDEX}(p, i)$ that is true if and only if $p = p_i$.

Using Lemma 10.4 we can encode a $x \in \{0,1\}^*$ by the numbers $(X, n)$ where $X = \prod_{i=1}^n p_i$ and $n = |x|$. We can then define the statement $\text{INDEX}(X, i)$ as

$$\forall_{p \in \mathbb{N}} \neg \text{PINDEX}(p, i) \lor \text{DIVIDES}(p, X) \quad (10.7)$$

where $\text{DIVIDES}(a, b)$, as before, is defined as $\exists_c \in \mathbb{N} a \times c = b$. Note that indeed if $X, n$ encodes the string $x \in \{0,1\}^*$, then for every $i < n$, $\text{INDEX}(X, i) = x_i$, since $p_i$ divides $X$ if and only if $x_i = 1$.

Thus all that is left to conclude the proof of Theorem 10.2 is to prove Lemma 10.4, which we now proceed to do.

**Proof.** The sequence of prime numbers we consider is the following: we define $p_i$ to be the smallest prime number that is in the interval $[i + 1, 2i + 1]$. It is known by Bertrand’s postulate that there exists such a prime number for every $i \in \mathbb{N}$. Given this, the definition of $\text{PINDEX}(p, i)$ is simple:

$$(p > i) \land (p < 2 \times i + 1) \land \left( \forall_{p'} \neg \text{PRIME}(p') \lor (p' \leq i) \lor (p' \geq p) \right) \quad (10.8)$$

We leave it to the reader to verify that $\text{PINDEX}(p, i)$ is true iff $p = p_i$. ■
10.3 Hilbert’s Program and Gödel’s Incompleteness Theorem

“And what are these . . . vanishing increments? They are neither finite quantities, nor quantities infinitely small, nor yet nothing. May we not call them the ghosts of departed quantities?”, George Berkeley, Bishop of Cloyne, 1734.

The 1700’s and 1800’s were a time of great discoveries in mathematics but also of several crises. The discovery of calculus by Newton and Leibnitz in the late 1600’s ushered a golden age of problem solving. Many longstanding challenges succumbed to the new tools that were discovered, and mathematicians got ever better at doing some truly impressive calculations. However, the rigorous foundations behind these calculations left much to be desired. Mathematicians manipulated infinitesimal quantities and infinite series cavalierly, and while most of the time they ended up with the correct results, there were a few strange examples (such as trying to calculate the value of the infinite series \[1 - 1 + 1 - 1 + 1 + \ldots\]) which seemed to give out different answers depending on the method of calculation. This led to a growing sense of unease in the foundations of the subject which was addressed in works of mathematicians such as Cauchy, Weierstrass, and Reimann, who eventually placed analysis on firmer foundations, giving rise to the \(\varepsilon\)’s and \(\delta\)’s that students taking honors calculus grapple with to this day.

In the beginning of the 20th century, there was an effort to replicate this effort, in greater rigor, to all parts of mathematics. The hope was to show that all the true results of mathematics can be obtained by starting with a number of axioms, and deriving theorems from them using logical rules of inference. This effort was known as the Hilbert program, named after the very same David Hilbert we mentioned above. Alas, Theorem 10.1 yields a devastating blow to this program, as it implies that for any valid set of axioms and inference laws, there will be unsatisfiable Diophantine equations that cannot be proven unsatisfiable using these axioms and laws. To formalize that, we make the following definition:

**Definition 10.3 — Proof systems for diophantine equations.** A proof system for Diophantine equations is defined by a finite subset \(A \subseteq \{0, 1\}^*\) of axioms and a finite set of functions \(I_1, \ldots, I_m\) (known as inference rules) where each \(I_j : (\{0, 1\}^*)^{k_j} \rightarrow \{0, 1\}^*\) is a function mapping a tuple of \(k_j\) strings to a string.
A valid proof in the system $(A, I_1, \ldots, I_m)$ of the unsatisfiability of a diophantine equation $P(x_1, \ldots, x_t) = 0$ consists of a sequence $p_1, \ldots, p_n$ of strings such that for every $i$, either $p_i \in A$ or there exists $j \in [m]$ and $i_1, \ldots, i_j < i$ such that $p_i = I_j(p_{i_1}, \ldots, p_{i_j})$ and $p_n$ is a string representing the statement “$\forall x_1, \ldots, x_t \in \mathbb{N} P(x_1, \ldots, x_t) \neq 0$.”

A proof system $(A, I_1, \ldots, I_m)$ is sound if there is no valid proof of a false statement. That is, for every diophantine equation $P(x_1, \ldots, x_t) = 0$, if there is a proof $(p_1, \ldots, p_n)$ that the equation is unsatisfiable then it is indeed unsatisfiable.

The formal definition is a bit of a mouthful, but what it states the natural notion of a logical proof for the unsatisfiability of an equation. Namely, that such a proof will consist of $n$ lines, where each line is either an axiom or is derived from the previous lines by some rule of inference. We do not make any restriction on what the axioms or rules should be, except that they should not allow us to prove false statements. Hilbert believed that for all of mathematics, and in particular for settling diophantine equations, it should be possible to find some set of axioms and rules of inference that would allow to derive all true statements. However, he was wrong:

**Theorem 10.5 — Gödel’s Incompleteness Theorem.** For every valid proof system $(A, I_1, \ldots, I_m)$, there exists a diophantine equation $P(x_1, \ldots, x_t) = 0$ such that there is no $x_1, \ldots, x_t \in \mathbb{N}$ that satisfy it, but yet there is no proof in the system $(A, I_1, \ldots, I_m)$ for the statement “$\forall x_1 \in \mathbb{N} \cdots \forall x_t \in \mathbb{N} P(x_1, \ldots, x_t) \neq 0$.”

**Proof.** Suppose otherwise, that there exists such a system. Then we can define the following algorithm $S$ that computes the function $HASSOL : \{0, 1\}^* \rightarrow \{0, 1\}$ described in Theorem 10.1. The algorithm will work as follows:

- On input a Diophantine equation $P(x_1, \ldots, x_t) = 0$, for $k = 1, 2, \ldots$ do the following:

  1. Check for all $x_1, \ldots, x_t \in \{0, \ldots, k\}$ whether $x_1, \ldots, x_t$ satisfies the equation. If so then halt and output 1.

  2. For all $n \in \{1, \ldots, k\}$ and all strings $p_1, \ldots, p_n$ of length at most $k$, check whether $(p_1, \ldots, p_n)$ is a valid proof of “$\forall x_1 \in \mathbb{N} \cdots \forall x_t \in \mathbb{N} P(x_1, \ldots, x_t) \neq 0$”. If so then halt and output 0.

Note that checking if a list $(p_1, \ldots, p_n)$ is a valid proof can be done in finite time since there is only a finite number of axioms and inference rules. Under the assumption that for every diophantine equation...
that is unsatisfiable, there is a proof that certifies it, this algorithm will always halt and output 0 or 1, and moreover, the answer will be correct. Hence we reach a contradiction to Theorem 10.1.

Note that if we considered proof systems for more general quantified integer statements, then the existence of a true but yet unprovable statement would follow from Theorem 10.2. Indeed, that was the content of Gödel’s original incompleteness theorem which was proven in 1931 way before the MRDP Theorem (and initiated the line of research which resulted in the latter theorem). Another way to state the result is that every proof system that is rich enough to express quantified integer statements is either inconsistent (can prove both a statement and its negation) or incomplete (cannot prove all true statements).

Examining the proof of Theorem 10.5 shows that it yields a more general statement (see Exercise 10.1): for every uncomputable function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ and every sound axiomatic proof system $S$ (that is characterized by a finite number of axioms and inference rules), there is some input $x$ for which the proof system $S$ is not able to prove neither that $F(x) = 0$ nor that $F(x) \neq 0$ (see Exercise 10.1).

Also, the proof of Theorem 10.5 can be extended to yield Gödel’s second incompleteness theorem which, informally speaking, says for that every proof system $S$ rich enough to express quantified integer statements, the following holds:

- There is a quantified integer statement $\varphi$ that is true if and only if $S$ is consistent.

- There is no proof in $S$ for $\varphi$.

Thus once we pass a sufficient level of expressiveness, we cannot find a proof system that is strong enough to prove its own consistency. This in particular showed that Hilbert’s second problem (which was about finding an axiomatic provably-consistent basis for arithmetic) was also unsolvable.

10.4 Lecture summary

- Uncomputable functions include also functions that seem to have nothing to do with NAND++ programs or other computational models such as determining the satisfiability of diophantine equations.
• This also implies that for any finite axiomatic system $S$, there are interesting statements $X$ (namely of the form “$F(x) = 0$” for an uncomputable function $F$) such that $S$ is not able to prove either $X$ or its negation.

10.5 Exercises

Exercise 10.1 — title. For every representation of logical statements as strings, we can define as in Definition 10.3 an axiomatic proof system to consist of a finite set of strings $A$ and a finite set of rules $I_0, \ldots, I_{m-1}$ with $I_j : (\{0,1\}^*)^k_i \rightarrow \{0,1\}^*$ such that a proof $(s_1, \ldots, s_n)$ that $s_n$ is true is valid if for every $i$, either $s_i \in A$ or is some $j \in [m]$ and are $i_1, \ldots, i_k < i$ such that $s_i = I_j(s_{i_1}, \ldots, s_{i_k})$. A system is sound if whenever there is no false $s$ such that there is a proof that $s$ is true, Prove that for every uncomputable function $F : \{0,1\}^* \rightarrow \{0,1\}$ and every sound axiomatic proof system $S$ (that is characterized by a finite number of axioms and inference rules), there is some input $x$ for which the proof system $S$ is not able to prove neither that $F(x) = 0$ nor that $F(x) \neq 0$. ■

10.6 Bibliographical notes

6

10.7 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

10.8 Acknowledgements

Thanks to Alex Lombardi for pointing out an embarrassing mistake in the description of Fermat’s Last Theorem. (I said that it was open for exponent 11 was before Wiles’ work.)
Efficient computation

“The problem of distinguishing prime numbers from composite and of resolving the latter into their prime factors is ... one of the most important and useful in arithmetic ... Nevertheless we must confess that all methods ... are either restricted to very special cases or are so laborious ... they try the patience of even the practiced calculator ... and do not apply at all to larger numbers.”, Carl Friedrich Gauss, 1798

“For practical purposes, the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite.”, Jack Edmunds, “Paths, Trees, and Flowers”, 1963

“Sad to say, but it will be many more years, if ever before we really understand the Mystical Power of Tiveness ... 2-SAT is easy, 3-SAT is hard, 2-dimensional matching is easy, 3-dimensional matching is hard. Why? oh, Why?” Eugene Lawler

So far we were concerned with which functions are computable and which ones are not. But now we return to quantitative considerations and study the time that it takes to compute functions mapping strings to strings, as a function of the input length. One of the interesting phenomena of computing is that there is often a kind of “zero one law” for running time, where for many natural problems, they either can be solved in polynomial running time (e.g., something like \(O(n^2)\) or \(O(n^3)\)), or require exponential (e.g., at least \(2^{\Omega(n)}\) or \(2^{\Omega(\sqrt{n})}\)) running time. The reasons for this phenomenon are still not fully understood, but some light on this is shed by the concept of \(NP\)
An equivalent viewpoint is that an undirected graph is like a directed graph with the property that whenever the edge $\overrightarrow{u v}$ is present then so is the edge $\overrightarrow{v u}$.

In an undirected graph, the adjacency matrix $A$ is symmetric, in the sense that $A_{i,j} = A_{j,i}$.\footnote{An equivalent viewpoint is that an undirected graph is like a directed graph with the property that whenever the edge $\overrightarrow{u v}$ is present then so is the edge $\overrightarrow{v u}$.}

Completeness, which we will encounter later.

In this lecture we will survey some examples of computational problems, for some of which we know efficient (e.g., $n^c$-time for a small constant $c$) algorithms, and for others the best known algorithms are exponential. We want to get a feel as to the kinds of problems that lie on each side of this divide and also see how some seemingly minor changes in formulation can make the (known) complexity of a problem “jump” from polynomial to exponential.

In this lecture, we will not formally define the notion of running time, and so use the same notion of an $O(n)$ or $O(n^2)$ time algorithms as the one you’ve seen in an intro to CS course: “I know it when I see it”. In the next lecture, we will define this notion precisely, using our NAND++ and NAND« programming languages. One of the nice things about the theory of computation is that it turns out that, like in the context of computability, the details of the precise computational model or programming language don’t matter that much, especially if you mostly care about the distinction between polynomial and exponential time.

11.1 Some computational problems

Let us start with a few examples of computational problems that we might want to solve. Many of the problems will involve graphs. We have already encountered graphs in the context of Boolean circuits, but let us now quickly recall the basic notation. A graph $G$ consists of a set of vertices $V$ and edges $E$ where each edge is a pair of vertices. In a directed graph, an edge is an ordered pair $(u,v)$, which we sometimes denote as $\overrightarrow{u v}$. In an undirected graph, an edge is an unordered pair (or simply a set) $\{u,v\}$.\footnote{In an undirected graph, the adjacency matrix $A$ is symmetric, in the sense that it satisfies $A_{i,j} = A_{j,i}$.} We will assume graphs are undirected and simple (i.e., containing no parallel edges or self-loops) unless stated otherwise. We typically will think of the set of vertices as simply the set $[n]$ of the numbers from 0 till $n-1$. Graphs can be represented either in the adjacency list representation, which is a list of $n$ lists, with the $i^{th}$ list corresponding to the neighbors of the $i^{th}$ vertex, or the adjacency matrix representation, which is an $n \times n$ matrix $A$ with $A_{i,j}$ equalling 1 if the edge $\overrightarrow{u v}$ is present and equalling 0 otherwise.\footnote{An equivalent viewpoint is that an undirected graph is like a directed graph with the property that whenever the edge $\overrightarrow{u v}$ is present then so is the edge $\overrightarrow{v u}$.} We can transform between these two representations using $O(n^2)$ operations, and hence for our purposes we will mostly consider them as equivalent. We will sometimes consider labeled or weighted graphs, where we assign a label or a number to the edges or vertices of the graph, but mostly we will try to keep things simple and stick to the basic notion of an unlabeled, unweighted, simple
There is a reason that graphs are so ubiquitous in computer science and other sciences. They can be used to model a great many of the data that we encounter. These are not just the “obvious” networks such as the road network (which can be thought of as a graph of whose vertices are locations with edges corresponding to road segments), or the web (which can be thought of as a graph whose vertices are web pages with edges corresponding to links), or social networks (which can be thought of as a graph whose vertices are people and the edges correspond to friend relation). Graphs can also denote correlations in data (e.g., graph of observations of features with edges corresponding to features that tend to appear together), casual relations (e.g., gene regulatory networks, where a gene is connected to gene products it derives), or the state space of a system (e.g., graph of configurations of a physical system, with edges corresponding to states that can be reached from one another in one step).

Figure 11.1: Some examples of graphs found from the Internet.

We now give some examples of computational problems on graphs. As mentioned above, to keep things simple, we will restrict attention to undirected simple graphs. In all cases the input graph $G = (V, E)$ will have $n$ vertices and $m$ edges. To avoid trivialities, we will always assume the graph is connected (every two vertices can be reached from one another by traversing edges), which in particular means that $m \geq n - 1$ (can you see why?).
11.1.1 Finding the shortest path in a graph

The shortest path problem is the task of, given a graph \( G = (V, E) \) and two vertices \( s, t \in V \), to find the length of the shortest path between \( s \) and \( t \). That is, we want to find the smallest number \( k \) such that there are vertices \( v_0, v_1, \ldots, v_k \) with \( v_0 = s, v_k = t \) and for every \( i \in \{0, \ldots, k-1\} \) an edge between \( v_i \) and \( v_{i+1} \). If each vertex has at least two neighbors then there can be an exponential number of paths from \( s \) to \( t \), but fortunately we do not have to enumerate them all to find the shortest path. We can do so by performing a breadth first search (BFS), enumerating \( s \)'s neighbors, and then neighbors' neighbors, etc. in order. If we maintain the neighbors in a list we can perform a BFS in \( O(n^2) \) time, while using a queue we can do this in \( O(m) \) time.\(^3\)

11.1.2 Finding the longest path in a graph

The longest path problem is the task of, given a graph \( G = (V, E) \) and two vertices \( s, t \in V \), to find the length of the longest simple (i.e., non-intersecting) path between \( s \) and \( t \). While a priori this may seem less motivated than the shortest path, it is a natural optimization problem related to the Hamiltonian path problem which asks for a maximally long simple path (i.e., path that visits all \( n \) vertices once) between \( s \) and \( t \), as well as the notorious traveling salesman problem (TSP) of finding (in a weighted graph) a maximally long path of cost at most \( w \). TSP is a classical optimization problem, with applications ranging from planning and logistics to DNA sequencing and astronomy.

A priori it is not clear that finding the longest path should be harder than finding the shortest path, but this turns out to be the case. While we know how to find the shortest path in \( O(n) \) time, for the longest path problem we have not been able to significantly improve upon the trivial brute force algorithm to try all paths, and the best algorithm takes \( \Omega(c^n) \) time for some constant \( c \) which is a little smaller than 2 but not that much smaller.\(^4\)

11.1.3 Finding the minimum cut in a graph

Given a graph \( G = (V, E) \), a cut is simply a subset \( S \) of \( V \), and the edges cut by \( S \) are those edges where one of their endpoints is in \( S \) and the other is in \( \overline{S} = V \setminus S \). If \( s, t \in V \) then an \( s, t \) cut is a cut such that \( s \in S \) and \( t \in \overline{S} \). The minimum \( s, t \) cut problem is the task of

\(^3\) Since we assume \( m \geq n - 1 \), \( O(m) \) is the same as \( O(n + m) \). As we mentioned above, we state the running time of BFS as \( O(m) \) even though it is typically thought of as an \( O(m) \) time algorithm, to avoid dependence on the model. Dijkstra’s algorithm is a well-known generalization of BFS to weighted graphs.

\(^4\) At the moment the best record is \( c \sim 1.65 \) or so. Even beating the trivial \( O(n!) \) bound is not that simple, see Exercise 11.1.
finding, given \( s \) and \( t \), the minimum number \( k \) such that there is an \( s,t \) cut cutting \( k \) edges.\(^5\)

The minimum \( s,t \) cut problem appears in many applications. Minimum cuts often correspond to *bottlenecks*. For example, in a communication network the minimum cut between \( s \) and \( t \) corresponds to the smallest number of edges that, if dropped, will disconnect \( s \) from \( t \). Similar applications arise in scheduling and planning. There are also other applications as well: for *image segmentation*, we can define a graph whose vertices are pixels and whose edges correspond to neighboring pixels of distinct colors. If we want to separate the foreground from the background then we can pick (or guess) a foreground pixel \( s \) and background pixel \( t \) and ask for a minimum cut between them.

Once again, a priori we might worry that we’d need to enumerate over all the \( 2^n \) subsets of \( S \). Fortunately, we can solve the minimum \( s,t \) cut problem efficiently. There are several algorithms to do so, but many of them rely on the *Max Flow Min Cut* that says that the minimum cut between \( s \) and \( t \) equals the maximum amount of flow we can send from \( s \) to \( t \), if every edge has unit capacity. For example, this directly implies that the value of the minimum cut problem is the

\(^5\) We can also define the problem of finding the minimum \( s,t \) cut in the graph over all pairs \( s,t \). Though note that if we can solve the minimum \( s,t \) cut problem in time \( T(n) \) then we can solve the global minimum cut in time \( O(T(n)n^2) \).
solution for the following linear program:

\[
\begin{align*}
\max_{x \in \mathbb{R}^m} & \quad F_s(x) - F_t(x) \\
\text{s.t.} & \quad \forall u \in \{s, t\}, F_u(x) = 0
\end{align*}
\]  

(11.1)

where for every vertex \( u \) and \( x \in \mathbb{R}^m \),

\[ F_u(x) = \sum_{e \in E, u \in e} x_e. \]

Since there is a polynomial-time algorithm for linear programming, the minimum cut (or, equivalently, maximum flow) problem can be solved in polynomial time. In fact, there are much better algorithms for this problem, with currently the record standing at \( O(\min\{m^{10/7}, m\sqrt{n}\}) \).  

\[ \text{TODO: add references in bibliographic notes: Madry, Lee-Sidford} \]

11.1.4 Finding the maximum cut in a graph

We can also define the maximum cut problem of finding, given a graph \( G = (V, E) \) the subset \( S \subseteq V \) that maximizes the number of edges cut by \( S \). Like its cousin the minimum cut problem, the maximum cut problem is also very well motivated. For example, it arises in VLSI design, and also has some surprising relation to analyzing the Ising model in statistical physics.

Once again, a priori it might not be clear that the maximum cut problem should be harder than minimum cut but this turns out to be the case. We do not know of an algorithm that solves this problem much faster than the trivial “brute force” algorithm that tries all \( 2^n \) possibilities for the set \( S \).

11.1.5 A note on convexity

![Figure 11.3](image)

\textbf{Figure 11.3}: In a convex function (right figure) the global minimum is the only local minimum, and we can find it by a local-search algorithm which can be thought of as dropping a marble and letting it “slide down” until it reaches the global minimum. In contrast, a non-convex function (right figure) might have an exponential number of local minima in which any local-search algorithm could get stuck.

There is an underlying reason for the sometimes radical difference between the difficulty of maximizing and minimizing a function over
a domain. If $D \subseteq \mathbb{R}^n$, then a function $f : D \to \mathbb{R}$ is convex if for every $x, y \in D$ and $p \in [0, 1]$ 
$f(px + (1 - p)y) \leq pf(x) + (1 - p)f(y)$.

That is, $f$ applied to the $p$-weighted midpoint between $x$ and $y$ is smaller than the $p$-weighted average value of $f$. If $D$ itself is convex (which means that if $x, y$ are in $D$ then so is the line segment between them), then this means that if $x$ is a local minimum of $f$ then it is also a global minimum. The reason is that if $f(y) < f(x)$ then every point $z = px + (1 - p)y$ on the line segment between $x$ and $y$ will satisfy $f(z) \leq pf(x) + (1 - p)f(y) < f(x)$ and hence in particular $x$ cannot be a local minimum. Intuitively, local minima of functions are much easier to find than global ones: after all, any “local search” algorithm that keeps finding a nearby point on which the value is lower, will eventually arrive at a local minima.\(^8\) Indeed, under certain technical conditions, we can often efficiently find the minimum of convex functions, and this underlies the reason problems such as minimum cut and shortest path are easy to solve. On the other hand, maximizing a convex function (or equivalently, minimizing a concave function) can often be a hard computational task.

The minimum cut problem is not a priori a convex minimization task, because the set of potential cuts is discrete. However, it turns out that we can embed it in a continuous and convex set via the maximum flow problem. The “max flow min cut” theorem ensuring that this embedding is “tight” in the sense that the minimum “fractional cut” that we obtain through the maximum-flow linear program will be the same as the true minimum cut. Unfortunately, we don’t know of such a tight embedding in the setting of the maximum cut problem.

The issue of convexity arises time and again in the context of computation. For example, one of the basic tasks in machine learning is empirical risk minimization. That is, given a set of labeled examples $(x_1, y_1), \ldots, (x_m, y_m)$, where each $x_i \in \{0, 1\}^n$ and $y_i \in \{0, 1\}$, we want to find the function $h : \{0, 1\}^n \to \{0, 1\}$ from some class $H$ that minimizes the error in the sense of minimizing the number of $i$’s such that $h(x_i) \neq y_i$. Like in the minimum cut problem, to make this a better behaved computational problem, we often embed it in a continuous domain, including functions that could output a real number and replacing the condition $h(x_i) \neq y_i$ with minimizing some continuous loss function $\ell(h(x_i), y_i)$.\(^9\) When this embedding is convex then we are guaranteed that the global minimizer is unique and can be found in polynomial time. When the embedding is non convex, we have no such guarantee and in general there can be many global or local minima. That said, even if we don’t find the global (or even a local) minima, this continuous embedding can still help us. In particular, when running a local improvement algorithm such as

\(^8\) One example of such a local search algorithm is gradient descent which takes a small step in the direction that would reduce the value by the most amount based on the current derivative. There are also algorithms that take advantage of the second derivative (hence are known as second order methods) to potentially converge faster.

\(^9\) We also sometimes replace or enhance the condition that $h$ is in the class $H$ by adding a regularizing term of the form $R(h)$ to the minimization problem, where $R : H \to \mathbb{R}$ is some measure of the “complexity” of $h$. As a general rule, the larger or more “complex” functions $h$ we allow, the easier it is to fit the data, but the more danger we have of “overfitting”.
Gradient Descent, we might still find a function \( h \) that is “useful” in the sense of having a small error on future examples from the same distribution.\(^{10} \)

11.1.6 The 2SAT problem

Not all computational problems arise from graphs. A propositional formula \( \varphi \) involves \( n \) variables \( x_1, \ldots, x_n \) and the logical operators \( \text{AND} (\wedge) \), \( \text{OR} (\vee) \), and \( \text{NOT} (\neg) \) (also denoted as \( \bar{\cdot} \)). We say that such a formula is in conjunctive normal form (CNF for short) if it is an OR of ANDs of variables or their negations (we call a term of the form \( x_i \) or \( \bar{x}_i \) a literal). For example, this is a CNF formula

\[
(x_7 \vee \bar{x}_{22} \vee x_{15}) \wedge (x_{37} \vee x_{22}) \wedge (x_{55} \vee \bar{x}_7)
\]

(11.2)

We say that a formula is a \( k \)-CNF if it is an AND or ORs where each OR involves exactly \( k \) literals. The 2SAT problem is to find out, given a 2-CNF formula \( \varphi \), whether there is an assignment \( x \in \{0, 1\}^n \) that satisfies \( \varphi \), in the sense that it makes it evaluate to 1 or “True”.

Determining the satisfiability of Boolean formulas arises in many applications and in particular in software and hardware verification, as well as scheduling problems. The trivial, brute-force, algorithm for 2SAT will enumerate all the \( 2^n \) assignments \( x \in \{0, 1\}^n \) but fortunately we can do much better.

The key is that we can think of every constraint of the form \( \ell_i \vee \ell_j \) (where \( \ell_i, \ell_j \) are literals, corresponding to variables or their negations) as an implication \( \bar{\ell}_i \Rightarrow \ell_j \), since it corresponds to the constraints that if the literal \( \ell_i = \bar{\ell}_i \) is true then it must be the case that \( \ell_j \) is true as well. Hence we can think of \( \varphi \) as a directed graph between the \( 2n \) literals, with an edge from \( \ell_i \) to \( \ell_j \) corresponding to an implication from the former to the latter. It can be shown that \( \varphi \) is unsatisfiable if and only if there is a variable \( x_i \) such that there is a directed path from \( x_i \) to \( \bar{x}_i \) as well as a directed path from \( \bar{x}_i \) to \( x_i \) (see Exercise 11.2). This reduces 2SAT to the (efficiently solvable) problem of determining connectivity in directed graphs.

