

BOAZ BARAK

# INTRODUCTION TO THEORETICAL COMPUTER SCIENCE

Friday 27<sup>th</sup> October, 2017 19:28

Copyright © 2017 Boaz Barak

This work is licensed under a [Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International” license](#).



# *Contents*

<i>Preface</i>	21
<i>Mathematical Background</i>	27
1 <i>Introduction</i>	57
2 <i>Computation and Representation</i>	73
3 <i>Defining computation</i>	95
4 <i>Syntactic sugar, and computing every function</i>	117
5 <i>Code as data, data as code</i>	135
6 <i>Physical implementations of NAND programs</i>	151
7 <i>Loops and infinity</i>	177
8 <i>Indirection and universality</i>	199

9	<i>Equivalent models of computation</i>	215
10	<i>Is every function computable?</i>	239
11	<i>Restricted computational models</i>	253
12	<i>Is every theorem provable?</i>	269
13	<i>Efficient computation</i>	287
14	<i>Modeling running time</i>	305
15	<i>Polynomial-time reductions</i>	321
16	<i>NP, NP completeness, and the Cook-Levin Theorem</i>	333
17	<i>Advanced hardness reductions (advanced lecture)</i>	347
18	<i>What if P equals NP?</i>	349
19	<i>Probability Theory 101</i>	361
20	<i>Probabilistic computation</i>	375
21	<i>Modeling randomized computation</i>	385
22	<i>Hardness vs. Randomness</i>	395

23	<i>Derandomization from worst-case assumptions (advanced lecture)</i>	397
24	<i>Cryptography</i>	399
25	<i>Algorithms and society</i>	401
26	<i>Compression, coding, and information</i>	403
27	<i>Space bounded computation</i>	405
28	<i>Streaming, sketching, and automata</i>	407
29	<i>Proofs and algorithms</i>	409
30	<i>Interactive proofs (advanced lecture)</i>	411
31	<i>Data structure lower bounds</i>	413
32	<i>Quantum computing</i>	415
33	<i>Shor's Algorithm (advanced lecture)</i>	417
	<i>Appendix: The NAND* Programming Languages</i>	419
	<i>Bibliography</i>	433



# *Contents (detailed)*

## *Preface*      21

0.1	<i>To the student</i>	23
0.1.1	<i>Is this effort worth it?</i>	23
0.2	<i>To potential instructors</i>	24
0.3	<i>Acknowledgements</i>	25

## *Mathematical Background*      27

0.4	<i>A mathematician's apology</i>	27
0.5	<i>A quick overview of mathematical prerequisites</i>	28
0.6	<i>Basic discrete math objects</i>	29
0.6.1	<i>Sets</i>	29
0.6.2	<i>Special sets</i>	31
0.6.3	<i>Functions</i>	32
0.6.4	<i>Graphs</i>	35
0.6.5	<i>Logic operators and quantifiers.</i>	37
0.6.6	<i>Quantifiers for summations and products</i>	38
0.6.7	<i>Parsing formulas: bound and free variables</i>	39
0.6.8	<i>Asymptotics and big-Oh notation</i>	40
0.6.9	<i>Some "rules of thumbs" for big Oh notation</i>	42
0.7	<i>Proofs</i>	42
0.7.1	<i>Proofs and programs</i>	43
0.8	<i>Extended example: graph connectivity</i>	43
0.8.1	<i>Mathematical induction</i>	45
0.8.2	<i>Proving the theorem by induction</i>	46
0.8.3	<i>Writing down the proof</i>	48
0.9	<i>Proof writing style</i>	51
0.9.1	<i>Patterns in proofs</i>	52
0.10	<i>Non-standard notation</i>	54

<i>0.11 Exercises</i>	55
<i>0.12 Bibliographical notes</i>	56
<i>0.13 Acknowledgements</i>	56
<b>1 Introduction</b>	<b>57</b>
<i>1.0.1 Example: A faster way to multiply</i>	60
<i>1.0.2 Beyond Karatsuba's algorithm</i>	65
<i>1.0.3 Advanced note: matrix multiplication</i>	65
<i>1.0.4 Algorithms beyond arithmetic</i>	66
<i>1.0.5 On the importance of negative results.</i>	67
<i>1.1 Lecture summary</i>	68
<i>1.1.1 Roadmap to the rest of this course</i>	69
<i>1.2 Exercises</i>	70
<i>1.3 Bibliographical notes</i>	71
<i>1.4 Further explorations</i>	71
<i>1.5 Acknowledgements</i>	72
<b>2 Computation and Representation</b>	<b>73</b>
<i>2.1 Examples of binary representations</i>	75
<i>2.1.1 Representing natural numbers</i>	75
<i>2.1.2 Representing (potentially negative) integers</i>	76
<i>2.1.3 Representing rational numbers</i>	77
<i>2.1.4 Representing real numbers</i>	77
<i>2.1.5 Can we represent reals exactly?</i>	78
<i>2.2 Beyond numbers</i>	82
<i>2.2.1 Finite representations</i>	83
<i>2.2.2 Prefix free encoding</i>	83
<i>2.2.3 Making representations prefix free</i>	85
<i>2.2.4 Representing letters and text</i>	86
<i>2.2.5 Representing vectors, matrices, images</i>	86
<i>2.2.6 Representing graphs</i>	87
<i>2.2.7 Representing lists</i>	87
<i>2.2.8 Notation</i>	88
<i>2.3 Defining computational tasks</i>	88
<i>2.3.1 Advanced note: beyond computing functions</i>	90
<i>2.4 Lecture summary</i>	91
<i>2.5 Exercises</i>	91
<i>2.6 Bibliographical notes</i>	93

2.7	<i>Further explorations</i>	94
2.8	<i>Acknowledgements</i>	94
3	<i>Defining computation</i>	95
3.1	<i>Defining computation</i>	97
3.2	<i>The NAND Programming language</i>	99
3.2.1	<i>Adding one-bit numbers</i>	101
3.2.2	<i>Formal definitions</i>	102
3.2.3	<i>Computing a function: formal definition</i>	103
3.3	<i>Canonical input and output variables</i>	106
3.4	<i>Composing functions</i>	107
3.5	<i>Proving the composition theorems</i>	109
3.5.1	<i>Example: Adding two-bit numbers</i>	111
3.5.2	<i>Composition in NAND programs</i>	114
3.6	<i>Lecture summary</i>	114
3.7	<i>Exercises</i>	114
3.8	<i>Bibliographical notes</i>	115
3.9	<i>Further explorations</i>	115
3.10	<i>Acknowledgements</i>	115
4	<i>Syntactic sugar, and computing every function</i>	117
4.1	<i>Some useful syntactic sugar</i>	118
4.1.1	<i>Constants</i>	118
4.1.2	<i>Conditional statements</i>	119
4.1.3	<i>Functions / Macros</i>	119
4.1.4	<i>Bounded loops</i>	120
4.1.5	<i>Example:</i>	121
4.1.6	<i>More indices</i>	121
4.1.7	<i>Non-Boolean variables, lists and integers</i>	122
4.1.8	<i>Storing integers</i>	122
4.2	<i>Adding and multiplying n bit numbers</i>	124
4.2.1	<i>Multiplying numbers</i>	125
4.3	<i>Functions beyond arithmetic</i>	125
4.3.1	<i>Constructing a NAND program for LOOKUP</i>	126
4.4	<i>Computing every function</i>	128
4.4.1	<i>Proof of NAND's Universality</i>	128
4.5	<i>The class <math>\text{SIZE}_{n,m}(T)</math></i>	131
4.6	<i>Lecture summary</i>	131

4.7	<i>Exercises</i>	132
4.8	<i>Bibliographical notes</i>	133
4.9	<i>Further explorations</i>	133
4.10	<i>Acknowledgements</i>	133
5	<i>Code as data, data as code</i>	135
5.1	<i>A NAND interpreter in NAND</i>	136
5.2	<i>Concrete representation for NAND programs</i>	137
5.3	<i>A NAND interpreter in NAND</i>	140
5.4	<i>A Python interpreter in NAND</i>	143
5.5	<i>Counting programs, and lower bounds on the size of NAND programs</i>	145
5.6	<i>Lecture summary</i>	147
5.7	<i>Exercises</i>	148
5.8	<i>Bibliographical notes</i>	149
5.9	<i>Further explorations</i>	149
5.10	<i>Acknowledgements</i>	150
6	<i>Physical implementations of NAND programs</i>	151
6.1	<i>Physical implementation of computing devices.</i>	152
6.2	<i>Transistors and physical logic gates</i>	152
6.3	<i>Gates and circuits</i>	156
6.4	<i>Representing programs as graphs</i>	157
6.5	<i>Composition from graphs</i>	162
6.6	<i>General Boolean circuits: a formal definition</i>	164
6.7	<i>Neural networks</i>	166
6.8	<i>Biological computing</i>	166
6.9	<i>Cellular automata and the game of life</i>	167
6.10	<i>Circuit evaluation algorithm</i>	167
6.10.1	<i>Advanced note: evaluating circuits in quasilinear time.</i>	168
6.11	<i>The physical extended Church-Turing thesis</i>	168
6.11.1	<i>Attempts at refuting the PECTT</i>	170
6.12	<i>Lecture summary</i>	175
6.13	<i>Exercises</i>	175
6.14	<i>Bibliographical notes</i>	176
6.15	<i>Further explorations</i>	176
6.16	<i>Acknowledgements</i>	176

7	<i>Loops and infinity</i>	177
	7.1 <i>The NAND++ Programming language</i>	178
	7.1.1 <i>Computing the index location</i>	180
	7.1.2 <i>Infinite loops and computing a function</i>	181
	7.2 <i>A spoonful of sugar</i>	182
	7.2.1 <i>Inner loops via syntactic sugar</i>	183
	7.2.2 <i>Controlling the index variable</i>	184
	7.2.3 <i>"Simple" NAND++ programs</i>	185
	7.3 <i>Uniformity, and NAND vs NAND++</i>	186
	7.3.1 <i>Growing a NAND tree</i>	187
	7.4 <i>NAND++ Programs as tuples</i>	189
	7.4.1 <i>Configurations</i>	191
	7.4.2 <i>Deltas</i>	196
	7.5 <i>Lecture summary</i>	197
	7.6 <i>Exercises</i>	198
	7.7 <i>Bibliographical notes</i>	198
	7.8 <i>Further explorations</i>	198
	7.9 <i>Acknowledgements</i>	198
8	<i>Indirection and universality</i>	199
	8.1 <i>The NAND« programming language</i>	200
	8.1.1 <i>Simulating NAND« in NAND++</i>	201
	8.1.2 <i>Example</i>	204
	8.2 <i>The "Best of both worlds" paradigm</i>	205
	8.2.1 <i>Let's talk about abstractions.</i>	206
	8.3 <i>Universality: A NAND++ interpreter in NAND++</i>	208
	8.3.1 <i>Representing NAND++ programs as strings</i>	209
	8.3.2 <i>A NAND++ interpreter in NAND«</i>	210
	8.3.3 <i>A Python interpreter in NAND++</i>	211
	8.4 <i>Lecture summary</i>	212
	8.5 <i>Exercises</i>	212
	8.6 <i>Bibliographical notes</i>	213
	8.7 <i>Further explorations</i>	213
	8.8 <i>Acknowledgements</i>	213
9	<i>Equivalent models of computation</i>	215
	9.1 <i>Turing machines</i>	216

<b>9.2</b>	<i>Turing Machines and NAND++ programs</i>	218
9.2.1	<i>Simulating Turing machines with NAND++ programs</i>	218
9.2.2	<i>Simulating NAND++ programs with Turing machines</i>	220
<b>9.3</b>	<i>"Turing Completeness" and other Computational models</i>	223
9.3.1	<i>RAM Machines</i>	223
9.3.2	<i>Imperative languages</i>	223
<b>9.4</b>	<i>Lambda calculus and functional programming languages</i>	224
9.4.1	<i>The "basic" <math>\lambda</math> calculus objects</i>	226
9.4.2	<i>How basic is "basic"?</i>	229
9.4.3	<i>List processing and recursion without recursion</i>	230
9.4.4	<i>The Y combinator</i>	231
<b>9.5</b>	<i>Other models</i>	233
9.5.1	<i>Parallel algorithms and cloud computing</i>	233
9.5.2	<i>Game of life, tiling and cellular automata</i>	233
<b>9.6</b>	<i>Our models vs standard texts</i>	234
<b>9.7</b>	<i>The Church-Turing Thesis</i>	236
<b>9.8</b>	<i>Lecture summary</i>	236
<b>9.9</b>	<i>Exercises</i>	237
<b>9.10</b>	<i>Bibliographical notes</i>	237
<b>9.11</b>	<i>Further explorations</i>	237
<b>9.12</b>	<i>Acknowledgements</i>	237
<b>10</b>	<i>Is every function computable?</i>	239
10.1	<i>The Halting problem</i>	241
10.1.1	<i>Is the Halting problem really hard?</i>	243
10.1.2	<i>Reductions</i>	244
10.2	<i>Impossibility of general software verification</i>	245
10.2.1	<i>Rice's Theorem</i>	248
10.3	<i>Lecture summary</i>	250
10.4	<i>Exercises</i>	250
10.5	<i>Bibliographical notes</i>	251
10.6	<i>Further explorations</i>	251
10.7	<i>Acknowledgements</i>	251
<b>11</b>	<i>Restricted computational models</i>	253
11.1	<i>Turing completeness as a bug</i>	254
11.2	<i>Regular expressions</i>	256
11.3	<i>Context free grammars.</i>	262

<i>11.4 Unrestricted grammars</i>	267
<i>11.5 Lecture summary</i>	267
<i>11.6 Exercises</i>	267
<i>11.7 Bibliographical notes</i>	267
<i>11.8 Further explorations</i>	268
<i>11.9 Acknowledgements</i>	268
<b>12 <i>Is every theorem provable?</i></b>	<b>269</b>
<i>12.1 Unsolvability of Diophantine equations</i>	269
<i>12.1.1 "Baby" MRDP Theorem: hardness of quantified Diophantine equations</i>	271
<i>12.2 Proving the unsolvability of quantified integer statements.</i>	273
<i>12.2.1 Quantified mixed statements and computation traces</i>	273
<i>12.2.2 "Unraveling" NAND++ programs and quantified mixed integer statements</i>	274
<i>12.2.3 Reducing mixed statements to integer statements</i>	278
<i>12.3 Hilbert's Program and Gödel's Incompleteness Theorem</i>	279
<i>12.3.1 The Gödel statement</i>	282
<i>12.4 Lecture summary</i>	283
<i>12.5 Exercises</i>	283
<i>12.6 Bibliographical notes</i>	284
<i>12.7 Further explorations</i>	284
<i>12.8 Acknowledgements</i>	285
<b>13 <i>Efficient computation</i></b>	<b>287</b>
<i>13.1 Problems on graphs</i>	288
<i>13.1.1 Finding the shortest path in a graph</i>	290
<i>13.1.2 Finding the longest path in a graph</i>	291
<i>13.1.3 Finding the minimum cut in a graph</i>	291
<i>13.1.4 Finding the maximum cut in a graph</i>	295
<i>13.1.5 A note on convexity</i>	295
<i>13.2 Beyond graphs</i>	297
<i>13.2.1 The 2SAT problem</i>	297
<i>13.2.2 The 3SAT problem</i>	298
<i>13.2.3 Solving linear equations</i>	298
<i>13.2.4 Solving quadratic equations</i>	299
<i>13.3 More advanced examples</i>	299
<i>13.3.1 The permanent (mod 2) problem</i>	299
<i>13.3.2 The permanent (mod 3) problem</i>	300
<i>13.3.3 Finding a zero-sum equilibrium</i>	300

<i>13.3.4 Finding a Nash equilibrium</i>	301
<i>13.3.5 Primality testing</i>	301
<i>13.3.6 Integer factoring</i>	302
<i>13.4 Our current knowledge</i>	302
<i>13.5 Lecture summary</i>	303
<i>13.6 Exercises</i>	303
<i>13.7 Bibliographical notes</i>	304
<i>13.8 Further explorations</i>	304
<i>13.9 Acknowledgements</i>	304
<b>14 Modeling running time</b>	<b>305</b>
<i>14.1 NAND« vs NAND++</i>	308
<i>14.2 Efficient universal machine: a NAND« interpreter in NAND«</i>	310
<i>14.3 Time hierarchy theorem</i>	312
<i>14.4 Simulating NAND« or NAND++ programs with NAND programs</i>	314
<i>14.5 Simulating NAND with NAND++?</i>	316
<i>14.5.1 Uniform vs. Nonuniform computation: A recap</i>	317
<i>14.6 Extended Church-Turing Thesis</i>	317
<i>14.7 Lecture summary</i>	318
<i>14.8 Exercises</i>	319
<i>14.9 Bibliographical notes</i>	320
<i>14.10 Further explorations</i>	320
<i>14.11 Acknowledgements</i>	320
<b>15 Polynomial-time reductions</b>	<b>321</b>
<i>15.0.1 Decision problems</i>	322
<i>15.1 Reductions</i>	322
<i>15.2 Some example reductions</i>	323
<i>15.2.1 Reducing 3SAT to quadratic equations</i>	324
<i>15.3 The independent set problem</i>	325
<i>15.4 Reducing Independent Set to Maximum Cut</i>	328
<i>15.5 Reducing 3SAT to Longest Path</i>	330
<i>15.6 Exercises</i>	331
<i>15.7 Bibliographical notes</i>	332
<i>15.8 Further explorations</i>	332
<i>15.9 Acknowledgements</i>	332

<b>16</b>	<b><i>NP, NP completeness, and the Cook-Levin Theorem</i></b>	<b>333</b>
<b>16.1</b>	<b><i>The class NP</i></b>	<b>333</b>
<b>16.1.1</b>	<b><i>Examples:</i></b>	<b>334</b>
<b>16.1.2</b>	<b><i>From NP to 3SAT</i></b>	<b>335</b>
<b>16.1.3</b>	<b><i>What does this mean?</i></b>	<b>336</b>
<b>16.2</b>	<b><i>The Cook-Levin Theorem</i></b>	<b>338</b>
<b>16.2.1</b>	<b><i>The NANDSAT Problem, and why it is NP hard.</i></b>	<b>338</b>
<b>16.2.2</b>	<b><i>The 3NAND problem</i></b>	<b>340</b>
<b>16.2.3</b>	<b><i>From 3NAND to 3SAT</i></b>	<b>343</b>
<b>16.2.4</b>	<b><i>Wrapping up</i></b>	<b>343</b>
<b>16.3</b>	<b><i>Lecture summary</i></b>	<b>344</b>
<b>16.4</b>	<b><i>Exercises</i></b>	<b>344</b>
<b>16.5</b>	<b><i>Bibliographical notes</i></b>	<b>344</b>
<b>16.6</b>	<b><i>Further explorations</i></b>	<b>344</b>
<b>16.7</b>	<b><i>Acknowledgements</i></b>	<b>345</b>
<b>17</b>	<b><i>Advanced hardness reductions (advanced lecture)</i></b>	<b>347</b>
<b>17.1</b>	<b><i>Lecture summary</i></b>	<b>347</b>
<b>17.2</b>	<b><i>Exercises</i></b>	<b>347</b>
<b>17.3</b>	<b><i>Bibliographical notes</i></b>	<b>347</b>
<b>17.4</b>	<b><i>Further explorations</i></b>	<b>347</b>
<b>17.5</b>	<b><i>Acknowledgements</i></b>	<b>347</b>
<b>18</b>	<b><i>What if P equals NP?</i></b>	<b>349</b>
<b>18.1</b>	<b><i>Search to decision reduction</i></b>	<b>350</b>
<b>18.2</b>	<b><i>Quantifier elimination</i></b>	<b>352</b>
<b>18.2.1</b>	<b><i>Approximating counting problems</i></b>	<b>353</b>
<b>18.3</b>	<b><i>What does all of this imply?</i></b>	<b>354</b>
<b>18.4</b>	<b><i>Can P <math>\neq</math> NP be neither true nor false?</i></b>	<b>356</b>
<b>18.5</b>	<b><i>Is P = NP “in practice”?</i></b>	<b>357</b>
<b>18.6</b>	<b><i>What if P <math>\neq</math> NP?</i></b>	<b>358</b>
<b>18.7</b>	<b><i>Lecture summary</i></b>	<b>359</b>
<b>18.8</b>	<b><i>Exercises</i></b>	<b>359</b>
<b>18.9</b>	<b><i>Bibliographical notes</i></b>	<b>359</b>
<b>18.10</b>	<b><i>Further explorations</i></b>	<b>359</b>
<b>18.11</b>	<b><i>Acknowledgements</i></b>	<b>359</b>

19	<i>Probability Theory 101</i>	361
	19.1 <i>Random coins</i>	361
	19.1.1 <i>Random variables</i>	363
	19.1.2 <i>More general sample spaces.</i>	364
	19.2 <i>Correlations and independence</i>	364
	19.2.1 <i>Independent random variables</i>	366
	19.2.2 <i>Collections of independent random variables.</i>	367
	19.3 <i>Concentration</i>	367
	19.4 <i>Chebyshev's Inequality</i>	369
	19.5 <i>The Chernoff bound</i>	370
	19.6 <i>Lecture summary</i>	371
	19.7 <i>Exercises</i>	371
	19.8 <i>Bibliographical notes</i>	373
	19.9 <i>Further explorations</i>	373
	19.10 <i>Acknowledgements</i>	373
20	<i>Probabilistic computation</i>	375
	20.1 <i>Finding approximately good maximum cuts.</i>	376
	20.1.1 <i>Amplification</i>	377
	20.1.2 <i>What does this mean?</i>	377
	20.1.3 <i>Solving SAT through randomization</i>	378
	20.1.4 <i>Bipartite matching.</i>	380
	20.2 <i>Lecture summary</i>	383
	20.3 <i>Exercises</i>	383
	20.4 <i>Bibliographical notes</i>	384
	20.5 <i>Further explorations</i>	384
	20.6 <i>Acknowledgements</i>	384
21	<i>Modeling randomized computation</i>	385
	21.1 <i>The power of randomization</i>	387
	21.2 <i>Simulating RNAND programs by NAND programs</i>	387
	21.3 <i>Derandomizing uniform computation</i>	388
	21.4 <i>Pseudorandom generators</i>	390
	21.4.1 <i>Existence of pseudorandom generators</i>	392
	21.4.2 <i>Usefulness of pseudorandom generators</i>	394
	21.5 <i>Lecture summary</i>	394
	21.6 <i>Exercises</i>	394

21.7	<i>Bibliographical notes</i>	394
21.8	<i>Further explorations</i>	394
21.9	<i>Acknowledgements</i>	394
22	<i>Hardness vs. Randomness</i>	395
22.1	<i>Lecture summary</i>	395
22.2	<i>Exercises</i>	395
22.3	<i>Bibliographical notes</i>	395
22.4	<i>Further explorations</i>	395
22.5	<i>Acknowledgements</i>	395
23	<i>Derandomization from worst-case assumptions (advanced lecture)</i>	397
23.1	<i>Lecture summary</i>	397
23.2	<i>Exercises</i>	397
23.3	<i>Bibliographical notes</i>	397
23.4	<i>Further explorations</i>	397
23.5	<i>Acknowledgements</i>	397
24	<i>Cryptography</i>	399
24.1	<i>Lecture summary</i>	399
24.2	<i>Exercises</i>	399
24.3	<i>Bibliographical notes</i>	399
24.4	<i>Further explorations</i>	399
24.5	<i>Acknowledgements</i>	399
25	<i>Algorithms and society</i>	401
25.1	<i>Lecture summary</i>	401
25.2	<i>Exercises</i>	401
25.3	<i>Bibliographical notes</i>	401
25.4	<i>Further explorations</i>	401
25.5	<i>Acknowledgements</i>	401
26	<i>Compression, coding, and information</i>	403
26.1	<i>Lecture summary</i>	403
26.2	<i>Exercises</i>	403
26.3	<i>Bibliographical notes</i>	403

26.4	<i>Further explorations</i>	403
26.5	<i>Acknowledgements</i>	403
27	<i>Space bounded computation</i>	405
27.1	<i>Lecture summary</i>	405
27.2	<i>Exercises</i>	405
27.3	<i>Bibliographical notes</i>	405
27.4	<i>Further explorations</i>	405
27.5	<i>Acknowledgements</i>	405
28	<i>Streaming, sketching, and automata</i>	407
28.1	<i>Lecture summary</i>	407
28.2	<i>Exercises</i>	407
28.3	<i>Bibliographical notes</i>	407
28.4	<i>Further explorations</i>	407
28.5	<i>Acknowledgements</i>	407
29	<i>Proofs and algorithms</i>	409
29.1	<i>Lecture summary</i>	409
29.2	<i>Exercises</i>	409
29.3	<i>Bibliographical notes</i>	409
29.4	<i>Further explorations</i>	409
29.5	<i>Acknowledgements</i>	409
30	<i>Interactive proofs (advanced lecture)</i>	411
30.1	<i>Lecture summary</i>	411
30.2	<i>Exercises</i>	411
30.3	<i>Bibliographical notes</i>	411
30.4	<i>Further explorations</i>	411
30.5	<i>Acknowledgements</i>	411
31	<i>Data structure lower bounds</i>	413
31.1	<i>Lecture summary</i>	413
31.2	<i>Exercises</i>	413
31.3	<i>Bibliographical notes</i>	413
31.4	<i>Further explorations</i>	413

31.5	<i>Acknowledgements</i>	413
32	<i>Quantum computing</i>	415
32.1	<i>Lecture summary</i>	415
32.2	<i>Exercises</i>	415
32.3	<i>Bibliographical notes</i>	415
32.4	<i>Further explorations</i>	415
32.5	<i>Acknowledgements</i>	415
33	<i>Shor's Algorithm (advanced lecture)</i>	417
33.1	<i>Lecture summary</i>	417
33.2	<i>Exercises</i>	417
33.3	<i>Bibliographical notes</i>	417
33.4	<i>Further explorations</i>	417
33.5	<i>Acknowledgements</i>	417
<i>Appendix: The NAND* Programming Languages</i>		419
33.6	<i>NAND programming language specification</i>	419
33.6.1	<i>Syntax of NAND programs</i>	419
33.6.2	<i>Semantics of NAND programs</i>	420
33.7	<i>NAND++ programming language specification</i>	420
33.7.1	<i>Syntax of NAND++ programs</i>	421
33.7.2	<i>Semantics of NAND++ programs</i>	421
33.7.3	<i>Interpreting NAND programs</i>	422
33.8	<i>NAND« programming language specification</i>	425
33.8.1	<i>Syntax of NAND« programs</i>	425
33.8.2	<i>Semantics of NAND« programs</i>	426
33.9	<i>The "standard library"</i>	428
33.9.1	<i>Syntactic sugar for NAND</i>	428
33.9.2	<i>Encoding of integers, strings, lists.</i>	430
33.9.3	<i>Syntactic sugar for NAND++ / NAND«</i>	432
<i>Bibliography</i>		433



# 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, Sam Benkelman, Jarosław Błasiok, Christy Cheng, Daniel Chiu, Chi-Ning Chou, Michael Colavita, Robert Darley Waddilove, Juan Esteller, Leor Fishma, Mark Goldstein, Chan Kang, Estefania Lahera, Allison Lee, Ondřej Lengál, Emma Ling, Alex Lombardi, Lisa Lu, Aditya Mahadevan, Jacob Meyerson, Hamish Nicholson, Thomas Orton, Juan Perdomo, Aaron Sachs, Brian Sapozhnikov, John Seides, Alaisha Sharma, Alec Sun, Everett Sussman, Garrett Tanzer, Sarah Turnill, Salil Vadhan, Ryan Williams, Wanqian Yang, 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>.

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 produced 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 Tufte [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?*”.<sup>1</sup> To even phrase such a question,

<sup>1</sup> 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 ( $\in$ ), containment ( $\subseteq$ ), and set operations such as union, intersection, set difference and Cartesian product ( $\cup, \cap, \setminus$  and  $\times$ ).
- **Functions:** Including the notions of the *domain* and *range* of a function, properties such being *one-to-one* (also known as *injective*) or *onto* (also known as *surjective*) 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 ( $\wedge, \vee, \neg$ ) and the quantifiers “exists” and “forall” ( $\exists, \forall$ ).
- **Tuples and strings:** The notation  $\Sigma^k$  and  $\Sigma^*$  where  $\Sigma$  is some finite set which is called the *alphabet* (quite often  $\Sigma = \{0, 1\}$ ).
- **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:**  $O, o, \Omega, \omega, \Theta$  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.<sup>2</sup> 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

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

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

$$\text{EVEN} = \{0, 2, 4, \dots\}. \quad (2)$$

<sup>2</sup> Later in this course we will discuss how to extend the notion of cardinality to *infinite* sets.

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  $\{ \text{New York}, \text{Los Angeles}, \text{Chicago}, \text{Houston}, \text{Philadelphia}, \text{Phoenix}, \text{San Antonio}, \text{San Diego}, \text{Dallas} \}$  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 in  $T$ . 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  $(1, 2, 4, 9, \dots)$  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.<sup>3</sup>

**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  $s \in S$ ,  $t \in T$ , and  $u \in U$ . More generally, for every positive integer  $n$  and sets  $S_0, \dots, S_{n-1}$ , we denote by  $S_0 \times S_1 \times \dots \times S_{n-1}$  the set of ordered  $n$ -tuples  $(s_0, \dots, s_{n-1})$  where  $s_i \in S_i$  for every  $i \in \{0, \dots, n-1\}$ .

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.

<sup>3</sup> We can identify a sequence  $(a_0, a_1, a_2, \dots)$  of elements in some set  $S$  with a function  $A : \mathbb{N} \rightarrow S$  (where  $a_n = A(n)$  for every  $n \in \mathbb{N}$ ). Similarly, we can identify a  $k$ -tuple  $(a_0, \dots, a_{k-1})$  of elements in  $S$  with a function  $A : [k] \rightarrow S$ .

### o.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

$$\mathbb{N} = \{0, 1, 2, \dots\} \quad (3)$$

to be the set of all *natural numbers*, i.e., non-negative integers. For any natural number  $n$ , we define the set  $[n]$  as  $\{0, \dots, n - 1\} = \{k \in \mathbb{N} : k < n\}$ .<sup>4</sup>

We will also occasionally use the set  $\mathbb{Z} = \{\dots, -2, -1, 0, +1, +2, \dots\}$  of (negative and non-negative) *whole numbers*,<sup>5</sup> 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

$$\{0, 1\}^n = \{(x_0, \dots, x_{n-1}) : x_0, \dots, x_{n-1} \in \{0, 1\}\} \quad (4)$$

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  $(x_0, x_1, \dots, x_{n-1})$  as simply  $x_0 x_1 \cdots x_{n-1}$  and so for example

$$\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}. \quad (5)$$

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  $i \in \{n, \dots, n+m-1\}$ ,  $z_i = y_{i-n}$ .

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

<sup>4</sup> We start our indexing of both  $\mathbb{N}$  and  $[n]$  from 0, while many other texts index those sets from 1. Starting from zero or one is simply a convention that doesn't make much difference, as long as one is consistent about it.

<sup>5</sup> The letter Z stands for the German word "Zahlen", which means *numbers*.

which is

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

Another way to write this set is as

$$\{0,1\}^* = \{0,1\}^0 \cup \{0,1\}^1 \cup \{0,1\}^2 \cup \dots \quad (7)$$

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 “”.<sup>6</sup>

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

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

For example, if  $\Sigma = \{a, b, c, d, \dots, z\}$  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  $n \leq i < n+m$ .

<sup>6</sup> 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”.

### 0.6.3 Functions

If  $S$  and  $T$  are sets, a *function*  $F$  mapping  $S$  to  $T$ , denoted by  $F : S \rightarrow T$ , associates with every element  $x \in S$  an element  $F(x) \in T$ . The set  $S$  is known as the *domain* of  $F$  and the set  $T$  is known as the *range* or *co-domain* of  $F$ . 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

Is the same as defining  $F(x) = (x \bmod 2)$ . If  $F : S \rightarrow T$  satisfies that  $F(x) \neq F(y)$  for all  $x \neq y$  then we say that  $F$  is *one-to-one* (also known as an *injective* function or simply an *injection*).

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* (also known as a *surjective* function or simply a *surjection*). A function that is both one-to-one and onto is

Input	Output
0	0
1	1
2	0
3	1
4	0
5	1
6	0
7	1
8	0
9	1

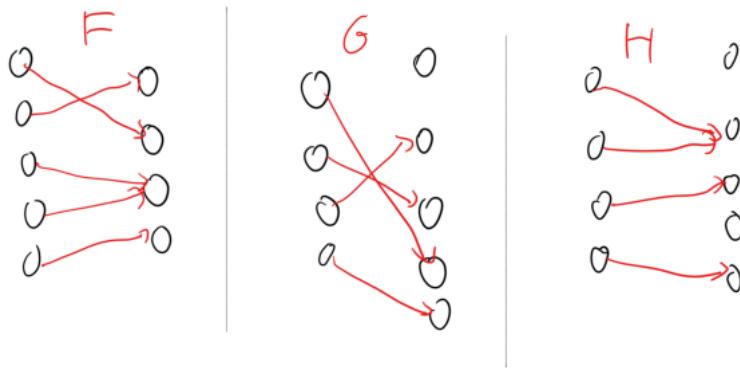
known as a *bijective* function or simply a *bijection*. If  $S = T$  then a bijection from  $S$  to  $T$  is also known as a *permutation*. If  $F : S \rightarrow T$  is a bijection then for every  $y \in T$  there is a unique  $x \in S$  s.t.  $F(x) = y$ . We denote this value  $x$  by  $F^{-1}(y)$ . 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  $f : S \rightarrow T$ . In particular, the cardinality of a set  $S$  is defined  $n$  if there is a bijection from  $S$  to the set  $\{0, \dots, n - 1\}$ . As we will see later in this course, this is a definition that can generalize to defining the cardinality of *infinite* sets.

**Partial functions:** We will sometimes be interested in *partial* functions from  $S$  to  $T$ . This is a generalization of the notion of a function to consider also  $F$  that is not necessarily defined on every element of  $S$ . For example, the partial 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. When we say “function” without any qualifier then we mean a *total* function. That is, the notion of partial functions is a strict generalization of functions, and so a partial function *not* necessarily a function. The set of partial functions is a proper superset of the set of total functions, since a partial function is allowed to be defined on all its input elements. When we want to emphasize that a function  $f$  from  $A$  to  $B$  might not be total, we will write  $f : A \rightarrow_p B$ . We can think of a partial function  $F$  from  $S$  to  $T$  also as a total function from  $S$  to  $T \cup \{\perp\}$  where  $\perp$  is some special “failure symbol”, and so instead of saying that  $F$  is undefined at  $x$ , we can say that  $F(x) = \perp$ .