11.1.7 The 3SAT problem

The 3SAT problem is the task of determining satisfiability for 3CNFs. One might think that the difference between two and three would not make that much of a difference for complexity. One would be wrong. Despite much effort, do not know of a significantly better than brute
force algorithm for 3SAT (the best known algorithms take roughly $1.3^n$ steps).

Interestingly, a similar issue arises time and again in computation, where the difference between two and three often corresponds to the difference between tractable and intractable. As Lawler’s quote alludes to, we do not fully understand the reasons for this phenomenon, though the notions of NP completeness we will see in the next lecture does offer a partial explanation. It may be related to the fact that optimizing a polynomial often amounts to equations on its derivative. The derivative of a quadratic polynomial is linear, while the derivative of a cubic is quadratic, and, as we will see, the difference between solving linear and quadratic equations can be quite profound.

11.1.8 Solving linear equations

One of the most useful problems that people have been solving time and again is solving $n$ linear equations in $n$ variables. That is, solve equations of the form

$$
\begin{align*}
    a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} &= b_0 \\
    a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} &= b_1 \\
    \vdots & \quad \vdots \\
    a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} &= b_{n-1}
\end{align*}
$$

where $\{a_{ij}\}_{i,j \in [n]}$ and $\{b_i\}_{i \in [n]}$ are real (or rational) numbers. More compactly, we can write this as the equations $Ax = b$ where $A$ is an $n \times n$ matrix, and we think of $x, b$ are column vectors in $\mathbb{R}^n$.

The standard Gaussian elimination algorithm can be used to solve such equations in polynomial time (i.e., determine if they have a solution, and if so, to find it).\textsuperscript{11} As we discussed above, if we are willing to allow some loss in precision, we even have algorithms that handle linear inequalities, also known as linear programming.

11.1.9 Solving quadratic equations

Suppose that the equations involve also quadratic terms of the form $a_{ij}x_j x_k$. That is, suppose that we are given a set of quadratic polynomials $p_1, \ldots, p_m$ and consider the equations $\{p_j(x) = 0\}$. To avoid

\textsuperscript{11} To analyze this fully we need to ensure that the bit complexity of the numbers involved does not grow too much, but fortunately we can indeed ensure this using Cramer’s rule. Also, as is usually the case when talking about real numbers, we do not care much for the distinction between solving equations exactly and solving them to arbitrarily good precision.
issues with bit representations, we will always assume that the equations contain the constraints \( \{ x_i^2 - x_i = 0 \}_{i \in [n]} \) and hence the solutions can be assumed to lie in \( \{0,1\}^n \). This is a very well motivated problem which in particular generalizes the classical quadratic assignment problem. Once again, we do not know a much better algorithm for this problem than the one that enumerates over all the \( 2^n \) possibilities.

11.1.10 The permanent (mod 2) problem

Given an \( n \times n \) matrix \( A \), the permanent of \( A \) is the sum over all permutations \( \pi \) (i.e., \( \pi \) is a member of the set \( S_n \) of one-to-one and onto functions from \( [n] \) to \( [n] \)) of the product \( \prod_{i=0}^{n-1} A_{i, \pi(i)} \). The permanent of a matrix is a natural quantity, and has been studied in several contexts including combinatorics and graph theory. It also arises in physics where it can be used to describe the quantum state of multiple boson particles (see here and here).

If the entries of \( A \) are integers, then we can also define a Boolean function \( \text{perm}_2(A) \) which will output the result of the permanent modulo 2. A priori computing this would seem to require enumerating over all \( n! \) possibilities. However, it turns out we can compute \( \text{perm}_2(A) \) in polynomial time! The key is that modulo 2, \(-x\) and \(+x\) are the same quantity and hence the permanent modulo 2 is the same as taking the following quantity modulo 2:

\[
\sum_{\pi \in S_n} \text{sign}(\pi) \prod_{i=0}^{n-1} A_{i, \pi(i)} \quad (11.4)
\]

where the sign of a permutation \( \pi \) is a number in \( \{ +1, -1 \} \) which can be defined in several ways, one of which is that \( \text{sign}(\pi) \) equals \(+1\) if the number of swaps that “Bubble” sort performs starting an array sorted according to \( \pi \) is even, and it equals \(-1\) if this number is odd.\(^{12}\)

It seems like we replaced one formula involving a sum over \( n! \) terms with an even more complicated formula, which does not appear useful for making the problem easier. But fortunately Eq. (11.4) also has an alternative description: it is simply the determinant of the matrix \( A \), which can be computed using a process similar to Gaussian elimination.

\(^{12}\) It turns out that this definition is independent of the sorting algorithm, and for example if \( \text{sign}(\pi) = -1 \) then one cannot sort an array ordered according to \( \pi \) using an even number of swaps.
11.1.11 The permanent (mod 3) problem

Emboldened by our good fortune above, we might hope to be able to compute the permanent modulo any prime \( p \) and perhaps in full generality. Alas, we have no such luck. We do not know of a much better than brute force algorithm to even compute the permanent modulo 3.

11.1.12 Finding a zero-sum equilibrium

A zero sum game is a game between two players where the payoff for one is the same as the penalty for the other. That is, whatever the first player gains, the second player loses. As much as we want to avoid them, zero sum games do arise in life, and the one good thing about them is that at least we can compute the optimal strategy.

A zero sum game can be specified by an \( n \times n \) matrix \( A \), where if player 1 chooses action \( i \) and player 2 chooses action \( j \) then player one gets \( A_{i,j} \) and player 2 loses the same amount. The famous “min max” theorem by John von Neumann states that if we allow probabilistic or “mixed” strategies (where a player does not choose a single action but rather a distribution over actions) then it does not matter who plays first and the end result will be the same. Mathematically the min max theorem is that if we let \( \Delta_n \) be the set of probability distributions over \([n]\) (i.e., non-negative columns vectors in \( \mathbb{R}^n \) whose entries sum to 1) then

\[
\max_{p \in \Delta_n} \min_{q \in \Delta_n} p^\top A q = \min_{q \in \Delta_n} \max_{p \in \Delta_n} p^\top A q \tag{11.5}
\]

The min-max theorem turns out to be a corollary of linear programming duality, and indeed the value of Eq. (11.5) can be computed efficiently by a linear program.

11.1.13 Finding a Nash equilibrium

Fortunately, not all real-world games are zero sum, and we do have more general games, where the payoff of one player is not necessarily the loss of the other. John Nash won the Nobel prize for showing that there is a notion of equilibrium for such games as well. In many economic texts it is taken as an article of faith that when actual agents are involved in such a game then they reach a Nash equilibrium. However, unlike zero sum games, we do not know of an efficient
algorithm for finding a Nash equilibrium given the description of a
general (non zero sum) game. In particular this means that, despite
economists’ intuitions, there are games for which natural strategies
will take exponential number of steps to converge to an equilibrium.

11.1.14 Primality testing

Another classical computational problem that has been of interest
since the ancient greeks is to determine whether a given number $N$
is prime or composite. Clearly we can do so by trying to divide it
with all the numbers in $2, \ldots, N - 1$, but this would take at least $N$
steps which is exponential in its bit complexity $n = \log N$. We can
reduce these $N$ steps to $\sqrt{N}$ by observing that if $N$ is a composite of
the form $N = PQ$ then either $P$ or $Q$ is smaller than $\sqrt{N}$. However, it
turns out we can do radically better. In the 1970’s, Rabin and Miller
gave probabilistic algorithms to determine whether a given number
$N$ is prime or composite in time $\text{poly}(n)$ for $n = \log N$. We will
discuss the probabilistic model of computation later in this course.
In 2002, Agrawal, Kayal, and Saxena found a deterministic $\text{poly}(n)$
time algorithm for this problem. This is surely a development that
mathematicians from Archimedes till Gauss would have found
exciting.

11.1.15 Integer factoring

Given that we can efficiently determine whether a number $N$ is prime
or composite, we could expect that in the latter case we could also
efficiently find the factorization of $N$. Alas, no such algorithm is
known. In a surprising and exciting turn of events, the non existence
of such an algorithm has been used as a basis for encryptions, and
indeed it underlies much of the security of the world wide web. We
will return to the factoring problem later in this course. We remark
that we do know much better than brute force algorithms for this
problem. While the brute force algorithms would require $2^{\Omega(n)}$ time
to factor an $n$-bit integer, there are known algorithms running in time
$2^{O(\sqrt{n})}$ and also algorithms that are widely believed (though not fully
rigorously analyzed) to run in time $2^{O(n^{1/3})}$.

11.2 Our current knowledge

The difference between an exponential and polynomial time algo-

rithm might seem merely “quantitative” but it is in fact extremely
significant. As we’ve already seen, the brute force exponential time algorithm runs out of steam very very fast, and as Edmonds says, in practice there might not be much difference between a problem where the best algorithm is exponential and a problem that is not solvable at all. Thus the efficient algorithms we mention above are widely used and power many computer science applications. Moreover, a polynomial-time algorithm often arises out of significant insight to the problem at hand, whether it is the “max-flow min-cut” result, the solvability of the determinant, or the group theoretic structure that enables primality testing. Such insight can be useful regardless of its computational implications.

At the moment we do not know whether the “hard” problems are truly hard, or whether it is merely because we haven’t yet found the right algorithms for them. However, we will now see that there are problems that do inherently require exponential time. We just don’t know if any of the examples above fall into that category.

11.3 Lecture summary

- There are many natural problems that have polynomial-time algorithms, and other natural problems that we’d love to solve, but for which the best known algorithms are exponential.
• Often a polynomial time algorithm relies on discovering some hidden structure in the problem, or finding a surprising equivalent formulation for it.

• There are many interesting problems where there is an exponential gap between the best known algorithm and the best algorithm that we can rule out. Closing this gap is one of the main open questions of theoretical computer science.

11.4 Exercises

Exercise 11.1 — exponential time algorithm for longest path. Give a \(\text{poly}(n)2^n\) time algorithm for the longest path problem in \(n\) vertex graphs.\(^{13}\) ■

Exercise 11.2 — 2SAT algorithm. For every 2CNF \(\varphi\), define the graph \(G_\varphi\) on \(2n\) vertices corresponding to the literals \(x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\), such that there is an edge \(\ell_i \rightarrow \ell_j\) iff the constraint \(\ell_i \lor \ell_j\) is in \(\varphi\). Prove that \(\varphi\) is unsatisfiable if and only if there is some \(i\) such that there is a path from \(x_i\) to \(\bar{x}_i\) and from \(\bar{x}_i\) to \(x_i\) in \(G_\varphi\). Show how to use this to solve 2SAT in polynomial time. ■

Exercise 11.3 — Regular expressions. A regular expression over the binary alphabet is a string consisting of the symbols \(\{0, 1, \emptyset, (), \ast, \}\). An expression \(\text{exp}\) corresponds to a function mapping \(\{0, 1\}^* \rightarrow \{0, 1\}\), where the \(0\) and \(1\) expressions correspond to the functions that map only output 1 on the strings \(0\) and \(1\) respectively, and if \(\text{exp}, \text{exp}'\) are expressions corresponding to the functions, \(f'\) then \((\text{exp})|(\text{exp}')\) corresponds to the function \(f \lor f', (\text{exp})(\text{exp}')\) corresponds to the function \(g\) such that for \(x \in \{0, 1\}^n\), \(g(x) = 1\) if there is some \(i \in [n]\) such that \(f(x_0, \ldots, x_i) = 1\) and \(f(x_{i+1}, \ldots, x_{n-1}) = 1\), and \((\text{exp})^*\) corresponds to the function \(g\) such that \(g(x) = 1\) if either \(x = \emptyset\) or there are strings \(x_1, \ldots, x_k\) such that \(x\) is the concatenation of \(x_1, \ldots, x_k\) and \(f(x_i) = 1\) for every \(i \in [k]\).

Prove that for every regular expression \(\text{exp}\), the function corresponding to \(\text{exp}\) is computable in polynomial time. Can you show that it is computable in \(O(n)\) time? ■

11.5 Bibliographical notes

Eugene Lawler’s quote on the “mystical power of twoness” was taken from the wonderful book “The Nature of Computation” by Moore and Mertens. See also this memorial essay on Lawler by Lenstra.
11.6 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

11.7 Acknowledgements
Modeling running time

“When the measure of the problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of . . . the order of difficulty of [an] algorithm .. is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes”, Jack Edmonds, “Paths, Trees, and Flowers”, 1963

“The computational complexity of a sequence is to be measured by how fast a multitape Turing machine can print out the terms of the sequence. This particular abstract model of a computing device is chosen because much of the work in this area is stimulated by the rapidly growing importance of computation through the use of digital computers, and all digital computers in a slightly idealized form belong to the class of multitape Turing machines.”, Juris Hartmanis and Richard Stearns, “On the computational complexity of algorithms”, 1963.

In the last lecture we saw examples of efficient algorithms, and made some claims about their running time, but did not give a mathematically precise definition for this concept. We do so in this lecture. The definition is actually quite simple: we define the complexity of computing a function \( F \) as the number of NAND\( \ast \) steps that are required to compute it. (We can also use NAND++; this choice turns out not to matter much at the coarse resolution we are interested in.) Since we think of functions that can take as input a string of arbitrary length, we need to measure this number as a function of the length of the input. For example, if a function \( F \) can be computed by a NAND\( \ast \) program that on inputs of length \( n \) takes \( O(n) \) steps then we will
think of $F$ as “efficiently computable”, while if any NAND« program requires $2^\Omega(n)$ steps to compute $F$ then we consider $F$ “intractable". The formal definition for running time the following:

**Definition 12.1 — Running time.** Let $T : \mathbb{N} \rightarrow \mathbb{N}$. We say that a (partial) function $F : \{0,1\}^* \rightarrow \{0,1\}^*$ is computable in $T(n)$ NAND« time if there is a NAND« program $P$ such that for every $x \in \{0,1\}^*$ for which $F(x)$ is defined, on input $x$, $P$ runs for at most $T(|x|)$ steps and outputs $F(x)$. Similarly, we say that $F$ is computable in $T(n)$ NAND++ time if there is a NAND++ program that computes it on $x$ in $T(|x|)$ steps.

We let $\text{TIME}_{\leq}(T(n))$ and $\text{TIME}_{=}(T(n))$ denote the set of partial functions that are computable in $T(n)$ time by NAND« and NAND++ programs respectively. We let $\text{TIME}<_{\leq}(T(n))$ denote the restriction of $\text{TIME}_{\leq}(T(n))$ to total Boolean functions $F : \{0,1\}^* \rightarrow \{0,1\}$ and define $\text{TIME}=_{=}(T(n))$ analogously. ¹

We use NAND« programs as our “default” computational model for measuring time, and so if we say that $F$ is computable in $T(n)$ time without any qualifications, or write $\text{TIME}(T(n))$ without any subscript, we mean that this holds with respect to NAND« machines. However, as we will see, in most settings relevant to this course, the choice of computational model, whether it is NAND«, NAND++, or Turing or RAM machines of various flavors, will not make much difference.

**Nice time bounds.** When considering time bounds, we want to restrict attention to “nice” bounds such as $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^{\sqrt{n}})$, $O(2^n)$, etc. and avoid pathological examples such as non-monotone functions (where the time to compute a function on inputs of size $n$ could be smaller than the time to compute it on inputs of size $n' < n$) or other degenerate cases. Thus we make the following definition:

**Definition 12.2 — Nice functions.** A function $T : \mathbb{N} \rightarrow \mathbb{N}$ is a nice time bound function (or nice function for short) if:

* $T(n) \geq n$
* $T(n) \geq T(n')$ whenever $n \geq n'$
* The function $F_T : \{0,1\}^* \rightarrow \{0,1\}^*$ such that $F_T(x)$ is the binary representation of $T(|x|)$ is in $\text{TIME}(T(n))$.

All the functions mentioned above are “nice” per Definition 12.2,

¹ We use the “overline” notation because in the literature $\text{TIME}(T(n))$ is reserved for total functions that are Boolean (i.e., have a single bit of output) and so we respect this convention. The notation $\text{TIME}(T(n))$ is non-standard. Some texts use the prefix $F$ to indicate a complexity class containing non-Boolean function (such as the class $\text{FP}$ of non-Boolean functions computable in polynomial time) and the prefix “promise” to indicate a class that could contain also partial functions (which are also known as promise problems), and so one can also use promise-$\text{FTIME}(T)$ to denote the class we define as $\text{TIME}(T(n))$. However, we find $\text{TIME}(T(n))$ to be less cumbersome. Most of the time it is clear from the context whether a function $F$ is total and/or Boolean, and so you can typically safely ignore the $\ast$ notation and the distinction between $\text{TIME}(T(n))$ and $\text{TIME}(T(n))$. ²
and from now on we will only care about the class \( \text{TIME}(T(n)) \) or \( \overline{\text{TIME}}(T(n)) \) when \( T \) is a “nice” function.

The two main time complexity classes we will be interested in are the following:

- **Polynomial time**: We say that a total Boolean function is *computable in polynomial time* if it is in the class \( P = \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c) \). Similarly, we define \( \overline{P} = \bigcup_{c \in \mathbb{N}} \overline{\text{TIME}}(n^c) \).\(^2\)

- **Exponential time**: We say that a Boolean total function is computable in exponential time if it is in the class \( \text{EXP} = \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{n^c}) \). Similarly, we define \( \overline{\text{EXP}} = \bigcup_{c \in \mathbb{N}} \overline{\text{TIME}}(2^{n^c}) \).

Since exponential time is much larger than polynomial time, clearly \( P \subseteq \text{EXP} \) (and \( \overline{P} \subseteq \overline{\text{EXP}} \)). All of the problems we listed in the last lecture are in \( \text{EXP} \), but as we’ve seen, for some of them there are much better algorithms that demonstrate that they are in fact in \( \overline{P} \).

<table>
<thead>
<tr>
<th>P</th>
<th>EXP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>Longest Path</td>
</tr>
<tr>
<td>Min cut</td>
<td>Max cut</td>
</tr>
<tr>
<td>2SAT</td>
<td>3SAT</td>
</tr>
<tr>
<td>Linear eqs</td>
<td>Quad. eqs</td>
</tr>
<tr>
<td>Zerosum</td>
<td>Nash</td>
</tr>
<tr>
<td>Determinant</td>
<td>Permanent</td>
</tr>
<tr>
<td>Primality</td>
<td>Factoring</td>
</tr>
</tbody>
</table>

A table of the examples from the previous lecture. All these problems are in \( \text{EXP} \) but the only the ones on the left column are currently known to be in \( \overline{P} \) (i.e., have a polynomial-time algorithm).

### 12.1 NAND« vs NAND++

We have seen that for every NAND« program \( P \) there is a NAND++ program \( P' \) that computes the same function as \( P \). It turns out that the \( P' \) is not much slower than \( P \). That is, we can prove the following theorem:

**Theorem 12.1** — Efficient simulation of NAND« with NAND++.

There are absolute constants \( a, b \) such that for every partial function \( F \) and nice function \( T : \mathbb{N} \to \mathbb{N} \), if \( F \in \overline{\text{TIME}}(T(n)) \) then there is a NAND++ program \( P' \) that computes \( F \) in \( T'(n) = a \cdot T(n)^b \). That is,
\[ \text{TIME}_{\text{<<}}(T(n)) \subseteq \text{TIME}_{++}(aT(n)^b) \]

Figure 12.1: The path an index variable takes in a NAND++ program

(The constant \( b \) can be easily shown to be at most four, and with more effort can be made arbitrarily close to two.)

Proof. We only sketch the proof as it follows very closely the simulation of NAND« programs by NAND++ programs that we have already seen. First note that we can simulate all the index operations using only the operations of incrementing and decrementing a single index \( i \) with polynomial overhead. The reason is that these operations are enough to copy an index to another array, and all the operations of adding, multiplying, shifting etc. can be carried out in polynomial time. Now we need to simulate a program that only decrements or increments its index variable \( i \) (perhaps based on the value of another boolean variable) with a NAND++ program where the index variable travels using the path described in Fig. 12.1. In the NAND++ program if we want to, for example, increment the variable while we are in a decrementing phase, then we’ll have to wait until the round is finished. The worst case is if we always have to wait a whole round in which case to simulate \( T \) steps we will need something like \( 2 + 4 + \cdots + 2T = O(T^2) \) steps.

Theorem 12.1 means that we could have defined \( \overline{\text{EXP}} \) and \( \overline{\text{P}} \) equally well using NAND++ instead of NAND«, as these are the same up to polynomial factors. More generally, the equivalence between NAND++ and NAND« allows us to pick our favorite one depending on the task at hand. When we want to design an algorithm, we can use the extra power and convenience afforded by NAND«. When we want to analyze a program, we can describe it in the simpler form of NAND++. 

12.2 Efficient universal machine: a NAND« interpreter in NAND«

We have seen before the NAND« or “interpreter” for NAND++. Examining this program, we can see that it has polynomial (in fact
linear) overhead, and hence, combining this with Theorem 12.1, we get that we have a universal NAND« program with polynomial overhead. But in fact, by directly simulating NAND« programs, we can do better with only polylogarithmic overhead:

**Theorem 12.2 — Efficient universality of NAND«.** There is an \( O(n \log n) \)-step NAND« program that computes the partial function \( \text{TIMEDEVAL} : \{0,1\}^* \to \{0,1\}^* \) defined as follows:

\[
\text{TIMEDEVAL}(P, x, 1^T) = P(x) \tag{12.1}
\]

if \( P \) is a valid representation of a NAND« program which produces an output on \( x \) within at most \( T \) steps.

**Proof.** Once again we only sketch the proof. The definition of executing a NAND« program is given in Appendix A. It involves maintaining variables \( pc \) and \( i \) for the program counter and index variable, as well as an index for the current line that is being executed. If a program involves \( L \) different variable identifiers, we can store all the variables in a single array \( \text{vars} \) such that if \( \text{foo} \) is the \( \ell \)-th identifier then the value of \( \text{foo}_{\langle j \rangle} \) will be stored in \( \text{vars}_{\langle L j + \ell \rangle} \). Evaluating every line can be done in about \( O(L) \) operators which is a constant independent of the input length. ■

12.3 Time hierarchy theorem

We have seen that there are uncomputable functions, but are there functions that can be computed, but only at an exorbitant cost? For example, is there a function that \emph{can} be computed in time \( 2^n \), but \emph{can not} be computed in time \( 2^{0.9n} \)? It turns out that the answer is \textbf{Yes}:

**Theorem 12.3 — Time Hierarchy Theorem.** For every nice function \( T \), there is a function \( F : \{0,1\}^* \to \{0,1\} \) in \( \text{TIME}(T(n) \log^2 n) \setminus \text{TIME}(T(n)) \).

**Proof.** Recall the Halting function \( \text{HALT} : \{0,1\}^* \to \{0,1\} \) that was defined as follows: \( \text{HALT}(P, x) \) equals 1 for every program \( P \) and input \( x \) s.t. \( P \) halts on input \( x \), and is equal to 0 otherwise. We cannot use the Halting function of course, as it is uncomputable and hence not in \( \text{TIME}(T'(n)) \) for any function \( T' \). However, we will use the following variant of it:

We define the \emph{Bounded Halting} function \( \text{HALT}_T(P) \) to equal 1 for every program \( P \) s.t. \( P \) halts when given \emph{its own string representation}
as input within $100T(|P|)$ steps.

On one hand, using the universal NAND++ program, we can evaluate $HALT_T(P)$ in $O(T(|P|) \log T(|P|))$ steps. On the other hand, we claim that $HALT_T \not\in TIME(T(n))$. The proof is very reminiscent of the proof that $HALT$ is not computable. Assume, toward the sake of contradiction, that there is some program $P^*$ that computes $HALT_T(P)$ within $T(|P|)$ steps. Then, define $Q$ to be the program that on input a program $P$ does the following:

1. Computes $b = P^*(Q) = HALT_T(Q)$ (at a cost of at most $T(|P|)$ steps, under our assumptions).

2. If $b = 1$ then it goes into an infinite loop, otherwise it halts.

We reach a contradiction by splitting into cases according to whether or not $Q$ halts when given itself as input within $2T(|Q|)$ steps. On the one hand, if $Q$ does halt, then $HALT_T(Q) = 1$, and hence under our assumption that $P^*$ computes $HALT_T$ then $Q$ will not halt. On the other hand, if $Q$ does not halt, then $HALT_T(Q) = 0$, and hence under our assumption that $P^*$ computes $HALT_T(Q)$ in $T(|Q|)$ steps, then $Q$ will halt within $2T(|Q|)$ steps. In either case we get a contradiction. ■

The time hierarchy theorem tells us that there are functions we can compute in $O(n^2)$ time but not $O(n)$, in $2^n$ time, but not $2^{\sqrt{n}}$, etc.. In particular there are most definitely functions that we can compute in time $2^n$ but not $O(n)$. We have seen that we have no shortage of natural functions for which the best known algorithm requires roughly $2^n$ time, and that many people have invested significant effort in trying to improve that. However, unlike in the finite vs. infinite case, for all of the examples above at the moment we do not know how to rule out even an $O(n)$ time algorithm. We will however see that there is a single unproven conjecture that would imply such a result for most of these problems.

12.3.1 Note: Oh-tilde or a “Baker’s Oh”

The choice of what to take as “elementary operations” can sometimes make a difference as to the asymptotic running time. For example, the set of arithmetic operations we allowed in NAND$\ast$ was inherited from C and is somewhat arbitrary. What if we wanted to add another operation? A “reasonable” operation on integers of size at most $T$ can be computed in a number of steps that is polynomial in their representation which is $\log T$ bits. Fortunately, as $T$ grows, $\log T$, ...
and even polynomial factors in it, is extremely tiny compared to it, and hence we can think of such logarithmic terms as negligible and ignore them, at least in the context of this course. Hence, just as we drop constant terms with the Big-Oh notation, it often makes sense to ignore polylogarithmic terms as well.

**Definition 12.3 — Oh Tilde.** Let $f, g : \mathbb{N} \to \mathbb{N}$, we say that $f = \tilde{O}(g)$ if there are some constant $a, b, N_0 > 0$ such that for every $n > N_0$, $f(n) \leq ag(n)(\log g(n))^b$. We say that $f = \tilde{\Omega}(g)$ if $g = \tilde{O}(f)$.

We will often use the $\tilde{O}$ notation to suppress the differences between the NAND++ model and other, somewhat more permissive, models. Assuming you don’t mind a little cheating, when you see an $\tilde{O}(\cdot)$, you won’t suffer much in understanding if you pretend that it is the same as the “regular” $O(\cdot)$. Indeed, in most settings in this course we won’t even care so much about the difference between $O(n)$ and $O(n^3)$, let alone the difference between $O(n)$ and $O(n \log n)$.

Needless to say, when one is implementing actual algorithms on actual machines, constant and logarithmic factors could make all the difference in the world, and having a model that is faithful to the actual architecture we execute it on can be very important. However, at the coarse resolution we are mostly interested here, the differences between these models will not be so important.
12.4 Simulating NAND+ or NAND++ programs with NAND programs

We have seen two measures of “computation cost” for functions. For a finite function $G : \{0,1\}^n \rightarrow \{0,1\}^m$, we said that $G \in \text{SIZE}(T)$ if there is a $T$-line NAND program that computes $G$. We saw that every function mapping $\{0,1\}^n$ to $\{0,1\}^m$ can be computed using at most $O(m2^n)$ lines. For infinite functions $F : \{0,1\}^* \rightarrow \{0,1\}^*$, we can define the “complexity” by the smallest $T$ such that $F \in \text{TIME}(T(n))$. Is there a relation between the two?

For simplicity, let us restrict attention to functions $F : \{0,1\}^* \rightarrow \{0,1\}$. For every such function, define $F_n : \{0,1\}^n \rightarrow \{0,1\}$ to be the restriction of $F$ to inputs of size $n$. It turns out that we do have at least one relation between the NAND++ complexity of $F$ and the NAND complexity of the functions $\{F_n\}$.

**Theorem 12.4 — Nonuniform computation contains uniform computation.**
There is some $c \in \mathbb{N}$ s.t. for every $F : \{0,1\}^* \rightarrow \{0,1\}$ in $\text{TIME}_{++}(T(n))$ and every $n \in \mathbb{N}$, $F_n$ is in $\text{SIZE}(10 \cdot T(n))$.

**Proof.** The proof follows by the “unraveling” argument that we’ve already seen in the proof of Godel’s Theorem. Given a NAND++ program $P$ and some function $T(n)$, we can construct a NAND program on $n$ inputs and with less than $10T(n)$ lines by simply putting “unraveling the main loop” of $P$ and hence putting $T(n)/L$ copies of $P$ one after the other, where $L$ is the number of lines in $P$, replacing any instance of $i$ with the numerical value of $i$ for that iteration. The only thing left is to check for the case that the loop value is assigned a value. We do this by adding special variable $noop$ which is initialized to 0. If loop is ever equal to 0 at the end of an iteration, then we assign 1 to $noop$. Also, we replace any assignment of a value to $y_{\langle j \rangle}$ with a conditional statement that only applies it if $noop$ equals 1.

Algorithmic version: the “NAND++ to NAND compiler”: The transformation of the NAND++ program $P$ to the NAND program $Q_P$ is itself algorithmic. Thus we can also phrase this result as follows:

**Theorem 12.5 — NAND++ to NAND compiler.** There is an $\tilde{O}(n)$-time NAND++ program $\text{COMPILE}$ such that on input a NAND++ program $P$, and strings of the form $1^n, 1^m, 1^T$ outputs a NAND
program \(Q_P\) of at most \(O(T \log T)\) lines with \(n\) bits of inputs and \(m\) bits of output, such that: For every \(x \in \{0,1\}^n\), if \(P\) halts on input \(x\) within fewer than \(T\) steps and outputs some string \(y \in \{0,1\}^m\), then \(Q_P(x) = y\).

Since NAND« programs can be simulated by NAND++ programs with polynomial overhead, we see that we can simulate a \(T(n)\) time NAND« program on length \(n\) inputs with a \(\text{poly}(T(n))\) size NAND program.

### 12.5 Simulating NAND with NAND++?

We have seen that every function in \(\text{TIME}(T(n))\) is in \(\text{SIZE}(\text{poly}(T(n)))\). One can ask if there is an inverse relation. Suppose that \(F\) is such that \(F_n\) has a “short” NAND program for every \(n\). Can we say that it must be in \(\text{TIME}(T(n))\) for some “small” \(T\)?

The answer is \textbf{no}. Indeed, consider the following “unary halting function” \(UH : \{0,1\}^* \to \{0,1\}\) defined as follows: \(UH(x) = 1\) if and the binary representation of \(|x|\) corresponds to a program \(P\) such that \(P\) halts on input \(P\). \(UH\) is uncomputable, since otherwise we could compute the halting function by transforming the input program \(P\) into the integer \(n\) whose representation is the string \(P\), and then running \(UH(1^n)\) (i.e., \(UH\) on the string of \(n\) ones). On the other hand, for every \(n\), \(UH_n(x)\) is either equal to 0 for all inputs \(x\) or equal to 1 on all inputs \(x\), and hence can be computed by a NAND program of a \textit{constant} number of lines.

The issue here is \textit{uniformity}. For a function \(F : \{0,1\}^* \to \{0,1\}\), if \(F\) is in \(\text{TIME}(T(n))\) then we have a \textit{single} algorithm that can compute \(F_n\) for every \(n\). On the other hand, \(F_n\) might be in \(\text{SIZE}(T(n))\) for every \(n\) using a completely different algorithm for every input length. While this can be a real issue, in most natural settings the difference between uniformity and non-uniformity does not seem to arise. In particular, in all the example problems in this lecture, as the input size \(n\) grows, we do not know of NAND programs that are significantly smaller than what would be implied by the best known algorithm (i.e., NAND++ program). Thus, if you pretend that \(\text{TIME}(T(n))\) (or \(\text{TIME}(T(n))\)) is roughly the same as \(\text{SIZE}(T(n))\), you will be right more often than wrong.
12.5.1 Uniform vs. Nonuniform computation: A recap

To summarize, the two models of computation we have described so far are:

- NAND programs, which have no loops, can only compute finite functions, and the time to execute them is exactly the number of lines they contain. These are also known as straightline programs or Boolean circuits.

- NAND++ programs, which include loops, and hence a single program can compute a function with unbounded input length. These are equivalent (up to polynomial factors) to Turing Machines or (up to polylogarithmic factors) to RAM machines.

For a function \( F : \{0,1\}^* \to \{0,1\} \) and some nice time bound \( T : \mathbb{N} \to \mathbb{N} \), we know that:

- If \( F \) is computable in time \( T(n) \) then there is a sequence \( \{P_n\} \) of NAND programs with \( |P_n| = \tilde{O}(T(n)) \) such that \( P_n \) computes \( F_n \) (i.e., restriction of \( F \) to \( \{0,1\}^n \)) for every \( n \).

- The reverse direction is not necessarily true - there are examples of functions \( F : \{0,1\}^n \to \{0,1\} \) such that \( F_n \) can be computed by even a constant size NAND program but \( F \) is uncomputable.

Note that the \( \textsc{Eval} \) function, that takes as input a NAND program \( P \) and an input \( x \), and outputs \( P(x) \), can be computed by a NAND++ program in \( \tilde{O}(|P|) \) time. Hence if \( F \) has the property that it is computable by a sequence \( \{P_n\} \) of programs of \( T(n) \) size, then there is a in fact an \( \tilde{O}(T(n)) \) time NAND++ program \( P^* \) that can compute \( F \) if it is only given for every \( n \) the program \( P_n \) as “advice”. For this reason, nonuniform computation is sometimes known as computation with advice. The class \( \text{SIZE}(\text{poly}(n)) \) is sometimes denoted as \( \text{P}_{/\text{poly}} \), where the \( /\text{poly} \) stands for giving the polynomial time algorithm a polynomial amount of “advice” - some string of information that depends only on the input length but not on the particular input.

12.6 Extended Church-Turing Thesis

We have mentioned the Church-Turing thesis, that posits that the definition of computable functions using NAND++ programs captures the definition that would be obtained by all physically realizable computing devices. The extended Church Turing is the statement that the same holds for efficiently computable functions, which is typically interpreted as saying that NAND++ programs can simulate every phys-
ically realizable computing device with polynomial overhead. Like the Church-Turing thesis itself, the extended Church-Turing thesis is in the asymptotic setting and does not directly yield an experimentally testable prediction. However, it can be instantiated with more concrete bounds on the overhead, which would yield predictions such as the Physical Extended Church-Turing Thesis we mentioned before, that are experimentally testable. As we mentioned, quantum computing poses a serious challenge to the extended Church-Turing thesis. However, it still seems that the extended Church-Turing thesis is fundamentally correct, in the sense that, while we do need to adapt it to account for the possibility of quantum computing, its broad outline remains unchanged. In particular, out of all the example problems mentioned in the previous lecture, as far as we know, the complexity of only one—integer factoring—is affected by modifying our model to include quantum computers as well.

12.7 Lecture summary

• We can define the time complexity of a function using NAND++ programs, and similarly to the notion of computability, this appears to capture the inherent complexity of the function.

• There are many natural problems that have polynomial-time algorithms, and other natural problems that we’d love to solve, but for which the best known algorithms are exponential.

• The time hierarchy theorem shows that there are some problems that can be solved in exponential, but not in polynomial time. However, we do not know if that is the case for the natural examples that we described in this lecture.

12.8 Exercises

Exercise 12.1 — Composition of polynomial time. Prove that if $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$ are in $\overline{P}$ then their composition $F \circ G$, which is the function $H$ s.t. $H(x) = F(G(x))$, is also in $\overline{P}$.

Exercise 12.2 — Non composition of exponential time. Prove that there is some $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$ s.t. $F, G \in \text{EXP}$ but $F \circ G$ is not in $\text{EXP}$.\footnote{TODO: check that this works, idea is that we can do bounded halting.}

Exercise 12.3 — Oblivious program. We say that a NAND++ program $P$ is oblivious if there is some functions $T : \mathbb{N} \rightarrow \mathbb{N}$ and $i : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ such that for every input $x$ of length $n$, it holds that:
* $P$ halts when given input $x$ after exactly $T(n)$ steps.

* For $t \in \{1, \ldots, T(n)\}$, after $P$ executes the $t^{th}$ step of the
  execution the value of the index $i$ is equal to $t(n, i)$. In parti-
  cular this value does not depend on $x$ but only on its length.4 Let $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be such that there is some function $m : \mathbb{N} \rightarrow \mathbb{N}$ satisfying $|F(x)| = m(|x|)$ for every $x$, and let $P$ be a NAND++
  program that computes $F$ in $T(n)$ time for some nice $T$. Then
  there is an oblivious NAND++ program $P'$ that computes $F$ in time $O(T^2(n) \log T(n))$. ■

Exercise 12.4 — Evaluating NAND programs. Let $NANDEVAL : \{0, 1\}^* \rightarrow
\{0, 1\}$ be the function that maps an $n$-input NAND++ program $P$
and a string $x \in \{0, 1\}^n$ to $P(x)$. 1. Prove that $NANDEVAL \in
TIME(\tilde{O}(T(n)^2))$. For extra credit prove that $NANDEVAL \in
TIME(\tilde{O}(T(n)))$.

2. Let $COMPILE$ be the function from Theorem 14.5 that maps a
NAND++ program $P$ and strings $1^n, 1^m, 1^T$ to an $n$-input $m$-output
NAND program $Q_P$ such that for every $x \in \{0, 1\}^n$, if $P(x)$ outputs
$y \in \{0, 1\}^m$ within $T$ steps then $Q_P(x) = y$. We saw that
$COMPILE \in TIME(\tilde{O}(n))$. Use that to show that $TIMEDEVAL \in
TIME(\tilde{O}(n))$. ■

12.9 Bibliographical notes

6

12.10 Further explorations

Some topics related to this lecture that might be accessible to ad-
vanced students include: (to be completed)

12.11 Acknowledgements

4 An oblivious program $P$ cannot
compute functions whose output length
is not a function of the input length,
though this is not a real restriction, as
we can always embed variable output
functions in fixed length ones using
some special “end of output” marker.

5 TODO: Add exercise showing NAND
is like NAND++ with advice. Mention
the definition of $P/poly$.

6 TODO: add reference to best algo-
rithm for longest path - probably the
Bjorklund algorithm.
Let us consider several of the problems we have encountered before:

- Finding the longest path in a graph
- Finding the maximum cut in a graph
- The 3SAT problem
- Solving quadratic equations

All of these have the following properties:

- These are important problems, and people have spent significant effort on trying to find better algorithms for them.
- They have trivial exponential time algorithms that involve enumerating all possible solutions.
- At the moment the best known algorithms are not much better than the trivial one in the worst-case.

In this lecture we will see that, despite their apparent differences, all these problems are *computationally equivalent*, in the sense that solving one of them immediately implies solving the others. This phenomenon, known as **NP completeness**, is one of the surprising discoveries of theoretical computer science, and we will see that it has far-reaching ramifications.
13.0.1 Decision problems

For reasons of technical conditions rather than anything substantial, we will concern ourselves in this lecture with decision problems or Boolean functions. Thus, we will model all the problems as functions mapping $\{0, 1\}^*$ to $\{0, 1\}$:

- The 3SAT problem corresponds to the function $3SAT$ that maps a 3CNF formula $\varphi$ to 1 if there exists some assignment $x$ that satisfies it and to 0 otherwise.

- The quadratic equations problem corresponds to the function $QUADEQ$ that maps a set of quadratic equations $E$ to 1 if there is an assignment $x$ that satisfies all equations and to 0 otherwise.

- The longest path problem corresponds to the function $LONGPATH$ that maps a graph $G$ and a number $k$ to 1 if there is a simple path in $G$ of length at least $k$ and maps $(G, k)$ to 0 otherwise.

- The maximum cut problem corresponds to the function $MAXCUT$ that maps a graph $G$ and a number $k$ to 1 if there is a cut in $G$ that cuts at least $k$ edges and maps $(G, k)$ to 0 otherwise.

13.1 Reductions

Suppose that $F, G : \{0, 1\}^* \rightarrow \{0, 1\}$ are two functions. How can we show that they are “computationally equivalent”? The idea is that we show that an efficient algorithm for $F$ would imply an efficient algorithm for $G$ and vice versa. The key to this is the notion of a reduction:

**Definition 13.1 — Reductions.** Let $F, G : \{0, 1\}^* \rightarrow \{0, 1\}^*$. We say that $F$ reduces to $G$, denoted by $F \leq_p G$ if there is a polynomial-time computable $R : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for every $x \in \{0, 1\}^*$,

$$F(x) = G(R(x)) \ .$$

(13.1)

We say that $F$ and $G$ have equivalent complexity if $F \leq_p G$ and $G \leq_p H$.

If $F \leq_p G$ and $G$ is computable in polynomial time, then $F$ is computable in polynomial time as well. Indeed, Eq. (13.1) shows a way how to compute $F$ by applying the polynomial-time reduction $R$ and then the polynomial-time algorithm for computing $F$. 
13.2 Some example reductions

We will now use reductions to show that the problems above- 3SAT, Quadratic Equations, Maximum Cut, and Longest Path- are indeed computationally equivalent to one another. We start by reducing 3SAT to the latter three problems, demonstrating that solving either of them will solve it 3SAT.

![Diagram showing reduction from 3SAT to other problems]

Figure 13.1: Our first stage in showing equivalence is to reduce 3SAT to the three other problems

13.3 Reducing 3SAT to quadratic equations

Let us now see our first example of a reduction. We will show how to reduce 3SAT to the problem of Quadratic Equations.

**Theorem 13.1 — Hardness of quadratic equations.**

\[ 3\text{SAT} \leq_p \text{QUADEQ} \tag{13.2} \]

where 3SAT is the function that maps a 3SAT formula \( \varphi \) to 1 if it is satisfiable and to 0 otherwise, and QUADEQ is the function that maps a set \( E \) of quadratic equations over \( \{0, 1\}^n \) to 1 if its satisfiable and to 0 otherwise.

To do so, we need to give a polynomial-time transformation of every 3SAT formula \( \varphi \) into a set of quadratic equations \( E \). Recall that a 3SAT formula \( \varphi \) is a formula such as \( (x_{17} \lor \overline{x}_{101} \lor x_{57}) \land (x_{18} \lor \overline{x}_{19} \lor \overline{x}_{101}) \lor \cdots \). That is, \( \varphi \) is composed of the AND of \( m \) 3SAT clauses where a 3SAT clause is the OR of three variables or their
negation. A quadratic equations instance $E$, is composed of a list of equations, each of involving a sum of variables or their products, such as $x_1 x_2 - x_1 + 2x_3 = 2$, etc.. Recall that we restrict attention to $\{0, 1\}$ valued variables for simplicity (or, equivalently, assume that the instance contains the equations $x_i^2 - x_i = 0$ for every $i$.)

There is a natural way to map a 3SAT instance into a set of equations, and that is to map a clause such as $(x_1 \vee \bar{x}_2 \vee x_5)$ to the equation $(1 - x_1)(1 - \bar{x}_2)x_5 = 0$. We can map a formula $\varphi$ with $m$ clauses into a set $E$ of $m$ such equations such that there is an $x$ with $\varphi(x) = 1$ if and only if there is an assignment to the variables that satisfies all the equations of $E$. The problem is that the equations in $E$ will not be quadratic but cubic: they contain terms of degree three. So, to finish the reduction it will suffice to show that we can map any set of cubic equations $E$ into a set of quadratic equations $E'$ such that $E$ is satisfiable if and only if $E'$ is.

The idea is that for every two variables $x_i$ and $x_j$, we add an extra variables $y_{i,j}$ and $z_{i,j}$ and a set of quadratic equations that, if satisfied, guarantee that $y_{i,j} = x_i x_j$. Once we do that, we can replace cubic terms of the form $x_i x_j x_k$ with the quadratic term $y_{i,j} x_k$ in the new variables. We can do so by adding the following equations

$$
x_i y_{i,j} - y_{i,j} = 0 \\
x_j y_{i,j} - y_{i,j} = 0 \\
y_{i,j} + 1 - x_i - x_j - z_{i,j} = 0
$$

(13.3)

Note that this system can be satisfied by setting $y_{i,j} = x_i x_j$ and $z_{i,j} = (1 - x_i)(1 - x_j)$. It turns out this is the only solution

**Lemma 13.2** Every assignment to $\{x_i, y_{i,j}, z_{i,j}\}$ that satisfies the equations above must satisfy $y_{i,j} = x_i x_j$

We leave proving Lemma 13.2 as Exercise 13.1. Using this lemma, we can transform the cubic system $E$ in the variables $\{x_i\}_{i \in [n]}$ to an equivalent quadratic system $E'$ in the variables $\{x_i, y_{i,j}, z_{i,j}\}_{i,j \in [n]}$. Note that the transformation (which involves a simple translation of every 3SAT clause to a constant number of equations) can be easily carried out in polynomial (in fact linear) time. Since the original system was equivalent to the 3SAT instance it is not hard to see that we get:

- **(Completeness)** If $\varphi$ has a satisfying assignment $x$ then $E'$ has a satisfying assignments $(x, y, z)$.
- **(Soundness)** If $E'$ has a satisfying assignment $(x, y, z)$ then $\varphi$ has a
satisfying assignment.

Thus if we define $E' = R(\phi)$, then we see that for every 3SAT formula $\phi$, $3\text{SAT}(\phi) = \text{QUADEQ}(R(\phi))$, showing that $3\text{SAT} \leq_p \text{QUADEQ}$ and completing the proof of Theorem 13.1.

13.4 Reducing 3SAT to Maximum Cut

13.5 Reducing 3SAT to Longest Path

The two reductions above might not have seemed so surprising, since quadratic equations and max cut are at least somewhat similar to 3SAT in the sense that they are constraint satisfaction problems, which are about trying to find an assignment $x \in \{0, 1\}^n$ (or equivalently a set $S \subseteq [n]$) that satisfies as many local constraints (such as quadratic equations or cutting edges) as possible. But we will now show that 3SAT reduces the the longest path problem as well, which seems to be of a different nature.

Theorem 13.3 — Hardness of longest path.

$$3\text{SAT} \leq_p \text{LONGPATH} \quad (13.4)$$

To prove Theorem 13.3 need to show how to transform a 3CNF formula $\phi$ into a graph $G$ and two vertices $s, t$ such that $G$ has a path of length at least $k$ if and only if $\phi$ is satisfiable. The idea of the reduction is sketched in Fig. 13.2 and Fig. 13.3. We build a graph $G$ that “snakes” from $s$ to $t$ as follows. After $s$ we add a sequence of $n$ long loops. Each loop has an “upper path” and a “lower path”. A simple path cannot take both the upper path and the lower path, and so it will need to take exactly one of them to reach $s$ from $t$.

Our intention is that a path in the graph will correspond to an assignment $x \in \{0, 1\}^n$ in the sense that taking the upper path in the $i$th loop corresponds to assigning $x_i = 1$ and taking the lower path corresponds to assigning $x_i = 0$. When we are done snaking through all the $n$ loops corresponding to the variables to reach $t$ we need to pass through $m$ “obstacles”: for each clause $j$ we will have a small gadget consisting of a pair of vertices $s_j, t_j$ that have three paths between them. For example, if the $j$th clause had the form $x_{17} \lor \overline{x}_{55} \lor x_{72}$ then one path would go through a vertex in
the lower loop corresponding to \(x_{17}\), one path would go through a vertex in the upper loop corresponding to \(x_{35}\) and the third would go through the lower loop corresponding to \(x_{72}\). We see that if we went in the first stage according to a satisfying assignment then we will be able to find a free vertex to travel from \(s_j\) to \(t_j\). We link \(t_1\) to \(s_2\), \(t_2\) to \(s_3\), etc and link \(t_m\) to \(t\). Thus a satisfying assignment would correspond to a path from \(s\) to \(t\) that goes through one path in each loop corresponding to the variables, and one path in each loop corresponding to the clauses. We can make the loop corresponding to the variables long enough so that we must take the entire path in each loop in order to have a fighting chance of getting a path as long as the one corresponds to a satisfying assignment. But if we do that, then the only way if we are able to reach \(t\) is if the paths we took corresponded to a satisfying assignment, since otherwise we will have one clause \(j\) where we cannot reach \(t_j\) from \(s_j\) without using a vertex we already used before.

![Figure 13.2: We can transform a 3SAT formula \(\varphi\) into a graph \(G\) such that the longest path in the graph \(G\) would correspond to a satisfying assignment in \(\varphi\). In this graph, the black colored part corresponds to the variables of \(\varphi\) and the blue colored part corresponds to the vertices. A sufficiently long path would have to first “snake” through the black part, for each variable choosing either the “upper path” (corresponding to assigning it the value True) or the “lower path” (corresponding to assigning it the value False). Then to achieve maximum length the path would traverse through the blue part, where to go between two vertices corresponding to a clause such as \(x_{17} \vee x_{32} \vee x_{37}\), the corresponding vertices would have to have been not traversed before.](image)

### 13.6 Reducing to 3SAT from... everything?

So far we have shown that 3SAT is no harder than Quadratic Equations, Maximum Cut, and Longest Path. But to show that they are
equivalent we need to give reductions in the other direction, reducing each one of these problems to 3SAT as well. It turns out we can reduce all three problems to 3SAT in one fell swoop. In fact, this result extends far beyond these particular problems: every problem that corresponds to finding a solution that can be easily verified can be reduced to 3SAT. We make the following definition:

**Definition 13.2 — NP.** We say that $F : \{0, 1\}^* \rightarrow \{0, 1\}$ is in NP if there exists some constants $a, b \in \mathbb{N}$ and $G : \{0, 1\}^* \rightarrow \{0, 1\}$ such that $G \in \mathbb{P}$ and for every $x \in \{0, 1\}^n$

$$F(x) = 1 \iff \exists y \in \{0, 1\}^{anb} \text{ s.t. } G(x, y) = 1 \quad (13.5)$$

The name NP stands for “nondeterministic polynomial time” and is used for historical reasons, see the bibliographical notes.

### 13.6.1 Examples:

- 3SAT is in NP since for every $\ell$-variable formula $\varphi$, $3SAT(\varphi) = 1$ if and only if there exists a satisfying assignment $x \in \{0, 1\}^\ell$ such that $\varphi(x) = 1$, and we can check this condition in polynomial time.\(^2\)

- QUADEQ is in NP since for every $\ell$ variable instance of quadratic equations $E$, $QUADEQ(E) = 1$ if and only if there exists an

\(^2\)Note that an $\ell$ variable formula $\varphi$ is represented by a string of length at least $\ell$, and we can use some “padding” in our encoding so that the assignment to $\varphi$’s variables is encoded by a string of length exactly $|\varphi|$. We can always use this padding trick, and so one can think of the condition Eq. (13.5) as simply stipulating that the “solution” $y$ to the problem $x$ is of size at most $poly(|x|)$.\(^3\)
assignment \( x \in \{0, 1\}^f \) that satisfies \( E \) and we can check this condition in polynomial time.

- **LONGPATH** is in **NP** since for every graph \( G \) and integer \( k \),
  \( \text{LONGPATH}(G, k) = 1 \) if and only if there exists a simple path \( P \) in \( G \) that is of length at least \( k \), and we can check this condition in polynomial time.

- **MAXCUT** is in **NP** since for every graph \( G \) and integer \( k \),
  \( \text{MAXCUT}(G, k) = 1 \) if and only if there exists a cut \( (S, \bar{S}) \) in \( G \) that cuts at least \( k \) edges, and we can check this condition in polynomial time.

### 13.6.2 From **NP** to 3SAT

There are many, many, more examples of interesting functions we would like to compute that are easily shown to be in **NP**. What is quite amazing is that if we can solve 3SAT then we can solve all of them!

**Theorem 13.4 — Cook-Levin Theorem.** For every \( F \in \text{NP} \), \( F \leq_p 3\text{SAT} \).

We will see the proof of **Theorem 13.4** in the next lecture, but note that it immediately implies that QUADEQ, LONGPATH, and MAXCUT all reduce to 3SAT. In fact, combining it with the reductions we’ve seen, it implies that all these problems are equivalent! To reduce QUADEQ to LONGPATH, we can first reduce QUADEQ to 3SAT using **Theorem 13.4** and use the reduction we’ve seen from 3SAT to LONGPATH. There is of course nothing special about QUADEQ here- by combining **Theorem 13.4** with the reduction we saw, we see that just like 3SAT, every \( F \in \text{NP} \) reduces to LONGPATH, and the same is true for QUADEQ and MAXCUT. All these problems are in some sense “the hardest in **NP**” in the sense that an efficient algorithm for one of them would imply an efficient algorithm for all the problems in **NP**. This motivates the following definition

**Definition 13.3 — NP completeness.** We say that \( G : \{0, 1\}^* \rightarrow \{0, 1\} \) is **NP hard** if for every \( F \in \text{NP} \), \( F \leq_p G \). We say that \( G \) is **NP complete** if \( G \) is **NP hard** and it is in **NP**.