**Basic facts about functions:** Verifying that you can prove the following results is an excellent way to brush up on functions:

- If  $F : S \rightarrow T$  and  $G : T \rightarrow U$  are one-to-one functions, then their composition  $H : S \rightarrow U$  defined as  $H(s) = G(F(s))$  is also one to one.
- If  $F : S \rightarrow T$  is one to one, then there exists an onto function  $G : T \rightarrow S$  such that  $G(F(s)) = s$  for every  $s \in S$ .
- If  $G : T \rightarrow S$  is onto then there exists a one-to-one function  $F : S \rightarrow T$  such that  $G(F(s)) = s$  for every  $s \in S$ .
- If  $S$  and  $T$  are finite sets then the following conditions are equivalent to one another: **(a)**  $|S| \leq |T|$ , **(b)** there is a one-to-one function  $F : S \rightarrow T$ , and **(c)** there is an onto function  $G : T \rightarrow S$ .



**Figure 1:** We can represent finite functions as a directed graph where we put an edge from  $x$  to  $f(x)$ . The *onto* condition corresponds to requiring that every vertex in the range of the function has in-degree *at least* one. The *one-to-one* condition corresponds to requiring that every vertex in the range of the function has in-degree *at most* one. In the examples above  $F$  is an onto function,  $G$  is one to one, and  $H$  is neither onto nor one to one.



You can find the proofs of these results in many discrete math texts, including for example, section 4.5 in the [Leham-Leighton-Meyer notes](#). However, I strongly suggest you try to prove them on your own, or at least convince yourself that they are true by proving special cases of those for small sizes (e.g.,  $|S| = 3, |T| = 4, |U| = 5$ ).

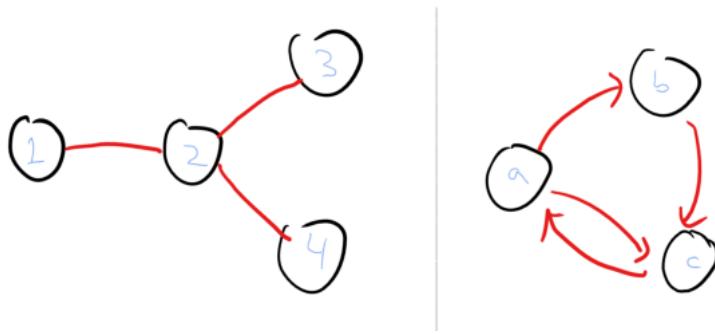
Let us prove one of these facts as an example:

**Lemma 0.1** If  $S, T$  are non-empty sets and  $F : S \rightarrow T$  is one to one, then there exists an onto function  $G : T \rightarrow S$  such that  $G(F(s)) = s$  for every  $s \in S$ .

*Proof.* Let  $S, T$  and  $F : S \rightarrow T$  be as in the Lemma's statement, and choose some  $s_0 \in S$ . We will define the function  $G : T \rightarrow S$  as follows: for every  $t \in T$ , if there is some  $s \in S$  such that  $F(s) = t$  then set  $G(t) = s$  (the choice of  $s$  is well defined since by the one-to-one property of  $F$ , there cannot be two distinct  $s, s'$  that both map to  $t$ ). Otherwise, set  $G(t) = s_0$ . Now for every  $s \in S$ , by the definition of  $G$ , if  $t = F(s)$  then  $G(t) = G(F(s)) = s$ . Moreover, this also shows that  $G$  is *onto*, since it means that for every  $s \in S$  there is some  $t$  (namely  $t = F(s)$ ) such that  $G(t) = s$ .  $\blacksquare$

#### 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* and *directed*.<sup>7</sup>



**Figure 2:** An example of an undirected and a directed graph. The undirected graph has vertex set  $\{1, 2, 3, 4\}$  and edge set  $\{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{2, 4\}\}$ . The directed graph has vertex set  $\{a, b, c\}$  and the edge set  $\{(a, b), (b, c), (c, a), (a, c)\}$ .

<sup>7</sup> 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  $uv$  and  $vu$  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  $u$  has. A *path* in the graph is a tuple  $(u_0, \dots, u_k) \in V^k$ , 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  $(u_0, \dots, u_{k-1})$  where all the  $u_i$ 's are distinct. A *cycle*

is a path  $(u_0, \dots, u_k)$  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  $(u_0, \dots, u_k)$  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 0.2** In any undirected graph  $G = (V, E)$ , the sum of the degrees of all vertices is equal to twice the number of edges.

**Lemma 0.2** 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 0.3** 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 0.3** can be shown by simply attaching a path of the form  $(u, u_1, u_2, \dots, u_{k-1}, v)$  to a path of the form  $(v, u'_1, \dots, u'_{k'-1}, w)$  to obtain the path  $(u, u_1, \dots, u_{k-1}, v, u'_1, \dots, u'_{k'-1}, w)$  that connects  $u$  to  $w$ .

**Lemma 0.4** 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 0.4** can be shown by “shortcutting” any non simple path of the form  $(u, u_1, \dots, u_{i-1}, w, u_{i+1}, \dots, u_{j-1}, w, u_{j+1}, \dots, u_{k-1}, v)$  where the same vertex  $w$  appears in both the  $i$ -th and  $j$ -position, to obtain the shorter path  $(u, u_1, \dots, u_{i-1}, w, u_{j+1}, \dots, u_{k-1}, v)$ .



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  $\vec{uv}$ . If the edge  $\vec{uv}$  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  $\vec{uv}$  and  $\vec{vu}$  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  $(u_0, \dots, u_k) \in V^k$ , 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  $(u_0, \dots, u_{k-1})$  where all the  $u_i$ 's are distinct and a *cycle* is a path  $(u_0, \dots, u_k)$  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 0.5** 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 0.6** 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 0.7** 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.



**Graph terminology** The word *graph* in the sense above was coined by the mathematician Sylvester in 1878 in analogy with the chemical graphs used to visualize molecules. There is an unfortunate confusion with the more common usage of the term as a way to plot data, and in particular a plot of some function  $f(x)$  as a function of  $x$ . We can merge these two meanings by thinking of a function  $f : A \rightarrow B$  as a special case of a directed graph over the vertex set  $V = A \cup B$  where we put the edge  $\overrightarrow{xf(x)}$  for every  $x \in A$ . In a graph constructed in this way every vertex in  $A$  has out-degree one.

The following [lecture of Berkeley CS70](#) provides an excellent overview of graph theory.

### 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  $P \wedge Q$ ) is the statement that is true if and only if both  $P$  and  $Q$  are true, and  $P$  OR  $Q$  (denoted as  $P \vee Q$ ) 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  $\bar{P}$ , 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  $x \in S$ .<sup>8</sup> 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:

$$\exists_{n \in \mathbb{N}} (n > 100) \wedge (\forall_{k \in \mathbb{N}} k + k + k \neq n) . \quad (10)$$

*“For sufficiently large  $n$ ”* One expression which comes up time and again is the claim that some statement  $P(n)$  is true “for sufficiently large  $n$ ”. What this means is that there exists an integer  $N_0$  such that  $P(n)$  is true for every  $n > N_0$ . We can formalize this as  $\exists_{N_0 \in \mathbb{N}} \forall_{n > N_0} P(n)$ .

### o.6.6 Quantifiers for summations and products

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

$$f(s_0) + f(s_1) + f(s_2) + \dots + f(s_{n-1}) , \quad (11)$$

and  $\prod_{x \in S} f(x)$  as shorthand for

$$f(s_0) \cdot f(s_1) \cdot f(s_2) \cdot \dots \cdot f(s_{n-1}) . \quad (12)$$

For example, the sum of the squares of all numbers from 1 to 100 can be written as

$$\sum_{i \in \{1, \dots, 100\}} i^2 . \quad (13)$$

Since summing up over intervals of integers is so common, there is a special notation for it, and for every two integers  $a \leq b$ ,  $\sum_{i=a}^b f(i)$

<sup>8</sup>In these notes we will place the variable that is bound by a quantifier in a subscript and so write  $\forall_{x \in S} P(x)$  whereas other texts might use  $\forall x \in S. P(x)$ .

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

$$\sum_{i=1}^{100} i^2. \quad (14)$$

### o.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 *bound* or *free* in this formula. For example, in the following statement  $n$  is free but  $a, b$  are bound by the  $\exists$  quantifier:

$$\exists_{a,b \in \mathbb{N}} (a \neq 1) \wedge (a \neq n) \wedge (n = a \times b) \quad (15)$$