**Theorem 13.4** and the reductions we’ve seen in this lecture show that despite their superficial differences, 3SAT, quadratic equations, longest path and maximum cut, are all **NP complete**. Thousands more problems have been shown to be **NP complete**, arising from all
science, mathematics, economics, engineering and many other fields. If (as is widely believed to be the case) there is no polynomial-time (or even $2^{O(n)}$ time) algorithm for 3SAT, then all of these problems cannot be computed by an efficient algorithm.

13.7 Complexocartography

Clearly $\text{NP} \supseteq \text{P}$, since if we can decide efficiently whether $F(x) = 1$, we can simply ignore any “solution” that we are presented with. Also, $\text{NP} \subseteq \text{EXP}$, since all the problems in $\text{NP}$ can be solved in exponential time by enumerating all the possible solutions. For the $\text{NP}$ complete ones, we believe that we cannot radically improve upon this trivial algorithm. Whether or not is true is the most important open question in computer science, commonly phrased as the $\text{P}$ vs $\text{NP}$ question- is it the case that $\text{P} = \text{NP}$?

One of the mysteries of computation is that people have observed a certain empirical “zero one law” or “dychotomy” in the computational complexity of natural problems, in the sense that they are either in $\text{P}$ (in fact often with a low exponent) or are $\text{NP}$ hard. However, it is believed that there are problems in $\text{NP}$ that are neither in $\text{P}$ not in $\text{NP}$, and in fact a result known as “Ladner’s Theorem” shows that if $\text{P} \neq \text{NP}$ then this is the case.

13.8 NP completeness as a barrier to understanding

• Many of the problems which we don’t know polynomial-time algorithms for are $\text{NP}$ complete, which means that finding a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for all of them.
• It is conjectured that $\text{NP} \neq \text{P}$ which means that we believe that polynomial-time algorithms for these problems are not merely unknown but are nonexistent.
• While an $\text{NP}$ hardness result means for example that a full fledged “textbook” solution to a problem such as MAX-CUT that is as
Figure 13.4: A rough illustration of the (conjectured) status of problems in exponential time. Darker colors correspond to higher running time, and the circle in the middle is the problems in \( P \). \( NP \) is a (conjectured to be proper) superclass of \( P \) and the \( NP \) complete problems (or NPC) for short are the “hardest” problems in \( NP \), in the sense that a solution for one of them implies solution for all other problems problems in \( NP \). It is conjectured that all the \( NP \) complete problems require at least \( \exp(n^e) \) time to solve for a constant \( \epsilon > 0 \), and many require \( \exp(\Omega(n)) \) time. The permanent is not believed to be contained in \( NP \) though it is \( NP \)-hard, which means that a polynomial-time algorithm for it implies that \( P = NP \).
clean and general as the algorithm for MIN-CUT probably does not exist, it does not mean that we need to give up whenever we see a MAX-CUT instance. Later in this course we will discuss several strategies to deal with NP hardness, including average-case complexity and approximation algorithms.

13.10 Exercises

Exercise 13.1 Prove Lemma 13.2

Exercise 13.2 — Transitivity of reductions. Prove that if $F \leq_p G$ and $G \leq_p H$ then $F \leq_p H$.

Exercise 13.3 — Poor man’s Ladner’s Theorem. Prove that if there is no $n^{O(\log^2 n)}$ time algorithm for 3SAT then there is some $F \in \text{NP}$ such that $F \notin \text{P}$ and $F$ is not NP complete.\(^5\)

13.11 Bibliographical notes

13.12 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

13.13 Acknowledgements

\(^5\) TODO: Maybe mention either in exercise or in body of the lecture some NP hard results motivated by science. For example, shortest superstring that is motivated by genome sequencing, protein folding, maybe others.

\(^6\) **Hint**: Use the function $F$ that on input a formula $\varphi$ and a string of the form $1^t$, outputs 1 if and only if $\varphi$ is satisfiable and $t = |\varphi|^{\log |\varphi|}$. 
The Cook-Levin Theorem

In this lecture we prove one of the most important theorems in Computer Science:

**Theorem 14.1 — Cook-Levin Theorem.** $3SAT$ is NP-complete.

The Cook-Levin Theorem is the underpinning to a great web of reductions from $3SAT$ to thousands of problems across great many fields. Some problems that have been shown NP complete include: minimum-energy protein folding, minimum surface-area foam configuration, map coloring, optimal Nash equilibrium, quantum state entanglement, minimum supersequence of a genome, minimum codeword problem, shortest vector in a lattice, minimum genus knots, positive Diophantine equations, integer programming, and many many more. The worst-case complexity of all these problems is (up to polynomial factors) equivalent to that of $3SAT$, and through the Cook-Levin Theorem, to all problems in NP.

Since (as we have already observed before) $3SAT$ is obviously in NP, to prove **Theorem 14.1** we need to show that $F \leq_p 3SAT$ for every $F \in$ NP. We will do so in three stages. We define two intermediate problems: NANDSAT and 3NAND. We will shortly show the definitions of these two problems, but **Theorem 14.1** will follow from the following three theorems:

**Theorem 14.2** NANDSAT is NP-complete.

**Theorem 14.3** NANDSAT $\leq_p$ 3NAND.

**Theorem 14.4** 3NAND $\leq_p$ 3SAT.
Together Theorem 14.2, Theorem 14.3, and Theorem 14.4 immediately imply that 3SAT is NP complete (can you see why?), hence establishing Theorem 14.1. We now prove them one by one, providing the requisite definitions as we go along.

14.1 The NANDSAT Problem, and why it is NP complete.

We define the NANDSAT problem as follows: its input is a NAND program $P$ (represented as a string as usual), and we define $\text{NANDSAT}(P) = 1$ if and only if there is some input $x$ such that $P(x) = 1$. Since given $P$ and $x$, we can check in polynomial (in fact $\tilde{O}(|P|)$ time) whether $P(x) = 1$, NANDSAT is in NP. Hence the proof of Theorem 14.2 will follow by showing that NANDSAT is NP hard, or in other words, that $F \leq_p \text{NAND}$ for every $F \in \text{NP}$.

Let $F \in \text{NP}$. Hence there are constants $a, b, c \in \mathbb{N}$ and a NAND++ program $P_F$ such that for every $x \in \{0, 1\}^n$, $F(x) = 1$ if and only if there exists some $y \in \{0, 1\}^{an^b}$ such that $P_F(xy)$ outputs 1 within $n^c$ steps. Recall that we had the following “NAND++ to NAND compiler”:

**Theorem 14.5 — NAND++ to NAND compiler.** There is an $\tilde{O}(n)$-time NAND++ program $\text{COMPILE}$ such that on input a NAND++ program $P$, and strings of the form $1^n, 1^m, 1^T$ outputs a NAND program $Q_P$ of at most $O(T \log T)$ lines with $n$ bits of inputs and $m$ bits of output, such that:

For every $x \in \{0, 1\}^n$, if $P$ halts on input $x$ within fewer than $T$ steps and outputs some string $y \in \{0, 1\}^m$, then $Q_P(x) = y$.

Therefore, we can transform the NAND++ program $P_F$ into a NAND program $Q_F$ of at most $\tilde{O}(n^c)$ lines taking $n + an^b$ bits of input and outputting one bit, such that $P_F(xy) = Q_F(xy)$ for every $x \in \{0, 1\}^n$ and $y \in \{0, 1\}^n$.

Now if $Q_F$ has $t$ lines, then we can transform $Q_F$ and $x$ into a NAND program $Q_{F,x}$ with $t' + 4$ lines that takes $an^b$ bits of inputs, and satisfies $P_x(y) = Q_{F,x}(x, y)$ for every $y \in \{0, 1\}^{an^b}$. The transformation is very simple. Recall that by adding four lines to the program, we can ensure that we have variables zero and one that are equal to 0 and 1 respectively. Now for every $i \in \{0, \ldots, n - 1\}$, we can replace any mention of $x_{-i}$ in the program with either zero or one depending on whether $x_i$ is equal to 0 or 1 respectively. Then for $i = n, \ldots, n + an^b - 1$, we can replace $x_{-i}$ with $x_{-i} - n$ so we make the
last \(an^b\) inputs of \(Q_F\) become the first inputs of \(Q_{F,x}\).

By our construction, \(NAND(Q_{F,x}) = 1\) if and only if there is \(y \in \{0,1\}^m\) such that \(P_F(xy) = 1\), which by definition, happens if and only if \(F(x) = 1\). Hence the polynomial-time transformation \(x \mapsto Q_{F,x}\) is a reduction from \(F\) to \(NANDSAT\), completing the proof of Theorem 14.2.

14.2 The 3NAND problem

The 3NAND problem is defined as follows: the input is a logical formula on a set of variables \(z_1, \ldots, z_m\) which is an AND of constraints of the form \(z_i = NAND(z_j, z_k)\). The output is 1 if and only if there is an assignment to the \(z\)'s that satisfies all these constraints.

To prove Theorem 14.3 we need to give a reduction from \(NANDSAT\) to 3NAND. Given a NAND program \(P\) with \(n\) inputs, one outputs, and \(m\) lines, we define a 3NAND formula \(\psi\) as follows. The formula \(\psi\) will have \(m + n\) variables \(z_1, \ldots, z_{m+n+1}\). For every \(i \in \{1, \ldots, m\}\), the variable \(z_i\) will correspond to the \(i^{th}\) line of the program \(P\). The variables \(z_{m+1}, \ldots, z_{m+n}\) will correspond to the input variables \(x_0 \ldots x_{n-1}\) of \(P\). The last variable \(z_{m+n+1}\) is added for convenience, where we ensure that it is always the negation of \(z_{m+n}\) by adding the constraint \(z_{m+n+1} = NAND(z_{m+n}, z_{m+n})\).

If the \(i^{th}\) line in the program \(P\) has the form

\[
\text{var := var'} NAND \text{ var''}
\]

then we add a constraint of the form \(z_i = NAND(z_j, z_k)\), where we choose \(j\) and \(k\) as follows. If \(\text{var}'\) is an input variable \(x_{\ell}\), then we choose \(j = n + \ell\). Similarly, if \(\text{var}''\) is an input variable \(x_{\ell'}\), then we choose \(k = n + \ell'\). If \(\text{var}'\) is a workspace variable then we let \(j\) be the index of the last line prior to \(i\) in which \(\text{var}'\) was assigned. Similarly, if \(\text{var}''\) is a workspace variable then we let \(k\) be the index of the last line prior to \(i\) in which \(\text{var}''\) was assigned.

Finally, if \(i_0\) is the last line in which the output variable \(y_0\) was assigned, then we add the constraint \(z_{i_0} = NAND(z_{m+n}, z_{m+n+1})\), which (since we constrained \(z_{m+n+1} = 1 - z_{m+n}\)) is equivalent to constraining \(z_{i_0} = 1\).

We make the following claim

**Lemma 14.6** There is \(x \in \{0,1\}^n\) s.t. \(P(x) = 1\) if and only if there is \(z \in \{0,1\}^{m+n+1}\) s.t. \(\psi(z) = 1\).
We reduce $\text{NANDSAT}$ to $3\text{NAND}$ by mapping a program $P$ to a formula $\psi$ where we have a variable for each line and input variable of $P$, and add a constraint to ensure that the variables are consistent with the program. We also add a constraint that the final output is 1. One can show that there is an input $x$ such that $P(x) = 1$ if and only if there is a satisfying assignment for $\psi$.

**Proof.** Suppose that there is such an $x$, and consider the execution of $P$ on $x$. For $i = 1, \ldots, m$ we let $z_i$ be the value that is assigned to a variable in the $i^{th}$ line, for $j = 0, \ldots, n - 1$, we let $z_{n+1+j} = x_j$, and we let $z_{m+n+1} = 1 - z_{m+n}$. By the semantics of the NAND program, the value $z_i$ will correspond to the NAND of the values of the variables corresponding to $z_j$ and $z_k$. Hence we see that every one of our constraints of the form $z_i = \text{NAND}(z_j, z_k)$ is satisfied, and moreover since the final output is 1, the last constraint is satisfied as well.

In the other direction, suppose that there is an assignment $z \in \{0, 1\}^{m+n+1}$ s.t. $\psi(z) = 1$, and let $x \in \{0, 1\}^n$ s.t. $x_j = z_{n+1+j}$ for every $j \in \{0, \ldots, n - 1\}$. We claim that $P(x) = 1$. Indeed note that, because $z$ satisfies all constraints of $\psi$, as we execute the program $P$ on $x$, the value assigned in the $i^{th}$ line is equal to $z_i$. Hence in particular the value that is finally assigned to the output variable will equal to 1.  

This claim means that the polynomial time map $P \mapsto \psi$ that transform a NAND program to a 3NAND formula satisfies that $\text{NANDSAT}(P) = 3\text{NAND}(\psi)$ and hence this is a reduction demonstrating $\text{NANDSAT} \leq_p 3\text{NANT}$ and concluding the proof of Theorem 14.3.
14.3 Concluding the proof of Cook-Levin

To conclude the proof of Theorem 14.1, we need to show Theorem 14.4. That is, to reduce 3NAND to 3SAT. The reduction is actually quite simple. Since $NAND(z, z') = \overline{z \land z'} = \overline{z} \lor \overline{z'}$, the constraint

$$z_i = NAND(z_j, z_k)$$  \hfill (14.1)

is the same as

$$z_i \Rightarrow (\overline{z_j} \lor \overline{z_k}) \land (\overline{z_j} \lor \overline{z_k}) \Rightarrow z_i$$  \hfill (14.2)

where $\Rightarrow$ is the logical implication operator, defined as $a \Rightarrow b = \overline{a} \lor b$.

Hence Eq. (14.1) is the same as

$$(\overline{z_i} \lor \overline{z_j} \lor \overline{z_k}) \land ((z_j \land z_k) \lor z_i)$$  \hfill (14.3)

which is the same as

$$(\overline{z_i} \lor \overline{z_j} \lor \overline{z_k}) \land (z_i \lor z_j) \land (z_i \lor z_k)$$  \hfill (14.4)

Hence by replacing every 3NAND constraint of $\psi$ with three 3OR constraints as above we can transform a 3NAND formula $\psi$ to an equivalent 3CNF formula $\phi$, thus completing the proof.\(^1\)

14.4 Lecture summary

2

14.5 Exercises

3

14.6 Bibliographical notes

4

\(^1\) The resulting formula will have some of the OR’s involving only two variables. If we wanted to insist on each formula involving three distinct variables we can always add a “dummy variable” $z_{m+n+2}$ and include it in all the OR’s involving only two variables. We leave it as an exercise to show that the new formula will be satisfiable if and only if the previous one was.

\(^2\) TODO: Add summary

\(^3\) TODO: add exercises

\(^4\) TODO: credit surveys of Avi, Madhu

14.7 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)
14.8 Acknowledgements
Advanced hardness reductions (advanced lecture)

PLAN: show NP hardness of the permanent (maybe also mention Tutte polynomial?), maybe others such as the Dichotomy theorem. Perhaps assume hardness of unique-SAT, which we will show in the randomness section. Maybe say something about PCP reductions?

15.1 Lecture summary

15.2 Exercises

15.3 Bibliographical notes

15.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

15.5 Acknowledgements
What if $P$ equals $NP$?

“We should have no fear . . . we will be protected by God.”, President Donald J. Trump, inauguration speech, 2017

“No more half measures, Walter”, Mike Ehrmantraut in “Breaking Bad”, 2010.

“The evidence in favor of $[P \neq NP]$ and [its algebraic counterpart] is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures.”, Volker Strassen, laudation for Leslie Valiant, 1986.

We have mentioned that the question of whether $P = NP$, which is equivalent to whether there is a polynomial-time algorithm for 3SAT, is the great open question of Computer Science. But why is it so important? In this lecture, we will try to figure out the implications of such an algorithm.

First, let us get one qualm out of the way. Sometimes people say, “What if $P = NP$ but the best algorithm for 3SAT takes $n^{100}$ time?” Well, $n^{100}$ is much larger than, say, $2^{\sqrt{n}}$ for any input shorter than $10^{60}$ bits which is way way larger than the world’s total storage capacity (estimated at a “mere” $10^{21}$ bits or about 200 exabytes at the time of this writing). So qualitatively, this question can be thought of as “what if the complexity of 3SAT is exponential for all inputs that we will ever encounter, but then grows much smaller than that?” To me this sounds like the computer science equivalent of asking “what if
the laws of physics change completely once they are out of the range of our telescopes?”. Sure, this is a valid possibility, but wondering about it does not sound like the most productive use of our time.

So, as the saying goes, we’ll keep an open mind, but not so open that our brains fall out, and assume from now on that:

- There is a mathematical god.

and

- She does not “pussyfoot around” or take “half measures”. If she decided to make 3SAT easy then this problem will have an $10^6 \cdot n$ (or at worst $10^6n^2$) time algorithm, and if she decided to make 3SAT hard, then for every $n$, 3SAT on $n$ variables cannot be solved by a NAND program of fewer than $2^{10^{-6}n}$ lines.

So far most of our evidence points to the latter possibility of 3SAT being exponentially hard, but we have not ruled out the former possibility either. In this lecture we will explore some of its consequences.

### 16.1 Search to decision reduction

A priori, having a fast algorithm for 3SAT might not seem so impressive. Sure, it will allow us to decide the satisfiability of not just 3CNF formulas but also quadratic equations, as well as find out whether there is a long path in a graph, and solve many other decision problems. But this is not typically what we want to do. It’s not enough to know if a formula is satisfiable- we want to discover the actual actual satisfying assignment. Similarly, it’s not enough to find out if a graph has a long path- we want to actually find the path.

It turns out that if we can solve these decision problems, we can solve the corresponding search problems as well:

**Theorem 16.1 — Search vs Decision.** Suppose that $P = NP$. Then for every polynomial-time NAND++ program $P$ and $a, b \in \mathbb{N}$, there is a polynomial time NAND++ program $FIND$ such that for every $x \in \{0, 1\}^n$, if there exists $y \in \{0, 1\}^{arb}$ satisfying $P(xy) = 1$, then $FIND(x)$ finds some string $y'$ satisfying this condition.

While the theorem’s statement is a bit of a mouthful, it in particular implies that if $P = NP$ then we can we can find a satisfying assignment for every satisfiable 3CNF formula, find a simple path of length at least $k$ in any graph that has one, etc..
Proof. The idea behind the proof is simple. If $P = NP$ then for every polynomial-time NAND++ program $P$ and $a, b \in \mathbb{N}$, there is a polynomial-time algorithm $\text{STARTSWITH}$ that on input $x \in \{0, 1\}^*$ and $z \in \{0, 1\}^\ell$, outputs 1 if and only if there exists some $y \in \{0, 1\}^{anb}$ such that the first $\ell$ bits of $y$ are equal to $z$ and $P(xy) = 1$. Indeed, we leave it as an exercise to verify that the $\text{STARTSWITH}$ function is in $NP$ and hence can be solved in polynomial time if $P = NP$.

Now for any program $P$ and $a, b \in \mathbb{N}$, we can implement $\text{FIND}(x)$ as follows:

1. For $\ell = 0, \ldots, anb - 1$ do the following:
   
   a. let $b_0 = \text{STARTSWITH}(xz_0 \cdots z_{\ell-1}0)$ and $b_1 = \text{STARTSWITH}(xz_0 \cdots z_{\ell-1}1)$
   
   b. If $b_0 = 1$ then $z_\ell = 0$, otherwise $z_\ell = 1$. 2. Output $z_0, \ldots, z_{anb-1}$

To analyze the $\text{FIND}$ algorithm note that it makes $2anb - 1$ invocations to $\text{STARTSWITH}$ and hence if the latter is polynomial-time then so is $\text{FIND}$. Now suppose that $x$ is such that there exists some $y$ satisfying $P(xy) = 1$. We claim that at every step $\ell = 0, \ldots, anb - 1$, we maintain the invariant that there exists $y \in \{0, 1\}^{anb}$ whose first $\ell$ bits are $z$ s.t. $P(xy) = 1$. Note that this claim implies the theorem, since in particular it means that for $\ell = anb - 1$, $z$ satisfies $P(xz) = 1$.

We prove the claim by induction. For $\ell = 0$ this holds vacuously. Now for every $\ell$, if the call $\text{STARTSWITH}(xz_0 \cdots z_{\ell-1}0)$ returns 1 then we are guaranteed the invariant by definition of $\text{STARTSWITH}$. Now under our inductive hypothesis, there is $y_0, \ldots, y_{anb-1}$ such that $P(xz_0, \ldots, z_{\ell-1}y_0, \ldots, y_{anb-1}) = 1$. If the call to $\text{STARTSWITH}(xz_0 \cdots z_{\ell-1}0)$ returns 0 then it must be the case that $y_\ell = 1$, and hence when we set $z_\ell = 1$ we maintain the invariant.

16.2 Quantifier elimination

So, if $P = NP$ then we can solve all $NP$ search problems in polynomial time. But can we do more? Yes we can!

An $NP$ decision problem can be thought of the task of deciding the truth of a statement of the form

$$\exists x P(x)$$  \hspace{1cm} (16.1)

for some NAND program $P$. But we can think of more general statements such as

$$\exists x \forall y P(x, y)$$  \hspace{1cm} (16.2)
or
\[ \exists x \forall y \exists z P(x, y, z). \]  

(16.3)

For example, given an \( n \)-input NAND program \( P \), we might want to find the smallest NAND program \( P' \) that computes the same function as \( P \). The question of whether there is such \( P' \) of size at most \( k \) can be phrased as
\[ \exists p' \forall x |P'| \leq k \land P(x) = P'(x). \]  

(16.4)

It turns out that if \( P = \text{NP} \) then we can solve these kinds of problems as well.

**Theorem 16.2 — Polynomial hierarchy collapse.** If \( P = \text{NP} \) then for every \( a \in \mathbb{N} \) there is a polynomial-time algorithm that on input a NAND program \( P \) on \( an \) inputs, returns 1 if and only if
\[ \exists x_1 \in \{0, 1\}^n \forall x_2 \in \{0, 1\}^n \cdots Q x_a \in \{0, 1\}^n P(x_1, \ldots, x_a) \]  

(16.5)

where \( Q \) is either \( \exists \) or \( \forall \) depending on whether \( a \) is odd or even respectively.

**Proof.** We prove the theorem by induction. We assume that there is a polynomial-time algorithm \( \text{SOLVE}_{a-1} \) that can solve the problem Eq. (16.5) for \( a - 1 \) and use that to solve the problem for \( a \). On input a NAND program \( P \), we will create the NAND program \( S_P \) that on input \( x_1 \in \{0, 1\}^n \), outputs \( 1 - \text{SOLVE}_{a-1}(1 - P_{x_1}) \) where \( P_{x_1} \) is a NAND program that on input \( x_2, \ldots, x_a \in \{0, 1\}^n \) outputs \( P(x_1, \ldots, x_n) \). Now note that by the definition of \( \text{SOLVE} \)
\[ \exists x_1 \in \{0, 1\}^n S_P(x_1) = \]  

\[ \exists x_1 \text{SOLVE}_{a-1}(P_{x_1}) = \]  

\[ \exists x_1 \exists x_2 \cdots Q x_a P(x_1, \ldots, x_a) \]  

(16.6)

Hence we see that if we can solve the satisfiability problem for \( S_P \) then we can solve Eq. (16.5). \( \square \)

This algorithm can also solve the search problem as well: find the value \( x_1 \) that certifies the truth of Eq. (16.5). We note that while this algorithm is polynomial time, the exponent of this polynomial blows up quite fast. If the original NANDSAT algorithm required \( \Omega(n^2) \) solving \( a \) levels of quantifiers would require time \( \Omega(n^{2^a}) \).\(^1\)

\(^1\) We do not know whether such loss is inherent. As far as we can tell, it’s possible that the quantified boolean formula problem has a linear-time algorithm. We will however see later in this course that it satisfies a notion known as \( \text{PSpace} \)-hardness which is even stronger than \( \text{NP} \)-hardness.
16.2.1 Approximating counting problems

Given a NAND program $P$, if $P = \text{NP}$ then we can find an input $x$ (if one exists) such that $P(x) = 1$, but what if there is more than one $x$ like that? Clearly we can’t efficiently output all such $x$’s: there might be exponentially many. But we can get an arbitrarily good multiplicative approximation (i.e., a $1 \pm \epsilon$ factor for arbitrarily small $\epsilon > 0$) for the number of such $x$’s as well as output a (nearly) uniform member of this set. We will defer the details to later in this course, when we learn about randomized computation.

16.3 What does all of this imply?

So, what will happen if we have a $10^6 n$ algorithm for 3SAT? We have mentioned that NP hard problems arise in many contexts, and indeed scientists, engineers, programmers and others routinely encounter such problems in their daily work. A better 3SAT algorithm will probably make their life easier, but that is the wrong place to look for the most foundational consequences. Indeed, while the invention of electronic computers did of course make it easier to do calculations that people were already doing with mechanical devices and pen and paper, the main applications computers are used for today were not even imagined before their invention.

An exponentially faster algorithm for all NP problems would be no less radical improvement (and indeed, in some sense more) than the computer itself, and it is as hard for us to imagine what it would imply, as it was for Babbage to envision today’s world. For starters, such an algorithm would completely change the way we program computers. Since we could automatically find the “best” (in any measure we chose) program that achieves a certain task, we will not need to define how to achieve a task, but only specify tests as to what would be a good solution, and could also ensure that a program satisfies an exponential number of tests without actually running them.

The possibility that $P = \text{NP}$ is often described as “automating creativity” and there is something to that analogy, as we often think of a creative solution, as a solution that is hard to discover, once that “spark” hits, is easy to verify. But there is also an element of hubris to that statement, implying that the most impressive consequence of such an algorithmic breakthrough will be that computers would succeed in doing something that humans already do today. Indeed, as the manager of any mass-market film or music studio will tell
you, creativity already is to a large extent automated, and as in most professions, we should expect to see the need for humans in this process diminish with time even if $P \neq NP$.