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

```
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

$$\exists_{x,y \in \mathbb{N}} (x \neq 1) \wedge (x \neq n) \wedge (n = x \times y) \quad (16)$$

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

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

produces the same result.



**Aside: mathematical vs programming notation** Mathematical notation has a lot of similarities with programming language, and for the same reasons. Both are formalisms meant to convey complex concepts in a precise way. However, there are some cultural differences. In programming languages, we often try to use meaningful variable names such as `NumberOfVertices` while in math we often use short identifiers such as  $n$ . (Part of it might have to do with the tradition of mathematical proofs as being handwritten and verbally presented, as opposed to typed up and compiled.) One consequence of that is that in mathematics we often end up reusing identifier, and also “run out” of letters and hence use greek letters too, as well as distinguish between small and capital letters. Similarly, mathematical notation tends to use quite a lot of “overloading”, using operators such as  $+$  for a great variety of objects (e.g., real numbers, matrices, finite field elements, etc..), and assuming that the meaning can be inferred from the context. Both fields have a notion of “types”, and in math we often try to reserve certain letters for variables of a particular type. For example, variables such as  $i, j, k, \ell, m, n$  will often denote integers, and  $\epsilon$  will often denote a small positive real number. When reading or writing mathematical texts, we usually don’t have the advantage of a “compiler” that will check type safety for us. Hence it is important to keep track of the type of each variable, and see that the operations that are performed on it “make sense”.

### 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  $F, G : \mathbb{N} \rightarrow \mathbb{R}_+$ , 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  $\lim_{n \rightarrow \infty} \frac{F(n)}{G(n)} = 0$ . Similarly, if the limit  $\lim_{n \rightarrow \infty} \frac{F(n)}{G(n)}$  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  $\limsup_{n \rightarrow \infty} \frac{F(n)}{G(n)} < \infty$ .

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.

### o.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:  $n^a = o(2^{n^\epsilon})$  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 Oh 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  $F(n) \geq 0.001G(n)$ .

### o.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.<sup>9</sup> 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 0.8 — 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 0.8**, 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.

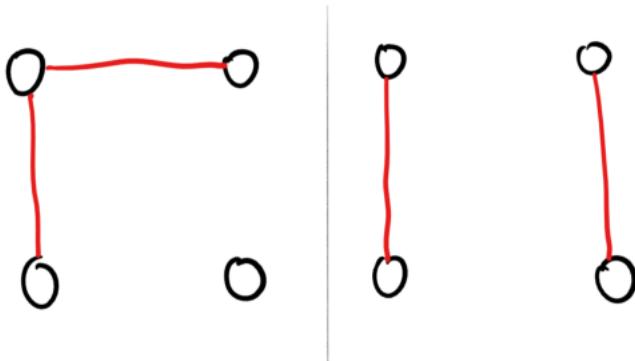
<sup>9</sup> This difference might be eroding with time, as more proofs are being written in a *machine verifiable form* and progress in artificial intelligence allows expressing programs in more human friendly ways, such as “programming by example”. Interestingly, much of the progress in automatic proof verification and proof assistants relies on a **much deeper correspondence** between *proofs* and *programs*. We *might* see this correspondence later in this course.

Before trying to prove [Theorem 0.8](#), 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  $(u_0, u_1, \dots, u_k)$  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 0.8](#) 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. 3](#), and in both there will remain a vertex that is not connected.



**Figure 3:** 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.

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*.

**P** 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 *proof by induction*.

### 0.8.1 Mathematical induction

*Induction* is simply an application of the self-evident **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  $Q(0), \dots, Q(k-1)$  are all true then  $Q(k)$  is true.<sup>10</sup> 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”.

<sup>10</sup> Usually proving **(b)** is the hard part, though there are examples where the “base case” **(a)** is quite subtle.

**R** **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-Leighoton book useful.

### o.8.2 Proving the theorem by induction

There are several ways to use induction to prove [Theorem o.8](#). 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:

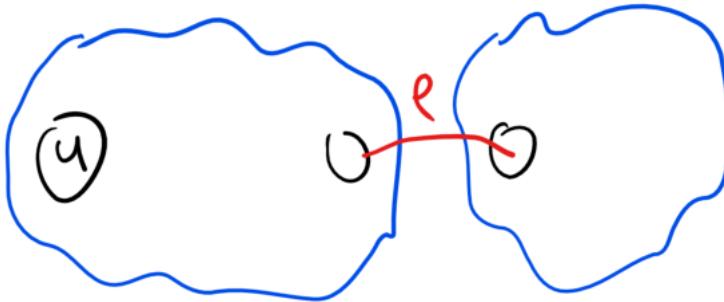
$Q(k)$  is “For every graph  $G = (V, E)$  with at most  $k$  edges and every  $u \in V$ , the number of vertices that are connected to  $u$  (including  $u$  itself) is at most  $k + 1$ ”

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  $Q(0), \dots, Q(k - 1)$  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 o.8](#).

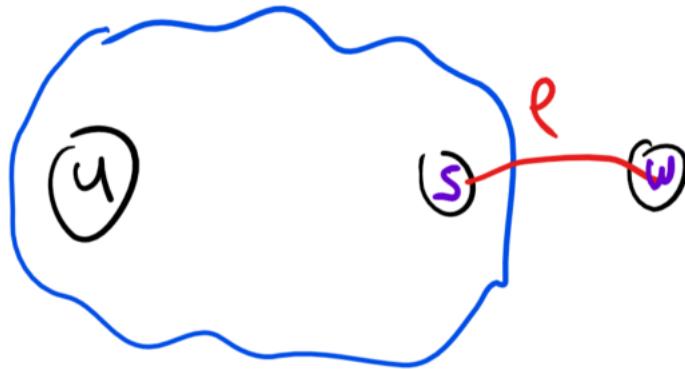
Proving **(a)** (i.e., the “base case”) is actually quite easy. The statement  $Q(0)$  says that if  $G$  has zero edges, then  $|C_G(u)| = 1$ , but this is clear because in a graph with zero edges,  $u$  is only connected to itself. The heart of the proof is, as typical with induction proofs, is in 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  $|C_G(u)| \leq |C_{G'}(u)| + 1 \leq k + 1$ .

**P** Please ensure that you understand why showing that  $|C_G(u)| \leq |C_{G'}(u)| + 1$  completes the inductive proof.



**Figure 4:** Removing a single edge  $e$  can greatly decrease the number of vertices that are connected to a vertex  $u$ .

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. 4. This might seem as a real stumbling 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 connect to  $u$  by at most one vertex.



**Figure 5:** Removing an edge  $e = \{s, w\}$  where  $w \in C_G(u)$  has degree one removes only  $w$  from  $C_G(u)$ .

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. 5. 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 0.2](#) 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 ■.<sup>11</sup> 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

<sup>11</sup> QED stands for “quod erat demonstrandum”, which is “What was to be demonstrated.” or “The very thing it was required to have shown.” in Latin.

or claim it belongs.

Let us see how the proof of [Theorem o.8](#) looks in this streamlined fashion. We start by repeating the theorem statement

**Theorem 0.9 — Minimum edges for connected graphs (restated).** Every connected undirected graph of  $n$  vertices has at least  $n - 1$  edges.

*Proof of Theorem o.9.* The proof will follow from the following lemma:

**Lemma 0.10** 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 o.10](#) implies the theorem:

**Proof of Theorem 0.9 from Lemma 0.10:** 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 o.10](#), 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 0.9 from Lemma o.10)**

We now turn to proving [Lemma o.10](#). Let  $G = (V, E)$  be an undirected graph of  $k$  edges and  $u \in V$ . We define  $C_G(u)$  to be the set of vertices connected to  $u$ . To complete the proof of [Lemma o.10](#), 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 o.10](#) 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 o.2](#),  $\sum_{v \in S} \deg(v) \leq \sum_{v \in V} \deg(v) = 2|E| = 2k$ . 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 0.4 implies that there must be some *simple* path  $(t_0, t_1, \dots, t_{i-1}, t_i)$  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 0.10 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 0.10 and hence also of Theorem 0.9. **QED (Lemma 0.10), QED (Theorem 0.9)**

■

The proof above used the observation that if the *average* of some  $n$  numbers  $x_0, \dots, x_{n-1}$  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 as-

sumption 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 0.11** 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  $2 = a^2/b^2$  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 (\*) 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  $\frac{a/2}{b/2} = \sqrt{2}$ , 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 0.12** For every natural number  $n \in N$ , either  $n$  or  $n + 1$  is even.

*Proof.* Let  $n \in N$  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  $A$  is true” and end with a derivation of  $B$  from  $A$ . Here is a simple example:

**Lemma 0.13** 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  $x = (-b + s)/(2a)$  satisfies

$$ax^2 + bx + c = a(-b + s)^2/(4a^2) + b(-b + s)/(2a) + c = (b^2 - 2bs + s^2)/(4a) + (-b^2 + bs)/(2a) + c. \quad (17)$$

Rearranging the terms of Eq. (17) we get

$$s^2/(4a) + c - b^2/(4a) = (b^2 - 4ac)/(4a) + c - b^2/(4a) = 0 \quad (18)$$

■

**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  $X_1 \wedge X_2 \wedge X_3$  implies  $X$ .<sup>12</sup> Our proof of Theorem o.8 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  $C_1, \dots, C_k$ , 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)”:** 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  $X_0, X_1, \dots, X_k$ , and we prove that  $X_0$  is true, as well that  $X_0$  implies  $X_1$ , and that  $X_0 \wedge X_1$  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.

<sup>12</sup> As mentioned below,  $\wedge$  denotes the logical AND operator.

### 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  $\{0, \dots, n-1\}$ . In most texts it is defined as  $\{1, \dots, n\}$ . 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  $f : A \rightarrow B$  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  $f : A \rightarrow_p B$ .
- As we will see later on in the course, we will mostly describe our computational problems in the terms of computing a *Boolean function*  $f : \{0,1\}^* \rightarrow \{0,1\}$ . 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*.<sup>13</sup>
- 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  $(x \bmod y)$  to denote the “remainder” of  $x$  when divided by  $y$ . That is,  $(x \bmod y) = x - y\lfloor x/y \rfloor$ . In context when an integer is expected we’ll typically “silently round” the quantities to an integer. For example, if we say that  $x$  is a string of length  $\sqrt{n}$  then we’ll typically mean that  $x$  is of length  $\lceil \sqrt{n} \rceil$ . (In most such cases, it will not make a difference whether we round up or down.)
- Like most Computer Science texts, we default to the logarithm in base two. Thus,  $\log n$  is the same as  $\log_2 n$ .
- We will also use the notation  $f(n) = \text{poly}(n)$  as a short hand for  $f(n) = n^{O(1)}$  (i.e., as shorthand for saying that there is some constants  $a, b$  such that  $f(n) \leq a \cdot n^b$  for every sufficiently large  $n$ ). Similarly, we will use  $f(n) = \text{polylog}(n)$  as shorthand for  $f(n) = \text{poly}(\log n)$  (i.e., as shorthand for saying that there are some constant  $a, b$  such that  $f(n) \leq a \cdot (\log n)^b$  for every sufficiently

<sup>13</sup> Because the language notation is so prevalent in textbooks, we will occasionally remind the reader of this correspondence.

large  $n$ ).

### 0.11 Exercises

**Exercise 0.1 — Inclusion Exclusion.** 1. Let  $A, B$  be finite sets. Prove that  $|A \cup B| = |A| + |B| - |A \cap B|$ .

2. Let  $A_0, \dots, A_{k-1}$  be finite sets. Prove that  $|A_0 \cup \dots \cup A_{k-1}| \geq \sum_{i=0}^{k-1} |A_i| - \sum_{0 \leq i < j < k} |A_i \cap A_j|$ .
3. Let  $A_0, \dots, A_{k-1}$  be finite subsets of  $\{1, \dots, n\}$ , 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  $F : S \rightarrow T$  is one to one then  $|S| \leq |T|$ .

■

**Exercise 0.3** Prove that if  $S, T$  are finite and  $F : S \rightarrow T$  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  $n > 1$   $S_n \leq 5S_{\lfloor \frac{n}{5} \rfloor} + 2n$ . Prove by induction that  $S_n \leq 100n \log n$  for every  $n$ .

■

**Exercise 0.6** Describe the following statement in English words:

$$\forall_{n \in \mathbb{N}} \exists_{p > n} \forall_{a, b \in \mathbb{N}} (a \times b \neq p) \vee (a = 1).$$

■

**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.  $F(n) = n$ ,  $G(n) = 2^{\sqrt{\log n}}$

■

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

**Exercise 0.11 — Topological sort.** Prove that for every directed acyclic graph (DAG)  $G = (V, E)$ , there exists a map  $f : V \rightarrow \mathbb{N}$  such that  $f(u) < f(v)$  for every edge  $\overrightarrow{u v}$  in the graph.<sup>14</sup> ■

<sup>14</sup> Hint: Use induction on the number of vertices. You might want to first prove the claim that every DAG contains a *sink*: a vertex without an outgoing edge.

### 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

# 1

## *Introduction*

*"Computer Science is no more about computers than astronomy is about telescopes"*, attributed to Edsger Dijkstra.<sup>1</sup>

*"Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry."* , Paul Graham 2003.<sup>2</sup>

*"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

<sup>1</sup> 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.

<sup>2</sup> 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

MM

MMMMMM~~

MMMMMM~~

MMMMMM~~

MMMMMM~~MM~~MM~~MM~~MM~~MM~~MM~~MM~~DCCCCLVI

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 Eratosthene, 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.<sup>3</sup> 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  $i = 1, \dots, y$ : set  $result \leftarrow result + x$
3. Output  $result$

<sup>3</sup> For more on the actual algorithms the Babylonians used, see Knuth’s paper and Neugebauer’s classic book.

**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  $i = 0, \dots, n - 1$ : set  $result \leftarrow result + 10^i \times$

$y_i \times x$ , where  $y_i$  is the  $i$ -th digit of  $y$  (i.e.  $y = 10^0y_0 + 10^1y_1 + \dots + y_{n-1}10^{n-1}$ )  
 3. Output result

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.<sup>4</sup> 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.

<sup>4</sup> 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  $n^2$ .<sup>5</sup> 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.<sup>6</sup> Amazingly, Karatsuba’s algorithm is based on a faster way to multiply *two digit* numbers.

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

The gradeschool algorithm works by transforming the task of

<sup>5</sup> That is at least some  $n^2/C$  operations for some constant  $C$  or, using “Big Oh notation”,  $\Omega(n^2)$  operations. See the *mathematical background* section for a precise definition of big Oh notation.

<sup>6</sup> At the time of this writing, the **standard Python multiplication implementation** switches from the elementary school algorithm to Karatsuba’s algorithm when multiplying numbers larger than 1000 bits long.

**Figure 1.1:** The gradeschool multiplication algorithm illustrated for multiplying  $x = 10\bar{x} + \underline{x}$  and  $y = 10\bar{y} + \underline{y}$ . It uses the formula  $(10\bar{x} + \underline{x}) \times (10\bar{y} + \underline{y}) = 100\bar{x}\bar{y} + 10(\bar{x}\underline{y} + \underline{x}\bar{y}) + \underline{x}\underline{y}$ .

multiplying a pair of two digit number into *four* single-digit multiplications via the formula

$$(10\bar{x} + \underline{x}) \times (10\bar{y} + \underline{y}) = 100\bar{x}\bar{y} + 10(\bar{x}\underline{y} + \underline{x}\bar{y}) + \underline{x}\underline{y} \quad (1.1)$$

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

$$(10\bar{x} + \underline{x}) \times (10\bar{y} + \underline{y}) = 100\bar{x}\bar{y} + 10 \left[ (\bar{x} + \underline{x})(\bar{y} + \underline{y}) \right] - 10\bar{x}\bar{y} - (10 + 1)\underline{x}\underline{y} \quad (1.2)$$

which reduces multiplying the two-digit number  $x$  and  $y$  to computing the following three "simple" products:  $\bar{x}\bar{y}$ ,  $\underline{x}\underline{y}$  and  $(\bar{x} + \underline{x})(\bar{y} + \underline{y})$ .<sup>7</sup>

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

<sup>7</sup> The last term is not exactly a single digit multiplication as  $\bar{x} + \underline{x}$  and  $\bar{y} + \underline{y}$  are numbers between 0 and 18 and not between 0 and 9. As we'll see, it turns out that does not make much of a difference, since when we use the algorithm recursively, this term will have essentially half the number of digits as the original input.

**Figure 1.2:** Karatsuba’s multiplication algorithm illustrated for multiplying  $x = 10\bar{x} + \underline{x}$  and  $y = 10\bar{y} + \underline{y}$ . We compute the three orange, green and purple products  $\underline{\underline{xy}}$ ,  $\bar{x}\bar{y}$  and  $(\bar{x} + \underline{x})(\bar{y} + \underline{y})$  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  $3^\ell = n^{\log_2 3} \sim n^{1.58}$  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  $x = 10^m \bar{x} + \underline{x}$  and  $y = 10^m \bar{y} + \underline{y}$ .<sup>8</sup>
3. Use *recursion* to compute  $A = \bar{x}\bar{y}$ ,  $B = \underline{\underline{yy}}$  and  $C = (\bar{x} + \underline{x})(\bar{y} + \underline{y})$ . Note that all the numbers will have at most  $m+1$  digits.
4. Return  $(10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$

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

<sup>8</sup> 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  $x = 10^m \bar{x} + \underline{x}$  and  $y = 10^m \bar{y} + \underline{y}$ ,

$$x \times y = 10^n \bar{x} \cdot \bar{y} + 10^m (\bar{x}\bar{y} + \underline{x}\underline{y}) + \underline{x}\underline{y} \quad (1.3)$$

But we can also write the same expression as

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

which equals to the value  $(10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$  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  $\lfloor n/2 \rfloor$  digit numbers (namely  $\bar{x}\bar{y}$ ,  $\underline{x}\underline{y}$  and  $(\bar{x} + \underline{x})(\bar{y} + \underline{y})$ ) 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 case we will never run into these rounding issues. These kind of tricks work

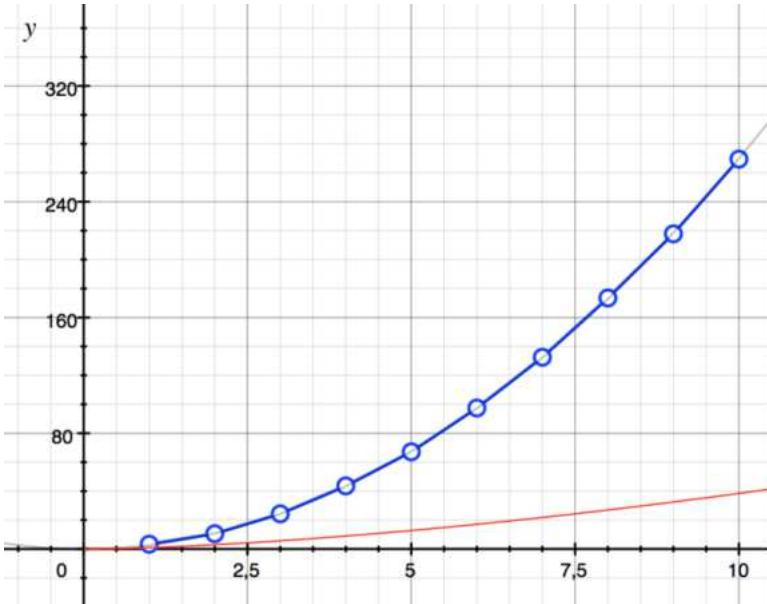


Figure 1.3: Running time of Karatsuba's algorithm vs. the Gradeschool algorithm.  
Figure by Marina Mele.

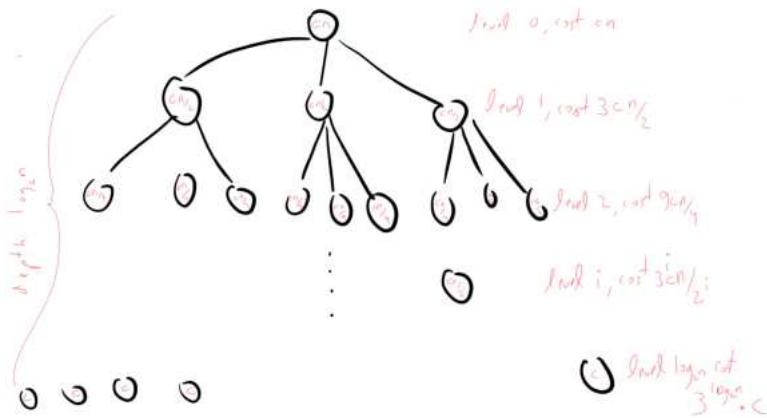


Figure 1.4: Karatsuba's algorithm reduces an  $n$ -bit multiplication to three  $n/2$ -bit multiplications, which in turn are reduced to nine  $n/4$ -bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth  $\log_2 n$ , where at the 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  $cn \sum_{i=0}^{\log_2 n} (3/2)^i \leq 2cn^{\log_2 3}$  by the formula for summing a geometric series.

not just in the context of multiplication algorithms but in many other cases as well. Thus most of the time we can safely ignore these kind of “rounding issues”.

### 1.0.2 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.<sup>9</sup> 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  $\begin{pmatrix} x_{0,0}y_{0,0} + x_{0,1}y_{1,0} & x_{0,0}y_{1,0} + x_{0,1}y_{1,1} \\ x_{1,0}y_{0,0} + x_{1,1}y_{1,0} & x_{1,0}y_{0,1} + x_{1,1}y_{1,1} \end{pmatrix}$ . 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

<sup>9</sup> The *Fourier transform* is a central tool in mathematics and engineering, used in a great number of applications. If you have not seen it yet, you will hopefully encounter it at some point in your studies.

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  $xy = \begin{pmatrix} t_1+t_4-t_5+t_7 & t_3+t_5 \\ t_2+t_4 & t_1+t_3-t_2+t_6 \end{pmatrix}$ . 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  $O(n^2 \text{polylog}(n))$  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 damped 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.<sup>10</sup>

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.<sup>11</sup> 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.

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

<sup>11</sup> 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.

## 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.

- 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  $T_1, T_2, T_3, \dots$  is a sequence of numbers such that  $T_2 \leq 10$  and for every  $n$ ,  $T_n \leq 3T_{\lceil n/2 \rceil} + Cn$ . Prove that  $T_n \leq 10Cn^{\log_2 3}$  for every  $n$ .<sup>12</sup>

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

<sup>12</sup> Hint: Use a proof by induction - suppose that this is true for all  $n$ 's from 1 to  $m$ , prove that this is true also for  $m+1$ .

**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  $x[0]+10*x[1]+100*x[2]+\dots$  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*

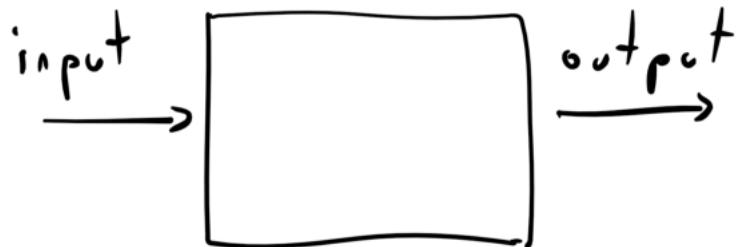
## 2

# 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*.



**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:** 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).<sup>1</sup>

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  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ . We can represent a number  $x \in \mathbb{N}$  as a string using the *binary basis*. Specifically, every natural number  $x$  can be written in a unique way as  $x = x_02^0 + x_12^1 + \dots + x_{n-1}2^{n-1}$  (or  $\sum_{i=0}^{n-1} x_i2^i$  for short) where  $x_0, \dots, x_{n-1}$  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  $(x_0, x_1, \dots, x_{n-1})$ .<sup>2</sup> For example, the number 35 is represented as the string  $(1, 1, 0, 0, 0, 1)$ .<sup>3</sup>

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.,  $D(x_0, \dots, x_{n-1}) = 2^0x_0 + 2^1x_1 + \dots + 2^{n-1}x_{n-1}$  for every  $x_0, \dots, x_{n-1} \in \{0, 1\}$ ). For the representation to be well defined, we need every natural number to be represented by some string, where two distinct numbers must

<sup>1</sup> 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.

<sup>2</sup> 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.

<sup>3</sup> 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  $(x_{n-1}, x_{n-2}, \dots, x_0)$  will of course work just as well. 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. A related, but not identical, distinction is the **Big Endian** vs **Little Endian** representation for integers in computing architecture.

have distinct representations. This corresponds to requiring the *encoding* function to be one-to-one, and the *decoding* function to be *onto*.

**P** 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  $\mathbb{Z} = \{\dots, -3, -2, -1, 0, +1, +2, +3, \dots\}$ ) by adding one more bit that represents the sign. So, the string  $(\sigma, x_0, \dots, x_{n-1}) \in \{0, 1\}^{n+1}$  will represent the number

$$(-1)^\sigma [x_0 2^0 + \dots + x_{n-1} 2^n] \quad (2.1)$$

The decoding function of a representation should always be *onto*, since every object must be represented by some string. However, it does not always have to be one to one. For example, in this particular representation 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?). We can also allow a *partial* decoding function for representations. For example, in the representation above there is no number that is represented by the empty string. But this is still a fine representation, since the decoding partial function is onto and the encoding function is the one-to-one total function  $E : \mathbb{Z} \rightarrow \{0, 1\}^*$  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.<sup>4</sup> 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

<sup>4</sup> In programming language, the compiler or interpreter determines the representation of the sequence of bits corresponding to a variable based on the variable's *type*.

will assume that we are fixing some canonical representation scheme such as the ones above. The choice of the particular representation 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.<sup>5</sup> For example, recall that we represent 4 as  $(0, 1)$  and 35 as  $(1, 1, 0, 0, 0, 1)$ , but the concatenation  $(0, 1, 1, 1, 0, 0, 0, 1)$  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  $(0, 1, 1, 1, 0, 0, 0, 1)$  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  $00110111100000011$ . 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 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?).

<sup>5</sup> Recall that the *concatenation* of two strings  $x$  and  $y$  is the string of length  $|x| + |y|$  obtained by writing  $y$  after  $x$ .

### 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  $x$ . For example, we can represent  $\pi$  by  $22/7$  with an error of about  $10^{-3}$  and if we wanted smaller error (e.g., about  $10^{-4}$ ) then we can use  $311/99$  and so on and so forth.

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 (positive or negative) integers of some prescribed sizes (determined by the desired accuracy) such that  $a \times 2^b$  is closest to  $x$ .<sup>6</sup> The reader might be (rightly) worried about this issue of approximation. In many (though not all) computational applications, one can make the accuracy tight enough so that this does not affect the final result, though sometimes we do need to be careful. This representation 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. The use of floating representation is the reason why in many programming systems printing the expression  $0.1+0.2$  will result in  $0.3000000000000004$  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.

### 2.1.5 Can we represent reals exactly?

Given the issues with floating point representation, we could ask whether we could represent real numbers *exactly* as strings. Unfortunately, the following theorem says this cannot be done

**Theorem 2.1 — Reals are uncountable.** There is no one-to-one function  $RtS : \mathbb{R} \rightarrow \{0, 1\}^*$ .<sup>7</sup>

Theorem 2.1 was proven by Georg Cantor in 1874.<sup>8</sup> The result (and the theory around it) was quite shocking to mathematicians at the time. By showing that there is no one-to-one map from  $\mathbb{R}$  to  $\{0, 1\}^*$  (or  $\mathbb{N}$ ), Cantor showed that these two infinite sets have “different forms of infinity” and that the set of real numbers  $\mathbb{R}$  is in some sense “bigger” than the infinite set  $\{0, 1\}^*$ . The notion that there are “shades of infinity” was deeply disturbing to mathematicians and philosophers at the time. The philosopher Ludwig Wittgenstein

<sup>6</sup> You can think of this as related to [scientific notation](#). In scientific notation we represent a number  $y$  as  $a \times 10^b$  for integers  $a, b$ . Sometimes we write this as  $y = aEb$ . For example, in many programming languages  $1.21E2$  is the same as  $121.0$ . In scientific notation, to represent  $\pi$  up to accuracy  $10^{-3}$  we will simply use  $3141 \times 10^{-3}$  and to represent it up to accuracy  $10^{-4}$  we will use  $31415 \times 10^{-4}$ .

<sup>7</sup>  $RtS$  stands for “reals to strings”.

<sup>8</sup> Cantor used the set  $\mathbb{N}$  rather than  $\{0, 1\}^*$ , but one can show that these two result are equivalent using the one-to-one maps between those two sets, see [Exercise 2.9](#). Saying that there is no one-to-one map from  $\mathbb{R}$  to  $\mathbb{N}$  is equivalent to saying that there is no onto map  $NtR : \mathbb{N} \rightarrow \mathbb{R}$  or, in other words, that there is no way to “count” all the real numbers as  $NtR(0), NtR(1), NtR(2), \dots$ . For this reason Theorem 2.1 is known as the *uncountability of the reals*.

called Cantor's results "utter nonsense" and "laughable". Others thought they were even worse than that. Leopold Kronecker called Cantor a "corrupter of youth", while Henri Poincaré said that Cantor's ideas "should be banished from mathematics once and for all". The tide eventually turned, and these days Cantor's work is universally accepted as the cornerstone of set theory and the foundations of mathematics. As we will see later in this course, Cantor's ideas also play a huge role in the theory of computation.

Now that we discussed the theorem's importance, let us see the proof. [Theorem 2.1](#) follows from the following two results:

**Lemma 2.2** Let  $\{0,1\}^\infty$  be the set  $\{f \mid f : \mathbb{N} \rightarrow \{0,1\}\}$  of functions from  $\mathbb{N}$  to  $\{0,1\}$ .<sup>9</sup> Then there is no one-to-one map  $FtS : \{0,1\}^\infty \rightarrow \{0,1\}^*$ .<sup>10</sup>

**Lemma 2.3** There *does* exist a one-to-one map  $FtR : \{0,1\}^\infty \rightarrow \mathbb{R}$ .<sup>11</sup>

[Lemma 2.2](#) and [Lemma 2.3](#) together imply [Theorem 2.1](#). To see why, suppose, towards the sake of contradiction, that there did exist a one-to-one function  $RtS : \mathbb{R} \rightarrow \{0,1\}^*$ . By [Lemma 2.3](#), there exists a one-to-one function  $FtR : \{0,1\}^\infty \rightarrow \mathbb{R}$ . Thus, under this assumption, since the composition of two one-to-one functions is one-to-one (see [Exercise 2.8](#)), the function  $FtS : \{0,1\}^\infty \rightarrow \{0,1\}^*$  defined as  $FtS(f) = RtS(FtR(f))$  will be one to one, contradicting [Lemma 2.2](#).

Now all that is left is to prove these two lemmas. We start by proving [Lemma 2.2](#) which is really the heart of [Theorem 2.1](#).

*Proof.* Let us assume, towards the sake of contradiction, that there exists a one-to-one function  $FtS : \{0,1\}^\infty \rightarrow \{0,1\}^*$ . Then, there is an *onto* function  $StF : \{0,1\}^* \rightarrow \{0,1\}^\infty$  (e.g., see [Lemma 0.1](#)). We will derive a contradiction by coming up with some function  $f^* : \mathbb{N} \rightarrow \{0,1\}$  such that  $f^* \neq StF(x)$  for every  $x \in \{0,1\}^*$ .

The argument for this is short but subtle. We need to construct some function  $f^* : \mathbb{N} \rightarrow \{0,1\}$  such that for every  $x \in \{0,1\}^*$ , if we let  $g = StF(x)$  then  $g \neq f^*$ . Since two functions are identical if and only if they agree on every input, to do this we need to show that there is *some*  $n \in \mathbb{N}$  such that  $f^*(n) \neq g(n)$ . (All these quantifiers can be confusing, so let's again recap where we are and where we want to get to. We assumed by contradiction there is a one-to-one  $FtS$  and hence an onto  $StF$ . To get our desired contradiction we need to show the *existence* of a single  $f^*$  such that for *every*  $x \in \{0,1\}^*$  there *exists*  $n \in \mathbb{N}$  on which  $f^*$  and  $g = StF(x)$  disagree.)

The idea is to construct  $f^*$  iteratively: for every  $x \in \{0,1\}^*$  we will

<sup>9</sup> We can also think of  $\{0,1\}^\infty$  as the set of all infinite *sequences* of bits, since a function  $f : \mathbb{N} \rightarrow \{0,1\}$  can be identified with the sequence  $(f(0), f(1), f(2), \dots)$ .

<sup>10</sup>  $FtS$  stands for "functions to strings".

<sup>11</sup>  $FtR$  stands for "functions to reals."

“ruin”  $f^*$  in one input  $n(x) \in \mathbb{N}$  to ensure that  $f^*(n(x)) \neq g(n(x))$  where  $g = StF(x)$ . If we are successful then this would ensure that  $f^* \neq StF(x)$  for every  $x$ . Specifically, for every  $x \in \{0,1\}^*$ , let  $n(x) \in \mathbb{N}$  be the number  $x_0 + 2x_1 + 4x_2 + \dots + 2^{k-1}x_{k-1} + 2^k$  where  $k = |x|$ . That is,  $n(x) = 2^k + \sum_{i=0}^{k-1} 2^i x_i$ . If  $x \neq x'$  then  $n(x) \neq n(x')$  (we leave verifying this as an exercise to you, the reader).

Now for every  $x \in \{0,1\}^*$ , we define

$$f^*(n(x)) = 1 - g(n(x)) \quad (2.2)$$

where  $g = StF(x)$ . For every  $n$  that is not of the form  $n = n(x)$  for some  $x$ , we set  $f^*(n) = 0$ . Eq. (2.2) is well defined since the map  $x \mapsto n(x)$  is one-to-one and hence we will not try to give  $f^*(n)$  two different values.

Now by Eq. (2.2), for every  $x \in \{0,1\}^*$ , if  $g = StF(x)$  and  $n = n(x)$  then  $f^*(n) = 1 - g(n) \neq g(n)$ . Hence  $StF(x) \neq f^*$  for every  $x \in \{0,1\}^*$ , contradicting the assumption that  $StF$  is onto. This proof is known as the “diagonal” argument, as the construction of  $f^*$  can be thought of as going over the diagonal elements of a table that in the  $n$ -th row and  $m$ -column contains  $StF(x)(m)$  where  $x$  is the string such that  $n(x) = n$ , see Fig. 2.3. ■

$X$	$n(x)$	$g = StF(x)$						
		$g(0)$	$g(1)$	$g(2)$	$g(3)$	$g(4)$	$g(5)$	$\dots$
$111$	$1$	$StF("")[0]$	$StF("")[1]$	$StF("")[2]$	$StF("")[3]$	$StF("")[4]$	$StF("")[5]$	$\dots$
$0$	$2$	$StF(0)[0]$	$StF(0)[1]$	$StF(0)[2]$	$StF(0)[3]$	$StF(0)[4]$	$StF(0)[5]$	$\dots$
$1$	$3$	$StF(1)[0]$	$StF(1)[1]$	$StF(1)[2]$	$StF(1)[3]$	$StF(1)[4]$	$StF(1)[5]$	$\dots$
$00$	$4$	$StF(00)[0]$	$StF(00)[1]$	$StF(00)[2]$	$StF(00)[3]$	$StF(00)[4]$	$StF(00)[5]$	$\dots$
$10$	$5$	$StF(10)[0]$	$StF(10)[1]$	$StF(10)[2]$	$StF(10)[3]$	$StF(10)[4]$	$StF(10)[5]$	$\dots$
$01$	$6$	$StF(01)[0]$	$StF(01)[1]$	$StF(01)[2]$	$StF(01)[3]$	$StF(01)[4]$	$StF(01)[5]$	$\dots$
$\vdots$	$\vdots$							

**Figure 2.3:** We construct a function  $f^*$  such that  $f^* \neq StF(x)$  for every  $x \in \{0,1\}^*$  by ensuring that  $f^*(n(x)) \neq StF(x)(n(x))$  for every  $x \in \{0,1\}^*$ . We can think of this as building a table where the columns correspond to numbers  $m \in \mathbb{N}$  and the rows correspond to  $x \in \{0,1\}^*$  (sorted according to  $n(x)$ ). If the entry in the  $x$ -th row and the  $m$ -th column corresponds to  $g(m)$  where  $g = StF(x)$  then  $f^*$  is obtained by going over the “diagonal” elements in this table (the entries corresponding to the  $x$ -th row and  $n(x)$ -th column) and ensuring that  $f^*(x)(n(x)) \neq StF(x)(n(x))$ .

**R** **Generalizing beyond strings and reals Lemma 2.2**  
 doesn't really have much to do with the natural numbers or the strings. An examination of the proof shows that it really shows that for *every* set  $S$ , there is no one-to-one map  $F : \{0,1\}^S \rightarrow S$  where  $\{0,1\}^S$  denotes the set  $\{f \mid f : S \rightarrow \{0,1\}\}$  of all Boolean functions with domain  $S$ . Since we can identify a subset  $V \subseteq S$  with its characteristic function  $f = 1_V$  (i.e.,  $1_V(x) = 1$  iff  $x \in V$ ), we can think of  $\{0,1\}^S$  also as the set of all *subsets* of  $S$ . This subset is sometimes called the *power set* of  $S$ . The proof of Lemma 2.2 can be generalized to show that there is no one-to-one map between a set and its power set. In particular, it means that the set  $\{0,1\}^{\mathbb{R}}$  is "even bigger" than  $\mathbb{R}$ . Cantor used these ideas to construct an infinite hierarchy of shades of infinity. The number of such shades turn out to be much larger than  $|\mathbb{N}|$  or even  $|\mathbb{R}|$ . He denoted the cardinality of  $\mathbb{N}$  by  $\aleph_0$ , where  $\aleph$  is the first letter in the Hebrew alphabet, and called the next largest infinite number by  $\aleph_1$ . Cantor also made the **continuum hypothesis** that  $|\mathbb{R}| = \aleph_1$ . We will come back to the very interesting story of this hypothesis later on in this course. **This lecture of Aaronson** mentions some of these issues (see also [this Berkeley CS 70 lecture](#)).

To complete the proof of Theorem 2.1, we need to show

**Lemma 2.3.** This requires some calculus background, but is otherwise straightforward. The idea is that we can construct a one-to-one map from  $\{0,1\}^\infty$  to the real numbers by mapping the function  $f : \mathbb{N} \rightarrow \{0,1\}$  to the number that has the infinite decimal expansion  $f(0).f(1)f(2)f(3)f(4)f(5)\dots$  (i.e., the number between 0 and 2 that is  $\sum_{i=0}^{\infty} f(i)10^{-i}$ ). We will now do this more formally. If you have not had much experience with limits of real series before, then the formal proof might be a little hard to follow. This part is not the core of Cantor's argument, nor are such limits very crucial to this course, so feel free to also just take Lemma 2.3 on faith and skip the formal proof.

*Proof of Lemma 2.3.* For every  $f \in \{0,1\}^\infty$  and  $n \in \mathbb{N}$ , we define  $S(f)_n = \sum_{i=0}^n f(i)10^{-i}$ . It is a known result (that we won't repeat here) that for every  $f : \mathbb{N} \rightarrow \{0,1\}$ , the sequence  $(S(f)_n)_{n=0}^{\infty}$  has a *limit*. That is, there exists some value  $x$  such that for every  $\epsilon > 0$ , if  $n$  is sufficiently large then  $|S_f(n) - x| < \epsilon$ . We define  $FtR(f)$  to be this value  $x$ . To show that  $FtR$  is one to one, we need to show that  $FtR(f) \neq FtR(g)$  for every distinct  $f, g : \mathbb{N} \rightarrow \{0,1\}$ . Let  $f \neq g$  be such functions, and let  $k$  be the smallest number for which  $f(k) \neq g(k)$ . Assume without loss of generality that  $f(k) = 0$  and

$g(k) = 1$ . Then, if  $S = \sum_{i=0}^{k-1} 10^{-i} f(i) = \sum_{i=0}^{k-1} 10^{-i} g(i)$ , we get that for every  $n > k + 1$ ,  $S(f)_n = S + \sum_{i=k+1}^n 10^{-i} f(i)$  and  $S(g)_n = S + 10^{-k} + \sum_{i=k+1}^n 10^{-i} g(i)$ . Clearly, the limit  $FtR(g)$  is at least  $S + 10^{-k}$ . On the other hand, we claim that for every  $n > k + 1$ ,  $S(f)_n \leq S + 2 \cdot 10^{-k-1} < S + 10^{-k}$ , and hence in particular  $FtR(f) < FtR(g)$ . Indeed, since  $f(i) \in \{0, 1\}$  for every  $i$ ,  $\sum_{i=k+1}^n f(i) 10^{-i} \leq \sum_{i=k+1}^n 10^{-i}$  which by formula for geometric series, equals  $10^{-k-1} \frac{1 - 10^{-(n-k-1)}}{1 - 10^{-1}} \leq 10^{-k-1}/0.9 \leq 2 \cdot 10^{-k-1}$ . ■

## 2.2 Beyond numbers

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

**Definition 2.1 — String representation.** Let  $\mathcal{O}$  be some set. A *representation scheme* for  $\mathcal{O}$  consists of a pair  $(E, D)$  where  $E : \mathcal{O} \rightarrow \{0, 1\}^*$  is a total one-to-one function,  $D : \{0, 1\}^* \rightarrow_p \mathcal{O}$  is a (possibly partial) function, and such that  $D$  and  $E$  satisfy that  $D(E(o)) = o$  for every  $o \in \mathcal{O}$ .  $E$  is known as the *encoding* function and  $D$  is known as the *decoding* function.

Note that the condition  $D(E(o)) = o$  for every  $o \in \mathcal{O}$  implies that  $D$  is *onto* (can you see why?). 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:

**Lemma 2.4** Suppose that  $E : \mathcal{O} \rightarrow \{0, 1\}^*$  is one-to-one. Then there exists a function  $D : \{0, 1\}^* \rightarrow \mathcal{O}$  such that  $D(E(o)) = o$  for every  $o \in \mathcal{O}$ .

*Proof.* Let  $o_0$  be some arbitrary element of  $\mathcal{O}$ . For every  $x \in \{0, 1\}^*$ , there exists either zero or a single  $o \in \mathcal{O}$  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  $o \in \mathcal{O}$ . ■

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 Finite representations

If  $\mathcal{O}$  is *finite*, then we can represent every object in  $\mathcal{O}$  as a string of length at most some number  $n$ . What is the value of  $n$ ? Let us denote the set  $\{x \in \{0,1\}^* : |x| \leq n\}$  of strings of length at most  $n$  by  $\{0,1\}^{\leq n}$ . To obtain a representation of objects in  $\mathcal{O}$  as strings in  $\{0,1\}^{\leq n}$  we need to come up with a one-to-one function from the former set to the latter. We can do so, if and only if  $|\mathcal{O}| \leq 2^{n+1} - 1$  as is implied by the following lemma:

**Lemma 2.5** For every two finite sets  $S, T$ , there exists a one-to-one  $E : S \rightarrow T$  if and only if  $|S| \leq |T|$ .

*Proof.* Let  $k = |S|$  and  $m = |T|$  and so write the elements of  $S$  and  $T$  as  $S = \{s_0, s_1, \dots, s_{k-1}\}$  and  $T = \{t_0, t_1, \dots, t_{m-1}\}$ . We need to show that there is a one-to-one function  $E : S \rightarrow T$  iff  $k \leq m$ . For the “if” direction, if  $k \leq m$  we can simply define  $E(s_i) = t_i$  for every  $i \in [k]$ . Clearly for  $i \neq j$ ,  $t_i = E(s_i) \neq E(s_j) = t_j$ , and hence this function is one-to-one. In the other direction, suppose that  $k > m$  and  $E : S \rightarrow T$  is some function. Then  $E$  cannot be one-to-one. Indeed, for  $i = 0, 1, \dots, m-1$  let us “mark” the element  $t_j = E(s_i)$  in  $T$ . If  $t_j$  was marked before, then we have found two objects in  $S$  mapping to the same element  $t_j$ . Otherwise, since  $T$  has  $m$  elements, when we get to  $i = m-1$  we mark all the objects in  $T$ . Hence, in this case  $E(s_m)$  must map to an element that was already marked before.<sup>12</sup> ■

Now the size of  $\{0,1\}^n$  is  $2^n$ , and the size of  $\{0,1\}^{\leq n}$  is only slightly bigger:  $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$  by the formula for a **geometric series**.

<sup>12</sup> This direction is sometimes known as the “Pigeon Hole Principle”: the principle that if you have pigeon coop with  $m$  holes, and  $k > m$  pigeons, then there must be two pigeon in the same hole.

### 2.2.2 Prefix free encoding

In our discussion of the representation of rational numbers, we used the “hack” of encoding the alphabet  $\{0,1,\cdot\}$  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 2.6 — 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  $(o_0, \dots, o_{k-1}) \in \mathcal{O}^*$ ,  $E'(o_0, \dots, o_{k-1}) = E(o_0)E(o_1)\cdots E(o_{k-1})$ .

P

Theorem 2.6 is one of those statements that are a little hard to parse, but in fact are fairly straightforward to prove once you understand what they mean. Thus I highly recommend that you pause here, make sure you understand statement of the theorem, and try to prove it yourself before proceeding further.



**Figure 2.4:** If we have a prefix-free representation of each object then we can concatenate the representations of  $k$  objects to obtain a representation for the tuple  $(o_1, \dots, o_k)$ .

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 2.6.* By Lemma 2.4, 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  $(o_0, \dots, o_{k-1})$  and  $(o'_0, \dots, o'_{k'-1})$  such that

$$E'(o_0, \dots, o_{k-1}) = E'(o'_0, \dots, o'_{k'-1}), \quad (2.3)$$

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. (2.3) the first  $|p| + |x_i|$  bits of the string  $x$  need to equal  $px_i$  and the first  $|p| + |x'_i|$  bits of  $x$  needs 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  $|x_i| > |x'_i|$ ,<sup>13</sup> 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  $x_0 \dots x_{k-1} = x_0 \dots x_{k-1}x'_k \dots x'_{k'-1}$ . 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?). ■

<sup>13</sup> This is one of our first uses of the phrase “without loss of generality”. Make sure you understand why it is justified! If you are not sure, you can try working out the case that  $|x_i| > |x'_i|$  and see why the proof is symmetric.

### 2.2.3 Making representations prefix free

Some natural representations are prefix free. For example, every *fixed output length* representation (i.e., one-to-one function  $E : \mathcal{O} \rightarrow \{0,1\}^n$ ) 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 2.7** Let  $E : \mathcal{O} \rightarrow \{0,1\}^*$  be a one-to-one function. Then there is a one-to-one prefix-free encoding  $\bar{E}$  such that  $|\bar{E}(o)| \leq 2|o| + 2$  for every  $o \in \mathcal{O}$ .

(P) For the sake of completeness, we will include the proof below, but it is a good idea for you to pause here and try to prove it yourself, using the same technique we used for representing rational numbers.

*Proof of Lemma 2.7.* Define the function  $PF : \{0,1\}^* \rightarrow \{0,1\}^*$  as follows  $PF(x) = x_0x_0x_1x_1\dots x_{n-1}x_{n-1}01$  for every  $x \in \{0,1\}^*$ . If  $E : \mathcal{O} \rightarrow \{0,1\}^*$  is the (potentially not prefix free) representation for  $\mathcal{O}$ , then we transform it into a prefix free representation  $\bar{E} : \mathcal{O} \rightarrow \{0,1\}^*$  by defining  $\bar{E}(o) = PF(E(o))$ .

To prove the lemma we need to show that (1)  $\bar{E}$  is one-to-one and (2)  $\bar{E}$  is prefix free. In fact (2) implies (1), since if  $\bar{E}(o)$  is never a prefix of  $\bar{E}(o')$  for every  $o \neq o'$  then in particular  $\bar{E}$  is one-to-one. Now suppose, toward the sake of contradiction, that there are  $o \neq o'$

in  $\mathcal{O}$  such that  $\overline{E}(o)$  is a prefix of  $\overline{E}(o')$ . (That is, if  $y = \overline{E}(o)$  and  $y' = \overline{E}(o')$ , then  $y_j = y'_j$  for every  $i < |y|$ .)

Define  $x = E(o)$  and  $x' = E(o')$ . Note that since  $E$  is one-to-one,  $x \neq x'$ . (Recall that two strings  $x, x'$  are distinct if they either differ in length or have at least one distinct coordinate.) Under our assumption,  $|PF(x)| \leq |PF(x')|$ , and since by construction  $|PF(x)| = 2|x| + 2$ , it follows that  $|x| \leq |x'|$ . If  $|x| = |x'|$  then, since  $x \neq x'$ , there must be a coordinate  $i \in \{0, \dots, |x| - 1\}$  such that  $x_i \neq x'_i$ . But since  $PF(x)_{2i} = x_i$ , we get that  $PF(x)_{2i} \neq PF(x')_{2i}$  and hence  $\overline{E}(o) = PF(x)$  is not a prefix of  $\overline{E}(o') = PF(x')$ . Otherwise (if  $|x| \neq |x'|$ ) then it must be that  $|x| < |x'|$ , and hence if  $n = |x|$ , then  $PF(x)_{2n} = 0$  and  $PF(x)_{2n+1} = 1$ . But since  $n < |x'|$ ,  $PF(x')_{2n}, PF(x')_{2n+1}$  is equal to either 00 or 11, and in any case we get that  $\overline{E}(o) = PF(x)$  is not a prefix of  $\overline{E}(o') = PF(x')$ . ■

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](#)).

#### 2.2.4 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.

#### 2.2.5 Representing vectors, matrices, images

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.<sup>14</sup> Thus an image of  $n$  pixels would be represented of a list

<sup>14</sup> We can restrict to three basic colors since (most) humans only have three types of cones in their retinas. We would have needed 16 basic colors to represent colors visible to the **Mantis Shrimp**.

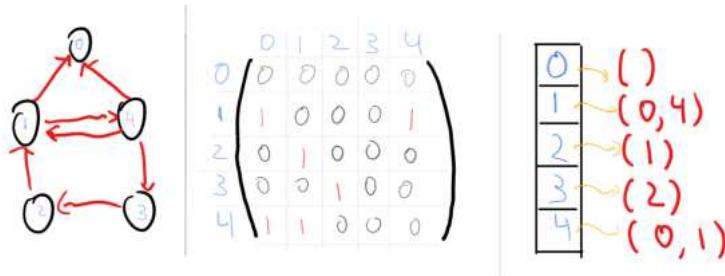
of  $n$  such length-three lists. A video can be represented as a list of images.<sup>15</sup>

### 2.2.6 Representing graphs

A *graph* on  $n$  vertices can be represented as an  $n \times n$  *adjacency matrix* whose  $(i, j)^{\text{th}}$  entry is equal to 1 if the edge  $(i, j)$  is present and is equal to 0 otherwise. That is, we can represent an  $n$  vertex directed graph  $G = (V, E)$  as a string  $A \in \{0, 1\}^{n^2}$  such that  $A_{i,j} = 1$  iff the edge  $\overrightarrow{i j} \in E$ . We can transform an undirected graph to a directed graph by replacing every edge  $\{i, j\}$  with both edges  $\overrightarrow{i j}$  and  $\overleftarrow{i j}$ .

Another representation for graphs is the *adjacency list* representation. That is, we identify the vertex set  $V$  of a graph with the set  $[n]$  where  $n = |V|$ , and represent the graph  $G = (V, E)$  as a list of  $n$  lists, where the  $i$ -th list consists of the out-neighbors of vertex  $i$ . The difference between these representations can be important for some applications, though for us would typically be immaterial.

<sup>15</sup> Of course these representations are rather wasteful and **much more** compact representations are typically used for images and videos, though this will not be our concern in this course.



**Figure 2.5:** Representing the graph  $G = (\{0, 1, 2, 3, 4\}, \{(1,0), (4,0), (1,4), (4,1), (2,1), (3,2), (4,3)\})$  in the adjacency matrix and adjacency list representations.

### 2.2.7 Representing lists

If we have a way of representing objects from a set  $\mathcal{O}$  as binary strings, then we can represent lists of these objects by 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  $E : \mathcal{O} \rightarrow \{0, 1\}^*$ , 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 0011,  $o_2$  is represented by 10011, and  $o_3$  is represented by 00111, then we can represent the nested list  $(o_1, (o_2, o_3))$  as the string "[0011, [10011, 00111]]" over the alphabet  $\Sigma$ . By encoding every element of  $\Sigma$  itself as a three-bit string, we can

transform any representation for objects  $\mathcal{O}$  into a representation that allows to represent (potentially nested) lists of these objects.

### 2.2.8 Notation

We will typically identify an object with its representation as a string. For example, if  $F : \{0,1\}^* \rightarrow \{0,1\}^*$  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  $\underline{x}$  and let  $\underline{y} = F(\underline{x})$ , then the integer  $y$  represented by the string  $\underline{y}$  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 2.6:** A computational process

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  $p_1 \leq \dots \leq p_k$  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  $F : \{0,1\}^* \rightarrow \{0,1\}$  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  $F$ .<sup>16</sup> Hence many texts refer to such as

<sup>16</sup> The language terminology is due to historical connections between the theory of computation and formal linguistics as developed by Noam Chomsky.

computational task as *deciding a language*.

For every particular function  $F$ , there can be several possible *algorithms* to compute  $F$ . We will be interested in questions such as:

- For a given function  $F$ , can it be the case that *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 *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 *algorithm*, which is what we’ll do in the next lecture.

### 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 *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 *promise problems*.

Another generalization is to consider *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 *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 *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:  $B(x) = (x_0, \dots, x_k)$  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:  $B(x) = (x_k, \dots, x_0)$  where  $x_i$  is defined as above for  $i = 0, \dots, k - 1$ .

c. Binary coded decimal representation:  $B(x) = (y_0, \dots, y_{n-1})$  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  $R : \mathbb{N} \rightarrow \{0, 1\}^*$  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  $B : \mathbb{N} \rightarrow \{0, 1\}^*$  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  $2^k - 1 = 1 + 2 + 4 + \dots + 2^{k-1}$ .

**Exercise 2.5 — Prefix-free encoding of tuples.** Suppose that  $F : \mathbb{N} \rightarrow \{0, 1\}^*$  is a one-to-one function that is *prefix free* in the sense that there is no  $a \neq b$  s.t.  $F(a)$  is a prefix of  $F(b)$ .

a. Prove that  $F_2 : \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}^*$ , defined as  $F_2(a, b) = F(a)F(b)$  (i.e., the concatenation of  $F(a)$  and  $F(b)$ ) is a one-to-one function.

b. Prove that  $F_* : \mathbb{N}^* \rightarrow \{0, 1\}^*$  defined as  $F_*(a_1, \dots, a_k) = F(a_1) \dots F(a_k)$  is a one-to-one function, where  $\mathbb{N}^*$  denotes the set of all finite-length lists of natural numbers.

**Exercise 2.6 — More efficient prefix-free transformation.** Suppose that  $F : O \rightarrow \{0, 1\}^*$  is some (not necessarily prefix free) representation of the objects in the set  $O$ , and  $G : \mathbb{N} \rightarrow \{0, 1\}^*$  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

length at most  $k + O(\log k)$ . c. 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 length at most  $k + \log k + O(\log \log k)$ .<sup>17</sup> ■

<sup>17</sup> Hint: Think recursively how to represent the length of the string.

**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  $x_0, \dots, x_{k-1}$ . 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  $PF : \{0, 1\}^* \rightarrow \{0, 1\}^*$  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 — Composition of one-to-one functions.** Prove that for every two one-to-one functions  $F : S \rightarrow T$  and  $G : T \rightarrow U$ , the function  $H : S \rightarrow U$  defined as  $H(x) = G(F(x))$  is one to one. ■

**Exercise 2.9 — Natural numbers and strings.** 1. We have shown that the natural numbers can be represented as strings. Prove that the other direction holds as well: that there is a one-to-one map  $StN : \{0, 1\}^* \rightarrow \mathbb{N}$ . ( $StN$  stands for “strings to numbers”.)

2. Recall that Cantor proved that there is no one-to-one map  $RtN : \mathbb{R} \rightarrow \mathbb{N}$ . Show that Cantor’s result implies [Theorem 2.1](#). ■

## 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  $\mathcal{O}$  into strings of length not much larger than the minimum of  $\log_2 |\mathcal{O}|$  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

### Learning Objectives:

- See that computation can be precisely modeled.
- Learn the NAND computational model.
- Comfort switching between description of NAND programs as *code* and as *tuples*.
- Begin acquiring skill of translating informal algorithms into NAND code.

# 3

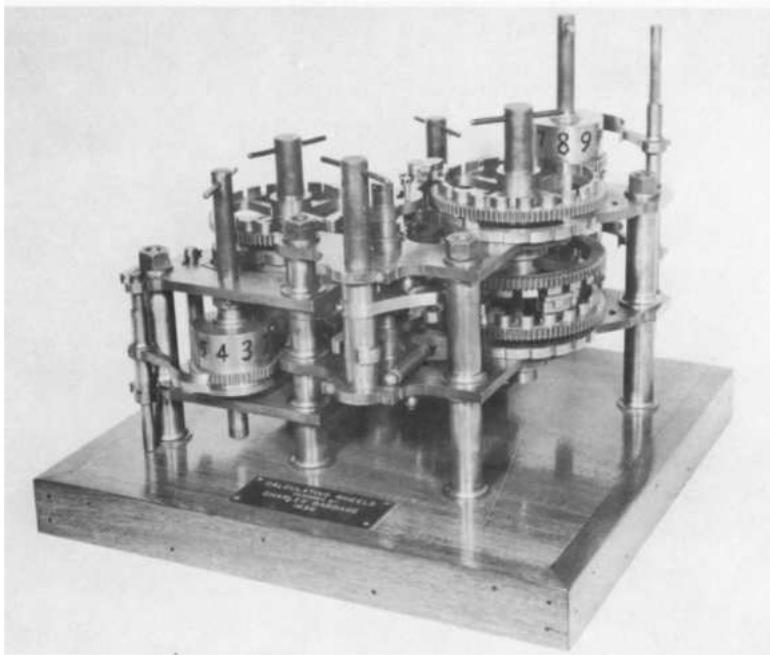
## Defining computation

*"there is no reason why mental as well as bodily labor should not be economized by the aid of machinery",*  
Charles Babbage, 1852

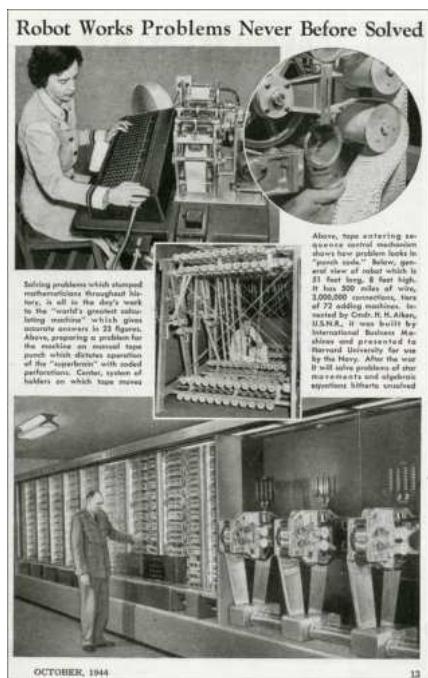
*"If, unwarned by my example, any man shall undertake and shall succeed in constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he 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.<sup>1</sup>

Here is how al-Khwarizmi described how to solve an equation of the form  $x^2 + bx = c$ .<sup>2</sup>

*[How to solve an equation of the form] “roots and squares are equal to numbers”: For instance “one square, and ten roots of the same, amount to thirty-nine dirhems” that is to say, what must be the square which, when increased by ten of its own root, amounts to thirty-nine? The solution is this: you halve the number of the roots, which in the present instance yields five. This you multiply by itself; the product is twenty-five. Add this to thirty-nine’ the sum is sixty-four. Now take the root of this, which is eight, and subtract from it half the number of roots, which is five; the remainder is three. This is the root of the square which you sought for; the square itself is nine.*

For the purposes of this course, we will need a much more precise way to define algorithms. Fortunately (or is it unfortunately?), at least at the moment, computers lag far behind school-age children in learning from examples. Hence in the 20th century people have come up with exact formalisms for describing algorithms, namely *programming languages*. Here is al-Khwarizmi’s quadratic equation solving algorithm described in the Python programming language:<sup>3</sup>

```
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
```

<sup>1</sup> Indeed, extrapolation from examples is still the way most of us first learn algorithms such as addition and multiplication, see Fig. 3.3)

<sup>2</sup> Translation from “The Algebra of Ben-Musa”, Fredric Rosen, 1831.

<sup>3</sup> 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

```
return val5 # This is the root of the square which you
sought for
```

### 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.<sup>4</sup> 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`.<sup>5</sup>

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_-<i>` or `y_-<i>` where  $i$  is a natural number) have a special meaning.<sup>6</sup> 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
```

<sup>4</sup> The terms `foo` and `bar` 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 <http://nandpl.org> for a full specification of the NAND programming language.

<sup>5</sup> The *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  $\text{NAND}(0,0) = \text{NAND}(0,1) = \text{NAND}(1,0) = 1$ , while  $\text{NAND}(1,1) = 0$ . If a variable hasn't been assigned a value, then its default value is zero.

<sup>6</sup> 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.