Nevertheless, artificial intelligence, like many other fields, will clearly be greatly impacted by an efficient $3\text{SAT}$ algorithm. For example, it is clearly much easier to find a better Chess-playing algorithm, when given any algorithm $P$, you can find the smallest algorithm $P'$ that plays Chess better than $P$. Moreover, much of machine learning (and statistical reasoning in general) is about finding “simple” concepts that explain the observed data, and with $NP = P$, we could search for such concepts automatically for any notion of “simplicity” we see fit. In fact, we could even “skip the middle man” and do an automatic search for the learning algorithm with smallest generalization error. Ultimately the field of Artificial Intelligence is about trying to “shortcut” billions of years of evolution to obtain artificial programs that match (or beat) the performance of natural ones, and a fast algorithm for $NP$ would provide the ultimate shortcut.$^2$

More generally, a faster algorithm for $NP$-problems would be immensely useful in any field where one is faced with computational or quantitative problems, which is basically all fields of science, math, and engineering. This will not only help with concrete problems such as designing a better bridge, or finding a better drug, but also with addressing basic mysteries such as trying to find scientific theories or “laws of nature”. In a fascinating talk physicist Nima Harkani Hamed discusses the effort of finding scientific theories in much the same language as one would describe solving an $NP$ problem, where the solution is easy-to-verify, or seems “inevitable”, once you see it, but to reach it you need to search through a huge landscape of possibilities, and often can get “stuck” at local optima:

```
“the laws of nature have this amazing feeling of inevitability... which is associated with local perfection.”
```

```
“The classical picture of the world is the top of a local mountain in the space of ideas. And you go up to the top and it looks amazing up there and absolutely incredible. And you learn that there is a taller mountain out there. Find it, Mount Quantum... they’re not smoothly connected... you’ve got to make a jump to go from classical to quantum... This also tells you why we have such major challenges in trying to extend our understanding of physics. We don’t have these knobs,
```

$^2$ Some people might claim that, if it indeed holds $P = NP$, then evolution should have already discovered the efficient $3\text{SAT}$ algorithm and perhaps we have to discover this algorithm too if we want to match evolution’s performance. At the moments there seems to be very little evidence for such a scenario. In fact we have some partial results showing that, regardless of whether $P = NP$, many types of “local search” or “evolutionary” algorithms require exponential time to solve $3\text{SAT}$ and other $NP$-hard problems.
Finding an efficient algorithm for NP amounts to always being able to search through an exponential space and find not just the “local” mountain, but the tallest peak.

But perhaps more than any computational speedups, a fast algorithm for NP problems would bring about a new type of understanding. In many of the areas where NP completeness arises, it is not as much a barrier for solving computational problems as a barrier for obtaining “closed form formulas” or other types of a more constructive descriptions of the behavior of natural, biological, social and other systems. A better algorithm for NP, even if it is “merely” only \(2^{\sqrt{n}}\) time, seems to require obtaining a new way to understand these types of systems, whether it is characterizing Nash equilibria, spin-glass configurations, entangled quantum states, of any of the other questions where NP is currently a barrier for analytical understanding. Such new insights would be very fruitful regardless of their computational utility.

16.4 Can \(P \neq \text{NP}\) be neither true nor false?

The Continuum Hypothesis, was a conjecture made by Georg Cantor in 1878, positing the non-existence of a certain type of infinite cardinality.\(^3\) This was considered one of the most important open problems in set theory, and settling its truth or falseness was the first problem put forward by Hilbert in his 1900 address we made before. However, using the developed by Gödel and Turing, in 1963 Paul Cohen proved that both the Continuum Hypothesis and its negation are consistent with the standard axioms of set theory (i.e., the Zermelo-Fraenkel axioms + the Axiom of choice, or “ZFC” for short).\(^4\)

Today many (though not all) mathematicians interpret this result as saying that the Continuum Hypothesis is neither true nor false, but rather is an axiomatic choice that we are free to make one way or the other. Could the same hold for \(P \neq \text{NP}\)?

In short, the answer is No. For example, suppose that we are trying to decide between the “\(3\text{SAT is easy}\)” conjecture (there is an \(10^9n\) time algorithm for \(3\text{SAT}\)) and the “\(3\text{SAT is hard}\)” conjecture (for every \(n\), any NAND program that solves \(n\) variable \(3\text{SAT}\) takes \(2^{10^{-6}n}\)).
lines), then, since for \( n = 10^8 \), \( 2^{10^{-6}n} > 10^6n \), this boils down to the finite question of deciding whether or not there is \( 10^{13} \) line NAND program deciding 3SAT on formulas with \( 10^8 \) variables.

If there is such a program then there is a finite proof of that, namely the proof is the 1TB file describing the program, and the verification is the (finite in principle though infeasible in practice) process of checking that it succeeds on all inputs.\(^5\) If there isn’t such a program then there is also a finite proof of that, though that would take longer since we would need to enumerate over all programs as well. Ultimately, since it boils down to a finite statement about bits and numbers, either the statement or its negation must follow from the standard axioms of arithmetic in a finite number of arithmetic steps. Thus we cannot justify our ignorance in distinguishing between the “3SAT easy” and “3SAT hard” cases by claiming that this might be an inherently ill-defined question. Similar reasoning (with different numbers) applies to other variants of the P vs NP question. We note that in the case that 3SAT is hard, it may well be that there is no short proof of this fact using the standard axioms, and this is a question that people have been studying in various restricted forms of proof systems.

16.5 Is P = NP “in practice”?

The fact that a problem is NP hard means that we believe there is no efficient algorithm that solve it in the worst case. It of course does not mean that every single instance of the problem is hard. For example, if all the clauses in a 3SAT instance \( \varphi \) contain the same variable \( x_i \) (possibly in negated form) then by guessing a value to \( x_i \) we can reduce \( \varphi \) to a 2SAT instance which can then be efficiently solved. Generalizations of this simple idea are used in “SAT solvers” which are algorithms that have solved certain specific interesting SAT formulas with thousands of variables, despite the fact that we believe SAT to be exponentially hard in the worst case. Similarly, a lot of problems arising in machine learning and economics are NP hard. And yet people manage to figure out prices (as economists like to point out, there is milk on the shelves) and distinguish cats from dogs. Hence people (and machines) seem to regularly succeed in solving interesting instances of NP-hard problems, typically by using some combination of guessing while making local improvements.

It is also true that there are many interesting instances of NP hard problems that we do not currently know how to solve. Across all application areas, whether it is scientific computing, optimization,
16.6 What if \( P \neq NP \)?

So \( P = NP \) would give us all kinds of fantastical outcomes. But we strongly suspect that \( P \neq NP \), and in fact that there is no much-better-than-brute-force algorithm for 3SAT. If indeed that is the case, is it all bad news?

One might think that impossibility results, telling you that you cannot do something, is the kind of cloud that does not have a silver lining. But in fact, as we already alluded to before, it does. A hard (in a sufficiently strong sense) problem in \( NP \) can be used to create a code that cannot be broken, a task that for thousands of years has been the dream of not just spies but many scientists and mathematicians over the generations. But the complexity viewpoint turned out to yield much more than simple codes, achieving tasks that people have not even dared to dream about. These include the notion of public key cryptography, allowing two people to communicate securely without ever having exchanged a secret key, electronic cash, allowing private secure transaction without a central authority, and secure multiparty computation, enabling parties to compute a joint function on private inputs without revealing any extra information about it. Also, as we will see, computational hardness can be used to replace the role of randomness in many settings.

Furthermore, while it is often convenient to pretend that computational problems are simply handed to us, and our job as computer scientists is to find the most efficient algorithm for them, this is not how things work in most computing applications. Typically even

control or more, people often encounter hard instances of \( NP \) problems on which our current algorithms fail. In fact, as we will see, all of our digital security infrastructure relies on the fact that some concrete and easy-to-generate instances of, say, 3SAT (or, equivalently, any other \( NP \) hard problem) are exponentially hard to solve.

Thus it would be wrong to say that \( NP \) is easy “in practice”, nor would it be correct to take \( NP \)-hardness as the “final word” on the complexity of a problem, particularly when we have more information on where our instances arise from. Understanding both the “typical complexity” of \( NP \) problems, as well as the power and limitations of certain heuristics (such as various local-search based algorithms) is a very active area of research. We will see more on these topics later in this course.

\footnote{Talk more about coping with \( NP \) hardness. Main two approaches are \textit{heuristics} such as SAT solvers that succeed on some instances, and \textit{proxy measures} such as mathematical relaxations that instead of solving problem \( X \) (e.g., an integer program) solve program \( X' \) (e.g., a linear program) that is related to that. Maybe give compressed sensing as an example, and least square minimization as a proxy for maximum a posteriori probability.}
formulating the problem to solve is a highly non-trivial task. When we discover that the problem we want to solve is NP-hard, this might be a useful sign that we used the wrong formulation for it.

Beyond all these, the quest to understand computational hardness, including the discoveries of lower bounds for restricted computational models, as well as new types of reductions (such as those arising from “probabilistically checkable proofs”), already had surprising positive applications to problems in algorithm design, as well as coding for both communication and storage. This is not surprising since, as we mentioned before, from group theory to the theory of relativity, the pursuit of impossibility results has often been one of the most fruitful enterprises of mankind.

16.7 Lecture summary

- The question of whether $P = NP$ is one of the most important and fascinating questions of computer science and science at large, touching on all fields of the natural and social sciences, as well as mathematics and engineering.

- Our current evidence and understanding supports the “SAT hard” scenario that there is no much-better-than-brute-force algorithm for 3SAT and many other NP-hard problems.

- We are very far from proving this however, and we will discuss some of the efforts in this direction later in this course.

- Understanding how to cope with this computational intractability, and even benefit from it, comprises much of the research in theoretical computer science.

16.8 Exercises

16.9 Bibliographical notes

7 TODO: Scott’s two surveys

16.10 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)
16.11 Acknowledgements
17

Probability Theory 101

Before we show how to use randomness in algorithms, let us do a quick review of some basic notions in probability theory. This is not meant to replace a course on probability theory, and if you have not seen this material before, I highly recommend you look at additional resources to get up to speed. Fortunately, we will not need many of the advanced notions of probability theory, but, as we will see, even the so called “simple” setting of tossing $n$ coins can lead to very subtle and interesting issues.

17.1 Random coins

The nature of randomness and probability is a topic of great philosophical, scientific and mathematical depth. Is there actual randomness in the world, or does it proceed in a deterministic clockwork fashion from some initial conditions set at the beginning of time? Does probability refer to our uncertainty of beliefs, or to the frequency of occurrences in repeated experiments? How can we define probability over infinite sets?

These are all important questions that have been studied and debated by scientists, mathematicians, statisticians and philosophers. Fortunately, we will not need to deal directly with these questions here. We will be mostly interested in the setting of tossing $n$ random, unbiased and independent coins. Below we define the basic probabilistic objects of events and random variables when restricted to this setting. These can be defined for much more general probabilistic experiments or sample spaces, and later on we will briefly discuss how this can be done, though the $n$-coin case is sufficient for almost everything we’ll need in this course.
If instead of “heads” and “tails” we encode the sides of each coin by “zero” and “one”, we can encode the result of tossing $n$ coins as a string in $\{0, 1\}^n$. Each particular outcome $x \in \{0, 1\}^n$ is obtained with probability $2^{-n}$. For example, if we toss three coins, then we obtain each of the 8 outcomes 000, 001, 010, 011, 100, 101, 110, 111 with probability $2^{-3} = 1/8$. We can also describe this experiment as choosing $x$ uniformly at random from $\{0, 1\}^n$, and hence we’ll use the shorthand $x \sim \{0, 1\}^n$ for it.

An event is simply a subset $A$ of $\{0, 1\}^n$. The probability of $A$, denoted by $\mathbb{P}_{x \sim \{0, 1\}^n}[A]$ (or $\mathbb{P}[A]$ for short, when the sample space is understood from the context), is the probability that a random $x$ chosen uniformly at random will be contained in $A$. Note that this is the same as $|A|/2^n$. For example, the probability that $x$ has an even number of ones is $\mathbb{P}[A]$ where $A = \{ x : \sum_i x_i = 0(\mod 2) \}$. Let us calculate this probability:

\textbf{Lemma 17.1}

$$
\mathbb{P}_{x \sim \{0, 1\}^n}[\sum_i x_i \text{ is even}] = 1/2
$$

\textbf{Proof.} Let $A = \{ x \in \{0, 1\}^n : \sum_i x_i = 0(\mod 2) \}$. Since every $x$ is obtained with probability $2^{-n}$, to show this we need to show that $|A| = 2^n/2 = 2^{n-1}$. For every $x_0, \ldots, x_{n-2}$, if $\sum_{i=0}^{n-2} x_i$ is even then $(x_0, \ldots, x_{n-1}, 0) \in A$ and $(x_0, \ldots, x_{n-1}, 1) \not\in A$. Similarly, if $\sum_{i=0}^{n-2} x_i$ is odd then $(x_0, \ldots, x_{n-1}, 1) \in A$ and $(x_0, \ldots, x_{n-1}, 0) \not\in A$. Hence, for every one of the $2^{n-1}$ prefixes $(x_0, \ldots, x_{n-2})$, there is exactly a single continuation of $(x_0, \ldots, x_{n-2})$ that places it in $A$. \hfill \blacksquare

We can also use the \textit{intersection} ($\cap$) and \textit{union} ($\cup$) operators to talk about the probability of both event $A$ \textit{and} event $B$ happening, or the probability of event $A$ \textit{or} event $B$ happening. For example, the probability $p$ that $x$ is has an even number of ones \textit{and} $x_0 = 1$ is the same as $\mathbb{P}[A \cap B]$ where $A = \{ x \in \{0, 1\}^n : \sum_i x_i = 0(\mod 2) \}$ and $B = \{ x \in \{0, 1\}^n : x_0 = 1 \}$. This probability is equal to 1/4: can you see why? Because intersection corresponds to considering the logical AND of the conditions that two events happen, while union corresponds to considering the logical OR, we will sometimes use the $\wedge$ and $\vee$ operators instead of $\cap$ and $\cup$, and so write this probability $p$ above also as

$$
\mathbb{P}_{x \sim \{0, 1\}^n}[\sum_i x_i = 0(\mod 2) \wedge x_0 = 1].
$$

If $A \subseteq \{0, 1\}^n$ is an event, then $\overline{A} = \{0, 1\}^n \setminus A$ corresponds to the event that $A$ does \textit{not} happen. Note that $\mathbb{P}[\overline{A}] = 1 - \mathbb{P}[A]$; can you see why?
17.1.1 Random variables

A random variable is a function \( X : \{0, 1\}^n \to \mathbb{R} \) that maps every outcome \( x \in \{0, 1\}^n \) to a real number \( X(x) \). For example, the sum of the \( x_i \)'s is a random variable. The expectation of a random variable \( X \), denoted by \( \mathbb{E}[X] \), is the average value that it will receive. That is,

\[
\mathbb{E}[X] = \sum_{x \in \{0,1\}^n} 2^{-n}X(x). \tag{17.3}
\]

If \( X \) and \( Y \) are random variables, then we can define \( X + Y \) as simply the random variable maps \( x \) to \( X(x) + Y(x) \). One of the basic and useful properties of the expectation is that it is linear:

**Lemma 17.2 — Linearity of expectation.**

\[
\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] \tag{17.4}
\]

**Proof.**

\[
\mathbb{E}[X + Y] = \sum_{x \in \{0,1\}^n} 2^{-n} (X(x) + Y(x)) = \\
\sum_{x \in \{0,1\}^n} 2^{-n}X(x) + \sum_{x \in \{0,1\}^n} 2^{-n}Y(x) = \\
\mathbb{E}[X] + \mathbb{E}[Y] \tag{17.5}
\]

Similarly, \( \mathbb{E}[kX] = k \mathbb{E}[X] \) for every \( k \in \mathbb{R} \). For example, using the linearity of expectation, it is very easy to show that the expectation of the sum of the \( x_i \)'s for \( x \sim \{0, 1\}^n \) is equal to \( n/2 \). Indeed, if we write \( X = \sum_{i=0}^{n-1} x_i \) then \( X = X_0 + \cdots + X_{n-1} \) where \( X_i \) is the random variable \( x_i \). Since for every \( i \), \( \mathbb{P}[X_i = 0] = 1/2 \) and \( \mathbb{P}[X_i = 1] = 1/2 \), we get that \( \mathbb{E}[X_i] = (1/2) \cdot 0 + (1/2) \cdot 1 = 1/2 \) and hence \( \mathbb{E}[X] = \sum_{i=0}^{n-1} \mathbb{E}[X_i] = n \cdot (1/2) = n/2 \). (If you have not seen discrete probability before, please go over this argument again until you are sure you follow it, it is a prototypical simple example of the type of reasoning we will employ again and again in this course.)

If \( A \) is an event, then \( 1_A \) is the random variable such that \( 1_A(x) \) equals 1 if \( x \in A \) and \( 1_A(x) = 0 \) otherwise. Note that \( \mathbb{P}[A] = \mathbb{E}[1_A] \) (can you see why?). Using this and the linearity of expectation, we can show one of the most useful bounds in probability theory:

**Lemma 17.3 — Union bound.** For every two events \( A, B \), \( \mathbb{P}[A \cup B] \leq \mathbb{P}[A] + \mathbb{P}[B] \)
Proof. For every $x$, the variable $1_{A \cup B}(x) \leq 1_A(x) + 1_B(x)$. Hence, $\mathbb{P}[A \cup B] = \mathbb{E}[1_{A \cup B}] \leq \mathbb{E}[1_A + 1_B] = \mathbb{E}[1_A] + \mathbb{E}[1_B] = \mathbb{P}[A] + \mathbb{P}[B]$. ■

The way we often use this in theoretical computer science is to argue that, for example, if there is a list of 1000 bad events that can happen, and each one of them happens with probability at most $1/10000$, then with probability at least $1 - 1000/10000 \geq 0.9$, no bad event happens.

17.1.2 More general sample spaces.

While in this lecture we assume that the underlying probabilistic experiment corresponds to tossing $n$ independent coins, everything we say easily generalizes to sampling $x$ from a more general finite set $S$ (and not-so-easily generalizes to infinite sets $S$ as well). A probability distribution over a finite set $S$ is simply a function $\mu : S \to [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$. We think of this as the experiment where we obtain every $x \in S$ with probability $\mu(s)$, and sometimes denote this as $x \sim \mu$. An event $A$ is a subset of $S$, and the probability of $A$, which we denote by $\mathbb{P}_\mu[A]$ is $\sum_{x \in A} \mu(x)$. A random variable is a function $X : S \to \mathbb{R}$, where the probability that $X = y$ is equal to $\sum_{x \in S \mid X(x) = y} \mu(x)$.

17.2 Correlations and independence

One of the most delicate but important concepts in probability is the notion of independence (and the opposing notion of correlations). Subtle correlations are often behind surprises and errors in probability and statistical analysis, and several mistaken predictions have been blamed on miscalculating the correlations between, say, housing prices in Florida and Arizona, or voter preferences in Ohio and Michigan. See also Joe Blitzstein’s aptly named talk “Conditioning is the Soul of Statistics”.

Two events $A$ and $B$ are independent if the fact that $A$ happened does not make $B$ more or less likely to happen. For example, if we think of the experiment of tossing 3 random coins $x \in \{0, 1\}^3$, and let $A$ be the event that $x_0 = 1$ and $B$ the event that $x_0 + x_1 + x_2 \geq 2$, then if $A$ happens it is more likely that $B$ happens, and hence these events are not independent. On the other hand, if we let $C$ be the event that

Todo: add exercise on simulating die tosses and choosing a random number in $[m]$ by coin tosses

Note: Another thorny issue is of course the difference between correlation and causation. Luckily, this is another point we don’t need to worry about in our clean setting of tossing $n$ coins.
$x_1 = 1$, then because the second coin toss is not affected by the result of the first one, the events $A$ and $C$ are independent.

Mathematically, we say that events $A$ and $B$ are independent if

$P[A \cap B] = P[A]P[B]$. If $P[A \cap B] > P[A]P[B]$ then we say that $A$ and $B$ are positively correlated, while if $P[A \cap B] < P[A]P[B]$ then we say that $A$ and $B$ are negatively correlated.

If we consider the above examples on the experiment of choosing $x \in \{0, 1\}^3$ then we can see that

$$P[x_0 = 1] = 1/2$$
$$P[x_0 + x_1 + x_2 \geq 2] = P[\{011, 101, 110, 111\}] = 4/8 = 1/2$$

but

$$P[x_0 = 1 \land x_0 + x_1 + x_2 \geq 2] = P[\{101, 110, 111\}] = 3/8 > (1/2)(1/2)$$

and hence, as we already observed, the events $\{x_0 = 1\}$ and $\{x_0 + x_1 + x_2 \geq 2\}$ are not independent and in fact positively correlated.

On the other hand $P[x_0 = 1 \land x_1 = 1] = P[\{110, 111\}] = 2/8 = (1/2)(1/2)$ and hence the events $\{x_0 = 1\}$ and $\{x_1 = 1\}$ are indeed independent.

**Conditional probability:** If $A$ and $B$ are events, and $A$ happens with nonzero probability then we define the probability that $B$ happens conditioned on $A$ to be $P[B|A] = P[A \cap B] / P[A]$. This corresponds to calculating the probability that $B$ happened if we already know that $A$ happened. Note that $A$ and $B$ are independent if and only if $P[B|A] = P[B]$.

**More than two events:** We can generalize this definition to more than two events. We say that events $A_1, \ldots, A_k$ are mutually independent if knowing that any set of them occurred or didn’t occur does not change the probability that an event outside the set occurs. Formally the condition is that for every subset $I \subseteq [k]$,

$$P[\land_{i \in I} A_i] = \prod_{i \in I} P[A_i]$$

For example, if $x \sim \{0, 1\}^3$, then the events $\{x_0 = 1\}$, $\{x_1 = 1\}$ and $\{x_2 = 1\}$ are mutually independent. On the other hand, the events $\{x_0 = 1\}$, $\{x_1 = 1\}$ and $\{x_0 + x_1 = 0(\mod 2)\}$ are not mutually independent even though every pair of these events is independent (can you see why?).
17.2.1 Independent random variables

We say that two random variables \(X\) and \(Y\) are independent if for every \(u, v \in \mathbb{R}\), the events \(\{X = u\}\) and \(\{Y = t\}\) are independent. That is, \(P[X = u \land Y = t] = P[X = u]P[Y = t]\). For example, if two random variables depend on the result of tossing different coins then they are independent:

**Lemma 17.4** Suppose that \(S = \{s_1, \ldots, s_k\}\) and \(T = \{t_1, \ldots, t_m\}\) are disjoint subsets of \(\{0, \ldots, n - 1\}\) and let \(X, Y : \{0, 1\}^n \to \mathbb{R}\) be random variables such that \(X = F(x_{s_1}, \ldots, x_{s_k})\) and \(Y = G(x_{t_1}, \ldots, x_{t_m})\) for some functions \(F : \{0, 1\}^k \to \mathbb{R}\) and \(G : \{0, 1\}^m \to \mathbb{R}\). Then \(X\) and \(Y\) are independent.

**Proof.** Let \(a, b \in \mathbb{R}\), and let \(A = \{x \in \{0, 1\}^k : F(x) = a\}\) and \(B = \{x \in \{0, 1\}^m : F(x) = b\}\). Since \(S\) and \(T\) are disjoint, we can reorder the indices so that \(S = \{0, \ldots, k - 1\}\) and \(T = \{k, \ldots, k + m - 1\}\) without affecting any of the probabilities. Hence we can write \(P[X = a \land X = b] = |C|/2^n\) where \(C = \{x_0, \ldots, x_{n-1} : (x_0, \ldots, x_{k-1}) \in A \land (x_k, \ldots, x_{k+m-1}) \in B\}\). Another way to write this using string concatenation is that \(C = \{xyz : x \in A, y \in B, z \in \{0, 1\}^{n-k-m}\}\), and hence \(|C| = |A||B|2^{n-k-m}\), which means that

\[
\frac{|C|}{2^n} = \frac{|A|}{2^k} \frac{|B|}{2^m} 2^{n-k-m} = P[X = a]P[Y = b] \quad (17.9)
\]

Note that if \(X\) and \(Y\) are independent then

\[
\mathbb{E}[XY] = \sum_{a,b} P[X = a \land Y = b]ab = \sum_a P[X = a]P[Y = b]ab = \left(\sum_a P[X = a]a\right) \left(\sum_b P[Y = b]b\right) = (17.10)
\]

\[
\mathbb{E}[X] \mathbb{E}[Y]
\]

(This is not an “if and only if”, see Exercise 17.2.)

If \(X\) and \(Y\) are independent random variables then so are \(F(X)\) and \(G(Y)\) for every functions \(F, G : \mathbb{R} \to \mathbb{R}\). This is intuitively true since learning \(F(X)\) can only provide us with less information than learning \(X\). Hence, if learning \(X\) does not teach us anything about \(Y\) (and so also about \(F(Y)\)) then neither will learning \(F(X)\). Indeed, to
prove this we can write

\[
P[F(X) = a \land G(Y) = b] = \sum_{x \text{ s.t. } F(x) = a, y \text{ s.t. } G(y) = b} P[X = x \land Y = y] = \sum_{x \text{ s.t. } F(x) = a} \sum_{y \text{ s.t. } G(y) = b} P[X = x] P[Y = y] = \left( \sum_{x \text{ s.t. } F(x) = a} P[X = x] \right) \cdot \left( \sum_{y \text{ s.t. } G(y) = b} P[Y = y] \right) = P[F(X) = a] P[G(Y) = b] \tag{17.11}\]

17.2.2 Collections of independent random variables.

We can extend the notions of independence to more than two random variables. We say that the random variables \(X_0, \ldots, X_{n-1}\) are mutually independent if for every \(a_0, \ldots, a_{n-1}\) then

\[
P[X_0 = a_0 \land \cdots \land X_{n-1} = a_{n-1}] = P[X_0 = a_0] \cdots P[X_{n-1} = a_{n-1}] \tag{17.12}\]

and similarly we have that

\[
\text{Lemma 17.5 — Expectation of product of independent random variables.} \quad \text{If} \quad X_0, \ldots, X_{n-1} \text{ are mutually independent then}
\]

\[
\mathbb{E}[\prod_{i=0}^{n-1} X_i] = \prod_{i=0}^{n-1} \mathbb{E}[X_i] \tag{17.13}
\]

\[
\text{Lemma 17.6 — Functions preserve independence.} \quad \text{If} \quad X_0, \ldots, X_{n-1} \text{ are mutually independent, and} \quad Y_0, \ldots, Y_{n-1} \text{ are defined as} \quad Y_i = F_i(X_i) \text{ for some functions} \quad F_0, \ldots, F_{n-1} : \mathbb{R} \rightarrow \mathbb{R}, \text{ then} \quad Y_0, \ldots, Y_{n-1} \text{ are mutually independent as well.}
\]

We leave proving Lemma 17.5 and Lemma 17.6 as Exercise 17.3. Exercise 17.4. It is good idea for you stop now and do these exercises to make sure you are comfortable with the notion of independence, as we will use it heavily later on in this course.

17.3 Concentration

The name “expectation” is somewhat misleading. For example, suppose that I place a bet on the outcome of 10 coin tosses, where if they all come out to be 1’s then I pay you 100000 dollars and
otherwise you pay me 10 dollars. If we let $X : \{0, 1\}^{10} \to \mathbb{R}$ be the random variable denoting your gain, then we see that

$$E[X] = 2^{-10} \cdot 100000 - (1 - 2^{-10})10 \sim 90 \quad (17.14)$$

but we don’t really “expect” the result of this experiment to be for you to gain 90 dollars. Rather, 99.9% of the time you will pay me 10 dollars, and you will hit the jackpot 0.01% of the times.

However, if we repeat this experiment again and again (with fresh and hence independent coins), then in the long run we do expect your average earning to be 90 dollars, which is the reason why casinos can make money in a predictable way even though every individual bet is random. For example, if we toss $n$ coins, then as $n$ grows the number of coins that come up ones will be more and more concentrated around $n/2$ according to the famous “bell curve” (see Fig. 17.1).

![Figure 17.1](image)

Figure 17.1: The probabilities we obtain a particular sum when we toss $n = 10, 20, 100, 1000$ coins converge quickly to the Gaussian/normal distribution.

Much of probability theory is concerned with so called concentration or tail bounds, which are upper bounds on the probability that a random variable $X$ deviates too much from its expectation. The first and simplest one of them is Markov’s inequality:

**Theorem 17.7 — Markov’s inequality.** If $X$ is a non-negative random variable then $P[X \geq kE[X]] \leq 1/k.$
Proof. Let $\mu = \mathbb{E}[X]$ and define $Y = 1_{X \geq k\mu}$. That is, $Y(x) = 1$ if $X(x) \geq k\mu$ and $Y(x) = 0$ otherwise. Note that by definition, for every $x$, $Y(x) \leq X/(k\mu)$. We need to show $\mathbb{E}[Y] \leq 1/k$. But this follows since $\mathbb{E}[Y] \leq \mathbb{E}[X/k(\mu)] = \mathbb{E}[X]/(k\mu) = \mu/(k\mu) = 1/k$. ■

Markov’s inequality says that a (non-negative) random variable $X$ can’t go too crazy and be, say, a million times its expectation, with significant probability. But ideally we would like to say that with high probability $X$ should be very close to its expectation, e.g., in the range $[0.99\mu, 1.01\mu]$ where $\mu = \mathbb{E}[X]$. This is not generally true, but does turn out to hold when $X$ is obtained by combining (e.g., adding) many independent random variables. This phenomena, variants of which are known as “law of large numbers”, “central limit theorem”, “invariance principles” and “Chernoff bounds”, is one of the most fundamental in probability and statistics, and one that we heavily use in computer science as well.

### 17.4 Chebyshev’s Inequality

A standard way to measure the deviation of a random variable from its expectation is using its standard deviation. For a random variable $X$, we define the variance of $X$ as $\text{Var}[X] = \mathbb{E}[X - \mu]^2$ where $\mu = \mathbb{E}[X]$, i.e., the variance is the average square distance of $X$ from its expectation. The standard deviation of $X$ is defined as $\sigma[X] = \sqrt{\text{Var}[X]}$.

Using Markov’s inequality we can control the probability that a random variable is too many standard deviations away from its expectation.

**Theorem 17.8 — Chebyshev’s inequality.** Suppose that $\mu = \mathbb{E}[X] = \mu$ and $\sigma^2 = \text{Var}[X]$. Then for every $k > 0$, $\mathbb{P}[|X - \mu| \geq k\sigma] \leq 1/k^2$.

*Proof.* The proof follows from Markov’s inequality. We define the random variable $Y = (X - \mu)^2$. Then $\mathbb{E}[Y] = \text{Var}[X] = \sigma^2$, and hence by Markov the probability that $Y > k^2\sigma^2$ is at most $1/k^2$. But clearly $(X - \mu)^2 \geq k^2\sigma^2$ if and only if $|X - \mu| \geq k\sigma$. ■

One example of how to use Chebyshev’s inequality is the setting when $X = X_1 + \cdots + X_n$ where $X_i$’s are independent and identically distributed (i.i.d for short) variables with values in $[0, 1]$ where each has expectation $1/2$. Since $\mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = n/2$, we would like to say that $X$ is very likely to be in, say, the interval $[0.499n, 0.501n]$. 

Using Markov’s inequality directly will not help us, since it will only tell us that $X$ is very likely to be at most $100n$ (which we already knew, since it always lies between 0 and $n$). However, since $X_1, \ldots, X_n$ are independent,

$$\text{Var}[X_1 + \cdots + X_n] = \text{Var}[X_1] + \cdots + \text{Var}[X_n].$$  \hspace{1cm} (17.15)

(We leave showing this to the reader as Exercise 17.5.)

For every random variable $X_i$ in $[0, 1]$, $\text{Var}[X_i] \leq 1$ (if the variable is always in $[0, 1]$, it can’t be more than 1 away from its expectation), and hence Eq. (17.15) implies that $\text{Var}[X] \leq n$ and hence $\sigma[X] \leq \sqrt{n}$. For large $n$, $\sqrt{n} \ll 0.01n$, and in particular if $\sqrt{n} \leq 0.01n/k$, we can use Chebyshev’s inequality to bound the probability that $X$ is not in $[0.499n, 0.501n]$ by $1/k^2$.

### 17.5 The Chernoff bound

Chebyshev’s inequality already shows a connection between independence and concentration, but in many cases we can hope for a quantitatively much stronger result. If, as in the example above, $X = X_1 + \cdots + X_n$ where the $X_i$’s are bounded i.i.d random variables of mean $1/2$, then as $n$ grows, the distribution of $X$ would be roughly the normal or Gaussian distribution, that is distributed according to the bell curve. This distribution has the property of being very concentrated in the sense that the probability of deviating $k$ standard deviation is not merely $1/k^2$ as is guaranteed by Chebyshev, but rather it is roughly $e^{-k^2}$. That is we have an exponential decay of the probability of deviation. This is stated by the following theorem, that is known under many names in different communities, though it is mostly called the “Chernoff bound” in the computer science literature:

**Theorem 17.9 — Chernoff bound.** If $X_1, \ldots, X_n$ are i.i.d random variables such that $X_i \in [0, 1]$ and $\mathbb{E}[X_i] = p$ for every $i$, then for every $\epsilon > 0$

$$\mathbb{P}\left[\left|\sum_{i=0}^{n-1} X_i - pn\right| > \epsilon n\right] \leq 2 \exp\left(-\frac{\epsilon^2 n}{2}\right) \hspace{1cm} (17.16)$$

We omit the proof, which appears in many texts, and uses Markov’s inequality on i.i.d random variables $Y_0, \ldots, Y_n$ that are of the form $Y_i = e^{\lambda X_i}$ for some carefully chosen parameter $\lambda$. See Exercise 17.8 for a proof of the simple (but highly useful and representative) case where each $X_i$ is $\{0, 1\}$ valued and $p = 1/2$. (See also Exercise 17.9 for a generalization.)
17.6 Lecture summary

- A basic probabilistic experiment corresponds to tossing \( n \) coins or choosing \( x \) uniformly at random from \( \{0, 1\}^n \).
- Random variables assign a real number to every result of a coin toss. The expectation of a random variable \( X \), is its average value, and there are several concentration results showing that under certain conditions random variables deviate significantly from their expectation only with small probability.

17.7 Exercises

Exercise 17.1 Give an example of random variables \( X, Y : \{0, 1\}^3 \to \mathbb{R} \) such that \( \mathbb{E}[XY] \neq \mathbb{E}[X] \mathbb{E}[Y] \).  

Exercise 17.2 Give an example of random variables \( X, Y : \{0, 1\}^3 \to \mathbb{R} \) such that \( X \) and \( Y \) are not independent but \( \mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y] \).  

Exercise 17.3 — Product of expectations. Prove Lemma 17.5  

Exercise 17.4 — Transformations preserve independence. Prove Lemma 17.6  

Exercise 17.5 — Variance of independent random variables. Prove that if \( X_0, \ldots, X_{n-1} \) are independent random variables then \( \text{Var}[X_0 + \cdots + X_{n-1}] = \sum_{i=0}^{n-1} \text{Var}[X_i] \).  

Exercise 17.6 — Entropy (challenge). Recall the definition of a distribution \( \mu \) over some finite set \( S \). Shannon defined the entropy of a distribution \( \mu \), denoted by \( H(\mu) \), to be \( \sum_{x \in S} \mu(x) \log(1/\mu(x)) \). The idea is that if \( \mu \) is a distribution of entropy \( k \), then encoding members of \( \mu \) will require \( k \) bits, in an amortized sense. In this exercise we justify this definition. Let \( \mu \) be such that \( H(\mu) = k \).

1. Prove that for every one to one function \( F : S \to \{0, 1\}^* \), \( \mathbb{E}_{X \sim \mu} |F(x)| \geq k \).

2. Prove that for every \( \epsilon \), there is some \( n \) and a one-to-one function \( F : S^n \to \{0, 1\}^* \), such that \( \mathbb{E}_{x \sim \mu^n} |F(x)| \leq n(k + \epsilon) \), where \( x \sim \mu \) denotes the experiments of choosing \( x_0, \ldots, x_{n-1} \) each independently from \( S \) using the distribution \( \mu \).  

Exercise 17.7 — Entropy approximation to binomial. Let \( H(p) = p \log(1/p) + (1-p) \log(1/(1-p)) \). Prove that for every \( p \in (0, 1) \)

\footnote{While you don’t need this to solve this exercise, this is the function that maps \( p \) to the entropy (as defined in Exercise 17.6) of the \( p \)-biased coin distribution over \( \{0, 1\} \), which is the function \( \mu : \{0, 1\} \to [0, 1] \) s.t. \( \mu(0) = 1 - p \) and \( \mu(1) = p \).}
and $\epsilon > 0$, if $n$ is large enough then
\[
2^{(H(p) - \epsilon)n} \left( \frac{n}{pn} \right) \leq 2^{(H(p) + \epsilon)n}
\]
(17.17)

where $\binom{n}{k}$ is the binomial coefficient $\frac{n!}{k!(n-k)!}$ which is equal to the number of $k$-size subsets of $\{0, \ldots, n-1\}$.

**Exercise 17.8 — Chernoff using Stirling.**

1. Prove that $\mathbb{P}_{x \sim \{0,1\}^n} \left[ \sum x_i = k \right] = \binom{n}{k}2^{-n}$.

2. Use this and **Exercise 17.7** to prove the Chernoff bound for the case that $X_0, \ldots, X_n$ are i.i.d. random variables over $\{0,1\}$ each equaling 0 and 1 with probability 1/2.

**Exercise 17.9 — Poor man’s Chernoff.** Let $X_0, \ldots, X_n$ be i.i.d random variables with $\mathbb{E}X_i = p$ and $\mathbb{P}[0 \leq X_i \leq 1] = 1$. Define $Y_i = X_i - p$.

1. Prove that for every $j_1, \ldots, j_n \in \mathbb{N}$, if there exists one $i$ such that $j_i$ is odd then $\mathbb{E}[\prod_{i=0}^{n-1} Y_i] = 0$.

2. Prove that for every $k$, $\mathbb{E}[\left(\sum_{i=0}^{n-1} Y_i\right)^k] \leq (10kn)^{k/2}7$.

3. Prove that for every $\epsilon > 0$, $\mathbb{P}\left[ \left| \sum Y_i \right| \geq \epsilon n \right] \geq 2^{-\epsilon^2 n / (10000 \log 1 / \epsilon)}$.

**Exercise 17.10 — Simulating distributions using coins.** Our model for probability involves tossing $n$ coins, but sometimes algorithm require sampling from other distributions, such as selecting a uniform number in $\{0, \ldots, M-1\}$ for some $M$. Fortunately, we can simulate this with an exponentially small probability of error: prove that for every $M$, if $n > k \log M$, then there is a function $F : \{0,1\}^n \to \{0, \ldots, M-1\} \cup \{\perp\}$ such that (1) The probability that $F(x) = \perp$ is at most $2^{-k}$ and (2) the distribution of $F(x)$ conditioned on $F(x) \neq \perp$ is equal to the uniform distribution over $\{0, \ldots, M-1\}$.

**Exercise 17.11 — Sampling.** Suppose that a country has 300,000,000 citizens, 52 percent of them prefer the color “green” and 48 percent of which prefer the color “orange”. Suppose we sample $n$ random citizens and ask them their favorite color (assume they will answer truthfully). What is the smallest value $n$ among the following choices so that the probability that the majority of the sample answers “green” is at most 0.05? a. 1,000 b. 10,000 c. 100,000 d. 1,000,000

**Exercise 17.12** Would the answer to **Exercise 17.11** change if the country had 300,000,000,000 citizens?

**Exercise 17.13 — Sampling (2).** Under the same assumptions as **Exercise 17.11**, what is the smallest value $n$ among the following choices

---

6 Hint: Use Stirling’s formula for approximating the factorial function.

7 Hint: Bound the number of tuples $j_0, \ldots, j_{n-1}$ such that every $j_i$ is even and $\sum j_i = k$.

8 Hint: Set $k = \lceil \epsilon^2 n / 1000 \rceil$ and then show that if the event $|\sum Y_i| \geq \epsilon n$ happens then the random variable $(\sum Y_i)^k$ is a factor of $e^{-1}$ larger than its expectation.

9 Hint: Think of $x \in \{0,1\}^n$ as choosing $k$ numbers $y_1, \ldots, y_k \in \{0, \ldots, 2\log M - 1\}$. Output the first such number that is in $\{0, \ldots, M-1\}$.
so that the probability that the majority of the sample answers “green” is at most $2^{-100}$? a. 1,000 b. 10,000 c. 100,000 d. 1,000,000 e. It is impossible to get such low probability since there are fewer than $2^{100}$ citizens.

10

17.8 Bibliographical notes

17.9 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

17.10 Acknowledgements
In early computer systems, much effort was taken to drive out randomness and noise. Hardware components were prone to nondeterministic behavior from a number of causes, whether it is vacuum tubes overheating or actual physical bugs causing short circuits (see Fig. 18.1). This motivated John von Neumann, one of the early computing pioneers, to write a paper on how to error correct computation, introducing the notion of redundancy.

So it is quite surprising that randomness turned out not just a hindrance but also a resource for computation, enabling to achieve tasks much more efficiently than previously known. One of the first applications involved the very same John von Neumann. While he was sick in bed and playing cards, Stan Ulam came up with the observation that calculating statistics of a system could be done much faster by running several randomized simulations. He mentioned this idea to von Neumann, who became very excited about it, as indeed it turned out to be crucial for the neutron transport calculations that were needed for development of the Atom bomb and later on the
hydrogen bomb. Because this project was highly classified, Ulam, von Neumann and their collaborators came up with the codeword “Monte Carlo” for this approach (based on the famous casinos where Ulam’s uncle gambled). The name stuck, and probabilistic algorithms are known as Monte Carlo algorithms to this day.¹

In this lecture, we will see some examples of probabilistic algorithms that use randomness to compute a quantity in a faster or simpler way than was known otherwise. We will describe the algorithms in an informal / “pseudo-code” way, rather than as NAND or NAND++ programs. In the next lecture we will discuss how to augment the NAND and NAND++ models to incorporate the ability to “toss coins”.

### 18.1 Finding approximately good maximum cuts.

Now that we have reviewed the basics of probability, let us see how we can use randomness to achieve algorithmic tasks. We start with the following example. Recall the maximum cut problem, of finding, given a graph $G = (V, E)$, the cut that maximizes the number of edges. This problem is NP-hard, which means that we do not know of any efficient algorithm that can solve it, but randomization enables a simple algorithm that can cut at least half of the edges:

---

¹ Some texts also talk about “Las Vegas algorithms” that always return the right answer but whose running time is only polynomial on the average. Since this Monte Carlo vs Las Vegas terminology is confusing, we will not use these terms anymore, and simply talk about probabilistic algorithms.
**Theorem 18.1 — Approximating max cut.** There is an efficient probabilistic algorithm that on input an $n$-vertex $m$-edge graph $G$, outputs a set $S$ such that the expected number of edges cut is at least $m/2$.

*Proof.* The algorithm is extremely simple: we choose $x$ uniformly at random in $\{0, 1\}^n$ and let $S$ be the set corresponding to $\{i : x_i = 1\}$. For every edge $e$, we let $X_e$ be the random variable such that $X_e(x) = 1$ if the edge $e$ is cut by $x$, and $X_e(x) = 0$ otherwise. For every edge $e = \{i, j\}$, $X_e(x) = 1$ if and only if $x_i \neq x_j$. Since the pair $(x_i, x_j)$ obtains each of the values 00, 01, 10, 11 with probability 1/4, the probability that $x_i \neq x_j$ is 1/2. Hence, $\mathbb{E}[X_e] = 1/2$ and if we let $X = \sum_e X_e$ over all the edges in the graph then $\mathbb{E}[X] = m(1/2) = m/2$.  

18.1.1 Amplification

Theorem 18.1 gives us an algorithm that cuts $m/2$ edges in expectation. But, as we saw before, expectation does not immediately imply concentration, and so a priori, it may be the case that when we run the algorithm, most of the time we don’t get a cut matching the expectation. Luckily, we can amplify the probability of success by repeating the process several times and outputting the best cut we find. Let $p$ be the probability that we cut at least $m/2$ edges. We claim that $p \geq 1/m$. Indeed, otherwise the expected number of edges cut would be at most

$$pm + (1 - p)(m/2 - 1) \leq pm + m/2 - 1 = m/2 + pm - 1 < m/2.$$  

(18.1)

So, if we repeat this experiment, for example, $1000m$ times, then the probability that we will never be able to cut at least $m/2$ edges is at most

$$(1 - 1/m)^{1000m} \leq 2^{-1000}$$  

(18.2)

(using the inequality $(1 - 1/m)^m \leq 1/e \leq 1/2$).

18.1.2 What does this mean?

We have shown a probabilistic algorithm that on any $m$ edge graph $G$, will output a cut of at least $m/2$ edges with probability at least $1 - 2^{-1000}$. Does it mean that we can consider this problem as “easy”?
Should we be somewhat wary of using a probabilistic algorithm, since it can sometimes fail?

First of all, it is important to emphasize that this is still a worst case guarantee. That is, we are not assuming anything about the input graph: the probability is only due to the internal randomness of the algorithm. While a probabilistic algorithm might not seem as nice as a deterministic algorithm that is guaranteed to give an output, to get a sense of what a failure probability of $2^{-1000}$ means, note that:

- The chance of winning the Massachusetts Mega Million lottery is one over $(75)^5 \cdot 15$ which is roughly $2^{-35}$. So $2^{-1000}$ corresponds to winning the lottery about 300 times in a row, at which point you might not care so much about your algorithm failing.

- The chance for a U.S. resident to be struck by lightning is about $1/700000$ which corresponds about $2^{-45}$ chance that you’ll be struck this very second, and again might not care so much about the algorithm’s performance.

- Since the earth is about 5 billion years old, we can estimate the chance that an asteroid of the magnitude that caused the dinosaurs’ extinction will hit us this very second is about $2^{-58}$. It is quite likely that even a deterministic algorithm will fail if this happens.

So, in practical terms, a probabilistic algorithm is just as good as a deterministic one. But it is still a theoretically fascinating question whether probabilistic algorithms actually yield more power, or is it the case that for any computational problem that can be solved by probabilistic algorithm, there is a deterministic algorithm with nearly the same performance.\(^2\) For example, we will see in Exercise 18.1 that there is in fact a deterministic algorithm that can cut at least $m/2$ edges in an $m$-edge graph. We will discuss this question in generality in future lectures. For now, let us see a couple of examples where randomization leads to algorithms that are better in some sense than what the known deterministic algorithms.

18.1.3 Solving SAT through randomization

The 3SAT is NP hard, and so it is unlikely that it has a polynomial (or even subexponential) time algorithm. But this does not mean that we can’t do at least somewhat better than the trivial $2^n$ algorithm for $n$-variable 3SAT. The best known worst-case algorithms for 3SAT are randomized, and are related to the following simple algorithm, variants of which are also used in practice:

\(^*\) This question does have some significance to practice, since hardware that generates high quality randomness at speed is nontrivial to construct.
Algorithm WalkSAT:

- On input an $n$ variable 3CNF formula $\varphi$ do the following for $T$ steps:
  - Choose a random assignment $x \in \{0, 1\}^n$ and repeat the following for $S$ steps:
    1. If $x$ satisfies $\varphi$ then output $x$.
    2. Otherwise, choose a random clause $(\ell_i \lor \ell_j \lor \ell_k)$ that $x$ does not satisfy, and choose a random literal in $\ell_i, \ell_j, \ell_k$ and modify $x$ to satisfy this literal.
    3. Go back to step 1.
- If all the $T \cdot S$ repetitions above did not result in a satisfying assignment then output Unsatisfiable

The running time of this algorithm is $S \cdot T \cdot \text{poly}(n)$, and so the key question is how small can we make $S$ and $T$ so that the probability that WalkSAT outputs Unsatisfiable on a satisfiable formula $\varphi$ will be small. It is known that we can do so with $ST = \tilde{O}((4/3)^n)$ (see Exercise 18.3), but we’ll show below a simpler analysis yielding $ST = \tilde{O}(\sqrt{3}^n) = \tilde{O}(1.74^n)$ which is still much better than the trivial $2^n$ bound.  

Theorem 18.2 — WalkSAT simple analysis. If we set $T = 100 \cdot 3^{n/2}$ and $S = n/2$, then the probability we output Unsatisfiable for a satisfiable $\varphi$ is at most $1/2$.

Proof. Suppose that $\varphi$ is a satisfiable formula and let $x^*$ be a satisfying assignment for it. For every $x \in \{0, 1\}^n$, denote by $\Delta(x, x^*)$ the number of coordinates that differ between $x$ and $x^*$. We claim that (*) in every local improvement step, with probability at least $1/3$ we will reduce $\Delta(x, x^*)$ by one. Hence, if the original guess $x$ satisfied $\Delta(x, x^*) \leq n/2$ (an event that, as we will show, happens with probability at least $1/2$) then with probability at least $(1/3)^{n/2} = \sqrt{3}^{-n/2}$ after $n/2$ steps we will reach a satisfying assignment. This is a pretty lousy probability of success, but if we repeat this $100\sqrt{3}^{n/2}$ times then it is likely that it that it will happen once.

To prove the claim (*) note that any clause that $x$ does not satisfy, it differs from $x^*$ by at least one literal. So when we change $x$ by one of the three literals in the clause, we have probability at least $1/3$ of decreasing the distance.

We now prove our earlier claim that with probability $1/2$ over
\[ x \in \{0,1\}^n, \Delta(x,x^*) \leq n/2. \] Indeed, consider the map \(FLIP: \{0,1\}^n \rightarrow \{0,1\}^n\) where \(FLIP(x_0,\ldots,x_{n-1}) = (1-x_0,\ldots,1-x_{n-1})\).

We leave it to the reader to verify that (1) \(FLIP\) is one to one, and (2) \(\Delta(FLIP(x),x^*) = n - \Delta(x,x^*)\). Thus, if \(A = \{x \in \{0,1\}^n : \Delta(x,x^*) \leq n/2\}\) then \(FLIP\) is a one-to-one map from \(\bar{A}\) to \(A\), implying that \(|A| \geq |\bar{A}|\) and hence \(\mathbb{P}[A] \geq 1/2\).

The above means that in any single repetition of the outer loop, we will end up with a satisfying assignment with probability \(\frac{1}{2} \cdot \sqrt{3^{-n}}\).

Hence the probability that we never do so in \(100\sqrt{3^n}\) repetitions is at most \((1 - \frac{1}{2\sqrt{3}})^{100\sqrt{3^n}} \leq (1/e)^{50}\). ■

18.1.4 Bipartite matching.

The matching problem is one of the canonical optimization problems, arising in all kinds of applications, including matching residents and hospitals, kidney donors and patients, or flights and crews, and many others. One prototypical variant is bipartite perfect matching. In this problem, we are given a bipartite graph \(G = (L \cup R, E)\) which has \(2n\) vertices partitioned into \(n\)-sized sets \(L\) and \(R\), where all edges have one endpoint in \(L\) and the other in \(R\). The goal is to determine whether there is a perfect matching which is a subset \(M \subseteq E\) of \(n\) disjoint edges. That is, \(M\) matches every vertex in \(L\) to a unique vertex in \(R\).

\[\text{Figure 18.2: The bipartite matching problem in the graph } G = (L \cup R, E) \text{ can be reduced to the minimum } s,t \text{ cut problem in the graph } G' \text{ obtained by adding vertices } s,t \text{ to } G, \text{ connecting } s \text{ with } L \text{ and connecting } t \text{ with } R.\]

The bipartite matching problem turns out to have a polynomial-time algorithm, since we can reduce finding a matching in \(G\) to finding a minimum cut (or equivalently, maximum flow) in a related graph \(G'\) (see Fig. 18.2). However, we will see a different probabilistic algorithm to determine whether a graph contains such a matching.

Let us label \(G\)'s vertices as \(L = \{\ell_0,\ldots,\ell_{n-1}\}\) and
Let \( R = \{ r_0, \ldots, r_{n-1} \} \). A matching \( M \) corresponds to a permutation \( \pi \in S_n \) (i.e., one-to-one and onto function \( \pi : [n] \to [n] \)) where for every \( i \in [n] \), we define \( \pi(i) \) to be the unique \( j \) such that \( M \) contains the edge \( \{ \ell, r_j \} \). Define an \( n \times n \) matrix \( A = A(G) \) where \( A_{ij} = 1 \) if and only if the edge \( \{ \ell, r_j \} \) is present and \( A_{ij} = 0 \) otherwise. The correspondence between matchings and permutations implies the following claim:

**Lemma 18.3 — Matching polynomial.** Define \( P = P(G) \) to be the polynomial mapping \( \mathbb{R}^{n^2} \) to \( \mathbb{R} \) where

\[
P(x_0, \ldots, x_{n-1}) = \sum_{\pi \in S_n} \left( \prod_{i=0}^{n-1} \operatorname{sign}(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_i \pi(i) \tag{18.3}
\]

Then \( G \) has a perfect matching if and only if \( P \) is not identically zero. That is, \( G \) has a perfect matching if and only if there exists some assignment \( x = (x_{i,j})_{i,j \in [n]} \in \mathbb{R}^{n^2} \) such that \( P(x) \neq 0 \).

**Proof.** If \( G \) has a perfect matching \( M^* \), then let \( \pi^* \) be the permutation corresponding to \( M \) and let \( x^* \in \mathbb{Z}^{n^2} \), defined as follows: \( x_{i,j} = 1 \) if \( j = \pi(i) \) and \( x_{i,j} = 0 \). Note that for every \( \pi \neq \pi^* \), \( \prod_{i=0}^{n-1} x_i \pi(i) = 0 \) but \( \prod_{i=0}^{n-1} x_i \pi^*(i) = 1 \) and hence \( P(x^*) \) will equal \( \prod_{i=0}^{n-1} A_{i,\pi^*(i)} \). But since \( M^* \) is a perfect matching in \( G \), \( \prod_{i=0}^{n-1} A_{i,\pi^*(i)} = 1 \).

On the other hand, suppose that \( P \) is not identically zero. By Eq. (18.3), this means that at least one of the terms \( \prod_{i=0}^{n-1} A_{i,\pi(i)} \) is not equal to zero. But then this permutation \( \pi \) must be a perfect matching in \( G \).