**P**

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.<sup>7</sup> When we run the program above on the input 01, we get the following trace:

```
Executing step 1: "u_0:=x_0_NAND_x_1" x_0 = 0, x_1 = 1, u
is assigned 1,
Executing step 2: "v_0:=x_0_NAND_u" x_0 = 0, u = 1, v is
assigned 1,
Executing step 3: "w_0:=x_1_NAND_u" x_1 = 1, u = 1, w is
assigned 0,
Executing step 4: "y_0:=v_0_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_0:=x_0_NAND_x_1" x_0 = 1, x_1 = 1, u
is assigned 0,
Executing step 2: "v_0:=x_0_NAND_u" x_0 = 1, u = 0, v is
assigned 1,
Executing step 3: "w_0:=x_1_NAND_u" x_1 = 1, u = 0, w is
assigned 1,
Executing step 4: "y_0:=v_0_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:

In other words, this program computes the *exclusive or* (also known as XOR) function.

<sup>7</sup> 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.

Input	Output
00	0
01	1
10	1
11	0

### 3.2.1 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  $ADD_1 : \{0,1\}^2 \rightarrow \{0,1\}^2$  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  $y_0 2^0 + y_1 2^1$  then the table of values for  $ADD_1$  is the following:

Input		Output	
$x\_0$	$x\_1$	$y\_0$	$y\_1$
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

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  $x\_1$ .<sup>8</sup> Thus we can compute one bit variable addition using the following program:

```
// 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,
```

<sup>8</sup> This is a special case of the general rule that when you add two digits  $x, x' \in \{0, 1, \dots, b-1\}$  over the  $b$ -ary basis (in our case  $b = 2$ ), then the output digit is  $x + x' \pmod{b}$  and the carry digit is  $\lfloor (x + x')/b \rfloor$ .

```

Executing step 4: "y_0:=v NAND w" v = 1, w = 1, y_0 is
    assigned 0,
Executing step 5: "y_1:=u 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

For a NAND program  $P$ , its *input length* is the largest number  $n$  such that  $P$  contains a variable of the form  $x_{\langle n-1 \rangle}$ .  $P$ 's *output length* is the largest number  $m$  such that  $P$  contains a variable of the form  $y_{\langle m-1 \rangle}$ .<sup>9</sup> Intuitively, if  $P$  is a NAND program with input length  $n$  and output length  $m$ , and  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$  is some function, then  $P$  computes  $F$  if for every  $x \in \{0, 1\}^n$  and  $y = F(x)$ , 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  $j \in [m]$ .

To make sure we have a precise and unambiguous definition of computation, we will now model NAND programs using sets and tuples, and recast the notion of computing a function in these terms.

<sup>9</sup> As mentioned in the appendix, we require that all output variables are assigned a value, and that the largest index used in an  $s$  line NAND program is smaller than  $s$ . In particular this means that an  $s$  line program can have at most  $s$  inputs and outputs.

**Definition 3.1 — NAND program.** A NAND program is a 4-tuple

$P = (V, X, Y, L)$  of the following form:

- $V$  (called the *variables*) is some finite set.
- $X$  (called the *input variables*) is a tuple of elements in  $V$ , i.e.  
 $X = (X_0, X_1, \dots, X_{n-1})$  for some  $n \in N$  where  $X_i \in V$  for all  $i \in [n]$ . We require that the elements of  $X$  are distinct:  $X_i \neq X_j$  for all  $i \neq j$  in  $[n]$ .
- $Y$  (called the *output variables*) is a tuple of elements in  $V$ , i.e.,  
 $Y = (Y_0, \dots, Y_{m-1})$  for some  $m \in \mathbb{N}$  where  $Y_j \in V$  for all  $j \in [m]$ . We require that the elements of  $Y$  are distinct (i.e.,  $Y_i \neq Y_j$  for all  $i \neq j$  in  $[m]$ ) and that they are disjoint from  $X$  (i.e.,  $Y_i \neq X_j$  for every  $i \in [n]$  and  $j \in [m]$ ).
- $L$  (called the *lines*) is a tuple of *triples* of  $V$ , i.e.,  $L \in (V \times V \times V)^*$ . Intuitively, if the  $\ell$ -th element of  $L$  is a triple  $(u, v, w)$  then this corresponds to the  $\ell$ -th line of the program being  $u := v \text{ NAND } w$ . We require that for every triple  $(u, v, w)$ ,  $u$  does not appear in  $X$  and  $v, w$  do not appear in  $Y$ . Moreover, we require that

for every  $v \in V$  (i.e., a member of  $V$  that is not equal to  $X_i$  for some  $i$ ),  $v$  is contained in some triple in  $L$ .

The *number of inputs* of  $P = (V, X, Y, Z)$  is equal to  $|X|$  and the *number of outputs* is equal to  $|Y|$ .



This definition is somewhat long and cumbersome, but really corresponds to a straightforward modelling of NAND programs, under the map that  $V$  is the set of all variables appearing in the program,  $X$  corresponds to the tuple  $(x_{\langle 0 \rangle}, x_{\langle 1 \rangle}, \dots, x_{\langle n-1 \rangle})$ ,  $Y$  corresponds to the tuple  $(y_{\langle 0 \rangle}, y_{\langle 1 \rangle}, \dots, y_{\langle m-1 \rangle})$  and  $L$  corresponds to the list of triples of the form  $(\text{foo}, \text{bar}, \text{baz})$  for every line  $\text{foo} := \text{bar NAND baz}$  in the program. Please pause here and verify that you understand this correspondence.

For example, one representation of the XOR program we described above is  $P = (V, X, Y, L)$  where

- $V = \{x_0, x_1, v, u, w, y_0\}$
- $X = (x_0, x_1)$
- $Y = (y_0)$
- $L = ((u, x_0, x_1), (v, x_0, u), (w, x_1, u), (y_0, v, w))$

But since we have the freedom of choosing arbitrary sets for our variables, we can also represent the same program as (for example)  $P' = (V', X', Y', L')$  where

- $V' = \{0, 1, 2, 3, 4, 5\}$
- $X' = (0, 1)$
- $Y' = (5)$
- $L' = ((3, 0, 1), (2, 0, 3), (4, 1, 3), (5, 2, 4))$

### 3.2.3 Computing a function: formal definition

Now that we defined NAND programs formally, we turn to formally defining the notion of computing a function. Before we do that, we will need to talk about the notion of the *configuration* of a NAND program. Such a configuration simply corresponds to the current line that is executed and the current values of all variables at a certain point in the execution. Thus we will model it as a pair  $(\ell, \sigma)$  where  $\ell$

is a number between 0 and the total number of lines in the program, and  $\sigma$  maps every variable to its current value.<sup>10</sup> The initial configuration has the form  $(0, \sigma_0)$  where 0 corresponds to the first line, and  $\sigma_0$  is the assignment of zeroes to all variables and  $x_i$ 's to the input variables. The final configuration will have the form  $(s, \sigma_s)$  where  $s$  is the number of lines (i.e., corresponding to “going past” the final line) and  $\sigma_s$  is the final values assigned to all variables, which in particular encodes also the values of the output variables.

For example, if we run the XOR program about on the input 11 then the configuration of the program evolves as follows:

```

x_0 x_1 v u w y_0
0. u := x_0 NAND x_1 : 0 1 0 0 0 0
1. v := x_0 NAND u : 0 1 0 1 0 0
2. w := x_1 NAND u : 0 1 1 1 0 0
3. y_0 := v NAND w : 0 1 1 1 0 0
4. (after halting) : 0 1 1 1 0 1

```

We now write the formal definition. As always, it is a good practice to verify that this formal definition matches the intuitive description above:

**Definition 3.2 — Configuration of a NAND program.** Let  $P = (V, X, Y, L)$  be a NAND program, and let  $n = |X|$ ,  $m = |Y|$  and  $s = |L|$ . A *configuration* of  $P$  is a pair  $(\ell, \sigma)$  where  $\ell \in [s + 1]$  and  $\sigma$  is a function  $\sigma : V \rightarrow \{0, 1\}$  that maps every variable of  $P$  into a bit in  $\{0, 1\}$ . We define  $\text{CONF}(P) = [s + 1] \times \{\sigma \mid \sigma : V \rightarrow \{0, 1\}\}$  to be the set of all configurations of  $P$ .<sup>11</sup>

If  $P$  has  $n$  inputs, then for every  $x \in \{0, 1\}^n$ , the *initial configuration of  $P$  with input  $x$*  is the pair  $(0, \sigma_0)$  where  $\sigma_0 : V \rightarrow \{0, 1\}$  is the function defined as  $\sigma_0(X_i) = x_i$  for every  $i \in [n]$  and  $\sigma_0(v) = 0$  for all variables not in  $X$ .

An execution of a NAND program can be thought of as simply progressing, line by line, from the initial configuration to the next one:

**Definition 3.3 — NAND next step function.** For every NAND program  $P = (V, X, Y, L)$ , the *next step function* of  $P$ , denoted by  $\text{NEXT}_P$ , is the function  $\text{NEXT}_P : \text{CONF}(P) \rightarrow \text{CONF}(P)$  that defined as follows:

For every  $(\ell, \sigma) \in \text{CONF}(P)$ , if  $\ell = |L|$  then  $\text{NEXT}_P(\ell, \sigma) = (\ell, \sigma)$ . Otherwise  $\text{NEXT}_P(\ell, \sigma) = (\ell + 1, \sigma')$  where  $\sigma' : V \rightarrow \{0, 1\}$

<sup>10</sup> The number  $\ell$  can be thought of as the “program counter” and refers to the line that is just about to be executed, when we number the lines from 0 till  $s - 1$  for some  $s \in \mathbb{N}$ . The program counter starts at 0, and after executing the last line (i.e., line number  $s - 1$ ), it equals  $s$ .

<sup>11</sup> Note that  $|\text{CONF}(P)| = (s + 1)2^{|V|}$ : can you see why?

is defined as follows:

$$\sigma'(x) = \begin{cases} NAND(\sigma(v), \sigma(w)) & x = u \\ \sigma(x) & \text{otherwise} \end{cases} \quad (3.1)$$

where  $(u, v, w) = L_\ell$  is the  $\ell$ -th triple (counting from zero) in  $L$ .

For every input  $x \in \{0, 1\}^n$  and  $\ell \in [s+1]$ , the  $\ell$ -th configuration of  $P$  on input  $x$ , denoted as  $conf_\ell(P, x)$  is defined recursively as follows:

$$conf_\ell(P, x) = \begin{cases} (0, \sigma_0) & \ell = 0 \\ NEXT_P(conf_{\ell-1}(P)) & \text{otherwise} \end{cases} \quad (3.2)$$

where  $(0, \sigma_0)$  is the initial configuration of  $P$  on input  $x$ .

We can now finally formally define the notion of computing a function:

**Definition 3.4 — Computing a function.** Let  $P = (V, X, Y, L)$  and  $n = |X|$ ,  $m = |Y|$  and  $s = |L|$ . Let  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ . We say that  $P$  computes  $F$  if for every  $x \in \{0, 1\}^n$ , if  $y = F(x)$  then  $conf_s(P, x) = (s, \sigma)$  where  $\sigma(Y_j) = y_j$  for every  $j \in [m]$ .

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



The formal specification of any programming language, no matter how simple, is often cumbersome, and the definitions above are no exception. You should go back and read them and make sure that you understand why they correspond to our informal description of computing a function via NAND programs. From this point on, we will not distinguish between the representation of a NAND program in terms of lines of codes, and its representation as a tuple  $P = (V, X, Y, L)$ .

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

**Theorem 3.1 — Computing XOR.**  $XOR_2 \in SIZE(4)$

Similarly, the addition program we presented shows that  $ADD_1 \in SIZE(5)$ .

### 3.3 Canonical input and output variables

The specific identifiers for NAND variables (other than the inputs and outputs) do not make any difference in the program's functionality, as long as we give separate variable distinct identifiers. For example, if I replace all instances of the variable `foo` with `boazisgreat` then, under the (unfortunately common) condition that `boazisgreat` was not used in the original program, the resulting program will still compute the same function. For convenience, it is sometimes useful to assume that all variables identifiers have some canonical form such as being either  $x_{\langle i \rangle}$ ,  $y_{\langle j \rangle}$  or `work_{\langle k \rangle}`. Similarly, while we allowed in [Definition 3.1](#) the variables to be members of some arbitrary set  $V$ , it is sometimes useful to assume that  $V$  is simply the set of numbers from 0 to some natural number (which can never be more than three times the number of lines  $s$ ). This motivates the following definition:

**Definition 3.5 — Canonical variables.** Let  $P = (V, X, Y, L)$  be a NAND program and let  $n = |X|$  and  $m = |Y|$ . We say that  $P$  has *canonical variables* if  $V = [t]$  for some  $t \in \mathbb{N}$ ,  $X = (0, 1, \dots, n - 1)$  and  $Y = (t - m, t - m + 1, \dots, t - 1)$ .

That is, in a canonical form program, the variables are the set  $[t]$ , where the input variables correspond to the first  $n$  variables and the outputs to the last  $m$  variables. Every program can be converted to an equivalent program of canonical form:

**Theorem 3.2 — Convert to canonical variables.** For every  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$  and  $s \in \mathbb{N}$ ,  $F$  can be computed by an  $s$ -line NAND program if and only if it can be computed by an  $s$ -line NAND program with canonical variables.

*Proof.* The “if” direction is trivial, since a NAND program with canonical variables is just a special case of a NAND program. For the “only if” direction, let  $P = (V, X, Y, L)$  be an  $s$ -line NAND program computing  $F$ , and let  $t = |V|$ . We define a bijection  $\pi : V \rightarrow [t]$  as follows:  $\pi(X_i) = i$  for all  $i \in [n]$ ,  $\pi(Y_j) = t - m + j$  for all  $j \in [m]$  and we map the remaining  $t - m - n$  elements of  $V$  to  $\{n, \dots, t - 1 - m\}$  in some arbitrary one to one way. (We can do so because the  $X$ 's and  $Y$ 's are distinct and disjoint.) Now define  $P' = (\pi(V), \pi(X), \pi(Y), \pi(L))$ , where by this we mean that we apply  $\pi$  individually to every element of  $V, X, Y$ , and the triples of  $L$ . Since (as we leave you to verify) the definition of configurations and computing a function are invariant under bijections of  $V, P'$

computes the same function as  $P$ . ■

Given [Theorem 3.2](#), since we only care about the functionality (and size) of programs, and not the labels of variables, we will always be able to assume “without loss of generality” that a given NAND program  $P$  has canonical form. A canonical form program  $P$  can also be represented as a triple  $(n, m, L)$  where  $n, m$  are (as usual) the inputs and outputs, and  $L$  is the lines. This is because we recover the original representation  $(V, X, Y, L)$  by simply setting  $X = (0, 1, \dots, n - 1)$ ,  $Y = (t - m, t - m + 1, \dots, t - 1)$  and  $V = [t]$  where  $t$  is one plus the largest number appearing in a triple of  $L$ . In the following we will freely move between these two representations. If  $n, m$  are known from the context, then a canonical form program can be represented simply by the list of triples  $L$ .

**Configurations of programs with canonical variables:** A *configuration* of a program with canonical variables is a pair  $(\ell, \sigma)$  where  $\sigma : [t] \rightarrow \{0, 1\}$  and  $t$  is the number of variables. We can and will identify such a function  $\sigma$  with a string of  $t$  bits. Thus we will often say that a configuration of a canonical program is a pair  $(\ell, \sigma)$  where  $\sigma \in \{0, 1\}^t$ .

### 3.4 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 3.3 — Computing four bit parity.**  $\text{XOR}_4 \in \text{SIZE}(12)$

We can prove [Theorem 3.3](#) 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 3.4 — Sequential composition of functions.** If  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$  is a function in  $\text{SIZE}(L)$  and  $G : \{0, 1\}^m \rightarrow \{0, 1\}^k$  is a function in  $\text{SIZE}(L')$  then  $G \circ F$  is in  $\text{SIZE}(L + L')$ , where  $G \circ F : \{0, 1\}^n \rightarrow \{0, 1\}^k$  is defined as the function that maps  $x \in \{0, 1\}^n$  to  $G(F(x))$ .

**Theorem 3.5 — Parallel composition of functions.** If  $F : \{0,1\}^n \rightarrow \{0,1\}^m$  is a function in  $\text{SIZE}(L)$  and  $G : \{0,1\}^{n'} \rightarrow \{0,1\}^{m'}$  is a function in  $\text{SIZE}(L')$  then  $F\|G$  is in  $\text{SIZE}(L + L')$ , where  $F\|G : \{0,1\}^{n+n'} \rightarrow \{0,1\}^{m+m'}$  is defined as the function that maps  $x \in \{0,1\}^{n+n'}$  to  $F(x_0, \dots, x_{n-1})G(x_n, \dots, x_{n+n'-1})$ .

(P)

We will prove [Theorem 3.4](#) and [Theorem 3.5](#) using our formal definition of NAND programs. But it is also possible to directly give syntactic transformations of the code of programs computing  $F$  and  $G$  to programs computing  $G \circ F$  and  $F \oplus G$  respectively. 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  $F\|G$ ).

Before proving [Theorem 3.4](#) and [Theorem 3.5](#), note that they do imply [Theorem 3.3](#). Indeed, it's easy to verify that for every  $x \in \{0,1\}^4$ ,

$$\text{XOR}_4(x) = \sum_{i=0}^3 x_i (\mod 3) = ((x_0 + x_1 \mod 2) + (x_2 + x_3 \mod 2) \mod 2) = \text{XOR}_2(\text{XOR}_2(x_0, x_1) \text{XOR}_2(x_2, x_3)) \quad (3.3)$$

and hence

$$\text{XOR}_4 = \text{XOR}_2 \circ (\text{XOR}_2 \oplus \text{XOR}_2). \quad (3.4)$$

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

We leave proving [Theorem 3.6](#) as [Exercise 3.3](#).

### 3.5 Proving the composition theorems

We now formally prove the “Sequential Composition Theorem”

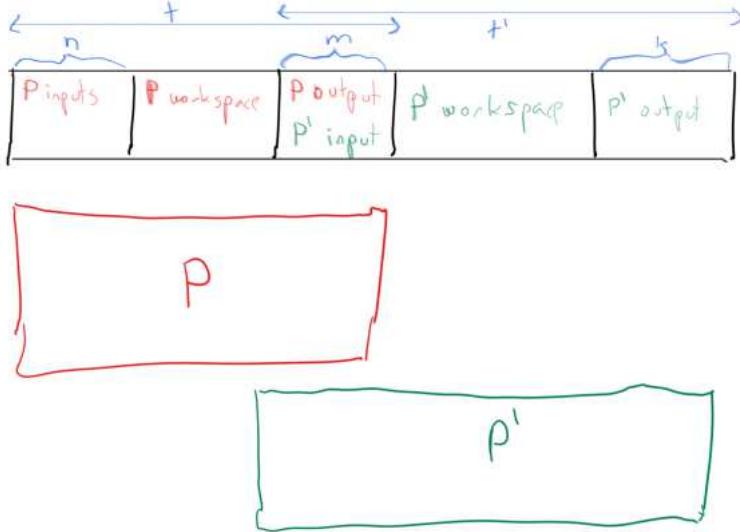
[Theorem 3.4](#), leaving the “parallel composition” as an exercise. The idea behind the proof is that given a program  $P = (V, X, Y, L)$  that computes  $F$  and a program  $P' = (V', X', Y', L')$  that computes  $G$ , we can hope to obtain a program  $P''$  that computes  $G \circ F$  by simply “copy and pasting” the code for  $P'$  after the code for  $P$ , replacing the inputs of  $G$  with the outputs of  $F$ . In our tuple notation this corresponds renaming the variables  $X'$  so they are the same as  $Y$ , and then making  $P'' = (V \cup V', X, Y', L'')$  where  $L''$  is obtained by simply concatenating  $L$  and  $L'$ .

It is an interesting exercise to try to prove that this transformation works. If you do so, you will find out that you simply can't make the proof go through. It turns out the issue is not about mere “formalities”. This transformation is simply not correct: if  $G$  and  $F$  use the same workspace variable `foo`, then the program  $P'$  might assume `foo` is initialized to zero, while the program  $P$  might assign `foo` a nonzero value. Thus in the proof we will need to take care of this issue, and ensure that  $P'$  and  $P$  use disjoint workspace variables. This is one example of a general phenomenon. Trying and failing to prove that a program or algorithm is correct often leads to discovery of bugs in it.

We now turn to the full proof. It is somewhat cumbersome since we have to **(1)** fully specify the transformation of  $P$  and  $P'$  to  $P''$  and **(2)** prove that the transformed program  $P''$  does actually compute  $G \circ F$ . Nevertheless, because proofs about computation can be subtle, it is important that you read carefully the proof and make sure you understand every step in it.

*Proof of Theorem 6.2.* Let  $P = (V, X, Y, L)$  and  $P' = (V', X', Y', L')$  be the programs for computing  $F$  and  $G$  respectively, and assume without loss of generality that they are in canonical form, and so  $X = [n]$ ,  $Y = X' = [m]$ , and  $Y' = [k]$ . Let  $t = |V|$ ,  $s = |L|$ ,  $t' = |V'|$ , and  $s' = |L'|$ . We will construct an  $s + s'$  line canonical form program  $P''$  with  $t + t' - m$  variables that computes  $G \circ F : \{0, 1\}^n \rightarrow \{0, 1\}^k$  (see [Fig. 3.5](#)). To specify  $P''$  we only need to define its set of lines  $L'' = (L''_0, L''_1, \dots, L''_{s+s'-1})$ . The first  $s$  lines of  $L''$  simply equal  $L$ . The next  $s'$  lines are obtained from  $L'$  by adding  $t - m$  to every label.<sup>12</sup> In

<sup>12</sup> Since we are representing programs in canonical form, each line is a triple of numbers, and adding  $t - m$  simply shifts the set of variables used by  $P'$  from  $\{0, \dots, t' - 1\}$  to the last  $t'$  elements of  $[t + t' - m]$  which is the set of variables of the composed program  $P''$ .



**Figure 3.5:** Given canonical-variables programs  $P$  and  $P'$  that compute  $F : \{0,1\}^n \rightarrow \{0,1\}^m$  and  $G : \{0,1\}^m \rightarrow \{0,1\}^k$  respectively, we create a program  $P''$  using  $t + t' - m$  variables to compute  $G \circ F$  where  $t$  and  $t'$  are the number of variables used by  $P$  and  $P'$  respectively. In this program, the variables corresponding to the outputs of  $P$  and the inputs of  $P'$  are shared, but all other variables are disjoint.

other words, for every  $\ell \in [s + s']$ ,

$$L''_\ell = \begin{cases} L_\ell & \ell < s \\ (L'_{\ell-s,0} + t - m, L'_{\ell-s,1} + t - m, L'_{\ell-s,2} + t - m) & \ell > s \end{cases} \quad (3.5)$$

where  $L' = ((L'_{0,0}, L'_{0,1}, L'_{0,2}), \dots, (L'_{t',0}, L'_{t',1}, L'_{t',2}))$ .

We now need to prove that  $P''$  computes  $G \circ F$ . We will do so by showing the following two claims:<sup>13</sup>

**Claim 1:** For every  $x \in \{0,1\}^n$  and  $\ell \in [s + 1]$ ,  $\text{conf}_\ell(P'', x) = (\ell, \sigma 0^{t'-m})$  where  $(\ell, \sigma) = \text{conf}_\ell(P, x)$ .<sup>14</sup>

**Claim 2:** For every  $x \in \{0,1\}^n$  and  $\ell \in \{s, \dots, s + s'\}$ ,  $\text{conf}_\ell(P'', x) = (\ell, z\sigma)$  where  $(\ell - t, \sigma) = \text{conf}_{\ell-t}(P', F(x))$  and  $z \in \{0,1\}^{t-m}$  is some string.

Claim 2 implies the theorem, since by our definition of  $P''$  computing the function  $G$ , it follows that if  $(t', \sigma) = \text{conf}_{t'}(P'', F(x))$  then the last  $k$  bits of  $\sigma$  correspond to  $G(F(x))$ . We will outline the proof of Claims 1 and 2:

**Proof outline of claim 1:** The proof follows because the lines  $L''_0, \dots, L''_{t-1}$  are identical to the lines of  $L$ , and hence they only touch the first  $t$  variables, and leave the remaining  $t' - m$  equal to 0.

<sup>13</sup> These two claims are easiest to understand by looking at Fig. 3.5. Claim 1 simply says that in the first  $s$  steps of the execution, the state of the first  $t$  variables corresponds to the state in the execution of  $P$ , and the last  $t' - m$  variables are untouched. Claim 2 says that in the following  $t'$  step, the state of the first  $t - m$  variables remains as they were at the end of the execution of  $P$ , and the last  $t'$  variables evolve according to the execution of  $P'$ . The variables  $\{t - m, \dots, t - 1\}$  are involved in both executions: they play the role of output variables for  $P$  and the role of input variables for  $P'$ .

<sup>14</sup> Recall that for a program in canonical form, we can think of the state  $\sigma$  as a string, and hence  $\sigma 0^{t'-m}$  means that we concatenate  $t' - m$  zeroes to this string. Similarly in Claim 2 below,  $z\sigma$  refers to the concatenation of the string  $z$  to  $\sigma$ .

**Proof outline of claim 2:** By Claim 1, by step  $t$  the configuration has the value  $F(x)$  on the variables  $t - m, \dots, t - 1$ . Since the lines  $L''_t, \dots, L''_{t+t'-1}$  only touch the variables  $t - m, \dots, t + t' - 1 - m$ , the last  $t'$  variables correspond to the same configuration as running the program  $P''$  on  $F(x)$ .

To make these outlines into full proofs, we need to use *induction*, so we can argue that for every  $\ell$ , if we maintained these properties up to step  $\ell - 1$ , then they are maintained in step  $\ell$  as well. We omit the full inductive proof, though working out it for yourself can be an excellent exercise in getting comfortable with such arguments. ■

### 3.5.1 Example: Adding two-bit numbers

Using composition, we can show how to add *two bit* numbers. That is, the function  $ADD_2 : \{0,1\}^4 \rightarrow \{0,1\}^3$  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. 3.6)

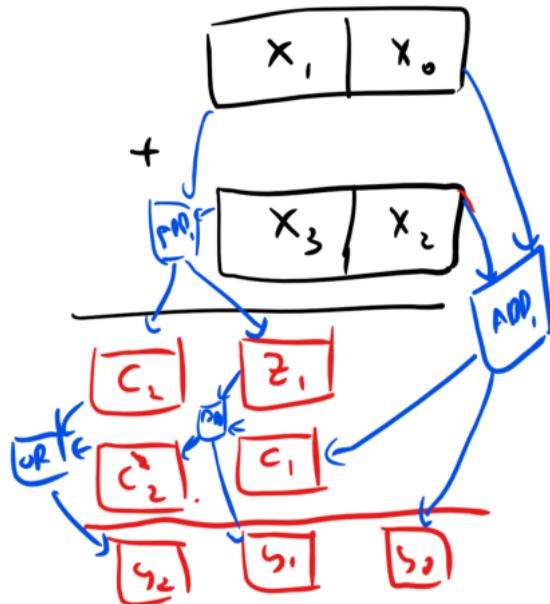
```

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 3.4 and Theorem 3.5. That is, we first compute  $(y_0, c_1, z_1, c_2) = ADD_1 \parallel ADD_1(x_0, x_2, x_1, x_3)$ , which we can do in 10 lines via Theorem 3.5, 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  $c_2 \text{ OR } c'_2$  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:
    
```



**Figure 3.6:** Adding two 2-bit numbers via the grade school algorithm.

```

// 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:

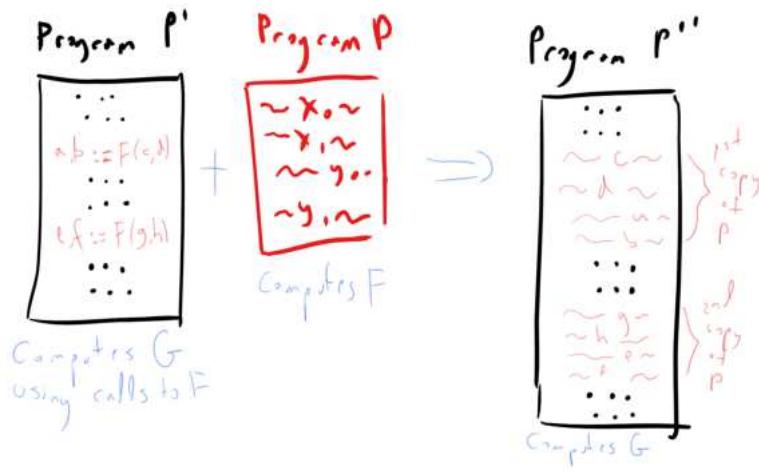
```

Executing step 1: "u:=x_0NANDx_2" x_0 = 0, x_2 = 1, u
is assigned 1,
Executing step 2: "v:=x_0NANDu" x_0 = 0, u = 1, v is
assigned 1,
Executing step 3: "w:=x_2NANDu" x_2 = 1, u = 1, w is
assigned 0,
Executing step 4: "y_0:=vNANDw" v = 1, w = 0, y_0 is
assigned 1,
Executing step 5: "c_1:=uNANDu" u = 1, u = 1, c_1 is
assigned 0,
Executing step 6: "u:=x_1NANDx_3" x_1 = 1, x_3 = 1, u
is assigned 0,
Executing step 7: "v:=x_1NANDu" x_1 = 1, u = 0, v is
assigned 1,
Executing step 8: "w:=x_3NANDu" x_3 = 1, u = 0, w is
assigned 1,
Executing step 9: "z_1:=vNANDw" v = 1, w = 1, z_1 is
assigned 0,
Executing step 10: "z'_1:=uNANDu" u = 0, u = 0, z'_1 is
assigned 1,
Executing step 11: "u:=z_1NANDc_1" z_1 = 0, c_1 = 0, u
is assigned 1,
Executing step 12: "v:=z_1NANDu" z_1 = 0, u = 1, v is
assigned 1,
Executing step 13: "w:=c_1NANDu" c_1 = 0, u = 1, w is
assigned 1,
Executing step 14: "y_1:=vNANDw" v = 1, w = 1, y_1 is
assigned 0,
Executing step 15: "u:=z'_1NANDz'_1" z'_1 = 1, z'_1 =
1, u is assigned 0,
Executing step 16: "v:=z_1NANDc_1" z_1 = 0, c_1 = 0, v
is assigned 1,
Executing step 17: "y_2:=uNANDv" 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 program  $P'$  that can compute the function  $G$  using  $t$  standard NAND lines and  $k$  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).



**Figure 3.7:** 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 3.6](#).<sup>15</sup>

<sup>15</sup> **Hint:** Prove by induction that for every  $n > 1$  which is a power of two,  $XOR_n \in SIZE(4(n - 1))$ . Then use this to prove the result for every  $n$ .

### 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)

### 3.10 Acknowledgements



### Learning Objectives:

- Get comfort with syntactic sugar or automatic translation of higher level logic to NAND code.
- More techniques for translating informal or higher level language algorithms into NAND.
- Learn proof of major result: every finite function can be computed by some NAND program.
- Start thinking *quantitatively* about number of lines required for computation.

## 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

```
foo := bar
```

into the valid NAND code:

```
notbar := bar NAND bar  
foo := notbar NAND notbar
```

the reason being that for every  $a \in \{0,1\}$ ,  $NAND(a,a) = NOT(a AND a) = NOT(a)$  and so in these two lines `notbar` is assigned the negation of `bar` and so `foo` is assigned the negation of the negation of `bar`, which is simply `bar`.

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.<sup>1</sup> We will use several such “syntactic sugar” constructs to make our descriptions of NAND programs shorter and simpler. However, these descriptions are merely 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 have the values 0 and 1 respectively by adding the lines

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

Note that since for every  $x \in \{0, 1\}$ ,  $NAND(x, \bar{x}) = 1$ , the variable `one` will get the value 1 regardless of the value of  $x_0$ , and the variable `zero` will get the value  $NAND(1, 1) = 0$ .<sup>2</sup> 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`.

<sup>1</sup> 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. One modern example is the [Babel](#) JavaScript syntax transformer, that converts JavaScript programs written using the latest features into a format that older Browsers can accept. It even has a [plug-in](#) architecture, that allows users to add their own syntactic sugar to the language.

<sup>2</sup> 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.

### 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 (cond) {
    ...
    some code here
    ...
}
```

To ensure that there is code that will only be executed when the variable `cond` is equal to 1. We can do so by replacing every variable `foo` that is assigned a value in the code by a variable `tempfoo` and then simply execute the code (*regardless* of whether `cond` is false or true). After the code is executed, we want to ensure that if `cond` is true then the value of every such variable `foo` is replaced with `tempfoo`, and otherwise the value of `foo` should be unchanged.<sup>3</sup> We do so by assigning to every variable `foo` the value  $\text{MUX}(\text{foo}, \text{tempfoo}, \text{cond})$  where  $\text{MUX} : \{0,1\}^3 \rightarrow \{0,1\}$  is the *multiplexer* function that on input  $(a, b, c)$  outputs  $a$  if  $c = 0$  and  $b$  if  $c = 1$ . This function has a 4-line NAND program:

```
nx_2 := x_2 NAND x_2
u := x_0 NAND nx_2
v := x_1 NAND x_2
y_0 := u NAND v
```

We leave it as [Exercise 4.2](#) to verify that this program does indeed compute the *MUX* function.

<sup>3</sup> We assume here for simplicity of exposition that `cond` itself is not modified by the code inside the `if` statement. Otherwise, we will copy it to a temporary variable as well.

### 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

```
...
function_code'
...

```

where `function_code'` is obtained by replacing all occurrences of `a` with `e,f`, `c` with `g,h`, `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 Bounded loops

We can use “copy paste” to implement a bounded variant of *loops*, as long we only need to repeat the loop a fixed number of times. For example, we can use code such as:

```
for i in [7,9,12] do {
    y_i := foo_i NAND x_i
}
```

as shorthand for

```
y_7 := foo_7 NAND x_7
y_9 := foo_9 NAND x_9
y_12 := foo_12 NAND x_12
```

More generally, we will replace code of the form

```
for i in RANGE do {
    code
}
```

where `RANGE` specifies a finite set  $I = \{i_0, \dots, i_{k-1}\}$  of natural numbers, with  $|R|$  copies of `code`, where for  $j \in [k]$ , we replace all occurrences of `_i` in the  $j$ -th copy with `_⟨i_j⟩`. We specify the set  $I = \{i_0, \dots, i_{k-1}\}$  by simply writing `[⟨i_0⟩, ⟨i_1⟩, …, ⟨i_{k-1}⟩]`. We will also use the `⟨beg⟩:⟨end⟩` notation so specify the interval  $\{beg, beg + 1, \dots, end - 1\}$ . So for example

```
for i in [3:5,7:10] do {
    foo_i := bar_i NAND baz_i
}
```

will be a shorthand for

```
foo_3 := bar_3 NAND baz_3
foo_4 := bar_4 NAND baz_4
```

```
foo_7 := bar_7 NAND baz_7
foo_8 := bar_8 NAND baz_8
foo_9 := bar_9 NAND baz_9
```

One can also consider fancier versions, including inner loops and allowing arithmetic expressions such as  $<5*i+3>$  in indices. The crucial point is that (unlike most programming languages) we do not allow the number of times the loop is executed to depend on the input, and so it is always possible to “expand out” the loop by simply copying the code the requisite number of times.

#### 4.1.5 Example:

Using these features, we can express the code of the  $ADD_2$  function we saw last lecture as

```
def c := AND(a,b) {
    notc := a NAND b
    c := notc NAND notc
}

def c := XOR(a,b) {
    u := a NAND b
    v := a NAND u
    w := b NAND u
    c := v NAND w
}

y_0 := XOR(x_0,x_2) // add first digit
c_1 := AND(x_0,x_2)
z_1 := XOR(x_1,x_2) // add second digit
c_2 := AND(x_1,x_2)
y_1 := XOR(c_1,y_2) // add carry from before
c'_2 := AND(c_1,y_2)
y_2 := XOR(c'_2,x_2)
```

#### 4.1.6 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  $PAIR : \mathbb{N}^2 \rightarrow \mathbb{N}$ . (For example, we can use  $PAIR(x,y) = 2^x 3^y$ , but there are also more efficient embeddings, see [Exercise 4.1](#).)

Hence we can replace any variable of the form  $\text{foo}_{\langle i \rangle, \langle j \rangle}$  with  $\text{foo}_{\langle \text{PAIR}(i, j) \rangle}$ , and similarly for three dimensional arrays.

#### 4.1.7 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

```
foo := "b"
```

would be a shorthand for the program

```
foo_0 := 0
foo_1 := 0
foo_2 := 1
```

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

```
foo := "be"
```

as a shorthand for

```
foo_{0,0} := 0
foo_{0,1} := 0
foo_{0,2} := 1
foo_{1,0} := 1
foo_{1,1} := 0
foo_{1,2} := 0
```

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

#### 4.1.8 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.<sup>4</sup> So, code such as

<sup>4</sup> 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.