As we’ve seen before, for every \( x \in \mathbb{R}^{n^2} \), we can compute \( P(x) \) by simply computing the determinant of the matrix \( A(x) \) which is obtained by replacing \( A_{i,j} \) with \( A_{i,j} x_{i,j} \). So, this reduces testing perfect matching to the zero testing problem for polynomials: given some polynomial \( P(\cdot) \), test whether \( P \) is identically zero or not. The intuition behind our randomized algorithm for zero testing is the following:

*If a polynomial is not identically zero, then it can’t have "too many" roots.*

This intuition sort of makes sense. For one variable polynomials, we know that a nonzero linear function has at most one root, a quadratic function (e.g., a parabola) has at most two roots, and generally a degree \( d \) equation has at most \( d \) roots. While in more than one variable there can be an infinite number of roots (e.g., the polynomial \( x_0 + y_0 \) vanishes on the line \( y = -x \)) it is still the case that...
A degree $d$ curve in one variable can have at most $d$ roots. In higher dimensions, a $n$-variate degree-$d$ polynomial can have an infinite number of roots though the set of roots will be an $n-1$ dimensional surface. Over a finite field $\mathbb{F}$, an $n$-variate degree $d$ polynomial has at most $d|\mathbb{F}|^{n-1}$ roots.

The set of roots is very “small” compared to the set of all inputs. For example, the root of a bivariate polynomial form a curve, the roots of a three-variable polynomial form a surface, and more generally the roots of an $n$-variable polynomial are a space of dimension $n-1$.

This intuition leads to the following simple randomized algorithm:

To decide if $P$ is identically zero, choose a “random” input $x$ and check if $P(x) \neq 0$.

This makes sense as if there are only “few” roots, then we expect that with high probability the random input $x$ is not going to be one of those roots. However, to transform into an actual algorithm, we need to make both the intuition and the notion of a “random” input precise. Choosing a random real number is quite problematic, especially when you have only a finite number of coins at your disposal, and so we start by reducing the task to a finite setting. We will use the following result

**Theorem 18.4 — Schwartz–Zippel lemma.** For every integer $q$, and polynomial $P : \mathbb{R}^m \to \mathbb{R}$ with integer coefficients. If $P$ has degree at most $d$ and is not identically zero, then it has at most $dq^{n-1}$ roots in the set $[q]^n = \{(x_0, \ldots, x_{m-1}) : x_i \in \{0, \ldots, q-1\}\}$.

We omit the (not too complicated) proof of Theorem 18.4. We remark that it holds not just over the real numbers but over any field.
as well. Since the matching polynomial $P$ of Lemma 18.3 has degree at most $n$, Theorem 18.4 leads directly to a simple algorithm for testing if it is nonzero:

**Algorithm Perfect-Matching:**
1. **Input:** Bipartite graph $G$ on $2n$ vertices $\{\ell_0, \ldots, \ell_{n-1}, r_0, \ldots, r_{n-1}\}$.
2. For every $i, j \in [n]$, choose $x_{i,j}$ independently at random from $[2n] = \{0, \ldots, 2n - 1\}$.
3. Compute the determinant of the matrix $A(x)$ whose $(i,j)^{th}$ entry corresponds equals $x_{i,j}$ if the edge $\{\ell_i, r_j\}$ is present and is equal to 0 otherwise.
4. Output no perfect matching if this determinant is zero, and output perfect matching otherwise.

This algorithm can be improved further (e.g., see Exercise 18.4). While it is not necessarily faster than the cut-based algorithms for perfect matching, it does have some advantages and in particular it turns out to be more amenable for parallelization. (It also has the significant disadvantage that it does not produce a matching but only states that one exists.) The Schwartz–Zippel Lemma, and the associated zero testing algorithm for polynomials, is widely used across computer science, including in several settings where we have no known deterministic algorithm matching their performance.

### 18.2 Lecture summary

- Using concentration results we can *amplify* in polynomial time the success probability of a probabilistic algorithm from a mere $1/p(n)$ to $1 - 2^{-q(n)}$ for every polynomials $p$ and $q$.
- There are several probabilistic algorithms that are better in various senses (e.g., simpler, faster, or other advantages) than the best known deterministic algorithm for the same problem.

### 18.3 Exercises

**Exercise 18.1** — Deterministic max cut algorithm. 4

**Exercise 18.2** — Simulating distributions using coins. Our model for probability involves tossing $n$ coins, but sometimes algorithm require sampling from other distributions, such as selecting a uniform number in $\{0, \ldots, M - 1\}$ for some $M$. Fortunately, we can simulate

*TODO: add exercise to give a deterministic max cut algorithm that gives $m/2$ edges. Talk about greedy approach.*
this with an exponentially small probability of error: prove that for every \( M \), if \( n > k \lceil \log M \rceil \), then there is a function \( F : \{0, 1\}^n \rightarrow \{0, \ldots, M - 1\} \cup \{\bot\} \) such that (1) The probability that \( F(x) = \bot \) is at most \( 2^{-k} \) and (2) the distribution of \( F(x) \) conditioned on \( F(x) \neq \bot \) is equal to the uniform distribution over \( \{0, \ldots, M - 1\} \).

**Exercise 18.3 — Better walksat analysis.** 1. Prove that for every \( \epsilon > 0 \), if \( n \) is large enough then for every \( x^* \in \{0, 1\}^n \) \( \Pr_{x \sim \{0, 1\}^n}[\Delta(x, x^*) \leq n/3] \leq 2^{-(1-H(1/3)-\epsilon)n} \) where \( H(p) = p \log(1/p) + (1 - p) \log(1/(1 - p)) \) is the same function as in Exercise 17.7.

2. Prove that \( 2^{1-H(1/3)+1/3} = (4/3) \).

3. Use the above to prove that for every \( \delta > 0 \) and large enough \( n \), if we set \( T = 1000 \cdot (4/3 + \delta)^n \) and \( S = n/3 \) in the WalkSAT algorithm then for every satisfiable 3CNF \( \varphi \), the probability that we output unsatisfiable is at most \( 1/2 \).

**Exercise 18.4 — Faster bipartite matching (challenge).**

**Bibliographical notes**


**Further explorations**

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

**Acknowledgements**
19

Modeling randomized computation

“Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.”, John von Neumann, 1951.

So far we have described probabilistic algorithms in an informal way, assuming that an operation such as “pick a string $x \in \{0, 1\}^n$” can be done efficiently. We have neglected to address two questions:

1. How do we actually efficiently obtain random strings in the physical world?

2. What is the mathematical model for randomized computations, and is it more powerful than deterministic computation?

We will return to the first question later in this course, but for now let’s just say that there are various random physical sources. User’s mouse movements, (non solid state) hard drive and network latency, thermal noise, and radioactive decay, have all been used as sources for randomness. For example, new Intel chips come with a random number generator built in. At the worst case, one can even include a coin tossing machine Fig. 19.1. We remark that while we can represent the output of these processes as binary strings which will be random from some distribution $\mu$, this distribution is not necessarily the same as the uniform distribution over $\{0, 1\}^n$.\(^1\) As we will discuss later (and is covered more in depth in a cryptography course), one typically needs to apply a “distillation” or randomness extraction process to the raw measurements to transform them to the uniform distribution.

In this lecture we focus on the second point - formally modeling probabilistic computation and studying its power. The first part is

\(^1\) Indeed, as this paper shows, even (real-world) coin tosses do not have exactly the distribution of a uniformly random string.
very easy. We define the RNAND programming language to include all the operations of NAND plus the following additional operation:

\[
\text{var :) RAND}
\]

where var is a variable. The result of applying this operation is that var is assigned a random bit in \(\{0, 1\}\). Similarly RNAND++ corresponds to NAND++ augmented with the same operation. (Every time the RAND operation is involved it returns a fresh independent random bit.) We can now define what it means to compute a function with a probabilistic algorithm:

**Definition 19.1 — Randomized circuits.** Let \(F\) be a (possibly partial) function mapping \(\{0, 1\}^n\) to \(\{0, 1\}^m\) and \(T \in \mathbb{N}\). We say that \(F \in \mathbf{BPSIZE}(T)\) if there is an \(n\)-input \(m\)-output RNAND program \(P\) of at most \(T\) lines so that for every \(x \in \{0, 1\}^m\),

\[
\mathbb{P}[P(x) = F(x)] \geq 2/3
\]

where this probability is taken over the random choices in the RAND operations.

Let \(F\) be a (possibly partial) function mapping \(\{0, 1\}^*\) to \(\{0, 1\}^*\) and \(T : \mathbb{N} \to \mathbb{N}\) be a nice function. We say that \(F \in \mathbf{BPTIME}(T(n))\) if there is an RNAND++ program \(P\) such that for every \(x \in \{0, 1\}^n\), \(P\) always halts with an output \(P(x)\) within at most \(T(|x|)\) steps and \(\mathbb{P}[P(x) = F(x)] \geq 2/3\), where again this probability is taken over the random choices in the RAND operations.
The prefix BP stands for “bounded probability”, and is used for historical reasons. As above, we will use $\text{BPTIME}(T(n))$ for the subset of $\text{BPTIME}(T(n))$ corresponding to total Boolean functions.

The number $2/3$ might seem arbitrary, but as we’ve seen in the previous lecture it can be amplified to our liking:

**Theorem 19.1 — Amplification.** Let $P$ be a NAND (or NAND++) program such that $\mathbb{P}[P(x) = F(x)] \geq 1/2 + \epsilon$, then there is a program $P'$ that with at most $10m/e^2$ times more lines (or runs in at most $m/e^2$ times more steps) such that $\mathbb{P}[P'(x) = F(x)] \geq 1 - 2^{-m}$.

**Proof.** The proof is the same as we’ve seen in the maximum cut example. We can run $P$ on input $x$ for $t = 10m/e^2$ times, using fresh randomness each one, to compute outputs $y_0, \ldots, y_{t-1}$. We output the value $y$ that appeared the largest number of times. Let $X_0$ be the random variable that is equal to 1 if $y_i = F(x)$ and equal to 0 otherwise. Then all the random variables $X_0, \ldots, X_{t-1}$ are i.i.d. and satisfy $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1] \geq 1/2 - \epsilon$. Hence the probability that $\sum X_i \leq t/2$ is at most $\exp(-\epsilon^2 t/4) \leq 2^{-m}$. ■

### 19.1 The power of randomization

A major question is whether randomization can add power to computation. We can phrase it mathematically as follows:

1. Is there some constant $c$ such that $\text{BPSIZE}(T) \subseteq \text{SIZE}(T^c)$ for every $T \in \mathbb{N}$?

2. Is there some constant $c$ such that $\text{BPTIME}(T(n)) \subseteq \text{TIME}(T(n)^c)$ for every nice $T : \mathbb{N} \rightarrow \mathbb{N}$? Specifically, is it the case that $\text{BPP} = \text{P}$ where $\text{BPP} = \bigcup_{a \in \mathbb{N}} \text{BPTIME}(n^a)$ stands for the class of functions that can be computed in probabilistic polynomial time.

### 19.2 Simulating RNAND programs by NAND programs

It turns out that question 1 is much easier to answer than question 2: RNAND is not really more powerful than NAND.

**Theorem 19.2 — Simulating RNAND with NAND.** For every $T \in \mathbb{N}$ and $F : \{0,1\}^n \rightarrow \{0,1\}^m$, if $F \in \text{RSIZE}(T)$ then $F \in \text{SIZE}(100nT)$. 
Proof. Suppose that $P$ is a $T$-line RNAND program such that $\Pr[P(x) = F(x)] \geq 2/3$. We use Theorem 19.1 to obtain an $50nT$ line program $P'$ such that

$$\Pr[P'(x) = F(x)] \geq 1 - 0.9 \cdot 2^{-n}.$$  \hfill (19.1)

Let $R \leq T$ be the number of lines using the RAND operations in $P'$. For every $r \in \{0, 1\}^R$, let $P'_r(x)$ be the result of executing $P'$ on $x$ where we use $r_i$ for the result of the $i$th RAND operation for every $i \in \{0, \ldots, R - 1\}$. For every $r \in \{0, 1\}^R$ and $x \in \{0, 1\}^n$, define a matrix $B$ whose rows are indexed by elements in $\{0, 1\}^R$ and whose columns are indexed by elements in $\{0, 1\}^n$ such that for every $r \in \{0, 1\}^R$ and $x \in \{0, 1\}^n$, $B_{r,x}$ equals 1 if $P'_r(x) \neq F(x)$ and $B_{r,x} = 0$ otherwise. We can rewrite Eq. (19.1) as follows: for every $x \in \{0, 1\}^n$

$$\sum_{r \in \{0, 1\}^R} B_{x,r} \leq 0.1 \cdot 2^{R-n}.$$ \hfill (19.2)

It follows that the total number of 1’s in this matrix is at most $0.12R$, which means that there is at least one row $r^*$ in this matrix such that $B_{r^*,x} = 0$ for every $x$. (Indeed, otherwise the matrix would have at least $2^R$ 1’s.)² Let $P^*$ be the following NAND program which is obtained by taking $P'$ and replacing the $i$th line of the form $\text{var} := \text{RAND}$ with the line $\text{var} := \langle r^*_i \rangle$. That is, we replace this with $\text{var} := \text{one NAND one}$ or $\text{var} := \text{zero NAND zero}$ based on whether $r^*_i$ is equal to 0 or 1 respectively, where we add as usual a couple of lines to initialize zero and one to 0 and 1. By construction, for every $x$, $P^*(x) = P'_r(x)$ and hence, since $B_{r^*,x} = 0$ for every $x \in \{0, 1\}^n$, it follows that $P^*(x) = F(x)$ for every $x \in \{0, 1\}^n$. \hfill \blacksquare

Note: Theorem 19.2 can also be proven using the Union Bound. That is, once we show that the probability of an error is smaller than $2^{-n}$, we can take a union bound over all $x$’s and so show that if we choose some random coins $r^*$ and fix them once and for all, then with high probability they will work for every $x \in \{0, 1\}^n$.

19.3 Derandomizing uniform computation

The proof of Theorem 19.2 can be summarized as follows: we can replace a $\text{poly}(n)$-time algorithm that tosses coins as it runs, with an algorithm that uses a single set of coin tosses $r^* \in \{0, 1\}^{\text{poly}(n)}$ which will be good enough for all inputs of size $n$. Another way to say it is that for the purposes of computing functions, we do not

² In fact, the same reasoning shows that at least a 0.9 fraction of the rows have this property.
need “online” access to random coins and can generate a set of coins “offline” ahead of time, before we see the actual input.

But the question of derandomizing uniform computation, or equivalently, NAND++ programs, is a whole different matter. For a NAND++ program we need to come up with a single deterministic algorithm that will work for all input lengths. That is, unlike the nonuniform NAND case, we cannot choose for every input length \( n \) some string \( r^* \in \{0,1\}^{\text{poly}(n)} \) to use as our random coins. Can we still do this, or does randomness add an inherent extra power for computation? This is a fundamentally interesting question but is also of practical significance. Ever since people started to use randomized algorithms during the Manhattan project, they have been trying to remove the need for randomness and replace it with numbers that are selected through some deterministic process. Throughout the years this approach has often been used successfully, though there have been a number of failures as well.\(^3\)

A common approach people used over the years was to replace the random coins of the algorithm by a “randomish looking” string that they generated through some arithmetic progress. For example, one can use the digits of \( \pi \) for the random tape. Using these type of methods corresponds to what von Neumann referred to as a “state of sin”. (Though this is a sin that he himself frequently committed, as generating true randomness in sufficient quantity was and still is often too expensive.) The reason that this is considered a “sin” is that such a procedure will not work in general. For example, it is easy to modify any probabilistic algorithm \( A \) such as the ones we have seen in the previous lecture, to an algorithm \( A' \) that is guaranteed to fail if the random tape happens to equal the digits of \( \pi \). This means that the procedure of “replacing the random tape by the digits of \( \pi \)” does not yield a general way to transform a probabilistic algorithm to a deterministic one that will solve the same problem. It does not mean that it always fails, but we have no good way to determine when this will work out.

This reasoning is not specific to \( \pi \) and holds for every deterministically produced string, whether it obtained by \( \pi, e \), the Fibonacci series, or anything else, as shown in the following result:

**Lemma 19.3 — Can’t replace tape deterministically.** There is a linear time probabilistic algorithm \( A \) such that for every \( x \in \{0,1\}^* \), \( \Pr[A(x) = 1] < 1/10 \) but for every \( n > 10 \) and fixed string \( r \in \{0,1\}^n \), there is some \( x \in \{0,1\}^n \) such that \( A(x;r) = 1 \) where \( A(x;r) \) denotes the execution of \( A \) on input \( x \) and where the randomness is supplied from \( r \).

\(^3\) One amusing anecdote is a recent case where scammers managed to predict the imperfect “pseudorandom generator” used by slot machines to cheat casinos. Unfortunately we don’t know the details of how they did it, since the case was sealed.
Proof. The algorithm $A$ is very simple. On input $x$ of length $n$, it tosses $n$ random coins $r_1, \ldots, r_n$ and outputs 1 if and only if $x_0 = r_0, x_1 = r_1, \ldots, x_n = r_9$ (if $n < 10$ then $A$ always outputs 0). Clearly $A$ runs in $O(n)$ steps and for every $x \in \{0, 1\}^*$, $P[A(x) = 1] \leq 2^{-10} < 0.1$. However, by definition, for every fixed string $r$ of length at least 10, $A(r;r) = 1$. ■

The proof of Lemma 19.3 might seem quite silly, but refers to a very serious issue. Time and again people have learned the hard way that one needs to be very careful about producing random bits using deterministic means. As we will see when we discuss cryptography, many spectacular security failures and break-ins were the result of using “insufficiently random” coins.

19.4 Pseudorandom generators

So, we can’t use any single string to “derandomize” a probabilistic algorithm. It turns out however, that we can use a collection of strings to do so. Another way to think about it is that we start by focusing on reducing (as opposed to eliminating) the amount of randomness needed. (Though we will see that if we reduce the randomness sufficiently, we can eventually get rid of it altogether.)

We make the following definition:

**Definition 19.2 — Pseudorandom generator.** A function $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^m$ is a $(T, \epsilon)$-pseudorandom generator if for every NAND program $P$ with $m$ inputs and one output of at most $T$ lines,

$$\left| \mathbb{P}_{s \sim \{0, 1\}^\ell}[P(G(s)) = 1] - \mathbb{P}_{r \sim \{0, 1\}^m}[P(r) = 1] \right| < \epsilon$$

(19.3)

This is a definition that’s worth reading more than once, and spending some time to digest it. First of all note that it takes several parameters:

- $T$ is the limit on the number of lines of the program $P$ that the generator needs to “fool”. The larger $T$ is, the stronger the generator.

- $\epsilon$ is how close is the output of the pseudorandom generator to the true uniform distribution over $\{0, 1\}^m$. The smaller $\epsilon$ is, the stronger the generator.
Figure 19.2: A pseudorandom generator $G$ maps a short string $s \in \{0,1\}^\ell$ into a long string $r \in \{0,1\}^m$ such that a small program $P$ cannot distinguish between the case that it is provided a random input $r \sim \{0,1\}^m$ and the case that it is provided a "pseudorandom" input of the form $r = G(s)$ where $s \sim \{0,1\}^\ell$. The short string $s$ is sometimes called the seed of the pseudorandom generator, as it is a small object that can be thought as yielding a large "tree of randomness".

- $\ell$ is the input length and $m$ is the output length. If $\ell \geq m$ then it is trivial to come up with such a generator: on input $s \in \{0,1\}^\ell$, we can output $s_0, \ldots, s_{m-1}$. In this case $\Pr_{s \sim \{0,1\}^\ell}[P(G(s)) = 1]$ will simply equal $\Pr_{r \in \{0,1\}^m}[P(r) = 1]$, no matter how many lines $P$ has. So, the smaller $\ell$ is and the larger $m$ is, the stronger the generator, and to get anything non-trivial, we need $m > \ell$.

We can think of a pseudorandom generator as a "randomness amplifier". It takes an input $s$ of $\ell$ bits, which we can assume per Eq. (19.3) that are truly random, and expands this into an output $r$ of $m > \ell$ pseudorandom bits. If $\epsilon$ is small enough (even $\epsilon = 0.1$ might be enough) then the pseudorandom bits will "look random" to any NAND program that is not too big. Still, there are two questions we haven’t answered:

- **What reason do we have to believe that pseudorandom generators with non-trivial parameters exist?**

- **Even if they do exist, why would such generators be useful to derandomize probabilistic algorithms?** After all, **Definition 19.2** does not involve RNAND++ programs but deterministic NAND programs with no randomness and no loops.
We will now (partially) answer both questions.

19.4.1 Existence of pseudorandom generators

For the first question, let us come clean and confess we do not know how to prove that interesting pseudorandom generators exist. By interesting we mean pseudorandom generators that satisfy that $\epsilon$ is some small constant (say $\epsilon < 1/3$, $m > \ell$, and the function $G$ itself can be computed in $\text{poly}(m)$ time. If we drop the last condition, then as shown in Lemma 19.4, there are pseudorandom generators where $m$ is exponentially larger than $\ell$.

Lemma 19.4 — Existence of inefficient pseudorandom generators. There is some absolute constant $C$ such that for every $\epsilon, T$, if $\ell > C(\log T + \log(1/\epsilon))$ and $m \leq T$, then there is an $(T, \epsilon)$ pseudorandom generator $G : \{0, 1\}^{\ell} \rightarrow \{0, 1\}^m$.

The proof uses an extremely useful technique known as the “probabilistic method” which is not too hard technically but can be confusing at first. The idea is to give a “non constructive” proof of existence of the pseudorandom generator $G$ by showing that if $G$ was chosen at random, then the probability that it would be a valid $(T, \epsilon)$ pseudorandom generator is positive. In particular this means that there exists a single $G$ that is a valid $(T, \epsilon)$ pseudorandom generator. The probabilistic method is doubly-confusing in the current setting, since eventually $G$ is a deterministic function $G$ (as its whole point is to reduce the need for randomness). The probabilistic method is just a proof technique to demonstrate the existence of such a function. The above discussion might be rather abstract at this point, but would become clearer after seeing the proof.

Proof. Let $\epsilon, T, \ell, m$ be as in the lemma’s statement. We need to show that there exists a function $G : \{0, 1\}^{\ell} \rightarrow \{0, 1\}^m$ that “fools” every $T$ line program $P$ in the sense of Eq. (19.3). We will show that this follows from the following claim:

CLAIM: For every fixed NAND program $P$, if we pick $G$ at random then the probability that Eq. (19.3) is violated is at most $2^{-T^2}$.

Before proving the claim, let us see why it implies the lemma. Suppose we give some arbitrary ordering $P_1, \ldots, P_M$ for the NAND programs $P$ of at most $T$ lines where $M$ is the number of such programs, which we have seen in lecture 4 to be at most $2^{O(T \log T)} < 2^{2T^2}$ (for large enough $T$). Thus if we let $E_i$ be the event Eq. (19.3) is vio-
lated for program $P_i$ with respect to the random $G$, then by setting $C$ large enough we can ensure that $P[E_i] < 1/(10M)$ which means that by the union bound with probability 0.9, Eq. (19.3) holds for every program $P$ of at most $T$ lines. This means that $G$ is a $(T, \epsilon)$ pseudorandom generators

Hence conclude the proof of Lemma 19.4, it suffices to prove the claim. Choosing a random $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^m$ amounts to choosing $L = 2^\ell$ random strings $y_0, \ldots, y_{L-1} \in \{0, 1\}^m$ and letting $G(x) = y_x$ (identifying $\{0, 1\}^\ell$ and $[L]$ via the binary representation). Hence the claim amounts to showing that for every fixed function $P : \{0, 1\}^m \rightarrow \{0, 1\}$, if $L > 2^{C(\log T + \log \epsilon)}$ (which by setting $C > 4$, we can ensure is larger than $10T^2/\epsilon^2$) then the probability that

$$\frac{1}{L} \sum_{i=0}^{L-1} P(y_i) - \mathbb{P}_{s \sim \{0, 1\}^m}[P(s) = 1] > \epsilon$$

(19.4)

is at most $2^{-T^2}$. Eq. (19.4) follows directly from the Chernoff bound. If we let for every $i \in [L]$ the random variable $X_i$ denote $P(y_i)$, then since $y_0, \ldots, y_{L-1}$ is chosen independently at random, these are independently and identically distributed random variables with mean $\mathbb{P}_{s \sim \{0, 1\}^m}[P(s) = 1]$ and hence the probability that they deviate from their expectation by $\epsilon$ is at most $2 \cdot 2^{-e^2 L/2}$. ■

The fact that there exists a pseudorandom generator does not mean that there is one that can be efficiently computed. However, it turns out that we can turn complexity “on its head” and used the assumed non existence of fast algorithms for problems such as 3SAT to obtain pseudorandom generators that can then be used to transform probabilistic algorithms into deterministic ones. This is known as the Hardness vs Randomness paradigm. We will discuss this in the next lecture, but this set of results led researchers to believe the following conjecture:

**Optimal PRG conjecture:** There are some absolute constants $\delta > 0, c \in \mathbb{N}$ such that for every nice time-complexity function $m : \mathbb{N} \rightarrow \mathbb{N}$, satisfying $m(n) \leq 2^{\delta n}$, there is a function $G : \{0, 1\}^* \rightarrow \{0, 1\}^*$, in $\text{TIME}(m(n)^c)$ such that for every $\ell \in \mathbb{N}$, the restriction of $G$ to length $\ell$ inputs is a function $G_\ell : \{0, 1\}^\ell \rightarrow \{0, 1\}^{m(\ell)}$ that is a $(2^{\delta \ell}, 2^{-\delta \ell})$ pseudorandom generator.

TO BE CONTINUED (discussion of what this means)
19.4.2 Usefulness of pseudorandom generators

TO BE CONTINUED (show that optimal pseudorandom generators imply that $\mathsf{BPP} = \mathsf{P}$)

19.5 Lecture summary

19.6 Exercises

19.7 Bibliographical notes

19.8 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

19.9 Acknowledgements
20

Hardness vs. Randomness

PLAN: Derandomization from an average case assumption. Show how to get from one bit stretch to arbitrary stretch. Perhaps state the general theorem that if there is hard-on-average “planted problem” in NP then there is a pseudorandom generator that stretches one bit, and maybe show a proof that this follows from a very particular planted problem such as planted k-COLORING or noisy 3XOR or something along those lines. Use the XOR lemma which will be stated without proof.