```
foo := 5 // (1,0,1) in binary
```

will be shorthand for

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

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

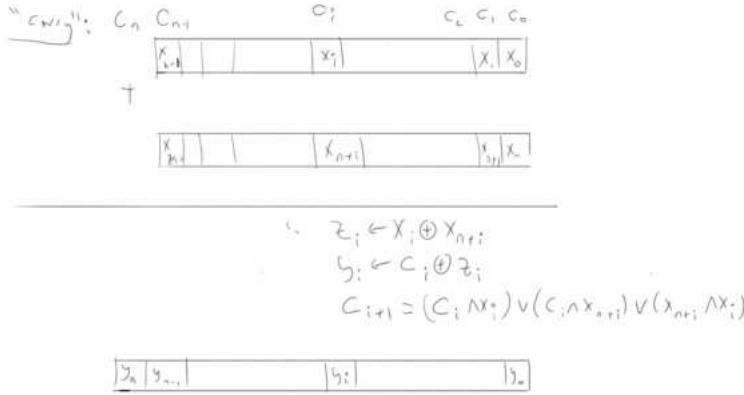
```
j := k + l
if (m*n>k) {
    code...
}
```

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  $100n$ .



**Figure 4.1:** Translating the grade school addition algorithm into a NAND program. If at the  $i^{th}$  stage, the  $i^{th}$  digits of the two numbers are  $x_i$  and  $x_{n-i}$  and the carry is  $c_i$ , then the  $i^{th}$  digit of the sum will be  $(x_i \oplus x_{n-i}) \oplus c_i$  and the new carry  $c_{i+1}$  will be equal to 1 if any two values among  $c_i, x_i, x_{n-i}$  are 1.

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

- \* 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  $\text{ATLEASTTWO} : \{0,1\}^3 \rightarrow \{0,1\}$  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

$$\begin{aligned} ATLEASTTWO(a, b, c) &= (a \wedge b) \vee (a \wedge c) \vee (b \wedge c) \\ &= NAND(NOT(NAND(NAND(a, b), NAND(a, c))), NAND(b, c)) \end{aligned} \tag{4.1}$$

We leave accounting for the number of lines, and verifying that it is smaller than  $100n$ , as an exercise to the reader. ■

See the website <http://nandpl.org> for an applet that produces, given  $n$ , a NAND program that computes  $ADD_n$ .<sup>5</sup>

<sup>5</sup> TODO: maybe add the example of the code of  $ADD_4$ ? (using syntactic sugar)

#### 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  $MULT_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$  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 4.6](#) 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  $LOOKUP_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$  is defined as follows: For every

$x \in \{0,1\}^{2^k}$  and  $i \in \{0,1\}^k$ ,

$$\text{LOOKUP}_k(x, i) = x_i \quad (4.2)$$

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

The function  $\text{LOOKUP}_1 : \{0,1\}^3 \rightarrow \{0,1\}$  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  $\text{LOOKUP}_k : \{0,1\}^{2^k+k} \rightarrow \{0,1\}$ . 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  $\text{LOOKUP}_k$  to compute  $\text{LOOKUP}_{k+1}$ . Let us first see how we do this for  $\text{LOOKUP}_2$ . 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  $\text{LOOKUP}_2(x, i)$  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  $\text{LOOKUP}_2(x, i)$  will equal  $x_2$  if  $i_0 = 0$  and will equal  $x_3$  if  $i_0 = 1$ . Another way to say this is that

$$\text{LOOKUP}_2(x_0, x_1, x_2, x_3, i_0, i_1) = \text{LOOKUP}_1(\text{LOOKUP}_1(x_0, x_1, i_0), \text{LOOKUP}_1(x_2, x_3, i_0), i_1) \quad (4.3)$$

That is, we can compute  $\text{LOOKUP}_2$  using three invocations of  $\text{LOOKUP}_1$ . The “pseudocode” for this program will be

```

z_0 := LOOKUP_1(x_0, x_1, x_4)
z_1 := LOOKUP_1(x_2, x_3, x_4)
y_0 := LOOKUP_1(z_0, z_1, x_5)

```

(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  $\text{LOOKUP}_3$  using two invocations of  $\text{LOOKUP}_2$  and one invocation of  $\text{LOOKUP}_1$ . That is, given input  $x = (x_0, \dots, x_7)$  and  $i = (i_0, i_1, i_2)$  for  $\text{LOOKUP}_3$ , if the most significant bit of the index  $i_2$  is 0, then the output of  $\text{LOOKUP}_3$  will

equal  $\text{LOOKUP}_2(x_0, x_1, x_2, x_3, i_0, i_1)$ , while if this index  $i_2$  is 1 then the output will be  $\text{LOOKUP}_2(x_4, x_5, x_6, x_7, i_0, i_1)$ , meaning that the following pseudocode can compute  $\text{LOOKUP}_3$ ,

```
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_10)
```

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  $k \geq 2$ ,  $\text{LOOKUP}_k(x_0, \dots, x_{2^k-1}, i_0, \dots, i_{k-1})$  is equal to

$$\text{LOOKUP}_1(\text{LOOKUP}_{k-1}(x_0, \dots, x_{2^{k-1}-1}, i_0, \dots, i_{k-2}), \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \dots, x_{2^k-1}, i_0, \dots, i_{k-2}), i_{k-1}) \quad (4.4)$$

*Proof.* If the most significant bit  $i_{k-1}$  of  $i$  is zero, then the index  $i$  is in  $\{0, \dots, 2^{k-1} - 1\}$  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  $a = \text{LOOKUP}_{k-1}(x_0, \dots, x_{2^{k-1}-1}, i_0, \dots, i_{k-1})$ . On the other hand, if this most significant bit  $i_{k-1}$  is equal to 1, then the index is in  $\{2^{k-1}, \dots, 2^k - 1\}$ , in which case the result of  $\text{LOOKUP}_k(x, i)$  is the same as  $b = \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \dots, x_{2^k-1}, i_0, \dots, i_{k-1})$ . 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 = LOOKUP_(k-1)(x_0, ..., x_(2^(k-1)-1), i_0, ..., i_(k-2))
b = LOOKUP_(k-1)(x_(2^(k-1)), ..., x_(2^k-1), i_0, ..., i_(k-2))
y_0 = 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

$$L(k) \leq 2L(k-1) + 4. \quad (4.5)$$

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

$$L(k) \leq 2 \cdot 4 \cdot (2^{k-1} - 1) + 4 = 4 \cdot 2^k - 8 + 4 = 4(2^k - 1) \quad (4.6)$$

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  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ , 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 4.8](#) you will show how to extend the proof for all values of  $m$ . A function  $F : \{0, 1\}^n \rightarrow \{0, 1\}$  can be specified by a table of its values for each one of the  $2^n$  inputs. For example, the table below describes one particular function  $G : \{0, 1\}^4 \rightarrow \{0, 1\}$ .<sup>6</sup>

We can see that for every  $x \in \{0, 1\}^4$ ,  $G(x) = \text{LOOKUP}_4(1100100100001111, x)$ . Therefore the following is NAND “pseudocode” to compute  $G$ :

```
G0000 := 1
G0001 := 1
G0010 := 0
G0011 := 0
G0100 := 1
G0101 := 0
```

<sup>6</sup> In case you are curious, this is the function that computes the digits of  $\pi$  in the binary basis.

Input ( $x$ )	Output ( $G(x)$ )
0000	1
0001	1
0010	0
0011	0
0100	1
0101	0
0110	0
0111	1
1000	0
1001	0
1010	0
1011	0
1100	1
1101	1
1110	1
1111	1

```

G0110 := 0
G0111 := 1
G1000 := 0
G1001 := 0
G1010 := 0
G1011 := 0
G1100 := 1
G1101 := 1
G1110 := 1
G1111 := 1
y_0 := LOOKUP(G0000,G0001,G0010,G0011,G0100,
               G0101,G0110,G0111,G1000,G1001,
               G1010,G1011,G1100,G1101,G1111,
               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  $G_{xxx} := \text{one NAND one}$  and a statement such as  $G_{xxx} := 1$  with  $G_{xxx} := \text{zero NAND zero}$ . The call to LOOKUP will be replaced by the NAND program that computes  $\text{LOOKUP}_4$ , but we will replace the variables  $i_0, \dots, i_3$  in this program with  $x_0, \dots, x_3$  and the variables  $x_0, \dots, x_{15}$  with  $G000, \dots, G1111$ .

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  $F00\dots0$  till  $F11\dots1$  so that for every  $z \in \{0,1\}^n$ , the variable corresponding to  $z$  is assigned the value  $F(z)$ .
2. Compute  $\text{LOOKUP}_n$  on the  $2^n$  variables initialized in the previous step, with the index variable being the input variables  $x_{\langle 0 \rangle}, \dots, x_{\langle 2^n - 1 \rangle}$ . That is, just like in the pseudocode for  $G$  above, we use  $y_{\langle 0 \rangle} := \text{LOOKUP}(F00\dots00, F00\dots01, \dots, F11\dots1, x_{\langle 0 \rangle}, \dots, x_{\langle n-1 \rangle})$

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  $k = \lceil \log(n - 2 \log n) \rceil$  (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  $w_0, \dots, w_{k-1}$  to  $F(a_0, \dots, a_{n-k-1}, w_0, \dots, w_{k-1})$ . On input  $x = x_0, \dots, x_{n-1}$ , 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  $F_a(x_{n-k}, \dots, x_{n-1})$ . One can verify that  $F(x) = \text{LOOKUP}_{n-k}(P, x_0, \dots, x_{n-k-1})$ . Since we can compute  $\text{LOOKUP}_{n-k}$  using  $O(2^{n-k})$  lines, if we can compute the string  $P$  (i.e., compute variables  $P_{\langle 0 \rangle}, \dots, P_{\langle 2^{n-k}-1 \rangle}$ ) 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  $x_0, \dots, x_{k-1} \mapsto F_a(x_0, \dots, x_{k-1})$  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  $F_a(x)$  and  $F_{a'}(x)$  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  $2^{2^k}$  unique functions, we can bound the total cost to compute  $P$  by  $O(2^{2^k} 2^k) + O(2^{n-k})$ . Now it just becomes a matter of calculation. By our choice of  $k$ ,  $2^k = n - 2 \log n$  and hence  $2^{2^k} = \frac{2^n}{n^2}$ . 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  $\text{LOOKUP}_{n-k}$ ) by  $O(\frac{2^n}{n^2} \cdot n) + O(2^n/n)$ , 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  $\text{ADD}_n \in \text{SIZE}_{2n,n+1}(100n)$  and  $\text{MULT}_n \in \text{SIZE}_{2n,2n}(10000n^{\log_2 3})$ .<sup>7</sup>

<sup>7</sup> TODO: check constants

## 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  $F : \mathbb{N}^2 \rightarrow \mathbb{N}$  such that for every  $x,y$ ,  $F(x,y) \leq 100 \cdot \max\{x,y\}^2 + 100$ .
3. For every  $k$ , show that there is a one-to-one map  $F : \mathbb{N}^k \rightarrow \mathbb{N}$  such that for every  $x_0, \dots, x_{k-1} \in \mathbb{N}$ ,  $F(x_0, \dots, x_{k-1}) \leq 100 \cdot (x_0 + x_1 + \dots + x_{k-1} + 100k)^k$ . ■

**Exercise 4.2 — Computing MUX.** Prove that the NAND program below computes the function  $MUX$  (or  $LOOKUP_1$ ) where  $MUX(a,b,c)$  equals  $a$  if  $c = 0$  and equals  $b$  if  $c = 1$ : ■

```
nx_2 := x_2 NAND x_2
u := x_0 NAND nx_2
v := x_1 NAND x_2
y_0 := u NAND v
```

**Exercise 4.3 — At least two.** Give a NAND program of at most 6 lines to compute  $ATLEASTTWO : \{0,1\}^3 \rightarrow \{0,1\}$  where  $ATLEASTTWO(a,b,c) = 1$  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 `if .. then .. else ..` statement, we can still implement it. Suppose that there is an  $s$ -line NAND program to compute  $F : \{0,1\}^n \rightarrow \{0,1\}$  and an  $s'$ -line NAND program to compute  $F' : \{0,1\}^n \rightarrow \{0,1\}$ . Prove that there is a program of at most  $s + s' + 10$  lines to compute the function  $G : \{0,1\}^{n+1} \rightarrow \{0,1\}$  where  $G(x_0, \dots, x_{n-1}, x_n)$  equals  $F(x_0, \dots, x_{n-1})$  if  $x_n = 0$  and equals  $F'(x_0, \dots, x_{n-1})$  otherwise. ■

**Exercise 4.5 — 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 4.6 — 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  $1000 \cdot n^2$  lines? ■

**Exercise 4.7 — Efficient multiplication (challenge).** Write a program using your favorite programming language that on input an integer  $n$ , outputs a NAND program that computes  $MULT_n$  and has at most  $10000n^{1.9}$  lines.<sup>8</sup> What is the smallest number of lines you can use to multiply two 2048 bit numbers? ■

<sup>8</sup> Hint: Use Karatsuba's algorithm

**Exercise 4.8 — Multibit function.** Prove that

- 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  $G(x) = (F(x), F'(x))$ .
- For every function  $F : \{0,1\}^n \rightarrow \{0,1\}^m$ , there is a NAND program of at most  $10m \cdot 2^n$  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



### Learning Objectives:

- Understand one of the most important concepts in computing: duality between code and data.
- Build up comfort in moving between different representations of programs.
- Follow the construction of a “universal NAND program” that can evaluate other NAND programs given their representation.
- See and understand the proof of a major result that complements the result last lecture: some functions require an *exponential* number of NAND lines to compute.

## 5

### *Code as data, data as code*

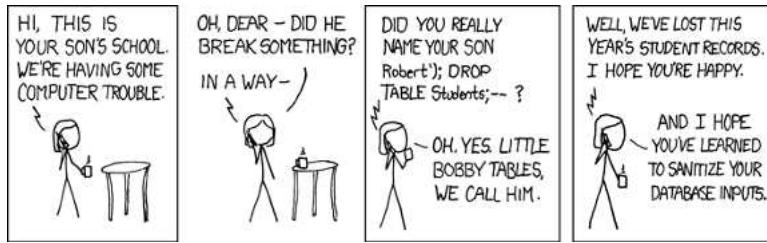
*“The term code script is, of course, too narrow. The chromosomal structures are at the same time instrumental in bringing about the development they foreshadow. They are law-code and executive power - or, to use another simile, they are architect’s plan and builder’s craft - in one.” , Erwin Schrödinger,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



**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

$$EVAL_{S,n,m} : \{0,1\}^{S+n} \rightarrow \{0,1\}^m \quad (5.1)$$

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  $m$  output bits, then  $EVAL_{S,n,m}(P, x)$  is the output of this program on input  $x$ .<sup>1</sup>

Of course to fully specify  $EVAL_{S,n,m}$ , we need to fix a precise representation scheme for NAND programs as binary strings. We can simply use the ASCII representation, though we will use a 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  $EVAL_{S,n,m}$  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 this NAND program  $U$  that computes  $EVAL_{S,n,m}$  a *bounded*

<sup>1</sup> If  $P$  does not describe a program then we don’t care what  $EVAL_{S,n,m}(P, x)$  is, but for concreteness we will set it to be  $0^m$ . Note that in this theorem we use  $S$  to denote the number of bits describing the program, rather than the number of lines in it. However, these two quantities are very closely related.

*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  $U$  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  $EVAL_{S,n,m} : \{0,1\}^{S+n} \rightarrow \{0,1\}^m$  defined above.

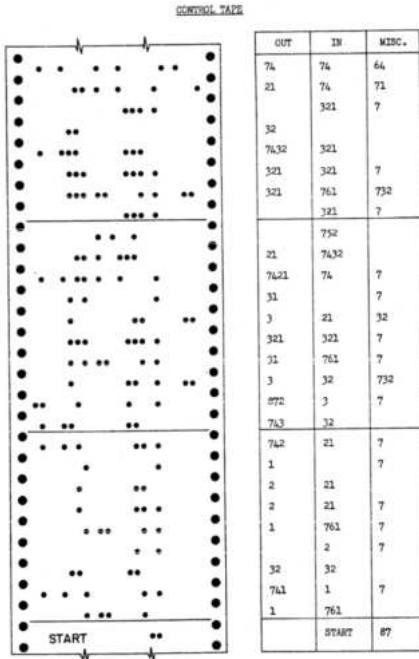
We will prove a weaker version of [Theorem 5.2](#), that will use a large number of  $O(S^2)$  lines instead of  $O(S \log S)$  as stated in the theorem. We will sketch how we can improve this proof and get the  $O(S \log S)$  bound in a future lecture. Unlike [Theorem 5.1](#), [Theorem 5.2](#) is not 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  $EVAL_{S,n,m}$  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  $EVAL_{S,n,m}$  in *Python*.<sup>2</sup> Finally, we will show how we can transform this Python program into a NAND program.

<sup>2</sup> We will not use much about Python, and a reader that has familiarity with programming in any language should be able to follow along.

## 5.2 Concrete representation for NAND programs

We can use the *canonical form* of NAND program (as per [Definition 3.5](#)) to represent it as a string. That is, if a NAND program has  $s$  lines and  $t$  distinct variables (where  $t \leq 3s$ ) then we encode every a line of the program such as `foo_54 := baz NAND blah_22` as the triple  $(a, b, c)$  where  $a, b, c$  are the numbers corresponding to `foo_54,bar,blah_22` respectively. We choose the ordering such that the numbers  $0, 1, \dots, n - 1$  encode the variables  $x_0, \dots, x_{n-1}$  and the numbers  $t - m, \dots, t - 1$  encode the variables  $y_0, \dots, y_{m-1}$ . Thus the representation of a program  $P$  of  $n$  inputs and  $m$  outputs is simply the list of triples of  $P$  in its canonical form. For example, 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.

XOR program:

```
u_0 := x_0 NAND x_1
v_0 := x_0 NAND u_0
w_0 := x_1 NAND u_0
y_0 := v_0 NAND w_0
```

is represented by the following list of four triples:

```
[[2, 0, 1], [3, 0, 2], [4, 1, 2], [5, 3, 4]]
```

Note that even if we renamed  $u$ ,  $v$  and  $w$  to `foo`, `bar` and `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 the list-of-tuples representation; for example, it can be done in 15 lines of Python.<sup>3</sup>

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

- We create an array `avars` of  $t$  integers. The value of the variable with label  $j$  will be stored in the  $j$ -th location of this array.
- We initialize the value of the input variables. We set  $i$  to be the index corresponding to the label  $x_{\langle i \rangle}$ , and so set the  $i$ -th coordinate

<sup>3</sup> If you're curious what these 15 lines are, see the appendix or the website <http://nandpl.org>.

of `avars` to  $x_i$  for every  $i \in [n]$ .

- For every triple  $(a, b, c)$  in the program's representation, we read from `avars` the values  $x, y$  of the variables  $b$  and  $c$  respectively, and then set the value of the variable indexed by  $a$  to  $NAND(x, y) = 1 - x \cdot y$ . That is, we set `avars[a] = 1-avars[b]*avars[c]`.
- The variables  $y_{\langle 0 \rangle}$  till  $y_{\langle m-1 \rangle}$  are given the indices  $t-m, \dots, t-1$  and so the output is `avars[t-m], \dots, avars[t-1]`.

The following is a *Python* function `EVAL` that on input  $L, n, m, x$  where  $L$  is a list of triples representing an  $n$ -input  $m$ -output program, and  $x$  is list of 0/1 values, returns the result of the execution of the NAND program represented by  $P$  on  $x$ :<sup>4</sup>

```
# Evaluates an n-input, m-output NAND program L on input x
# L is given in the canonical list of triples representation
# (first n variables are inputs and last m variables are
#   outputs)
def EVAL(L,n,m,x):
    s = len(L)
    avars = [0]*(3*s) # initialize variable array to zeroes,
                      # 3s is large enough to hold all variables
    avars[:n] = x # set first n vars to x

    for (a,b,c) in L: # evaluate each triple
        u = avars[b]
        v = avars[c]
        val = 1-u*v # i.e., the NAND of u and v
        avars[a] = val

    t = max([max(triple) for triple in L])+1 # num of vars in
                                                L

    return avars[t-m:] # output last m variables
```

For example, if we run

```
EVAL(
[[2, 0, 1], [3, 0, 2], [4, 1, 2], [5, 3, 4]],
2,
1,
[0,1]
)
```

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

<sup>4</sup> To keep things simple, we will not worry about the case that  $L$  does not represent a valid program of  $n$  inputs and  $m$  outputs. Also, there is nothing special about Python. We could have easily presented a corresponding function in JavaScript, C, OCaml, or any other programming language.

Accessing an element of the array `avars` at a given index takes a constant number of basic operations.<sup>5</sup> Hence (since  $n, m \leq s$  and  $t \leq 3s$ ), the program above will use  $O(s)$  basic operations.

### 5.3 A NAND interpreter in NAND

We now turn to actually proving [Theorem 5.2](#). To do this, it is of course not enough to give a Python program. We need to **(a)** give a precise representation of programs as binary strings, and **(b)** show how we compute the  $EVAL_{S,n,m}$  function on this representation by a NAND program.

First, if a NAND program has  $s$  lines, then since it can have at most  $3s$  distinct variables, it can be represented by a string of size  $S = 3s\lambda$  where  $\lambda = \lceil \log(3s) \rceil$ , by simply concatenating the binary representations of all the  $3s$  numbers (adding leading zeroes as needed to make each number represented by a string of exactly  $\lambda$  bits). So, our job is to transform, for every  $s, n, m$ , the Python code above to a NAND program  $U_{s,n,m}$  that computes the function  $EVAL_{S,n,m}$  for  $S = 3s\lambda$ . That is, given any representation  $r \in \{0,1\}^S$  of an  $s$ -line  $n$ -input  $m$ -output NAND program  $P$ , and string  $w \in \{0,1\}^n$ ,  $U_{s,n,m}(rw)$  outputs  $P(w)$ .

<sup>5</sup> 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.



Before reading further, try to think how *you* could give a “constructive proof” of [Theorem 5.2](#). That is, think of how you would write, in the programming language of your choice, a function `universal(s, n, m)` that on input  $s, n, m$  outputs the code for the NAND program  $U_{s,n,m}$  such that  $U_{s,n,m}$  computes  $EVAL_{S,n,m}$ . Note that there is a subtle but crucial difference between this function and the Python `EVAL` program described above. Rather than actually evaluating a given program  $P$  on some input  $w$ , the function `universal` should output the *code* of a NAND program that computes the map  $(P, w) \mapsto P(w)$ .

Let  $n, m, s \in \mathbb{N}$  be some numbers satisfying  $s \geq n$  and  $s \geq m$ . We now describe the NAND program  $U_{n,m,s}$  that computes  $EVAL_{S,n,m}$  for  $S = 3s\lambda$  and  $\lambda = \lceil \log(3s) \rceil$ . Our construction will follow very closely the Python implementation of `EVAL` above:<sup>6</sup>

1.  $U_{s,n,m}$  will contain variables `avars_0, ..., avars_{\langle 2^\lambda - 1 \rangle}`. (This corresponds to the line `avars = [0]*t` in the Python function `EVAL`.)

<sup>6</sup> We allow ourselves use of syntactic sugar in describing the program. We can always “unsweeten” the program later.

2. For  $i = 0, \dots, n - 1$ , we add the line  $\text{avars}_{\langle i \rangle} := \text{x}_{\langle 3s\lambda + i \rangle}$  to  $U_{s,n,m}$ . Recall that the input to  $\text{EVAL}_{s,n,m}$  is a string  $rw \in \{0,1\}^{3s\lambda+n}$  where  $r \in \{0,1\}^{3s\lambda}$  is the representation of the program  $P$  and  $w \in \{0,1\}^n$  is the input that the program should be applied on. Hence this step copies the input to the variables  $\text{avars}_0, \dots, \text{avars}_{\langle n-1 \rangle}$ . (This corresponds to the line  $\text{avars}[:n] = \text{x}$  in EVAL.)
3. For  $\ell = 0, \dots, s-1$  we add the following code to  $U_{s,n,m}$ :
- For all  $j \in [\lambda]$ , add the code  $\text{a}_{\langle j \rangle} := \text{x}_{\langle 3\ell\lambda + j \rangle}$ ,  $\text{b}_{\langle j \rangle} := \text{x}_{\langle 3\ell\lambda + \lambda + j \rangle}$  and  $\text{c}_{\langle j \rangle} := \text{x}_{\langle 3\ell\lambda + 2\lambda + j \rangle}$ . In other words, we add the code to copy to  $\text{a}, \text{b}, \text{c}$  the three  $\lambda$ -bit long strings containing the binary representation the  $\ell$ -th triple  $(a, b, c)$  in the input program. (This corresponds to the line `for (a,b,c) in L: in EVAL.`)
  - Add the code  $\text{u} := \text{LOOKUP}(\text{avars}_0, \dots, \text{avars}_{\langle 2^\lambda - 1 \rangle}, \text{b}_0, \dots, \text{b}_{\langle \lambda - 1 \rangle})$  and  $\text{v} := \text{LOOKUP}(\text{avars}_0, \dots, \text{avars}_{\langle 2^\lambda - 1 \rangle}, \text{c}_0, \dots, \text{c}_{\langle \lambda - 1 \rangle})$  where  $\text{LOOKUP}$  is the macro that computes  $\text{LOOKUP}_\lambda : \{0,1\}^{2^\lambda+\lambda} \rightarrow \{0,1\}$ . Recall that we defined  $\text{LOOKUP}_\lambda(A, i) = A_i$  for every  $A \in \{0,1\}^{2^\lambda}$  and  $i \in \{0,1\}^\lambda$  (using the binary representation to identify  $i$  with an index in  $[2^\lambda]$ ). Hence this code means that  $\text{u}$  gets the value of  $\text{avars}_{\langle b \rangle}$  and  $\text{v}$  gets the value of  $\text{avars}_{\langle c \rangle}$ . (This corresponds to the lines  $\text{u} = \text{avars}[\text{b}]$  and  $\text{v} = \text{avars}[\text{c}]$  in EVAL.)
  - Add the code  $\text{val} := \text{u} \text{ NAND } \text{v}$  (i.e.,  $\text{w}$  gets the value that should be stored in  $\text{avars}_{\langle a \rangle}$ ). (This corresponds to the line  $\text{val} = 1 - \text{u} * \text{v}$  in EVAL.)
  - Add the code  $\text{newvars}_0, \dots, \text{newvars}_{\langle 2^\lambda - 1 \rangle} := \text{UPDATE}(\text{avars}_0, \dots, \text{avars}_{\langle 2^\lambda - 1 \rangle}, \text{a}_0, \dots, \text{a}_{\langle \lambda - 1 \rangle}, \text{val})$ , where  $\text{UPDATE}$  is a macro that computes the function  $\text{UPDATE}_\lambda : \{0,1\}^{2^\lambda+\lambda+1} \rightarrow \{0,1\}^{2^\lambda}$  defined as follows: for every  $A \in \{0,1\}^{2^\lambda}$ ,  $i \in \{0,1\}^\lambda$  and  $v \in \{0,1\}$ ,  $\text{UPDATE}_\lambda(A, i, v) = A'$  such that  $A'_j = A_j$  for all  $j \neq i$  and  $A'_i = v$  (identifying  $i$  with an index in  $[2^\lambda]$ ). See below for discussions on how to implement  $\text{UPDATE}$  and other macros.
  - Add the code  $\text{avars}_{\langle j \rangle} := \text{newvars}_{\langle j \rangle}$  for every  $j \in [2^\lambda]$  (i.e., update  $\text{avars}$  to  $\text{newvars}$ ). (Steps 3.c and 3.d together correspond to the line  $\text{avars}[a] = \text{val}$  in EVAL.)

After adding all the  $s$  snippets above in Step 3, we add to the program the code  $\text{t}_0, \dots, \text{t}_{\langle \lambda - 1 \rangle} := \text{INC}(\text{MAX}(\text{avars}_0, \dots, \text{avars}_{2^\lambda}))$

where **MAX** is a macro that computes the function  $MAX_{2^\lambda, \lambda}$  and we define  $MAX_{s, \lambda} : \{0,1\}^{s\lambda} \rightarrow \{0,1\}^\lambda$  to take the concatenation of the representation of  $s$  numbers in  $[2^\lambda]$  and output the representation of the maximum number, and **INC** is a macro that computes the function  $INC_\lambda$  that increments a given number in  $[2^\lambda]$  by one. (This corresponds to the line `t = max([max(triple) for triple in L])+1` in EVAL.) We leave coming up with NAND programs for computing  $MAX_{s, \lambda}$  and  $INC_\lambda$  as an exercise for the reader.

5. Finally we add for every  $j \in [m]$ :

- (a) The code `idx_0, ..., idx_{\langle \lambda - 1 \rangle} := SUBTRACT(t_0, ..., t_{\langle \lambda \rangle}, z_0, ..., z_{\lambda-1})` where **SUBTRACT** is the code for subtracting two numbers in  $[2^\lambda]$  given in their binary representation, and each  $z_h$  is equal to either zero or one depending on the  $h$ -th digit in the binary representation of the number  $m - j$ .
- (b) `y_{\langle j \rangle} := LOOKUP( avars_0, ..., avars_{\langle 2^\lambda - 1 \rangle}, idx_0, ..., idx_{\langle \lambda - 1 \rangle} ).` (Steps 5.a and 5.b together correspond to the line `return avars[t:m:]` in EVAL.)

To complete the description of this program, we need to show that we can implement the macros for **LOOKUP**, **UPDATE**, **MAX**, **INC** and **SUBTRACT**:

- We have already seen the implementation of **LOOKUP**
- We leave the implementation of the arithmetic macros **MAX**, **INC**, and **SUBTRACT** as exercises for the reader. All of those can be done using a number of lines that is linear in the size of their input. That is  $MAX_{s, \lambda}$  can be computed in  $O(s\lambda)$  lines, and  $INC_\lambda$  and  $SUBTRACT_\lambda$  can be computed in  $O(\lambda)$  lines.
- For implementing the function **UPDATE**, note that for every indices  $i$ ,  $UPDATE_\lambda(A, i, v)_j = A_j$  unless  $j = i$  in which case  $UPDATE_\lambda(A, i, v)_i = v$ . Since we can use the syntactic sugar for **if** statements, computing **UPDATE** boils down to the function  $EQUAL_\lambda : \{0,1\}^{2\lambda} \rightarrow \{0,1\}$  such that  $EQUAL_\lambda(i_0, \dots, i_{\lambda-1}, j_0, \dots, j_{\lambda-1}) = 1$  if and only if  $i_k = j_k$  for every  $k \in [\lambda]$ .  $EQUAL_\lambda$  is equivalent to the AND of  $\lambda$  invocations of the function  $EQUAL_1 : \{0,1\}^2 \rightarrow \{0,1\}$  that checks if two bits are equal. Since each  $EQUAL_1$  (as a function on two inputs) can be computed in a constant number of lines, we can compute  $EQUAL_\lambda$  using  $O(\lambda)$  lines.

The total number of lines in  $U_{s,n,m}$  is dominated by the cost of step

3 above,<sup>7</sup> where we repeat  $s$  times the following:

1. Copying the  $\ell$ -th triple to the variables  $a,b,c$ . Cost:  $O(\lambda)$  lines.
2. Perform L00KUP on a  $2^\lambda = O(s)$  variables  $avar_0, \dots, avar_{\langle 2^\lambda - 1 \rangle}$ . Cost:  $O(2^\lambda) = O(s)$  lines.
3. Perform the UPDATE to update the  $2^\lambda$  variables  $avar_0, \dots, avar_{\langle 2^\lambda - 1 \rangle}$  to  $newvar_0, \dots, newvar_{\langle 2^\lambda - 1 \rangle}$ . Since UPDATE makes  $O(2^\lambda)$  calls to EQUAL $_\lambda$ , and each such call costs  $O(\lambda)$  lines, the total cost for UPDATE is  $O(2^\lambda \lambda) = O(s \log s)$  lines.
4. Copy  $newvar_0, \dots, newvar_{\langle 2^\lambda - 1 \rangle}$  to  $avar_0, \dots, avar_{\langle 2^\lambda - 1 \rangle}$ . Cost:  $O(2^\lambda)$  lines.

Since the loop of step 3 is repeated  $s$  times, the total number of lines in  $U_{s,n,m}$  is  $O(s^2 \log s)$  which (since  $S = \Omega(s \log s)$ ) is  $O(S^2)$ .<sup>8</sup> The NAND program above is less efficient than its Python counterpart, since NAND does not offer arrays with efficient random access. Hence for example the L00KUP operation on an array of  $s$  bits takes  $\Omega(s)$  lines in NAND even though it takes  $O(1)$  steps (or maybe  $O(\log s)$  steps, depending how we count) in *Python*. We might see in a future lecture how to improve this to  $O(s \log s)$ .