20.1 Lecture summary

20.2 Exercises

20.3 Bibliographical notes

20.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

20.5 Acknowledgements
21

Derandomization from worst-case assumptions (advanced lecture)

PLAN: Sketch the proof that BPP is in QuasiP if the permananet is subexponentially hard. Start by showing that an “ultra hard” function implies a generator with one bit expansion. Then talk about how with t disjoint blocks we can get t bit expansion. Then only sketch the NW generator how we can get exponential blowup with non-disjoint blocks. Next stage is to show how we get from worst-case to average case. Start by showing how we get weak average-case hardness for the permanent, then use without proof the XOR lemma.

21.1 Lecture summary

21.2 Exercises

21.3 Bibliographical notes

21.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

21.5 Acknowledgements
Cryptography

PLAN: Talk about one-time pad, then about getting short keys from pseudorandom generators. Brief sketch of advanced notions (leaving all details to CS 127)

Possibly: add lecture on *algorithms and society*. Could talk about how algorithms’ inputs and outputs today are interwoven with society (Google search, predictive policing, credit scores, adwords auctions, etc…). Talk about mechanism design, differential privacy, fairness, maybe also distributed computing and the consensus problem, and tie this into cryptocurrencies (“one cycle one vote”), talk about governance of the latter and “code as law”.

22.1 Lecture summary

22.2 Exercises

22.3 Bibliographical notes

22.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

22.5 Acknowledgements
Algorithms and society

PLAN: Talk about how algorithms interact with society - incentives, privacy, fairness. Maybe talk about cryptocurrencies (if we don’t talk about it in crypto)

23.1 Lecture summary

23.2 Exercises

23.3 Bibliographical notes

23.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

23.5 Acknowledgements
24

Compression, coding, and information

PLAN: Define entropy, talk about compression, Huffman coding, talk about channel capacity and error correcting codes.

24.1 Lecture summary

24.2 Exercises

24.3 Bibliographical notes

24.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

24.5 Acknowledgements
Space bounded computation

PLAN: Example of space bounded algorithms, importance of preserving space. The classes L and PSPACE, space hierarchy theorem.

25.1 Lecture summary

25.2 Exercises

25.3 Bibliographical notes

25.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

25.5 Acknowledgements
Streaming, sketching, and automata

PLAN: Define streaming model and constant-space computation, show that without loss of generality a constant space computation is streaming. State and prove (or perhaps only sketch?) the equivalence with regular expressions.

26.1 Lecture summary

26.2 Exercises

26.3 Bibliographical notes

26.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

26.5 Acknowledgements
27

Proofs and algorithms

PLAN: Talk about proofs and algorithms, zero knowledge proofs, correspondence between proofs and algorithms, Coq or other proof assistants, SAT solvers.

27.1 Lecture summary

27.2 Exercises

27.3 Bibliographical notes

27.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

27.5 Acknowledgements
Interactive proofs (advanced lecture)

PLAN: Define model of interactive proof, maybe motivate with Chess strategy, state without proof IP=PSPACE. Prove the IP for the Permanent

28.1 Lecture summary

28.2 Exercises

28.3 Bibliographical notes

28.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

28.5 Acknowledgements
Data structure lower bounds

PLAN: Lower bounds for data structures, cell-probe lower bounds using communication complexity.

29.1 Lecture summary

29.2 Exercises

29.3 Bibliographical notes

29.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

29.5 Acknowledgements
Quantum computing

PLAN: Talk about weirdness of quantum mechanics, double-slit experiment, Bell’s Inequality. Define QNAND programs that Have Hadamard gate. Give Simon’s algorithm, say something about Shor’s. State the “Quantum Physical Church Turing Thesis”

30.1 Lecture summary

30.2 Exercises

30.3 Bibliographical notes

30.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

30.5 Acknowledgements
Shor’s Algorithm (advanced lecture)

PLAN: Give proof of Shor’s Algorithm.

31.1 Lecture summary

31.2 Exercises

31.3 Bibliographical notes

31.4 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

31.5 Acknowledgements
Appendix: The NAND* Programming Languages

In this appendix we give a more formal specification of the NAND, NAND++ and NAND« programming languages. See the website http://nandpl.org for more information about these languages.

Note that the NAND programming language corresponds to Boolean circuits (with the NAND basis), the NAND++ programming language roughly corresponds to one-tape oblivious Turing machines, and the NAND« programming language roughly corresponds to RAM machines.

31.6 NAND programming language specification

31.6.1 Syntax of NAND programs

An unindexed variable identifier in NAND is a sequence of upper and lower case letters optionally ending with one or more occurrences of the prime symbol ‘, and that is not one of the following disallowed names: loop, validx, and i. For example, the following are valid unindexed variable identifiers: foo, bar’, Baz”. Invalid unindexed variable identifier include foo33, bo’az, Hey!, bar_22blah and i.

An indexed variable identifier has the form var_num where var is an unindexed variable identifier and num is a sequence of the digits 0-9. For example, the following are valid indexed variable identifiers: foo_19, bar’_73 and Baz”’_195. A variable identifier is either an indexed or an unindexed variable identifier. Thus both foo and Baz”’_68 are valid variable identifiers.

A NAND program consists of a finite sequence of lines of the form

vara := varb NAND varc

where vara, varb, varc are variable identifiers.
Variables of the form $x$ or $x_{\langle i \rangle}$ can only appear on the righthand side of the $\vdash$ operator and variables of the form $y$ or $y_{\langle i \rangle}$ can only appear on the lefthand side of the $\vdash$ operator. The number of inputs of a NAND program $P$ equals one plus the largest number $n$ such that a variable of the form $x_{\langle n \rangle}$ appears in the program, while the number of outputs of a NAND program equals one plus the largest number $j$ such that a variable of the form $y_{\langle j \rangle}$ appears in the program.

**Restrictions on indices:** If the variable identifiers are indexed, the index can never be larger than the number of lines in the program. If a variable of the form $y_{\langle j \rangle}$ appears in the program then $y_{\langle i \rangle}$ must appear in it for all $i < j$.

### 31.6.2 Semantics of NAND programs

To evaluate a NAND program $P$ with $n$ inputs and $m$ outputs on input $x_0, \ldots, x_{n-1}$ we initialize the all variables of the form $x_{\langle i \rangle}$ to $x_i$, and all other variables to zero. We then evaluate the program line by line, assigning to the variable on the lefthand side of the $\vdash$ operator the value of the NAND of the variables on the righthand side. In this evaluation, we identify $\text{foo}$ with $\text{foo}_{\langle 0 \rangle}$ and $\text{bar}_{\langle 79 \rangle}$ with $\text{bar}_{\langle 79 \rangle}$. That is, we only care about the numerical value of the index of a variable (and so ignore leading zeros) and if an index is not specified, we assume that it equals zero.

The output is the value of the variables $y_{\langle 0 \rangle}, \ldots, y_{\langle m - 1 \rangle}$. (If a variable of the form $y_{\langle i \rangle}$ has not been assigned a value in the program, then its value defaults to 0.)

Every NAND program $P$ on $n$ inputs and $m$ outputs can be associated with the function $F_P : \{0, 1\}^n \rightarrow \{0, 1\}^m$ such that for every $x \in \{0, 1\}^n$, $F_P(x)$ equals the output of $P$ on input $x$. We say that the function $P$ computes $F_P$. The NAND-complexity of a function $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ is the length (i.e., number of lines) of the smallest program $P$ that computes $F$.

### 31.7 NAND++ programming language specification

The NAND++ programming language adds to NAND the ability for loops and to access an unbounded amount of memory.
31.7.1 Syntax of NAND++ programs

Every valid NAND program is also a valid NAND++ program, but the NAND++ programming language adds to NAND the following operations:

- An indexed variable identifier can also have the form `var_i` where `var` is an unindexed variable identifier.

- The special variables `loop` and `validx` can appear in NAND++ programs. However, `loop` can only appear without an index and only on the lefthand side of the `:=` operator and `validx` can only appear on the righthand side of the `:=` operator. The variable `i` can only appear as an index to an unindexed variable identifier.

31.7.2 Semantics of NAND++ programs

Unlike a NAND program, a NAND++ program can be evaluated on inputs of every length. To evaluate a NAND++ program `P` on input `x ∈ \{0, 1\}^*` we do the following:

1. Set `pc = 0, r = 0, m = 0, i = 0, inc = +1` (`pc` stands for “program counter” and `r` is the current “rounds” of the the index variable).

2. For every line in `P` of the form `vara := varb NAND varc`, assign to the variable denoted by `vara` the NAND of the values of the variables denoted by `varb` and `varc`. If a variable on the righthand side has not received a value then its value is set to 0. If a variable has the form `foo_i` then we treat it as if it was `foo_{\langle i \rangle}` where `\langle i \rangle` denotes the current value of `i`. If a variable has the form `x_{\langle j \rangle}` then if `j < |x|` it gets the value `x_j` and otherwise it gets the value 0. If a variable has the form `validx_{\langle j \rangle}` then if `j < |x|` it gets the value 1 and otherwise it gets the value 0. If a variable on the lefthand side has the form `y_{\langle j \rangle}` then we let `m = max\{m, j + 1\}`.

3. If the variable `loop` has the value 0 then halt with output `y_0, \ldots, y_{m-1}` where `y_j` equals the value of `y_{\langle j \rangle}` if this variable has been assigned a value, and equals 0 if it hasn’t been assigned a value. Otherwise (if `loop` has value 1) then do the following:

   - If `i = 0` then set `r ← r + 1` (in this case we have completed a “round”) and `inc = +1`.

   - If `i = r` then set `inc = -1`.

   - Then set `i ← i + inc, pc ← pc + 1`, and go back to step 2.
We say that a NAND++ program \( P \) halts on input \( x \in \{0,1\}^* \) if when initialized with \( x \), \( P \) will eventually reach the point where \( \text{loop} \) equals 0 in Step 3 above and will output some value, which we denote by \( P(x) \). The number of steps that \( P \) takes on input \( x \) is defined as \( (pc + 1)\ell \) where \( pc \) is the value of the program counter \( pc \) at the time when the program halts and \( \ell \) is the number of lines in \( P \). If \( F : \{0,1\}^* \to \{0,1\}^* \) is a (potentially partial) function and \( P \) is a NAND++ program then we say that \( P \) computes \( F \) if for every \( x \in \{0,1\}^* \) on which \( F \) is defined, on input \( x \) the program \( P \) halts and outputs the value \( F(x) \). If \( P \) and \( F \) are as above and \( T : \mathbb{N} \to \mathbb{N} \) is some function, then we say that \( P \) computes \( F \) in time \( T \) if for every \( x \in \{0,1\}^* \) on which \( F \) is defined, on input \( x \) the program \( P \) halts within \( T(|x|) \) steps and outputs the value \( F(x) \).

### 31.7.3 Interpreting NAND programs

The NAND programming language is sufficiently simple so that writing an interpreter for it is an easy exercise. The website [http://nandpl.org](http://nandpl.org) contains an OCaml implementation that is more general and can handle many “syntactic sugar” transformation, but interpreting or compiling “plain vanilla” NAND can be done in few lines. For example, the following Python function parses a NAND program to the “list of tuples” representation:

```python
# Converts a NAND program from text to the list of tuples representation
# Assumes a program where no index is larger than the size of the program
def parse(prog):
    avars = { 'x':0, 'y':1, 'validx':2, 'loop':3 } # dictionary of indices of "workspace" variables
    n = max([int(var[2:]) for var in prog.split() if var[2]=='x_'])+1 # no of inputs
    m = max([int(var[2:]) for var in prog.split() if var[2]=='y_'])+1 # no of outputs

def var_idx(vname): # helper local function to return index of named variable
    vname = vname.split('_')
    name = vname[0]
    idx = int(vname[1]) if len(vname)>1 else 0
    return [avars.setdefault(name,len(avars)),idx]

    result = []
```
for line in prog.split('n'):
    if not line or line[0]=='#': continue # ignore empty
    and commented out lines
    (var1,assign,var2,op,var3) = line.split()
    result.append(var_idx(var1) + var_idx(var2) + var_idx(var3))
return (n,m,result)

As we discuss in the “code and data” lecture, we can execute a program given in the list of tuples representations as follows

# Evaluates an n-input, m-output NAND program P on input x
# P is given in the list of tuples representation
def EVAL(n,m,P,x):
    s = len(P) # no. of lines in the program
    t = 3*len(P) # maximum no. of unique labels
    avars = [0]*(t*s) # initialize array to 0
    for i in range(n): # initialize inputs to x
        avars[i*t] = x[i]
    for (a,i,b,j,k) in P: # evaluate every line of program
        avars[i*t+a] = 1-avars[j*t+b]*avars[k*t+c]

    # return y_0...y_(m-1) which is
    # avars[1],avars[t+1],...,avars[(m-1)*t+1]
    return [avars[i*t+1] for i in range(m)]

Moreover, if we want to compile NAND programs, we can easily transform them to C code using the following NAND2C function:

# Transforms a NAND program to a C function
# prog: string of program code
# n: number of inputs
# m: number of outputs
def NAND2C(prog,n,m):
    avars = { } # dictionary of indices of "workspace"
    variables
    for i in range(n):
        avars['x.'+str(i)] = i
    for j in range(m):
        avars['y.'+str(j)] = n+j

    def var_idx(vname): # helper local function to return
        # index of named variable
        index = vname.split('_')

        # The function is somewhat more complex than the minimum needed,
        # since it uses an array of bits to store the variables.
name = vname[0]
idx = int(vname[1]) if len(vname)>1 else 0
return avars.setdefault(name+'_'+str(idx),len(avars))

main = "\n"
for line in prog.split('n'):
    if not line or line[0]=='#' or line[0]=='//': continue  # ignore empty and commented out lines
    (var1,assign,var2,op,var3) = line.split()
    main += ' setbit(vars,{idx1}, ~(getbit(vars,{idx2}) &
               (getbit(vars,{idx2}))))\n'.format(
        idx1 = var_idx(var1), idx2 = var_idx(var2), idx3 =
        var_idx(var3))

Cprog = '''
#include <stdbool.h>

typedef unsigned long bfield_t[ (sizeof(long)-1+{numvars})/sizeof(long) ];
// long because that's probably what your cpu is best at
// The size_needed should be evenly divisible by sizeof(long) or
// you could (sizeof(long)-1+size_needed)/sizeof(long) to force it to round up

unsigned long getval(bfield_t vars, int idx) {{
    return 1 & (vars[idx / (8 * sizeof(long))] >> (idx %
               (8 * sizeof(long))));
}}

void setval(bfield_t vars, int idx, unsigned long v) {{
    vars[idx / (8 * sizeof(long))] = ((vars[idx / (8 *
                                      sizeof(long))] & ~(1<<b)) | (v<<b);
}}

unsigned long int *eval(unsigned long int *x) {{
    bfield_t vars = {{0}};
    int i;
    int j;
    unsigned long int y[{m}] = {{0}};
for(i=0;i<(n);++i) {{
    setval(vars,i,x[i])
}}
'''.format(n=n,m=m,numvars=len(avors))

Cprog = Cprog + main + '''
    for(j=0;j<(m);++j) {{
        y[j] = getval(vars,(n)+j)
    }}
    return y;
}}
'''.format(n=n,m=m)

return Cprog

31.8 NAND« programming language specification

The NAND« programming language allows *indirection*, hence using every variable as a pointer or index variable. Unlike the case of NAND++ vs NAND, NAND« cannot compute functions that NAND++ (and indeed any NAND« program can be “compiled to a NAND++ program) but it can be polynomially faster.

31.8.1 Syntax of NAND« programs

Like in NAND++, an indexed variable identifier in NAND« has the form foo_num where num is some absolute numerical constant or or foo_i where i is the special index variable. Every NAND++ program is also a valid NAND« program but in addition to lines of the form foo := bar NAND blah, in NAND« we allow the following operations as well:

- foo := bar (assignment)
- foo := bar + baz (addition)
- foo := bar - baz (subtraction)
- foo := bar » baz (right shift: \( \text{id}x \leftarrow \lfloor \text{foo}2^{-\text{bar}} \rfloor \))
- foo := bar « baz (left shift: \( \text{id}x \leftarrow \text{foo}2^{\text{bar}} \))
- foo := bar % baz (modular reduction)
- foo := bar * baz (multiplication)
• foo := bar / baz (integer division: idx ← \lfloor \frac{foo}{bar} \rfloor)
• foo := bar bAND baz (bitwise AND)
• foo := bar bXOR baz (bitwise XOR)
• foo := bar > baz (greater than)
• foo := bar < baz (smaller than)
• foo := bar == baz (equality)

Where foo, bar or baz are indexed or non-indexed variable identifiers but not the special variable i. However, we do allow i to be on the left hand side of the assignment operation, and hence can write code such as i := foo. As in NAND++, we only allow variables of the form x_... and validx_... and to be on the righthand side of the assignment operator and only allow variables of the form y_... to be on the lefthand side of the operator. In fact, by default we will only allow bit string valued NAND« programs which means that we only allow variables of the form y_... to be on the lefthand side of a line of the form y_... := foo NAND bar, hence guaranteeing that they are either 0 or 1. We might however sometimes consider drop the bit-string valued restriction and consider programs that can output integers as well.

31.8.2 Semantics of NAND« programs

Semantics of NAND« are obtained by generalizing to integer valued variables. Arithmetic operations are defined as expected except that we maintain the invariant that all variables always take values between 0 and the current value of the program counter (i.e., number of iterations of the program that have been completed). If an operation would result in assigning to a variable foo a number that is smaller than 0, then we assign 0 to foo, and if it assigns to foo a number that is larger than the program counter, then we assign the value of the program counter to foo. Just like C, we interpret any nonzero value as “true” or 1, and hence foo := bar NAND baz will assign to foo the value 0 if both bar and baz are not zero, and 1 otherwise. More formally, to evaluate a NAND++ program on inputs x_0, x_1, x_2, ... (which we will typically assume to be bits, but could be integers as well) we do the following:

1. Set pc = 0, m = 0, i = 0, (pc stands for “program counter” and r is the current “rounds” of the the index variable).

2. For every line in P, we do the following:
(a) If the line has the form \( vara := varb \text{ NAND } varc \), assign to the variable denoted by \( vara \) the NAND of the values of the variables denoted by \( varb \) and \( varc \) (interpreting 0 as false and nonzero as true). If a variable on the righthand side has not received a value then its value is set to 0.

- If a variable has the form \( \text{foo} \) without an index then we treat it as if it was \( \text{foo}_0 \).
- If a variable has the form \( \text{foo}_i \) then we treat it as if it is \( \text{foo}_{\langle j \rangle} \) where \( j \) denotes the current value of the index variable \( i \).
- If a variable has the form \( x_{\langle j \rangle} \) then if \( j < |x| \) it gets the value \( x_j \) and otherwise it gets the value 0.
- If a variable has the form \( \text{validx}_{\langle j \rangle} \) then if \( j < |x| \) it gets the value 1 and otherwise it gets the value 0.
- If a variable on the lefthand side has the form \( y_{\langle j \rangle} \) then we let \( m = \max\{m, j + 1\} \).

(b) If a line correspond to an index operation of the form \( \text{foo} := \text{bar } OP \text{ baz} \) where \( OP \) is one of the operations listed above then we evaluate \( \text{foo} \) and \( \text{bar} \) as in the step above and compute the value \( v \) to be the operation \( OP \) applied to the values of \( \text{foo} \) and \( \text{bar} \). We assign to the index variable corresponding to \( \text{idx} \) the value \( \max\{0, \min\{pc, v\}\} \).

If the variable \( \text{loop} \) has the value 0 then halt with output \( y_0, \ldots, y_{m-1} \) where \( y_j \) equals the value of \( y_{\langle j \rangle} \) if this variable has been assigned a value, and equals 0 if it hasn’t been assigned a value. Otherwise (if \( \text{loop} \) has value 1) then do the following. Set \( px \leftarrow pc + 1 \), set \( i = \text{INDEX}(pc) \) where \( \text{INDEX} \) is the function that maps the program counter value to the current value of \( i \), and go back to step 2.

Like a NAND++ program, the number of steps which a NAND« program \( P \) takes on input \( x \) is defined as \((pc + 1)\ell\) where \( pc \) is the value of the program counter at the point in which it halts and \( \ell \) is the number of lines in \( P \). Just like in NAND++, we say that a NAND« program \( P \) computes a partial function \( F : \{0,1\}^* \rightarrow \{0,1\}^* \) in time \( T : \mathbb{N} \rightarrow \mathbb{N} \) if for every \( x \in \{0,1\}^* \) on which \( F \) is defined, \( P(x) \) outputs \( F(x) \) within \( T(|x|) \) steps. Note that any NAND« program can be transformed into a NAND++ program that computes the same function, albeit at a polynomial loss in the number of steps.
31.9  The “standard library”

Additional features of NAND/NAND++/NAND« can be implemented via “syntactic sugar” transformations. The following is a “standard library” of features that can be assumed in writing programs for NAND/NAND++/NAND«. Whenever counting number of steps/lines in a program, or when feeding it as input to other programs, we assume that it is first transformed into its “sugar free” version that does not use any of these features.

31.9.1  Syntactic sugar for NAND

The following constructs are applicable in all the languages NAND/-NAND++/NAND«:

Variable assignment: foo := bar where foo,bar are variable identifiers corresponds to assigning the value of foo to bar.

Constants: We can assume that zero and one are assigned the values 0 and 1 respectively.

Multidimensional arrays: We can use multidimensional indices such as foo_{12,24} or foo_{i,j,k}. These can be transformed into a one-dimensional array via one-to-one embeddings of $\mathbb{N}^k$ in $\mathbb{N}$.

Conditionals: We can write code such as

```plaintext
if (foo) {
  code
}
```

to indicate that code will only be executed if foo is equal to 1.

Functions: We can define (non recursive) functions as follows:

```plaintext
def foo1,...,fook := Func(bar1,...,barl) {
  function_code
}
```

denotes code for a function that takes $l$ inputs bar1,...,barl and returns $k$ outputs foo1,...,fook. We can then invoke such a function by writing

```plaintext
blah1,...,blahk := Func(baz1,...,bazl)
```

this will be implemented by copy-pasting the code of Func and doing the appropriate search-and-replace of the variables, adding a
unique prefix to workspace variables to ensure there are no namespace conflicts.

We can use Functions also inside expressions, and so write code such as

\[
\text{foo} := \text{XOR}(\text{bar}, \text{baz})
\]

or

\[
\text{if AND}(\text{foo}, \text{bar}) \{ \\
\text{code} \\
\}
\]

where \text{XOR,AND} have been defined above.

\textbf{Operator overloading and infix notation:} For convenience of notation, we can define functions corresponding to standard operators \texttt{*},\texttt{+},\texttt{-}, etc. and then code such as \texttt{foo * bar} will correspond to \texttt{operator*(foo,bar)}.

\[2\]

\texttt{31.9.2 Encoding of integers, strings, lists.}

While NAND/NAND++/NAND\textsuperscript{*} natively only supports values of 0 and 1 for variables (and integers for indices in NAND++/NAND\textsuperscript{*}), in our standard library We will use the following default 8-symbol alphabet for specifying constants and strings in NAND/NAND++/NAND\textsuperscript{*}:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Encoding</th>
<th>Default semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>000</td>
<td>End of string/integer/list marker</td>
</tr>
<tr>
<td>0</td>
<td>001</td>
<td>The bit 0</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>The bit 1</td>
</tr>
<tr>
<td>{</td>
<td>011</td>
<td>Begin list</td>
</tr>
<tr>
<td>}</td>
<td>100</td>
<td>End list</td>
</tr>
<tr>
<td>,</td>
<td>101</td>
<td>Separate items</td>
</tr>
<tr>
<td>:</td>
<td>110</td>
<td>Define mapping</td>
</tr>
<tr>
<td>_</td>
<td>111</td>
<td>Space / “no op” / to be ignored</td>
</tr>
</tbody>
</table>

When we write code such as \texttt{foo := "str"} where \texttt{str} is a length \(n\) sequence from this alphabet that doesn’t contain \(\_\), then we mean that we store in \texttt{var\_\{0\} til var\_\{3n - 1\}\}} the encoding for the \(n\) symbols of \texttt{str}, while we store in \texttt{var\_\{3n\},var\_\{3n + 1\},var\_\{3n + 2\}\}} the encoding for \(\_\) (which is 000).
We will store integers in this encoding using their representation binary basis ending with .. So, for example, the integer 13 = $2^3 + 2^2 + 2^0$ will be stored by encoding 1011. And hence the shorthands \( \text{foo} := 13 \) and \( \text{foo} := "1011" \) will be transformed into identical NAND code. Arithmetic operations will be shorthand for the standard algorithms and so we can use code such as

\[
\begin{align*}
\text{foo} & := 12 \\
\text{bar} & := 1598 \\
\text{baz} & := \text{foo} \ast \text{bar}
\end{align*}
\]

Using multidimensional arrays we can also use code such as

\[
\begin{align*}
\text{foo}_0 & := 124 \\
\text{foo}_1 & := 57 \\
\text{foo}_2 & := 5459
\end{align*}
\]

**Lists:** We store lists and nested lists using the separators \([,],\),, with the binary encoding of each element. Thus code such as

\[
\text{foo} := [ 13 , 1 , 4 ]
\]

will be the same as

\[
\text{foo} := "[1011,01,001]"
\]

We can store in lists not just integers but any other object for which we have some canonical way to encode it into bits.

We can use the union operation on lists and so

\[
\text{foo} := [17,8] + [24,9]
\]

will be the same as

\[
\text{foo} := [17,8,24,9]
\]

we will use \( \text{in} \) as shorthand for the operation that scans a list to see if it contains an element and so in the code

\[
\begin{align*}
\text{foo} & := [12,127,8] \\
\text{bar} & := \text{8 in foo}
\end{align*}
\]

\( \text{bar} \) is assigned the value 1.

The \( \text{length} \) macro computes the length of a list.

Within lists the no-op character .. will be ignored, and so replacing characters with .. can be a way of removing items from the lists.

**Iterating:** We use the construct
for foo in bar {
    code
}

to execute code length(bar) times where each time foo will get
the current element.

3

4

31.9.3 Syntactic sugar for NAND++ / NAND«

In addition to the syntactic sugar above, in NAND++ and NAND«
we can also use the following constructs:

Indirect indexing: If we use code such as foo_bar where bar is
not an index variable then this is shorthand for copying the integer
encoded in bar into some index variable of the form i'' (with an
appropriate number of primes to make it unique) and then use
foo_i'''. Similarly, we will also use code such as idx := bar where
idx is an index variable and bar is a non-index variable to denote
that we copy to idx the integer that is encoded by bar. If bar does
not represent a number then we consider it as representing 0.

Inner loops: We can use the following loop construct

while (foo) {
    code
}

to indicate the loop will execute as long as foo equals 1. We can
nest loops inside one another, and also replace foo with another
expression such as a call to a function.

5

^TODO: potentially have a loop that
iterates over a list
3. Bibliography