## 5.4 A Python interpreter in NAND

To prove [Theorem 5.2](#) we essentially translated every line of the Python program for EVAL into an equivalent NAND snippet. It turns out that none of our reasoning was specific to the particular function EVAL. It is possible to translate *every* Python program into an equivalent NAND program of comparable efficiency.<sup>9</sup> 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.<sup>10</sup>

There are even simpler machine languages, such as the [LEG](#)

<sup>7</sup> It is a good exercise to verify that steps 1,2,4 and 5 above can be implemented in  $O(s \log s)$  lines.

<sup>8</sup> The website <http://nandpl.org> will (hopefully) eventually contain the implementation of the NAND program  $U_{s,n,m}$  where you can also play with it by feeding it various other programs as inputs.

<sup>9</sup> 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  $n$ .

<sup>10</sup> ARM stands for “Advanced RISC Machine” where RISC in turn stands for “Reduced instruction set computer”.

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.<sup>11</sup>

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.

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  $UNIV_n$  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

<sup>11</sup> The reverse direction of compiling NAND to C code, is much easier. We show code for a `NAND2C` function in the appendix.

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.**

$$|Size(s)| \leq 2^{O(s \log s)}. \quad (5.2)$$

That is, there are at most  $2^{O(s \log s)}$  functions computed by NAND programs of at most  $s$  lines.

Moreover, the implicit constant in the  $O(\cdot)$  notation in [Theorem 5.3](#) is at most 10.<sup>12</sup> The idea behind the proof is that we can represent every  $s$  line program by a binary string of  $O(s \log s)$  bits. Therefore the number of functions computed by  $s$ -line programs cannot be larger than the number of such strings, which is  $2^{O(s \log s)}$ . In the actual proof, given below, we count the number of representations a little more carefully, talking directly about triples rather than binary strings, although the idea remains the same.

<sup>12</sup> By this we mean that for all sufficiently large  $s$ ,  $|Size(s)| \leq 2^{10s \log s}$ .

*Proof.* Every NAND program  $P$  with  $s$  lines has at most  $3s$  variables. Hence, using our canonical representation,  $P$  can be represented by the numbers  $n, m$  of  $P$ 's inputs and outputs, as well as by the list  $L$  of  $s$  triples of natural numbers, each of which is smaller or equal to  $3s$ .

If two programs compute distinct functions then they have distinct representations. So we will simply count the number of such representations: for every  $s' \leq s$ , the number of  $s'$ -long lists of triples of numbers in  $[3s]$  is  $(3s)^{3s'}$ , which in particular is smaller than  $(3s)^{3s}$ . So, for every  $s' \leq s$  and  $n, m$ , the total number of representations of  $s'$ -line programs with  $n$  inputs and  $m$  outputs is smaller than  $(3s)^{3s}$ .

Since a program of at most  $s$  lines has at most  $s$  inputs and outputs, the total number of representations of all programs of at most  $s$

lines is smaller than

$$s \times s \times s \times (3s)^{3s} = (3s)^{3s+3} \quad (5.3)$$

(the factor  $s \times s$  arises from taking all of the at most  $s$  options for the number of inputs  $n$ , all of the at most  $s$  options for the number of outputs  $m$ , and all of the at most  $s$  options for the number of lines  $s'$ ). We claim that for  $s$  large enough, the righthand side of Eq. (5.3) (and hence the total number of representations of programs of at most  $s$  lines) is smaller than  $2^{4s \log s}$ . Indeed, we can write  $3s = 2^{\log(3s)} = 2^{\log 3 + \log s} \leq 2^{2 + \log s}$ , and hence the righthand side of Eq. (5.3) is at most  $(2^{2 + \log s})^{3s+3} = 2^{(2 + \log s)(3s+3)} \leq 2^{4s \log s}$  for  $s$  large enough.

For every function  $F \in \text{Size}(s)$  there is a program  $P$  of at most  $s$  lines that computes it, and we can map  $F$  to its representation as a tuple  $(n, m, L)$ . If  $F \neq F'$  then a program  $P$  that computes  $F$  must have an input on which it disagrees with any program  $P'$  that computes  $F'$ , and hence in particular  $P$  and  $P'$  have distinct representations. Thus we see that the map of  $\text{Size}(s)$  to its representation is one to one, and so in particular  $|\text{Size}(s)|$  is at most the number of distinct representations which is it at most  $2^{4s \log s}$ . ■



**Counting by ASCII representation** 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, 011, 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  $2^{2^n}$ . Note that this is *double exponential* in  $n$ , and hence even for small values of  $n$  (e.g.,  $n = 10$ ) the number of functions from  $\{0,1\}^n$  to  $\{0,1\}$  is truly astronomical.<sup>13</sup> 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.

<sup>13</sup> “Astronomical” here is an understatement: there are much fewer than  $2^{2^{10}}$  stars, or even particles, in the observable universe.

*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 by Theorem 5.3 the total number of such functions would be at most  $2^{10s \log s} \leq 2^{10 \log s \cdot 2^n/(100n)}$ . Since  $\log s = n - \log(100n) \leq n$  this means that the total number of such functions would be at most  $2^{2^n/10}$ , contradicting the fact that there are  $2^{2^n}$  of them. ■

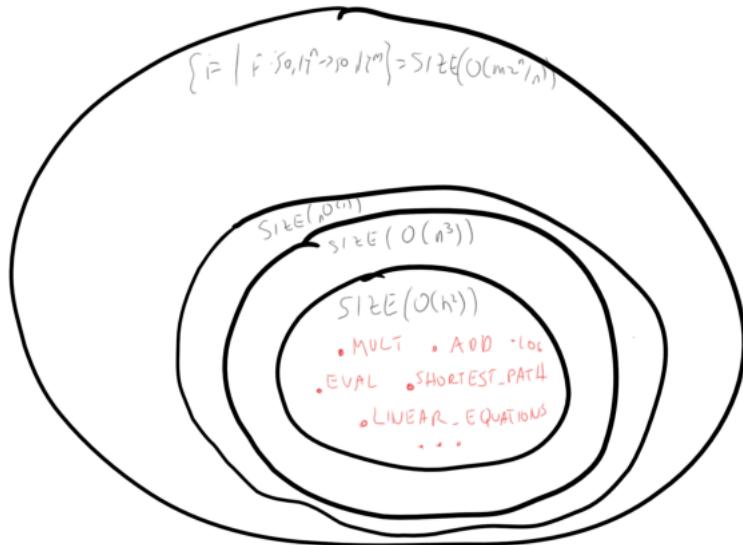
We have seen before that *every* function mapping  $\{0,1\}^n$  to  $\{0,1\}$  can be computed by an  $O(2^n/n)$  line program. We now see that this is tight in the sense that some functions do require such an astronomical number of lines to compute. In fact, as we explore in the exercises below, this 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 EVAL function) are the exception, rather than the rule.



**Advanced note: more efficient representation** The list of triples is not the shortest representation for NAND programs. As we will see in the next lecture, every NAND program of  $s$  lines and  $n$  inputs can be represented by a directed graph of  $s + n$  vertices, of which  $n$  have in-degree zero, and the  $s$  others have in-degree at most two. Using the adjacency list representation, such a graph can be represented using roughly  $2s \log(s+n) \leq 2s(\log s + O(1))$  bits. Using this representation we can reduce the implicit constant in Theorem 5.3 arbitrarily close to 2.

## 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



**Figure 5.3:** All functions mapping  $n$  bits to  $m$  bits can be computed by NAND programs of  $O(m2^n/n)$  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.

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.

## 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 — Random functions are hard (challenge).** Suppose  $n > 1000$  and that we choose a function  $F : \{0,1\}^n \rightarrow \{0,1\}$  at random, 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  $2^{-100}$ .<sup>14</sup> ■

**Exercise 5.4 — Circuit hierarchy theorem (challenge).** Prove that there is a constant  $c$  such that for every  $n$ , there is some function  $F : \{0,1\}^n \rightarrow \{0,1\}$  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.<sup>15</sup> ■

<sup>16</sup>

<sup>14</sup> Hint: An equivalent way to say this is that you need to prove that the set of functions that can be computed using at most  $2^n/(1000n)$  has fewer than  $2^{-100}2^n$  elements. Can you see why?

<sup>15</sup> Hint: Find an appropriate value of  $t$  and a function  $G : \{0,1\}^t \rightarrow \{0,1\}$  that can be computed in  $O(2^t/t)$  lines but can't be computed in  $\Omega(2^t/t)$  lines, and then extend this to a function mapping  $\{0,1\}^n$  to  $\{0,1\}$ .

<sup>16</sup> TODO: add exercise to do evaluation of  $T$  line programs in  $\tilde{O}(T^{1.5})$  time.

<sup>17</sup> TODO: EVAL is known as *Circuit Evaluation* typically. More references regarding oblivious RAM etc..

## 5.8 Bibliographical notes

<sup>17</sup>

## 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*

### Learning Objectives:

- Understand how NAND programs can map to physical processes in a variety of ways.
- Learn the model of *Boolean circuits* and get proficient in moving between description of a NAND program as a code and as a circuit or *labeled graph*.
- See that NAND is a *universal basis* for circuits, and examples for universal and non-universal families of gates.
- Understand the *physical extended Church-Turing thesis* that NAND programs capture all feasible computation in the physical world, and its physical and philosophical implications.

## 6

# 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  $F : \{0,1\}^n \rightarrow \{0,1\}$  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  $F : \{0,1\}^n \rightarrow \{0,1\}$  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).

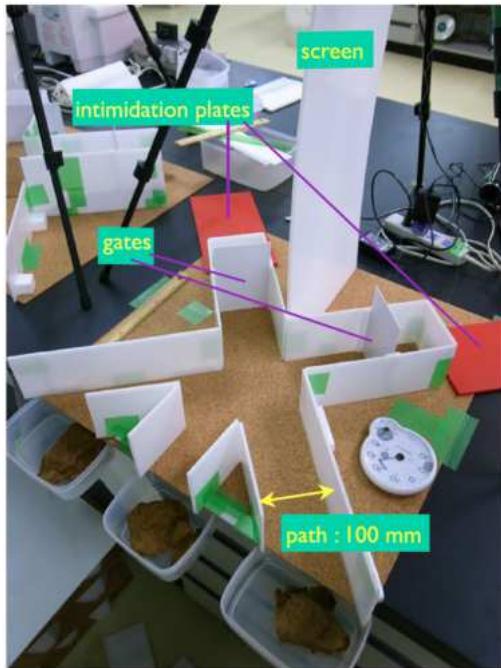
In this lecture we review some of these implementations, both so you can get an appreciation of how it is possible to directly translate NAND programs to the physical world, without going through the entire stack of architecture, operating systems, compilers, etc... as well as to emphasize that silicon-based 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

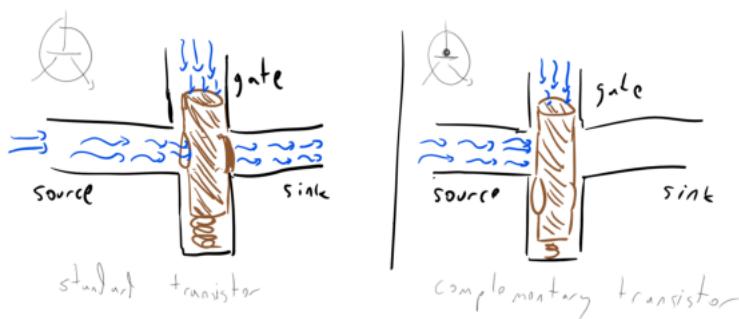
A *transistor* can be thought of as an electric circuit with two inputs, known as *source* and *gate* and an output, known as the *sink*. The gate controls whether current flows from the source to the sink. In a *standard transistor*, if the gate is “ON” then current can flow from the source to the sink and if it is “OFF” then it can’t. In a *complementary 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](#)).<sup>1</sup> However, the standard implementation uses electrical current. One of the original implementations used *vacuum tubes*.

<sup>1</sup> 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.



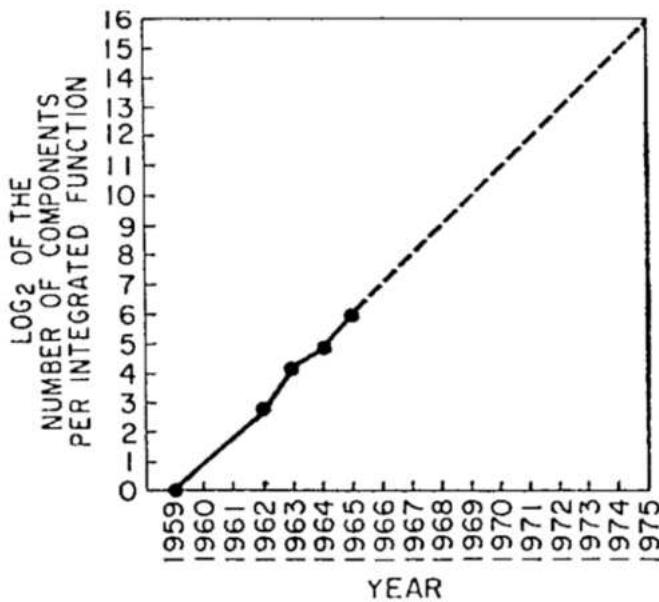
**Figure 6.1:** Crab-based logic gates from the paper “Robust soldier-crab ball gate” by Gunji, Nishiyama and Adamatzky. This is an example of an AND gate that relies on the tendency of two swarms of crabs arriving from different directions to combine to a single swarm that continues in the average of the directions.



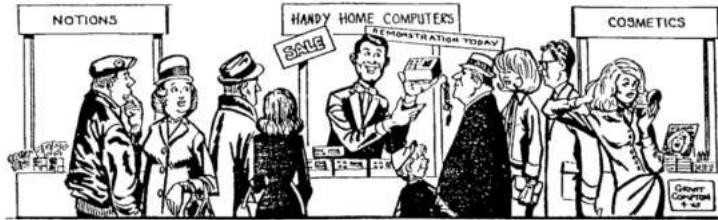
**Figure 6.2:** We can implement the logic of transistors using water. The water pressure from the gate closes or opens a faucet between the source and the sink.

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



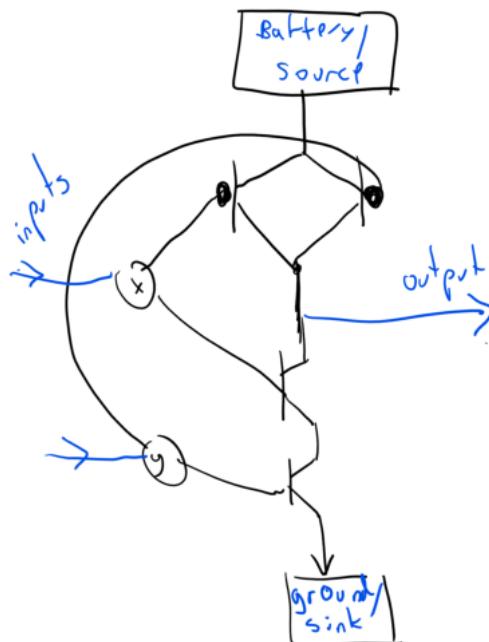
**Figure 6.4:** Gordon Moore's cartoon "predicting" the implications of radically improving transistor density.



**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).



**Figure 6.6:** Implementing a NAND gate using transistors.

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 *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  $NAND : \{0, 1\}^2 \rightarrow \{0, 1\}$ . We now show this more formally.

#### 6.4 Representing programs as graphs

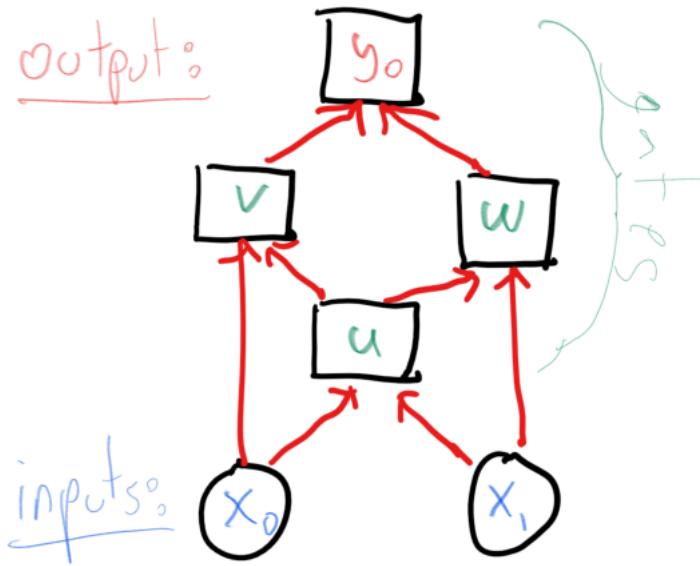
We now define NAND programs as circuits, using the notion of *directed acyclic graphs* (DAGs).

**P** If you are not comfortable with the definitions of graphs, and in particular directed acyclic graphs (DAGs), now would be a great time to go back to the “mathematical background” lecture, as well as some of the resources [here](#), and review these notions.

**Definition 6.1 — 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  $L : V \rightarrow_p S$  is the (partial) one-to-one labeling function that maps vertices into the set  $S = \{x_0, \dots, x_{\langle n-1 \rangle}, y_0, \dots, y_{\langle m-1 \rangle}\}$ . 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. 6.7](#), which describes the NAND circuit that corresponds to the 4-line NAND program we presented above for the  $XOR_2$  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



**Figure 6.7:** A NAND circuit for computing the  $XOR_2$  function. Note that it has exactly four gates, corresponding to the four lines of the NAND program we presented above. The green labels  $u, v, w$  for non-output gates are just for illustration and comparison with the NAND program, and are not formally part of the circuit.

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:

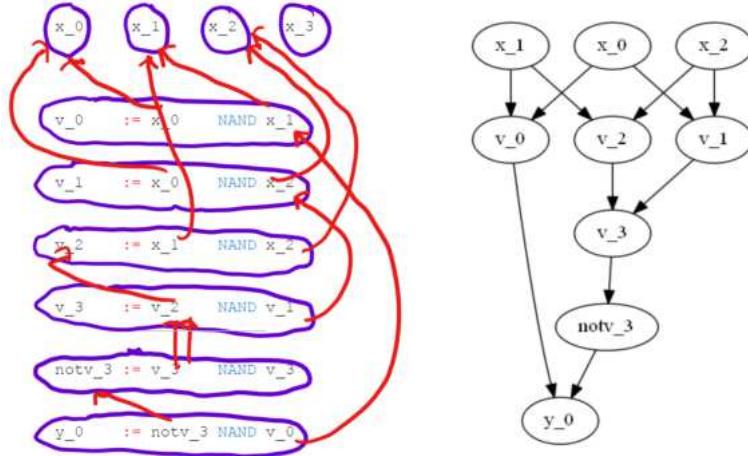
**Definition 6.2 — Computing a function by a NAND circuit.** Let  $F : \{0,1\}^n \rightarrow \{0,1\}^m$  and let  $C = (V, E, L)$  be a NAND circuit with  $n$  inputs and  $m$  outputs. We say that  $C$  computes  $F$  if there is a map  $Z : V \rightarrow \{0,1\}$ , such that for every  $x \in \{0,1\}^n$ , 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  $b = Z(w)$ , then  $Z(v) = NAND(a, b)$ . (If  $v$  has fewer than two neighbors then we replace either  $b$  or both  $a$  and  $b$  with zero in the condition above.)

**P** You should make sure you understand *why* Definition 6.2 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. Moreover, a priori it is not clear that for every circuit  $C$  and assignment  $x$  there is a map  $Z : V \rightarrow \{0,1\}$  that satisfies the conditions of Definition 6.2. However, this follows from Theorem 6.1

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 6.1 — Equivalence of circuits and straightline programs.** For every  $F : \{0,1\}^n \rightarrow \{0,1\}^m$  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.



**Figure 6.8:** The transformation of a NAND program to a circuit demonstrated on the program for the ATLEASTTWO function. Given a program of  $s$  lines and  $n$  inputs, we construct a circuit that is a graph with  $s + n$  vertices,  $s$  of which are gates and  $n$  of which are inputs. The neighbors of the vertex corresponding to a line of the program are the vertex corresponding to the lines in which the two variables on the righthand side of the assignment operators were last written to.

The idea behind the proof is simple (see Fig. 6.8 for an example). 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  $s$  gates (i.e., a graph of  $n + s$  vertices,  $n$  of which are *sources*), where each gate corresponds to a line in the program  $P$ . If line  $\ell$  involves the NAND of two variables  $\text{foo}$  and  $\text{bar}$

then if  $\ell'$  and  $\ell''$  are the lines where `foo` and `bar` were last assigned a value, then we add edges going into the gate corresponding to  $\ell$  from the gates corresponding to  $\ell', \ell''$ . (If one of the variables was an input variable, then we add an edge from that variable, if one of them was an uninitialized then we add no edge, and use the convention that it amounts to defaulting to zero.) 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  $\{(0, 0), \dots, (0, n - 1), (1, 0), \dots, (1, s - 1)\}$ . 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 NAND baz`, we put edges in the graph of the form  $\overrightarrow{(1, \ell') (1, \ell)}$  and  $\overrightarrow{(1, \ell'') (1, \ell)}$  where  $\ell'$  and  $\ell''$  are the last lines before  $\ell$  in which the variables `bar` and `baz` were assigned a value. If the variable `bar` and/or `baz` was not assigned a value prior to the  $\ell$ -th line and is not an input variable then we don’t add a corresponding edge. If the variable `bar` and/or `baz` is an input variable  $x_{\langle i \rangle}$  then we add the edge  $\overrightarrow{(0, i) (1, \ell)}$ . We label the vertices of the form  $(0, i)$  with  $x_{\langle i \rangle}$  for every  $i \in [n]$ . For every  $j \in [m]$ , let  $\ell$  be the last line in which the variable  $y_{\langle j \rangle}$  is assigned a value,<sup>2</sup> and label the vertex  $(1, \ell)$  with  $y_{\langle j \rangle}$ . Note that the vertices of the form  $(0, i)$  have in-degree zero, and all edges of the form  $\overrightarrow{(1, \ell') (1, \ell)}$  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  $y_{\langle j \rangle}$  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

<sup>2</sup> As noted in the appendix, valid NAND programs must assign a value to all their output variables.

execution of  $P$  on input  $x$ . Now, as per [Definition 6.2](#), 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  $y_{-\langle j \rangle}$  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  $\{v_0, \dots, v_{n+s-1}\}$  such that  $\overrightarrow{v_i v_j} \in E, v_i < v_j$ . Such a sorting can be found for every DAG (see also [Exercise 0.11](#)). 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  $x_{-\langle i \rangle}$ .

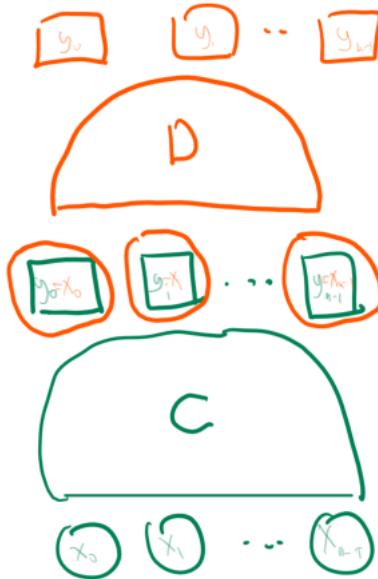
Now for  $\ell = 0, 1, \dots, n+s-1$  we will define a variable  $var(\ell)$  in our resulting program as follows: If  $\ell < n$  then  $var(\ell)$  equals  $x_{-\langle i \rangle}$ . If  $v_\ell$  is an output gate labeled with  $y_{-\langle j \rangle}$  then  $var(\ell)$  equals  $y_{-\langle j \rangle}$ . otherwise  $var(\ell)$  will be a temporary workspace variable  $temp_{-\langle \ell - n \rangle}$ . 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  $var(n+k) := var(i) \text{ NAND } var(j)$ . If  $v_k$  has fewer than two in-neighbors then we replace the corresponding variable with the variable 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 same 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 6.2](#). 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  $y_{-\langle j \rangle}$  then this value will equal  $y_j$ , meaning that on input  $x$  our program will output  $y = F(x)$ . ■

## 6.5 Composition from graphs

Given [Theorem 6.1](#), we can reprove our composition theorems in the circuit formalism, which has the advantage of making them more intuitive. That is, we can prove [Theorem 3.4](#) and [Theorem 3.5](#) by showing how to transform a circuits for  $F$  and  $G$  into circuits for  $F \circ G$  and  $F \oplus G$ . This is what we do now:

**Theorem 6.2 — 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  $G \circ F : \{0,1\}^n \rightarrow \{0,1\}^k$ .

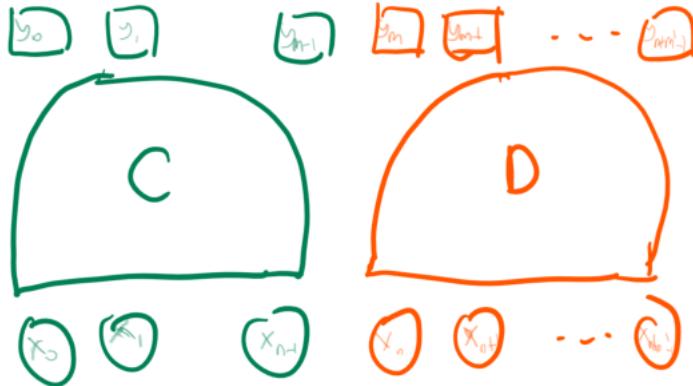


**Figure 6.9:** Given a circuit  $C$  computing  $F : \{0,1\}^n \rightarrow \{0,1\}^m$  and a circuit  $D$  computing  $G : \{0,1\}^m \rightarrow \{0,1\}^k$ , we obtain a circuit  $E$  computing  $G \circ F$  by identifying the inputs of  $D$  with the outputs of  $C$ . That is, the resulting circuit 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  $C$ .

*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. 6.9](#). 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 6.2](#)), 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  $G(F(x))$ . ■

**Theorem 6.3 — 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  $G : \{0, 1\}^{n'} \rightarrow \{0, 1\}^{m'}$  then there is a circuit  $E$  of size  $|C| + |D|$  computing the function  $G \oplus F : \{0, 1\}^{n+n'} \rightarrow \{0, 1\}^{m+m'}$ .



**Figure 6.10:** Given a circuit  $C$  computing  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$  and a circuit  $D$  computing  $G : \{0, 1\}^{n'} \rightarrow \{0, 1\}^{m'}$  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  $x_{-n}, \dots, x_{-n+n'-1}$  and  $y_{-m}, \dots, y_{-m+m'-1}$ .

*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. 6.10](#). 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  $x_0, \dots, x_{n'-1}$  to  $x_{-n}, \dots, x_{-n+n'-1}$  and the labelling of the outputs of  $D$  from  $y_0, \dots, y_{m'-1}$  to  $y_{-m}, \dots, y_{-m+m'-1}$ . 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  $F(x_0, \dots, x_{n-1})G(x_n, \dots, x_{n+n'-1})$ . ■

## 6.6 General Boolean circuits: a formal definition

We now define the notion of *general* Boolean circuits that can use any set  $B$  of gates and not just the NAND gate.

**Definition 6.3 — A basis of gates.** A *basis for Boolean circuits* is a finite set  $B = \{g_0, \dots, g_{c-1}\}$  of finite Boolean functions, where each function  $g \in B$  maps strings of some finite length (which we denote by  $\text{in}(g)$ ) to  $\{0, 1\}$ .

We now define the notion of a general Boolean circuit with gates from  $B$ .<sup>3</sup>

**Definition 6.4 — General Boolean circuits.** Let  $B$  be a basis for Boolean circuits. A *circuit over the basis  $B$*  (or  $B$ -circuit for short) with  $n$  inputs and  $m$  outputs is a labeled directed acyclic graph (DAG) over the vertex set  $[n+s]$  for  $s \in \mathbb{N}$ . The vertices  $\{0, \dots, n-1\}$  are known as the “input variables” and have in-degree zero. Every vertex apart from the input variables is known as a *gate*. Each such vertex is labeled with a function  $g \in B$  and has in-degree  $\text{in}(g)$ . The last  $m$  vertices  $\{n+s-m, \dots, n+s-1\}$  have out-degree zero and are known as the *output gates*. We denote the circuit as  $C = ([n+s], E, L)$  where  $[n+s], E$  are the vertices and edges of the circuit, and  $L : \{n, \dots, n+s-1\} \rightarrow B$  is the labeling function that maps vertices into the set  $B$ .

<sup>3</sup> Just as we defined canonical variables in [Definition 3.5](#), it will be convenient for us to assume that the vertex set of such a circuit is an interval of the form  $\{0, 1, 2, \dots, n+s\}$  for  $n, s \in \mathbb{N}$ , where the first  $n$  vertices correspond to the inputs and the last  $m$  vertices correspond to the outputs.



To make sure you understand this definition, stop and think how a Boolean circuits with AND, OR, and NOT gates corresponds to a  $B$ -circuit per [Definition 6.4](#), where  $B = \{\text{AND}, \text{OR}, \text{NOT}\}$  and  $\text{AND} : \{0, 1\}^2 \rightarrow \{0, 1\}$  is the function  $\text{AND}(a, b) = a \cdot b$ ,  $\text{OR}(a, b) \rightarrow \{0, 1\}$  is the function  $\text{OR}(a, b) = 1 - (1 - a)(1 - b)$  and  $\text{NOT} : \{0, 1\} \rightarrow \{0, 1\}$  is the function  $\text{NOT}(a) = 1 - a$ .<sup>4</sup>

The *size* of a circuit  $C$ , denoted by  $|C|$ , is the number of gates that it contains. An  $n$ -input  $m$ -output circuit  $C = ([n+s], E, L)$  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:

1. For the source vertices  $\{0, \dots, i-1\}$  we define  $\text{val}(i) = x_i$  for all  $i \in [n]$ .

<sup>4</sup> Another commonly used notation  $x \wedge y$  for  $\text{AND}(x, y)$ ,  $x \vee y$  for  $\text{OR}(x, y)$  and  $\bar{x}$  or  $\neg x$  for  $\text{NOT}(x)$ .

2. For a non source vertex  $v$  that is labeled with  $g \in B$ , if its incoming neighbors are vertices  $v_1, \dots, v_k$  (sorted in order) and their values have all been set then we let  $\text{val}(v) = f(\text{val}(v_1), \dots, \text{val}(v_k))$ .
3. Go back to step 2 until all vertices have values.
4. Output  $\text{val}(n+s-m), \dots, \text{val}(n+s-1)$ .

The *output* of the circuit  $C$  on input  $x$ , denoted by  $C(x)$ , is the string  $y \in \{0,1\}^m$  outputted by this process. We say that the circuit  $C$  *computes the function*  $F$  if for every  $x \in \{0,1\}^n$ ,  $C(x) = F(x)$ .

We have seen in [Theorem 4.5](#) that *every* function  $f : \{0,1\}^k \rightarrow \{0,1\}$  has a NAND program with at most  $10 \cdot 2^k$  lines, and hence [??](#) implies the following theorem (see [Exercise 6.2](#)):<sup>5</sup>

**Theorem 6.4 — 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  $S_{\text{NAND}}(f)$  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

$$S_{\text{NAND}}(f) \leq (10 \cdot 2^k) S_B(f) \quad (6.1)$$

<sup>5</sup> The bound that comes out of the proof of [Theorem 4.5](#) is  $5 \cdot 2^k$  and in fact can be easily optimized further. As  $k$  grows, we can also use the bound of  $O(2^k/k)$  mentioned in [Remark 4.4.1](#).

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  $B$  consists of the  $\wedge$  (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.



**Advanced note: depth** The *depth* of a Boolean circuit is the length of the longest path in it. The notion of depth is tightly connected to the *parallelism complexity* of the circuit. “Shallow” circuits are easier to parallelize, since a  $k$  long path we mean that we have a sequence of  $k$  gates that each needs to wait for the output of the other until it completes its computation.

It is a good exercise for you to verify that every function  $F : \{0,1\}^n \rightarrow \{0,1\}$  has a circuit that computes it which is of  $O(2^n)$  (in fact even  $O(2^n/n)$ ) size and  $O(\log n)$  depth. However, there are functions that require at least  $\log n/10$  depth (can you see why?). There are also functions for which the smallest size known circuits that compute them requires a much larger depth.

## 6.7 Neural networks

One particular basis we can use are *threshold gates*. For every vector  $w = (w_0, \dots, w_{k-1})$  of integers and integer  $t$  (some or all of whom could be negative), the *threshold function corresponding to  $w, t$*  is the function  $T_{w,t} : \{0,1\}^k \rightarrow \{0,1\}$  that maps  $x \in \{0,1\}^k$  to 1 if and only if  $\sum_{i=0}^{k-1} w_i x_i \geq t$ . For example, the threshold function  $T_{w,t}$  corresponding to  $w = (1, 1, 1, 1, 1)$  and  $t = 3$  is simply the majority function  $MAJ_5$  on  $\{0,1\}^5$ . The function  $NAND : \{0,1\}^2 \rightarrow \{0,1\}$  is the threshold function corresponding to  $w = (-1, -1)$  and  $t = -1$ , since  $NAND(x_0, x_1) = 1$  if and only if  $x_0 + x_1 \leq 1$  or equivalently,  $-x_0 - x_1 \geq -1$ .

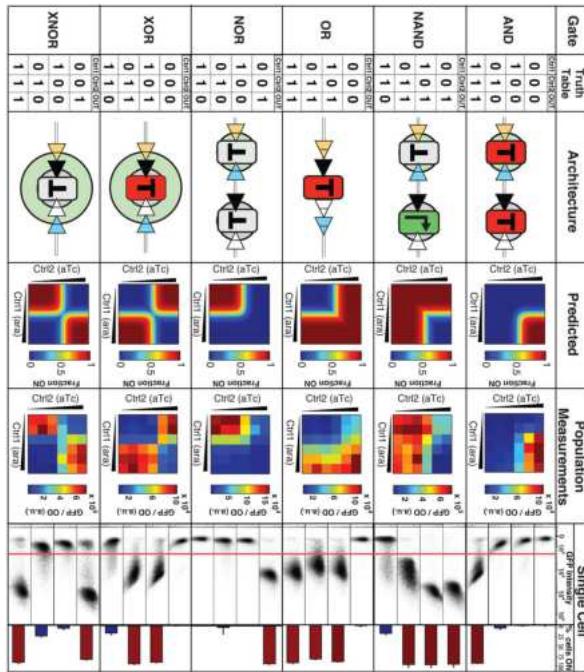
Threshold gates can be thought of as an approximation for *neuron cells* that make up the core of human and animal brains. To a first approximation, a neuron has  $k$  inputs and a single output and the neurons “fires” or “turns on” its output when those signals pass some threshold.<sup>6</sup> Hence 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 on  $k$  inputs 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 very different.

## 6.8 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  $x \wedge (\neg y)$  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. 6.11](#)). One motivation for DNA computing is to achieve increased parallelism or storage

<sup>6</sup> Typically we think of an input to neurons as being a real number rather than a binary string, but we can reduce to the binary case by representing a real number in the binary basis, and multiplying the weight of the bit corresponding to the  $i^{\text{th}}$  digit by  $2^i$ .

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.



**Figure 6.11:** Performance of DNA-based logic gates. Figure taken from paper of Bonnet et al, Science, 2013.

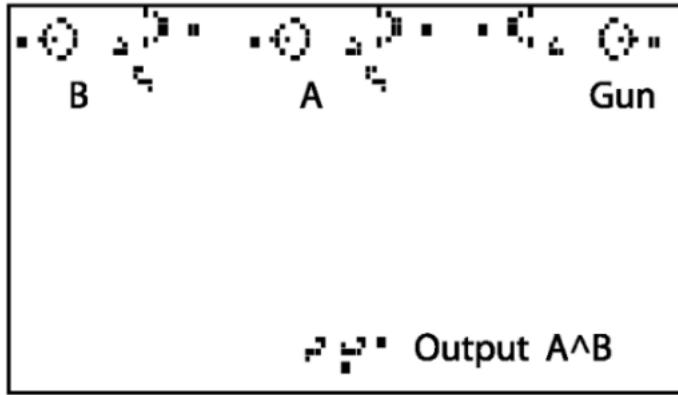
## 6.9 Cellular automata and the game of life

As we will discuss later, cellular automata such as Conway’s “Game of Life” can be used to simulate computation gates, see Fig. 6.12.

## 6.10 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 for every fixed basis  $B$  we can represent such a circuit by a string of length  $O(s \log s)$ .<sup>7</sup> We can define  $CIRCEVAL_{B,s,n,m}$  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

<sup>7</sup> The implicit constant in the  $O$  notation can depend on the basis  $B$ .



**Figure 6.12:** An AND gate using a “Game of Life” configuration. Figure taken from Jean-Philippe Rennard’s paper.

over  $B$ , and an input  $x \in \{0,1\}^n$ , and returns the evaluation of  $C$  on the input  $x$ .

Theorem 6.4 implies that every circuit  $C$  of  $s$  gates over a  $k$ -ary basis  $B$  can be transformed into a NAND program of  $s' = O(s \cdot 2^k)$  lines, and hence we can combine this transformation with last lecture’s evaluation procedure for NAND programs to conclude that  $\text{CIRCEVAL}$  for circuits of  $s$  gates over  $B$  can be computed by a NAND program of  $O(s'^2 \log s) = O(s^2 2^{2k} (\log s + k))$  lines.<sup>8</sup>

<sup>8</sup> In fact, as we mentioned, it is possible to improve this to  $O(s' \log^2 s') = O(s 2^k (\log s + k)^2)$  lines.

#### 6.10.1 Advanced note: evaluating circuits in quasilinear time.

We can improve the evaluation procedure, and evaluate  $s$ -size constant fan-in circuits (or NAND programs) in  $O(s \text{polylog}(s))$  lines.<sup>9</sup>

<sup>9</sup> TODO: add details here, use the notion of oblivious routing to embed any graph in a universal graph.

#### 6.11 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  $F : \{0,1\}^n \rightarrow \{0,1\}^m$  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. A priori it might seem rather extreme to hypothesize that our meager NAND model captures all possible physical computation. But yet, in more than a century of computing technologies, no one has yet built any scalable computing device that challenges this hypothesis.

We now discuss the “fine print” of the PECTT in more detail, as well as the (so far unsuccessful) challenges that have been raised against it. There is no single universally-agreed-upon formalization of “roughly  $s$  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, and ask that any such device can be simulated by a NAND program with a number of lines that is a polynomial (with not too large exponent) in the size of the system and the time it takes it to operate.

In other words, we can phrase the PECTT as stipulating 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 no 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.<sup>10</sup> 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 theoretically be measured) is roughly  $2^{-120}$  meters. The *Planck time*  $t_P$  (which is the time it takes for light to travel one Planck length) is about  $2^{-150}$  seconds. In the above setting, if a function  $F$  takes, say, 1KB of input (e.g., roughly  $10^4$  bits, which can encode a 100 by 100 bitmap image), and requires at least  $2^{0.8n} = 2^{0.8 \cdot 10^4}$  NAND lines to compute, then any physical system that computes it would require either volume of  $2^{0.2 \cdot 10^4}$  Planck length cubed, which is more than  $2^{1500}$  meters cubed or take at least  $2^{0.2 \cdot 10^4}$  Planck Time units, which is larger than  $2^{1500}$  seconds. To get a sense of how big that number is, note that the universe is only about  $2^{60}$  seconds old, and its observable radius is only roughly  $2^{90}$  meters. This suggests that it is possible to *empirically falsify* the PECTT by presenting a smaller-than-universe-size system that solves such a function.<sup>11</sup>

<sup>10</sup> 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.

<sup>11</sup> 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 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  $\omega(n)$  (let alone  $\Omega(2^n/n)$ ) lower bound on the number of lines that a NAND program needs to compute it.

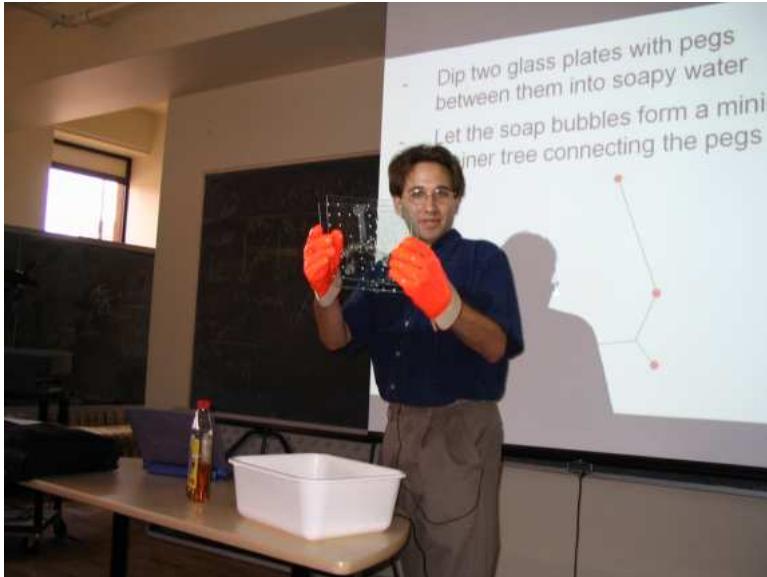
### 6.11.1 Attempts at refuting the PECTT

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  $x_1, \dots, x_n$ , we could cut  $n$  spaghetti noodles into lengths  $x_1, \dots, x_n$ , 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 PECTT 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  $(x_1, y_1), \dots, (x_m, y_m)$  (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 *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 *exponentially* in  $m$ , meaning that the PECTT would predict that if  $m$  is sufficiently large (such as 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  $(x_1, y_1), \dots, (x_m, y_m)$  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. 6.13](#)), 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.



**Figure 6.13:** Scott Aaronson tests a candidate device for computing Steiner trees using soap bubbles.

- **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 compute 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 milimiter, 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 possible.<sup>12</sup>
- **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 ma-

<sup>12</sup> We were extremely conservative in the suggested parameters for the PECTT, having assumed that as many as  $\ell_p^{-2} 10^{-6} \sim 10^{61}$  bits could potentially be stored in a millimeter radius region.

chines (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 *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 *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 *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.<sup>13</sup> 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

<sup>13</sup> 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.

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 [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.<sup>14</sup>

- **Quantum computation.** The most compelling attack on the Physical Extended Church Turing Thesis comes from the notion of *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. At the time of this writing, Scalable quantum computers have not yet been built, but it is a fascinating possibility, and one that does not seem to contradict any known law of nature. 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*).

<sup>14</sup> There are some well known scientists that have [advocated](#) that humans have inherent computational advantages over computers. See also [this](#).



**PECTT in practice** While even the precise phrasing of the PECTT, let alone understanding its correctness, is still a subject of research, some variant of it is already implicitly assumed in practice. A statement such as "this cryptosystem provides 128 bits of security" really means that (a) it is conjectured that there is no Boolean circuit (or, equivalently, a NAND gate) of size much smaller than  $2^{128}$  that can break the system,<sup>15</sup> and (b) we assume that no other physical mechanism can do better, and hence

it would take roughly a  $2^{128}$  amount of “resources” to break the system.

## 6.12 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*.

## 6.13 Exercises

**Exercise 6.1 — Relating NAND circuits and NAND programs.** Prove ??.

**Exercise 6.2 — Simulating all circuits with NAND programs.** Prove [Theorem 6.4](#)

**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 functions on two inputs, define  $\neg(x, y) = \bar{x}$ .)

2.  $B = \{\wedge, \vee\}$ .

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

4.  $B = \{LOOKUP_1, 0, 1\}$  where 0 and 1 are the constant functions as above and  $LOOKUP_1 : \{0, 1\}^3 \rightarrow \{0, 1\}$  satisfies  $LOOKUP_1(a, b, c)$  equals  $a$  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  $O(k)$  gates to compute the *NAND*

<sup>15</sup>We say “conjectured” and not “proved” because, while we can phrase such a statement as a precise mathematical conjecture, at the moment we are unable to *prove* such a statement for any cryptosystem. This is related to the P vs NP question we will discuss in future lectures

function (you can start by showing that there is a  $B$  circuit of at most  $O(k^{16})$  gates).<sup>16</sup>

**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.<sup>17</sup>

<sup>16</sup> Thanks to Alec Sun for solving this problem.

<sup>17</sup> TODO: check the right bound, and give it as a challenge program. Also say the conditions under which this can be improved to  $O(k)$  or  $\tilde{O}(k)$ .

### 6.14 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.

### 6.15 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.

### 6.16 Acknowledgements

### Learning Objectives:

- Learn the model of NAND++ program that involve loops.
- See some basic syntactic sugar for NAND++
- Get comfort with switching between representation of NAND++ programs as code and as tuples.
- Learn the notion of *configurations* for NAND++ programs.
- Understand the relation between NAND++ and NAND programs.

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<sup>1</sup>

"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

<sup>1</sup> Translation of "Sketch of the Analytical Engine" by L. F. Menabrea, Note G.

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 *parity* or XOR function  $\text{XOR} : \{0,1\}^* \rightarrow \{0,1\}$ , where  $\text{XOR}(x)$  equals 1 iff the number of 1's in  $x$  is odd. As simple as it is, the XOR function cannot be computed by a NAND program. Rather, for every  $n$ , we can compute  $\text{XOR}_n$  (the restriction of XOR to  $\{0,1\}^n$ ) using a different NAND program.

For example, here is the NAND program to compute  $XOR_5$ :

```

u := x_0 NAND x_1
v := x_0 NAND u
w := x_1 NAND u
s := v NAND w
u := s NAND x_2
v := s NAND u
w := x_2 NAND u
s := v NAND w
u := s NAND x_3
v := s NAND u
w := x_3 NAND u
s := v NAND w
u := s NAND x_4
v := s NAND u
w := x_4 NAND u
y_0 := v NAND w

```

This is rather repetitive, and more importantly, does not capture the fact that there is a *single* algorithm to compute the parity on all inputs. Typical programming language use the notion of *loops* to express such an algorithm, and so we might have wanted to use code such as:

```

# s is the "running parity", initialized to 0
while i < length(x):
    u := x_i NAND s
    v := s NAND u
    w := x_i NAND u
    s := v NAND w
    i++
    ns := s NAND s
    y_0 := ns NAND ns

```

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

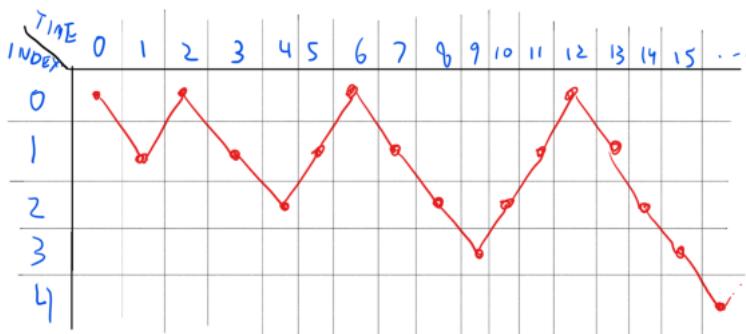
Keeping to our minimalist form, we will not add a `while` keyword to the NAND programming language. But we will extend this language in a way that allows for executing loops and accessing arrays of arbitrary length.

The main new ingredients are the following:

- We add a special variable `loop` with the following semantics: after executing the last line of the program, if `loop` is equal to one, then instead of halting, the program goes back to the first line. If `loop` is equal to zero after executing the last line then the program halts as is usual with NAND.<sup>2</sup>
- 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  $\langle i \rangle$  denotes the current value of the variable `i`). For example, if the current value of `i` is equal to 15, then `foo_i` corresponds to `foo_15`.<sup>3</sup> In the first loop of the program, `i` is assigned the value 0, but each time the program loops back to the first line, the value of `i` is updated in the following manner: in the  $k$ -th iteration the value of `i` equals  $I(k)$  where  $I = (I(0), I(1), I(2), \dots)$  is the following sequence (see Fig. 7.1):<sup>4</sup>

$$0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \dots \quad (7.1)$$

- 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. In particular, `validx_i` will equal to 1 if and only if the value of `i` is smaller than the length of the input.
- Like NAND programs, the output of a NAND++ program is the string  $y_0, \dots, y_{\langle k \rangle}$  where  $k$  is the largest integer such that  $y_{\langle k \rangle}$  was assigned a value.<sup>5</sup>



**Figure 7.1:** The value of `i` as a function of the current iteration. The variable `i` progresses according to the sequence  $0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \dots$ . At the  $k$ -th iteration the value of `i` equals  $k - r(r+1)$  if  $k \leq (r+1)^2$  and  $(r+1)(r+2) - k$  if  $k < (r+1)^2$  where  $r = \lfloor \sqrt{k+1/4} - 1/2 \rfloor$ .

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

<sup>2</sup> This corresponds to wrapping the entire program in one big loop that is executed at least once and continues as long as `loop` is equal to 1. For example, in the C programming language this would correspond with wrapping the entire program with the construct `do ... while (loop);`.

<sup>3</sup> Note that the variable `i`, like all variables in NAND, is a *global* variable, and hence all expressions of the form `foo_i, bar_i` etc. refer to the same value of `i`.

<sup>4</sup> TODO: Potentially change in the future to Salil's sequence  $INDEX(\ell) = min\ell - floor(sqrt(\ell))^2, ceiling(sqrt(\ell))^2 - \ell$  which has the form  $0, 0, 1, 1, 0, 1, 2, 2, 1, 0, 1, 2, 3, 3, 2, 1, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, \dots$

<sup>5</sup> To allow control of the output length, we also add a write-only array `invalidy`. If there exist  $j < k$  such that `invalidy_{<j>}=1` then we reduce the output length to  $j - 1$ . However, we will hardly use this array in this course, since we will almost always be interested in programs with a fixed output length (and in fact most often in programs with one bit of output).

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
tmp_1 := seen_i NAND seen_i
tmp_2 := x_i NAND tmp_1
val := tmp_2 NAND tmp_2

# Do s := s XOR val
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
```

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
```

### 7.1.1 Computing the index location

We say that a NAND program completed its *r-th round* when the index variable *i* completed the sequence:

$$0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \dots, 0, 1, \dots, r, r-1, \dots, 0 \quad (7.2)$$

This happens when the program completed

$$1 + 2 + 4 + 6 + \dots + 2r = r^2 + r + 1 \quad (7.3)$$

iterations of its main loop. (The last equality is obtained by applying the formula for the sum of an arithmetic progression.) This means that if we keep a “loop counter”  $k$  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  $r(r+1) \leq k$ , which (as you can verify) equals  $\lfloor \sqrt{k+1/4} - 1/2 \rfloor$ .

Thus the value of  $i$  in the  $k$ -th loop equals:

$$\text{index}(k) = \begin{cases} k - r(r+1) & k \leq (r+1)^2 \\ (r+1)(r+2) - k & \text{otherwise} \end{cases} \quad (7.4)$$

where  $r = \lfloor \sqrt{k+1/4} - 1/2 \rfloor$ . (We ask you to prove this in [Exercise 7.1](#).)

**R Variables as arrays** In NAND we allowed variables to have names such as `foo_17` but the numerical part of the identifier played essentially the same role as alphabetical part. In particular, NAND would be just as powerful if we didn’t allow any numbers in the variable identifiers. With the introduction of the special index variable `i`, in NAND++ things are different. It is best to think of each NAND++ variable `foo` as an *array*, with its  $j$ -th position corresponding to `foo_{\langle j \rangle}` (which in other programming languages would often be written as `foo[\langle j \rangle]`). Recall also our convention that a variable without an index such as `bar` is equivalent to `bar_0`, or the first position of the corresponding array. Of course we can think of variables as arrays in NAND as well, but since in NAND all indices are absolute numerical constants, this viewpoint does not make much of a difference as it does in NAND++.

### 7.1.2 Infinite loops and computing a function

One crucial difference between NAND and NAND++ programs is the following. Looking at a NAND program  $P$ , we can always tell how many inputs and how many outputs it has (by simply counting the number of `x_` and `y_` variables). Furthermore, we are guaranteed that if we invoke  $P$  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 follow-

ing NAND++ program would go into an infinite loop if the first bit of the input is zero:

```
loop := x_0 NAND x_0
```

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  $P(x) = \perp$ .

**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.

We say that a function  $F$  is *NAND++ computable* if there is a NAND++ program that computes it.

We will often drop the “NAND++” qualifier and simply call a function *computable* if it is NAND++ computable. This may seem “reckless” but, as we’ll see in future lectures, it turns out that being NAND++-computable is equivalent to being computable in essentially any reasonable model of computation.



**Notation** If  $F : \{0, 1\}^* \rightarrow \{0, 1\}$  is a Boolean function, then computing  $F$  is equivalent to deciding membership in the set  $L = \{x \in \{0, 1\}^* \mid F(x) = 1\}$ . Subsets of  $\{0, 1\}^*$  are known as *languages* in the literature. Such a language  $L \subseteq \{0, 1\}^*$  is known as *decidable* or *recursive* if the corresponding function  $F$  is computable.

## 7.2 A spoonful of sugar

Just like NAND, we can add a bit of “syntactic sugar” to NAND++ as well. These are constructs that can help us in expressing programs, though ultimately do not change the computational power of the model, since any program using these constructs can be “unsweetened” to obtain a program without them.

### 7.2.1 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 (cond) {
    LOOP_CODE
}
POSTLOOP_CODE
```

by

```
// startedloop is initialized to 0
// finishedloop is initialized to 0
if NOT(startedloop) {
    PRELOOP_CODE
    startedloop := 1
    temploop := loop
}
if NOT(finishedloop) {
    if (cond) {
        LOOP_CODE
        loop :=1
    }
    if NOT(cond) {
        finishedloop := 1
        loop := temploop
    }
}
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.

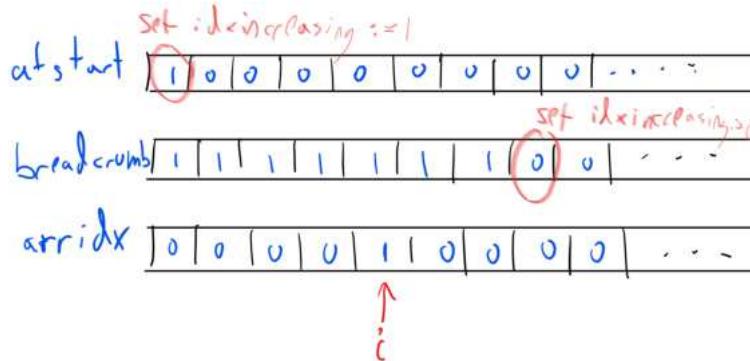


Please stop and verify that you understand why this transformation will result in a program that computes the same function as the original code with an inner loop.

### 7.2.2 Controlling the index variable

NAND++ is an *oblivious* programming model, in the sense that it gives us no means of controlling the index variable  $i$ . Rather to read, for example, the 1017-th index of the array `foo` (i.e., `foo_1017`) we need to wait until  $i$  will equal 1017.<sup>6</sup> However we can use syntactic sugar to simulate the effect of incrementing and decrementing  $i$ . That is, rather than having  $i$  move according to a fixed schedule, we can assume that we have the operation `i++ (foo)` that increments  $i$  if `foo` is equal to 1 (and otherwise leaves  $i$  in place), and similarly the operation `i- (bar)` that decrements  $i$  if `bar` is 1 and otherwise leaves  $i$  in place.

To achieve this, we start with the observation that in a NAND++ program we can know whether the index is increasing or decreasing. We achieve this using the Hansel and Gretel technique of leaving “breadcrumbs”. Specifically, we create an array `atstart` such that `atstart_0` equals 1 but `atstart_j` equals 0 for all  $j > 0$ , and an array `breadcrumb` where we set `breadcrumb_i` to 1 in every iteration. Then we can setup a variable `indexincreasing` and set it to 1 when we reach the zero index (i.e., when `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 `breadcrumb_i` is 0 and hence we have reached it for the first time). We can also maintain an array `arridx` that contains 0 in all positions except the current value of  $i$ .



**Figure 7.2:** We can simulate controlling the index variable  $i$  by keeping an array `atstart` letting us know when  $i$  reaches 0, and hence  $i$  starts increasing, and `breadcrumb` letting us know when we reach a point we haven't seen before, and hence  $i$  starts decreasing. If we are at a point in which the index is increasing but we want it to decrease then we mark our location on a special array `arridx` and enter a loop until the time we reach the same location again.

Now we can simulate incrementing and decrementing  $i$  by one by simply waiting until our desired outcome happens naturally. (This is similar to the observation that a bus is like a taxi if you're willing to wait long enough.) That is, if we want to increment  $i$  and

<sup>6</sup> Note that we *can* use variables with absolute numerical indices in the program, but they can only let us access a fixed number of locations (in particular smaller than the number of lines in the program). Since in NAND++ we typically think of inputs that are much longer than the number of lines, in general we will have to use the index variable  $i$  to access most of the memory locations.

`indexincreasing` equals 1 then we simply wait one step. Otherwise (if `indexincreasing` is 0) then we go into an inner loop in which we do nothing until we reach again the point when `arridx_i` is 1 and `indexincreasing` is equal to 1. Decrementing `i` is done in the analogous way.<sup>7</sup>

### 7.2.3 “Simple” NAND++ programs

When analyzing NAND++ programs, it will sometimes be convenient for us to restrict our attention to programs of a somewhat nicer form.

**Definition 7.2 — Simple NAND++ programs.** We say that a NAND++ program  $P$  is *simple* if it has the following properties:

- The only output variable it ever writes to is `y_0` (and so it computes a Boolean function).
- The last line of the program has the form `halted := loop NAND loop` and so the variable `halted` gets the value 1 when the program halts. Moreover, there is no other line in the program that writes to the variable `halted`.
- All lines that write to the variable `loop` or `y_0` are “guarded” by `halted` in the sense that we replace a line of the form `y_0 := foo NAND bar` with the (unsweetened equivalent to) `if NOT(halted) y_0 := foo NAND bar` and similarly `loop := blah NAND baz` is replaced with `if NOT(halted) loop := blah NAND baz`.
- It has an `indexincreasing` variable that is equal to 1 if and only if in the next iteration the value of `i` will increase by 1.
- It contains variables `zero` and `one` that are initialized to be 0 and 1 respectively, by having the first line be `one := zero NAND zero` and having no other lines that assign values to them.

Note that if  $P$  is a simple program then even if we continue its execution beyond the point it should have halted in, the value of the `y_0` and `loop` variables will not change. The following theorem shows that, in the context of Boolean functions, we can assume that every program is simple.<sup>8</sup>

**Theorem 7.1 — Simple program.** Let  $F : \{0, 1\}^* \rightarrow \{0, 1\}$  be a (possibly partial) Boolean function. If there is a NAND++ program that computes  $F$  then there is a simple NAND++ program  $P'$  that computes

<sup>7</sup> It can be verified that this transformation converts a program with  $T$  steps that used the `i++ (foo)` and `i- (bar)` operations into a program with  $O(T^2)$  that doesn’t use them.

<sup>8</sup> The restriction to Boolean functions is not very significant, as we can always encode a non Boolean function  $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$  by the Boolean function  $G(x, i) = F(x)_i$  where we treat the second input  $i$  as representing an integer. The crucial point is that we still allow the functions to have an unbounded *input length* and hence in particular these are functions that cannot be computed by plain “loop less” NAND programs.

$F$  as well.

*Proof.* We only sketch the proof, leaving verifying the full details to the reader. We prove the theorem by transforming the code of the program  $P$  to achieve a simple program  $P'$  without modifying the functionality of  $P$ . If  $P$  computes a Boolean function then it cannot write to any  $y_{\langle j \rangle}$  variable other than  $y_0$ . If  $P$  already used a variable named `halted` then we rename it. We then add the line `halted := loop NAND loop` to the end of the program, and replace all lines writing to the variables  $y_0$  and `loop` with their “guarded” equivalents. Finally, we ensure the existence of the variable `indexincreasing` using the “breadcrumbs” technique discussed above. ■

### 7.3 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.

### 7.3.1 Growing a NAND tree

If  $P$  is a NAND++ program and  $n, T \in \mathbb{N}$  are some numbers, then we can easily obtain a NAND program  $P' = \text{expand}_{T,n}(P)$  that, given any  $x \in \{0,1\}^n$ , runs  $T$  loop iterations of the program  $P$  and outputs the result. If  $P$  is a simple program, then we are guaranteed that, if  $P$  does not enter an infinite loop on  $x$ , then as long as we make  $T$  large enough,  $P'(x)$  will equal  $P(x)$ . To obtain the program  $P'$  we can simply place  $T$  copies of the program  $P$  one after the other, doing a “search and replace” in the  $k$ -th copy of any instances of  $\_i$  with the value  $\text{index}(k)$ , where the function  $\text{index}$  is defined as in Eq. (7.4). For example, Fig. 7.3 illustrates the expansion of the NAND++ program for parity.



**Figure 7.3:** The circuit corresponding to a NAND program for parity obtained by expanding the NAND++ program

We can also obtain such an expansion by using the `for ... do` .. syntactic sugar. For example, the NAND program below corresponds to running the parity program for 17 iterations, and computing  $\text{XOR}_5 : \{0,1\}^5 \rightarrow \{0,1\}$ . Its standard “unsweetened” version will have  $17 \cdot 10$  lines.<sup>9</sup>

```
for i in [0,1,0,1,2,1,0,1,2,3,2,1,0,1,2,3,4] do {
    tmp1 := seen_i NAND seen_i
    tmp2 := x_i NAND tmp1
    val := tmp2 NAND tmp2
```

<sup>9</sup> This is of course not the most efficient way to compute  $\text{XOR}_5$ . Generally, the NAND program to compute  $\text{XOR}_n$  obtained by expanding out the NAND++ program will require  $\Theta(n^2)$  lines, as opposed to the  $O(n)$  lines that is possible to achieve directly in NAND. However, in most cases this difference will not be so crucial for us.

```

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
}

```

In particular we have the following theorem

**Theorem 7.2 — Expansion of NAND++ to NAND.** For every simple NAND++ program  $P$  and function  $F : \{0,1\}^* \rightarrow \{0,1\}$ , if  $P$  computes  $F$  then for every  $n \in \mathbb{N}$  there exists  $T \in \mathbb{N}$  such that  $\text{expand}_{T,n}(P)$  computes  $F_n$ .

```

# Expand a NAND++ program and a given time bound T and n to
# an n-input T-line NAND program
def expand(P,T,n):
    result = ""

    for k in range(T):
        i=index(k)
        validx = ('one' if i<n else 'zero')
        result += P.replace('validx_i',validx).replace('x_i',
            'x_i' if i<n else 'zero')).replace('_i','_'+str(i))

    return result

def index(k):
    r = math.floor(math.sqrt(k+1/4)-1/2)
    return (k-r*(r+1) if k <= (r+1)*(r+1) else (r+1)*(r+2)-k)

```

*Proof.* We'll start with a “proof by code”. Above is a Python program `expand` to compute  $\text{expand}_{T,n}(P)$ . On input the code  $P$  of a NAND++ program and numbers  $T, n$ , `expand` outputs the code of the NAND program  $P'$  that works on length  $n$  inputs and is obtained by running  $T$  iterations of  $P$ :

If the original program had  $s$  lines, then for every  $\ell \in [sT]$ , line  $\ell$  in the output of `expand(P, T, n)` corresponds exactly to the line executed in step  $\ell$  of the execution  $P(x)$ .<sup>10</sup> Indeed, in step  $\ell$  of the execution of  $P(x)$ , the line executed is  $k = \ell \bmod s$ , and line  $\ell$  in the

<sup>10</sup> In the notation above (as elsewhere), we index both lines and steps from 0.

output of  $\text{expand}(P, T, n)$  is a copy of line  $k$  in  $P$ . If that line involved unindexed variables, then it is copied as is in the returned program result. Otherwise, if it involved the index  $_i$  then we replace  $i$  with the current value of  $i$ . Moreover, we replace the variable  $\text{validx}_i$  with either one or zero depending on whether  $i < n$ .

Now, if a simple NAND++ program  $P$  computes some function  $F : \{0,1\}^* \rightarrow \{0,1\}$ , then for every  $x \in \{0,1\}^*$  there is some number  $T_P(x)$  such that on input  $x$  halts within  $T(x)$  iterations of its main loop and outputs  $F(x)$ . Moreover, since  $P$  is simple, even if we run it for more iterations than that, the output value will not change. For every  $n \in \mathbb{N}$ , define  $T_P(n) = \max_{x \in \{0,1\}^n} T(x)$ . Then  $P' = \text{expand}_{T_P(n), n}(P)$  computes the function  $F_n : \{0,1\}^n \rightarrow \{0,1\}$  which is the restriction of  $F$  to  $\{0,1\}^n$ . ■

## 7.4 NAND++ Programs as tuples

Just like we did with NAND programs, we can represent NAND++ programs as tuples. A minor difference is that since in NAND++ it makes sense to keep track of indices, we will represent a variable  $\text{foo}_{\langle j \rangle}$  as a pair of numbers  $(a, j)$  where  $a$  corresponds to the identifier  $\text{foo}$ . Thus we will use a 6-tuple of the form  $(a, j, b, k, c, \ell)$  to represent each line of the form  $\text{foo}_{\langle j \rangle} := \text{bar}_{\langle k \rangle} \text{ NAND } \text{baz}_{\langle \ell \rangle}$ , where  $a, b, c$  correspond to the variable identifiers  $\text{foo}$ ,  $\text{bar}$  and  $\text{baz}$  respectively.<sup>11</sup> If one of the indices is the special variable  $i$  then we will use the number  $s$  for it where  $s$  is the number of lines (as no index is allowed to be this large in a NAND++ program). We can now define NAND++ programs in a way analogous to [Definition 3.1](#):

**Definition 7.3 — NAND++.** A NAND++ program is a 6-tuple  $P = (V, X, Y, \text{VALIDX}, \text{LOOP}, L)$  of the following form:

- $V$  (called the *variable identifiers*) is some finite set.
- $X \in V$  is called the *input identifier*.
- $Y \in V$  is called the *output identifier*.
- $\text{VALIDX} \in V$  is the *input length identifier*.
- $\text{LOOP} \in V$  is the *loop variable*.
- $L \in (V \times [s+1] \times V \times [s+1] \times V \times [s+1])^*$  is a list of 6-tuples of the form  $(a, j, b, k, c, \ell)$  where  $a, b, c \in V$  and  $j, k, \ell \in [s+1]$  for  $s = |L|$ . That is,  $L = ((a_0, j_0, b_0, k_0, c_0, \ell_0), \dots, (a_{s-1}, j_{s-1}, b_{s-1}, k_{s-1}, c_{s-1}, \ell_{s-1}))$  where for every  $t \in \{0, \dots, s-1\}$ ,  $a_t, b_t, c_t \in V$  and  $j_t, k_t, \ell_t \in$

<sup>11</sup> This difference between three tuples and six tuples is made for convenience and is not particularly important. We could have also represented NAND programs using six-tuples and NAND++ using three-tuples. Also recall that we use the convention that an unindexed variable identifier  $\text{foo}$  is equivalent to  $\text{foo}_0$ .

$[s + 1]$ . Moreover  $a_t \notin \{X, VALIDX\}$  for every  $t \in [s]$  and  $b_t, c_t \notin \{Y, LOOP\}$  for every  $t \in [s]$ .



This definition is long but ultimately translating a NAND++ program from code to tuples can be done in a fairly straightforward way. Please read the definition again to see that you can follow this transformation. Note that there is a difference between the way we represent NAND++ and NAND programs. In NAND programs, we used a different element of  $V$  to represent, for example,  $x_{17}$  and  $x_{35}$ . For NAND++ we will represent these two variables by  $(X, 17)$  and  $(X, 35)$  respectively where  $X$  is the input identifier. For this reason, in our definition of NAND++,  $X$  is a single element of  $V$  as opposed to a tuple of elements as in [Definition 3.1](#). For the same reason,  $Y$  is a single element and not a tuple as well.

Just as was the case for NAND programs, we can define a *canonical form* for NAND++ variables. Specifically in the canonical form we will use  $V = [t]$  for some  $t > 3$ ,  $X = 0, Y = 1, VALIDX = 2$  and  $LOOP = 3$ . Moreover, if  $P$  is *simple* in the sense of [Definition 7.2](#) then we will assume that the halted variable is encoded by 4, and the indexincreasing variable is encoded by 5. The canonical form representation of a NAND++ program is specified simply by a length  $s$  list of 6-tuples of natural numbers  $(a, j, b, k, c, \ell)$  where  $a, b, c \in [t]$  and  $j, k, \ell \in [s + 1]$ .

Here is a Python code to evaluate a NAND++ program given the list of 6-tuples representation:

```
# Evaluates a NAND++ program P on input x
# P is given in the list of tuples representation
# untested code
def EVALpp(P,x):
    vars = { 0:x , 2: [1]*len(x) } # vars[var][idx] is value
                                    # of var_idx.
    # special variables: 0:X, 1:Y, 2:VALIDX, 3:LOOP
    t = len(P)

    def index(k): # compute i at loop j
        r = math.floor(math.sqrt(k+1/4)-1/2)
        return (k-r*(r+1)) if k <= (r+1)*(r+1) else (r+1)*(r+2)
                  -k

    for i in range(1,t+1):
        if P[i][0] == 0:
            vars[0] = x
        elif P[i][0] == 1:
            vars[1] = x
        elif P[i][0] == 2:
            vars[2][P[i][1]] = 1
        elif P[i][0] == 3:
            vars[3] = 1
        elif P[i][0] == 4:
            vars[4] = 0
        elif P[i][0] == 5:
            vars[5] = 1
        else:
            print("Unknown variable")
            break
        if P[i][1] == 0:
            x = vars[0]
        elif P[i][1] == 1:
            x = vars[1]
        elif P[i][1] == 2:
            x = vars[2][index(P[i][2])]
        elif P[i][1] == 3:
            x = vars[3]
        elif P[i][1] == 4:
            x = vars[4]
        elif P[i][1] == 5:
            x = vars[5]
        else:
            print("Unknown variable")
            break
    return x
```

```

def getval(var,idx): # returns current value of var_idx
    if idx== t: idx = index(k)
    l = vars.setdefault(var,[])
    return l[idx] if idx<len(l) else 0

def setval(var,idx,v): # sets var_idx := v
    l = vars.setdefault(var,[])
    l.append([0]*(1+idx-len(l)))
    l[idx]=v
    vars[var] = l

k = 0
while True:
    for t in P:
        setval(t[0],t[1], 1-getval(t[2],t[3])*getval(t[4],t
            [5]))
    if not getval(3,0): break
    k += 1

return vars[1]

```

#### 7.4.1 Configurations

Just like we did for NAND programs, we can define the notion of a *configuration* and a *next step function* for NAND++ programs. That is, a configuration of a program  $P$  records all the state of  $P$  at a given point in the execution, and contains everything we need to know in order to continue from this state. The next step function of  $P$  maps a configuration of  $P$  into the configuration that occurs after executing one more line of  $P$ .

 Before reading onwards, try to think how *you* would define the notion of a configuration of a NAND++ program.

While we can define configurations in full generality, for concreteness we will restrict our attention to configurations of “simple” programs NAND++ programs in the sense of [Definition 7.2](#), that are given in a canonical form. Let  $P$  be a canonical form simple program, represented as a list of 6 tuples  $L = ((a_0, j_0, b_0, k_0, c_0, \ell_0), \dots, (a_{s-1}, j_{s-1}, b_{s-1}, k_{s-1}, c_{s-1}, \ell_{s-1}))$ . Let  $s$  be the number of lines and  $t$  be one more than the largest number appearing among the  $a$ 's,  $b$ 's or  $c$ 's.

Just like we did for NAND, a *configuration* of the program  $P$  will denote the current line being executed and the current value of all variables. For our convenience we will use a somewhat different encoding than we did for NAND. We will encode the configuration as a string  $\sigma \in \{0,1\}^*$ , which is composed of *blocks*, that is,  $\sigma$  will be the concatenation of  $\sigma^0, \dots, \sigma^{r-1}$  for some  $r \in \mathbb{N}$  (that will represent the maximum among  $n - 1$ , where  $n$  is the input length, the largest numerical index appearing in the program, and the largest index that the program has ever reached in the execution). Each block  $\sigma^i$  will be a string of length  $B$  (for some constant  $B$  depending on  $t, s$ ) that encodes the following:

- The values of variables indexed by  $i$  (e.g., `foo_<i>`, `bar_<i>`, etc.).
- Whether or not the block is “active” (i.e., whether the current value of the index variable `i` is  $i$ ), and in the latter case, the current line that is being executed.
- Whether this is the first or last block.



**High level points about configurations** For the sake of completeness, we will describe below precisely how configurations of NAND++ programs and the next-step function are defined. However, the details are as important as the high level points, which are the following: A configuration encodes all the information of the state of the program at a given step in the computation, including the values of all variables (both the Boolean variables and the special index variable `i`) and the current line number that is to be executed. The next step function of a program  $P$  updates that configuration by computing one line of the program, and updating the value of the variable that is assigned a value in this program. The variables involved in that line either have absolute numerical indices (in which case they are encoded in one of the first  $s$  blocks, as numerical indices can't be larger than the number of lines) or are indexed by the special variable `i` (in which case they are encoded in the active block). If the line is the last one in the program, the next step function also determines whether to halt based on the `loop` variable, and updates the active block based on whether the index will be increasing or decreasing.

We now describe a precise encoding for the configurations of a NAND++ program. Many of the choices below are made for convenience and other choices would be just as valid. We will think of encoding each block as using the alphabet  $\Sigma = \{\text{BB}, \text{EB}, 0, 1\}$ . (BB and EB stand for “begin block” and “end block” respectively; we can later

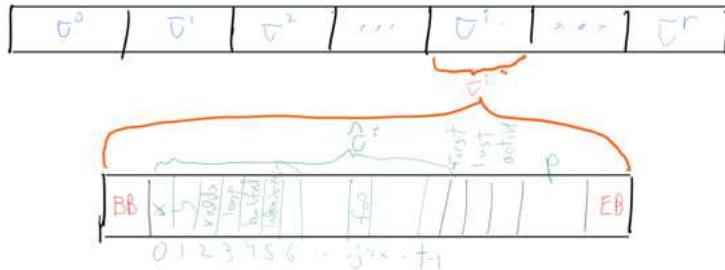
encode this as a binary string using the map  $0 \mapsto 00, 1 \mapsto 11, \text{BB} \mapsto 01, \text{EB} \mapsto 10$ .) In this alphabet  $\Sigma$ , every block  $\sigma^i$  will have the form

$$\sigma^i = \text{BB } \hat{\sigma}^i \text{ first last active } p \text{ EB} \quad (7.5)$$

where  $\hat{\sigma}^i$  is a string in  $\{0, 1\}^t$  that encodes the values of all the variables in the program indexed by  $i$ . That is, the  $a$ -th coordinate of  $\hat{\sigma}^i$  corresponds to the value of the variable represented by  $(a, i)$ . For example, if we encode `foo` by the number 11 then  $\hat{\sigma}_{11}^{17}$  corresponds to the value of `foo_17` at the given point in the execution. We use the same indexing of variables as in representations and so in particular coordinates 0, 1, 2, 3, 4, 5 of  $\hat{\sigma}^i$  correspond to the variables `x_i`, `y_i`, `validx_i`, `loop_i`, `halted_i`, `indexincreasing_i` respectively.<sup>12</sup>

The values *active*, *first*, and *last* are each bits that are set to 1 or 0 depending on whether the current block is *active* (i.e. the current value of *i* is  $i$ ), is the *first* block in the configuration and the *last* block, respectively. The parameter  $p$  is a string in  $\{0, 1\}^{\lceil \log(s+1) \rceil}$ , which (via the binary representation) we think of also as number in  $[s+1]$ . The value of  $p$  is equal to the current line that is about to be executed if the block is active, and to 0 if the block is not active. If  $p = s$  then this means that we have halted.

Note that in the alphabet  $\Sigma$ , our encoding takes 2 symbols for BB and EB,  $t$  symbols for  $\hat{\sigma}^i$ , three symbols for *first*, *last*, *active*, and  $\log[s+1]$  symbols for encoding  $p$ . Hence in the binary alphabet, each block  $\sigma^i$  will be encoded as a string of length  $B = 2(5 + t + \log([s+1]))$  bits, and a configuration will be encoded as a binary string of length  $(r+1)B$  where  $r$  is the largest index that the variable *i* has reached so far in the execution. See Fig. 7.4 for an illustration of the configuration.



**Figure 7.4:** A configuration of an  $s$ -line  $t$ -variable simple NAND++ program can be encoded as a string in  $\{0, 1\}^{rB}$ , the  $i$ -th block encodes the value of all variables of the form `foo_(i)`, as well as whether the block is first, last or active in the sense that  $i=i$  and in the latter case, also the index of the current line being executed.

<sup>12</sup> Recall that we identify an unindexed variable identifier such as `foo` with `foo_0`, and so in particular the values of `loop`, `halted` and `indexincreasing` are encoded in the block  $\sigma^0$ .

For a simple  $s$ -line  $t$ -variable NAND++ program  $P$  the *next configuration function*  $\text{NEXT}_P : \{0,1\}^* \rightarrow \{0,1\}^*$  is defined in the natural way.<sup>13</sup> That is, on input a configuration  $\sigma$ , one can compute  $\sigma' = \text{NEXT}_P(\sigma)$  as follows:

1. Scan the configuration  $\sigma$  to find the index  $i$  of the active block (block where the *active* bit is set to 1) and the current line  $p$  that needs to be executed (which is enc). We denote the new active block and current line in the configuration  $\sigma'$  by  $(i', p')$ .
2. If  $p = s$  then this  $\sigma$  a halting configuration and  $\text{NEXT}_P(\sigma) = \sigma$ . Otherwise we continue to the following steps:
3. Execute the line  $p$ : if the  $p$ -th tuple in the program is  $(a, j, b, k, c, \ell)$  then we update  $\sigma$  to  $\sigma'$  based on the value of this program. That is, in the configuration  $\sigma'$ , we encode the value of the variable corresponding to  $(a, j)$  as the NAND of the values of variables corresponding to  $(b, k)$  and  $(c, \ell)$ .<sup>14</sup>
4. Updating the value of  $i$ : if  $p = s - 1$  (i.e.,  $p$  corresponds to the last line of the program), then we check whether the value of the `loop` or `loop_0` variable (which by our convention is encoded as the variable with index 3 in the first block) and if so set in  $\sigma'$  the value  $p' = s$  which corresponds to a halting configuration. Otherwise,  $i$  is either incremented and decremented based on `indexincreasing` (which we can read from the first block). That is, we let  $i'$  be either  $i + 1$  and  $i - 1$  based on `indexincreasing` and modify the active block in  $\sigma'$  to be  $i'$ . (If  $i$  is the final block and  $i' = i + 1$  then we create a new block and mark it to be the last one.)
5. We update  $p' = p + 1 \bmod s$ , and encode  $p'$  in the active block of  $\sigma'$ .

One important property of  $\text{NEXT}_P$  is that to compute it we only need to access the blocks  $0, \dots, s - 1$  (since the largest absolute numerical index in the program is at most  $s - 1$ ) as well as the current active block and its immediate neighbors. Thus in each step,  $\text{NEXT}_P$  only reads or modifies a constant number of blocks.

Here is some Python code for the next step function:

```
# compute the next-step configuration
# Inputs:
# P: NAND++ program in list of 6-tuples representation (
#     assuming it has an "indexincreasing" variable)
# conf: encoding of configuration as a string using the
#       alphabet "B","E","0","1".
def next_step(P,conf):
```

<sup>13</sup> We define  $\text{NEXT}_P$  as a *partial* function, that is only defined on strings that are valid encoding of a configuration, and in particular have only a single block with its active bit set, and where the initial and final bits are also only set for the first and last block respectively. It is of course possible to extend  $\text{NEXT}_P$  to be a total function by defining it on invalid configurations in some way.

<sup>14</sup> Recall that according to the way we represent NAND++ programs as 6-tuples, if  $a$  is the number corresponding to the identifier `foo` then  $(a, j)$  corresponds to `foo_(j)` if  $j < s$ , and corresponds to `foo_(i)` if  $j = s$  where  $i$  is the current value of the index variable `i`.

```

s = len(P) # numer of lines
t = max([max(tup[0],tup[2],tup[4]) for tup in P])+1 #
    number of variables
line_enc_length = math.ceil(math.log(s+1,2)) # num of
    bits to encode a line
block_enc_length = t+3+line_enc_length # num of bits to
    encode a block (without bookends of "E","B")
LOOP = 3
INDEXINCREASING = 5
ACTIVEIDX = block_enc_length -line_enc_length-1 #
    position of active flag
FINALIDX = block_enc_length -line_enc_length-2 # position
    of final flag

def getval(var,idx):
    if idx<s: return int(blocks[idx][var])
    return int(active[var])

def setval(var,idx,v):
    nonlocal blocks, i
    if idx<s: blocks[idx][var]=str(v)
    blocks[i][var]=str(v)

blocks = [list(b[1:]) for b in conf.split("E")[:-1]] #
    list of blocks w/o initial "B" and final "E"

i = [j for j in range(len(blocks)) if blocks[j][ACTIVEIDX]
    ]=="1" ][0]
active = blocks[i]

p = int("".join(active[-line_enc_length:]),2) # current
    line to be executed

if p==s: return conf # halting configuration

(a,j,b,k,c,l) = P[p] # 6-tuple corresponding to current
    line# 6-tuple corresponding to current line
setval(a,j,1-getval(b,k)*getval(c,l))

new_p = p+1
new_i = i
if p==s-1: # last line
    new_p = (s if getval(LOOP,0)==0 else 0)
    new_i = (i+1 if getval(INDEXINCREASING,0) else i-1)

```

```

if new_i==len(blocks): # need to add another block and
    make it final
    blocks[len(blocks)-1][FINALIDX]="0"
    new_final = ["0"]*block_enc_length
    new_final[FINALIDX]="1"
    blocks.append(new_final)

blocks[i][ACTIVEIDX]=="0" # turn off "active" flag in
    old active block
blocks[i][ACTIVEIDX+1:ACTIVEIDX+1+line_enc_length]=[ "0
    "]*line_enc_length # zero out line counter in old
    active block
blocks[new_i][ACTIVEIDX]=="1" # turn on "active" flag
    in new active block
new_p_s = bin(new_p)[2:]
new_p_s = "0"*(line_enc_length-len(new_p_s))+new_p_s
blocks[new_i][ACTIVEIDX+1:ACTIVEIDX+1+line_enc_length] =
    list(new_p_s) # add binary representation of next line
    in new active block

return "".join(["B"+"".join(block)+"E" for block in
    blocks]) # return new configuration

```

#### 7.4.2 Deltas

Sometimes it is easier to keep track of merely the *changes* (sometimes known as “deltas”) in the state of a NAND++ program, rather than the full configuration. Since every step of a NAND++ program assigns a value to a single variable, this motivates the following definition:

**Definition 7.4 — Modification logs of NAND++ program.** The *modification log* (or “deltas”) of an  $s$ -line simple NAND++ program  $P$  on an input  $x \in \{0,1\}^n$  is the string  $\Delta$  of length  $sT + n$  whose first  $n$  bits are equal to  $x$  and the last  $sT$  bits correspond to the value assigned in each step of the program. That is, for every  $i \in [n]$ ,  $\Delta_i = x_i$  and for every  $\ell \in [sT]$ ,  $\Delta_{\ell+n}$  equals to the value that is assigned by the line executed in step  $\ell$  of the execution of  $P$  on input  $x$ , where  $T$  is the number of iterations of the loop that  $P$  does on input  $x$ .

If  $\Delta$  is the “deltas” of  $P$  on input  $x \in \{0,1\}^n$ , then for every  $\ell \in [Ts]$ ,  $\Delta_\ell$  is the same as the value assigned by line  $\ell$  of the NAND

program  $expand_{T',n}(P)$  where  $s$  is the number of lines in  $P$ , and for every  $T'$  which is at least the number of loop iterations that  $P$  takes on input  $x$ .

**R****Snapshots and deltas: what you need to remember**

The details of the definitions of configuration and deltas are not as important as the main points which are:

- \* A *configuration* is the full state of the program at a certain point in the computation. Applying the  $NEXT_P$  function to the current configuration yields the next configuration.
- \* Each configuration can be thought of as a string which is a sequence of constant-size *blocks*. The  $NEXT_P$  function only depends and modifies a constant number of blocks: the  $t$  first ones, the current active block, and its two adjacent neighbors.
- \* The “delta” or “modification log” of computation is a succinct description of how the configuration changed in each step of the computation. It is simply the string  $\Delta$  of length  $T$  such that for every  $\ell \in T$ ,  $\Delta_\ell$  denotes the value assigned in the  $\ell$ -th step of the computation.

Both configurations and Deltas are technical ways to capture the fact that computation is a complex process that is obtained as the result of a long sequence of simple steps.

## 7.5 Lecture summary

- NAND++ programs introduce the notion of *loops*, and allow us to capture a single algorithm that can evaluate functions of any input length.
- Running a NAND++ program for any finite number of steps corresponds to a NAND program. However, the key feature of NAND++ is that the number of iterations can depend on the input, rather than being a fixed upper bound in advance.
- A *configuration* of a NAND++ program encodes the state of the program at a given point in the computation. The *next step function* of the program maps the current configuration to the next one.

## 7.6 Exercises

**Exercise 7.1 — Compute index.** Suppose that  $t$  is the “iteration counter” of a NAND++ program, in the sense that  $t$  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  $t - r(r + 1)$  if  $t \leq (r + 1)^2$  and equals  $(r + 2)(r + 1) - t$  otherwise, where  $r = \lfloor \sqrt{t + 1/4} - 1/2 \rfloor$ . ■

## 7.7 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.

## 7.8 Further explorations

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

## 7.9 Acknowledgements

**Learning Objectives:**

- See the NAND« programming language.
- Understand how NAND« can be implemented as syntactic sugar on top of NAND++
- Understand the construction of a *universal* NAND« (and hence NAND++) program.

## 8

## *Indirection and universality*

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

"The programmer is in the unique position that ... he has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.", Edsger Dijkstra, "On the cruelty of really teaching computing science", 1988.

One of the most significant results we showed for NAND programs is the notion of *universality*: that a NAND program can evaluate other NAND programs. However, there was a significant caveat in this notion. To evaluate a NAND program of  $s$  lines, we needed to use a bigger number of lines than  $s$ . It turns out that NAND++ allows us to "break out of this cycle" and obtain a truly *universal* NAND++ program  $U$  that can evaluate all other programs, including programs that have more lines than  $U$  itself. (As we'll see in the next lecture, this is not something special to NAND++ but is a feature of many other computational models.) The existence of such a universal program has far reaching applications, and we will explore them in the rest of this course.

To describe the universal program, it will be convenient for us to introduce some extra "syntactic sugar" for NAND++. We'll use the name NAND« for the language of NAND++ with this extra syntactic sugar. The classes of functions computable by NAND++ and NAND« programs are identical, but NAND« can sometimes be more convenient to work with. Moreover, NAND« will be useful for

us later in the course when we will turn to modelling *running time* of algorithms.<sup>1</sup>

### 8.1 The NAND« programming language

We now define a seemingly more powerful programming language than NAND++: NAND« (pronounced “NAND shift”). 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.<sup>2</sup> 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:<sup>3</sup>

- *foo := bar* or *i := bar* (assignment)
- *foo := bar + baz* (addition)
- *foo := bar - baz* (subtraction)
- *foo := bar » baz* (right shift:  $foo \leftarrow \lfloor \frac{bar}{2^{baz}} \rfloor$ )
- *foo := bar << baz* (left shift:  $foo \leftarrow bar \cdot 2^{baz}$ )
- *foo := bar % baz* (modular reduction)
- *foo := bar \* baz* (multiplication)
- *foo := bar / baz* (integer division:  $foo \leftarrow \lfloor \frac{bar}{baz} \rfloor$ )
- *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)

The semantics of these operations are as expected except that we maintain the invariant that all variables always take values between

<sup>1</sup> Looking ahead, as we will see in the next lecture, NAND++ programs are essentially equivalent to *Turing Machines* (more precisely, their single-tape, oblivious variant), while NAND« programs are equivalent to *RAM machines*. Turing machines are typically the standard model used in computability and complexity theory, while RAM machines are used in algorithm design. As we will see, their powers are equivalent up to polynomial factors in the running time.

<sup>2</sup> 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, the NAND++/Turing-Machine model is indistinguishable from the NAND«/RAM-Machine model (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.

<sup>3</sup> Below *foo*, *bar* and *baz* are indexed or non-indexed variable identifiers (e.g., they can have the form *blah* or *blah\_12* or *blah\_i*), as usual, we identify an indexed identifier *blah* with *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.

0 and the current value of the iteration 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 iteration counter, then we assign the value of the iteration 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.

Apart from those operations, `NAND«` is identical to `NAND++`. For consistency, we still treat the variable `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 `foo_bar` though we can simulate it by first writing `i := bar` and then `foo_i`. We also maintain the invariant that at the beginning of each iteration, the value of `i` is set to the same value that it would have in a `NAND++` program (i.e., the function of the iteration counter stated in [Exercise 7.1](#)), though this can be of course overwritten by explicitly assigning a value to `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 `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«` programs that compute on integers.

### 8.1.1 Simulating `NAND«` in `NAND++`

The most important fact we need to know about `NAND«` 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 8.1 — `NAND++` and `NAND«` are equivalent.** For every (partial) function  $F : \{0,1\}^* \rightarrow \{0,1\}^*$ ,  $F$  is computable by a `NAND++` program if and only if  $F$  is computable by a `NAND«` program.

The rest of this section is devoted to outlining the proof of [Theo-](#)

**rem 8.1.** The “only if” direction of the theorem is immediate. After all, every NAND++ program  $P$  is in particular also a NAND« program, and hence if  $F$  is computable by a NAND++ program then it is also computable by a NAND« program. To show the “if” direction, we need to show how we can implement all the operations of NAND« in NAND++. In other words, we need to give a “NAND« to NAND++ compiler”.

Writing a compiler in full detail, and then proving that it is correct, is possible (and **has been done**) but is quite a time consuming enterprise, and not very illuminating. For our purposes, we need to convince ourselves that **Theorem 8.1** and that such a transformation exists, and we will do so by outlining the key ideas behind it.<sup>4</sup>

Let  $P$  be a NAND« program, we need to transform  $P$  into a NAND++ program  $P'$  that computes the same function as  $P$ . The idea will be that  $P'$  will simulate  $P$  “in its belly”. We will use the variables of  $P'$  to encode the state of the simulated program  $P$ , and every single NAND« step of the program  $P$  will be translated into several NAND++ steps by  $P'$ . We will do so in several steps:

**Step 1: Controlling the index and inner loops.** We have seen that we can add syntactic sugar for inner loops and incrementing/dereferencing the index (i.e., operations such as `i++ (foo)` and `i- (bar)`) to NAND++. Hence we can assume access to these operations in constructing  $P'$ . In particular, we can use such an inner loop to perform tasks such as copying the contents of one array (e.g., variables `foo_0`, ..., `foo_{k-1}` for some  $k$ ) to another.

**Step 2: Operations on integers.** We can use some standard prefix free encoding to represent an integer as an array of bits. For example, we can use the map  $pf : \mathbb{Z} \rightarrow \{0,1\}^*$  defined as follows. Given an integer  $z \in \mathbb{Z}$ , if  $z \geq 0$  then we define the string  $pf(z)$  as  $z_0z_1z_2\dots z_{n-1}z_n01$  (where  $n$  is the smallest number s.t.  $2^n > z$  and  $z_i$  is the binary digit of  $x$  corresponding to  $2^i$ ). If  $z < 0$  then we define  $pf(z) = 10pf(|z|)$ . We can then use the standard gradeschool algorithms to define NAND++ macros that perform arithmetic operations on the representation of integers (e.g., addition, multiplication, division, etc.).

**Step 3: Move index to specified location.** We can use the above operations to move the index to a location encoded by an integer. To do so, we can first move `i` to the zero location by decreasing it until the `atstart_i` equals 1 (where, as we've seen before, `atstart` is an array we can set up so `atstart_0` is 1 and `atstart_{j-1}` is zero for  $j \neq 0$ ). Now, if the variables `foo_0, foo_1, ...` encode the number

<sup>4</sup> The webpage [nandpl.org](http://nandpl.org) should eventually contain a program that transforms a NAND« program into an equivalent NAND++ program.

$k \in \mathbb{N}$  we can set the value of `i` to  $k$  as follows. We'll set `bar` to 1 and an inner loop that will proceed as long as `bar` is not zero. In this loop we will do the following:

- (1) If `foo` encodes 0 then set `bar` to zero.

- (2) Otherwise, we use a nested inner loop to decrement the number represented by `foo` by 1, and perform the operation `i++ (bar)`.

**Step 4: Maintaining an iteration counter and index.** The NAND++ program  $P'$  will simulate execution of the NAND« program  $P$ . Every step of  $P$  will be simulated by several steps of  $P'$ . We can use the above operations to maintain a variable `itercounter` and `index` that will encode the current step of  $P$  that is being executed and the current value of the special index variable `i` in the simulated program  $P$  (which does not have to be the same as the value of `i` in the NAND++ program  $P'$ ).

**Step 5: Embedding two dimensional arrays into one dimension.** If `foo` and `bar` encode the natural numbers  $x, y \in \mathbb{N}$ , then we can use NAND++ to compute the map  $PAIR : \mathbb{N}^2 \rightarrow \mathbb{N}$  where  $PAIR(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x$ . In [Exercise 8.1](#) we ask you to verify that  $PAIR$  is a one-to-one map from  $\mathbb{N}^2$  to  $\mathbb{N}$  and that there are NAND++ programs  $P_0, P_1$  such that for every  $x_0, x_1 \in \mathbb{N}$  and  $i \in \{0, 1\}$ ,  $P_i(PAIR(x_0, x_1)) = x_i$ . Using this  $PAIR$  map, we can assume we have access to two dimensional arrays in our NAND++ program.

**Step 6: Embedding an array of integers into a two dimensional bit array.** We can use the same encoding as above to embed a one-dimensional array `foo` of integers into a two-dimensional array `bar` of bits, where `bar_{<i>, <j>}` will encode the  $j$ -th bit in the representation of the integer `foo_{<i>}`. Thus we can simulate the integer arrays of the NAND« program  $P$  in the NAND++ program  $P'$ .

**Step 7: Simulating  $P$ .** Now we have all the components in place to simulate every operation of  $P$  in  $P'$ . The program  $P'$  will have a two dimensional bit array corresponding to any one dimensional array of  $P$ , as well as variables to store the iteration counter, index, as well as the `loop` variable of the simulated program  $P$ . Every step of  $P$  can now be translated into an inner loop that would perform the same operation on the representations of the state.

We omit the full details of all the steps above and their analysis, which are tedious but not that insightful.

### 8.1.2 Example

Here is a program that computes the function *PALINDROME* :

$\{0,1\}^* \rightarrow \{0,1\}$  that outputs 1 on  $x$  if and only if  $x_i = x_{|x|-i}$  for every  $i \in \{0, \dots, |x| - 1\}$ . This program uses NAND $\ll$  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 := 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
}

```

## 8.2 The “Best of both worlds” paradigm

The equivalence between NAND++ and NAND« allows us to choose the most convenient language for the task at hand:

- When we want to give a theorem about all programs, we can use NAND++ because it is simpler and easier to analyze. In particular, if we want to show that a certain function *can not* be computed, then we will use NAND++.
- When we want to show the existence of a program computing a certain function, we can use NAND«, because it is higher level and easier to program in. In particular, if we want to show that a function *can* be computed then we can use NAND«. In fact, because NAND« has much of the features of high level programming languages, we will often describe NAND« programs in an informal manner, trusting that the reader can fill in the details and translate the high level description to the precise program. (This is just like the way people typically use informal or “pseudocode” descriptions of algorithms, trusting that their audience will know to translate these descriptions to code if needed.)

Our usage of NAND++ and NAND« is very similar to the way people use in practice high and low level programming languages. When one wants to produce a device that executes programs, it is convenient to do so for very simple and “low level” programming language. When one wants to describe an algorithm, it is convenient to use as high level a formalism as possible.



**Recursion in NAND«** One high level tool we can use in describing NAND« programs is *recursion*. We can use the standard implementation of the *stack* data structure. That is, we let *stack* to be an array of integers  $\text{stack}_0, \dots, \text{stack}_{\langle k - 1 \rangle}$  and *stackpointer* will be the number  $k$  of items in the stack. We implement *push(foo)* by doing  $i := \text{stackpointer}$  and  $\text{stack}_i := \text{foo}$  and *pop()* by letting  $\text{stackpointer} := \text{stackpointer} - 1$ . By encoding strings as integers, we can have allow strings in our stack as well. Now we can implement



**Figure 8.1:** By having the two equivalent languages NAND++ and NAND«, we can “have our cake and eat it too”, using NAND++ when we want to prove that programs *can't* do something, and using NAND« or other high level languages when we want to prove that programs *can* do something.

recursion using the stack just as is done in most programming languages. First of all, we note that using loops and conditionals, we can implement “*goto*” statements in NAND«. Moreover, we can even implement “dynamic *gotos*”, in the sense that we can set integer labels for certain lines of codes, and have a *goto foo* operation that moves execution to the line labeled by *foo*. Now, if we want to make a call to a function *F* with parameter *bar* then we will push into the stack the label of the next line and *bar*, and then make a *goto* to the code of *F*. That code will pop its parameter from the stack, do the computation of *F*, and when it needs to resume execution, will pop the label from the stack and *goto* there.

### 8.2.1 Let's talk about abstractions.

At some point in any theory of computation course, the instructor and students need to have *the talk*. That is, we need to discuss the *level of abstraction* in describing algorithms. In algorithms courses, one typically describes algorithms in English, assuming readers can “fill in the details” and would be able to convert such an algorithm

into an implementation if needed. For example, we might describe the **breadth first search** algorithm to find if two vertices  $u, v$  are connected as follows:

1. Put  $u$  in queue  $Q$ .
2. While  $Q$  is not empty:
  - Remove the top vertex  $w$  from  $Q$
  - If  $w = v$  then declare “connected” and exit.
  - Mark  $w$  and add all unmarked neighbors of  $w$  to  $Q$ .
1. Declare “unconnected”.

We call such a description a *high level description*.

If we wanted to give more details on how to implement breadth first search in a programming language such as Python or C (or NAND« / NAND++ for that matter), we would describe how we implement the queue data structure using an array, and similarly how we would use arrays to implement the marking. We call such a “intermediate level” description an *implementation level* or *pseudocode* description. Finally, if we want to describe the implementation precisely, we would give the full code of the program (or another fully precise representation, such as in the form of a list of tuples).

We call this a *formal* or *low level* description.

While initially we might have described NAND, NAND++, and NAND« programs at the full formal level (and the [NAND website](#) contains more such examples), as the course continues we will move to implementation and high level description. After all, our focus is typically not to use these models for actual computation, but rather to analyze the general phenomenon of computation. That said, if you don’t understand how the high level description translates to an actual implementation, you should always feel welcome to ask for more details of your teachers and teaching fellows.

A similar distinction applies to the notion of *representation* of objects as strings. Sometimes, to be precise, we give a *low level specification* of exactly how an object maps into a binary string. For example, we might describe an encoding of  $n$  vertex graphs as length  $n^2$  binary strings, by saying that we map a graph  $G$  over the vertex  $[n]$  to a string  $x \in \{0, 1\}^{n^2}$  such that the  $n \cdot i + j$ -th coordinate of  $x$  is 1 if and only if the edge  $\overrightarrow{i j}$  is present in  $G$ . We can also use an *intermediate* or *implementation level* description, by simply saying that we represent a graph using the adjacency matrix representation. Finally, because we translating between the various representations of graphs (and

objects in general) can be done via a NAND« (and hence a NAND++) program, when talking in a high level we also suppress discussion of representation altogether. For example, the fact that graph connectivity is a computable function is true regardless of whether we represent graphs as adjacency lists, adjacency matrices, list of edge-pairs, and so on and so forth. Hence, in cases where the precise representation doesn't make a difference, we would often talk about our algorithms as taking as input an object  $O$  (that can be a graph, a vector, a program, etc.) without specifying how  $O$  is encoded as a string.

### 8.3 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 8.2 — Universality of NAND++.** There is a NAND++ program  $U$  that computes the partial function  $EVAL : \{0,1\}^* \rightarrow \{0,1\}^*$  defined as follows:

$$EVAL(P, x) = P(x) \quad (8.1)$$

for strings  $P, x$  such that  $P$  is a valid representation of a NAND++ program which produces an output on  $x$ . Moreover, for every input  $x \in \{0,1\}^*$  on which  $P$  does not halt,  $U(P, x)$  does not halt as well.

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« program into a NAND++ program, this means that even the seemingly “weaker” NAND++ programming language is powerful enough to simulate NAND« 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.

### 8.3.1 Representing NAND++ programs as strings

Before we can prove [Theorem 8.2](#), 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 triples” representation for NAND programs. We will assume that all variables are of the form `foo##` where `foo` is an identifier and `##` 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

`foo_<j> := bar_<k> NAND baz_<l>`

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. For variables that indexed by the special index `i`, we will encode the index by `s`, where `s` is the number of lines in the program. (There is no risk of conflict since we did not allow numerical indices larger or equal to the number of lines in the program.) We will set the identifiers of `x`, `y`, `validx` and `loop` to 0, 1, 2, 3 respectively. Therefore the representation of the parity program

```
tmp_1 := seen_i NAND seen_i
tmp_2 := x_i NAND tmp_1
val := tmp_2 NAND tmp_2
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 := z NAND z
stop := validx_i NAND validx_i
loop := stop NAND stop
```

will be

```
[[4, 1, 5, 11, 5, 11],
 [4, 2, 0, 11, 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, 11, 12, 0, 12, 0],
[13, 0, 2, 11, 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  $s$  (where  $s$  is the number of lines), we can simply use a string of length  $6\lceil \log(s+1) \rceil$  to represent them, starting with the prefix  $0^{s+1}1$  to encode  $s$ . 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* NAND++ program.

### 8.3.2 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  $x_0, \dots, x_{(6 \cdot \text{lines} - 1)}$  where  $\text{lines}$  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  $\text{lines}$  as this bound).

```
simloop := 3
totalvars := NumberVariables(x)
maxlines := Length(x) / 6
currenti := 0
currrentround := 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 $\ll$  program to a NAND $++$  one, we can also implement this interpreter in NAND $++$ , hence completing the proof of [Theorem 8.2](#).

### 8.3.3 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 $\ll$  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.

#### 8.4 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  $x$ .

#### 8.5 Exercises

**Exercise 8.1 — Pairing.** Let  $\text{PAIR} : \mathbb{N}^2 \rightarrow \mathbb{N}$  be the function defined as  $\text{PAIR}(x_0, x_1) = \frac{1}{2}(x_0 + x_1)(x_0 + x_1 + 1) + x_1$ .

1. Prove that for every  $x^0, x^1 \in \mathbb{N}$ ,  $\text{PAIR}(x^0, x^1)$  is indeed a natural number.
2. Prove that  $\text{PAIR}$  is one-to-one
3. Construct a NAND++ program  $P$  such that for every  $x^0, x^1 \in \mathbb{N}$ ,  $P(pf(x^0)pf(x^1)) = pf(\text{PAIR}(x^0, x^1))$ , where  $pf$  is the prefix-free encoding map defined above. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.
4. Construct NAND++ programs  $P_0, P_1$  such that for every  $x^0, x^1 \in \mathbb{N}$  and  $i \in N$ ,  $P_i(pf(\text{PAIR}(x^0, x^1))) = pf(x^i)$ . You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter. ■

**Exercise 8.2 — Single vs multiple bit.** Prove that for every  $F : \{0,1\}^* \rightarrow \{0,1\}^*$ , the function  $F$  is computable if and only if the following function  $G : \{0,1\}^* \rightarrow \{0,1\}$  is computable, where  $G$  is defined as

$$\text{follows: } G(x, i, \sigma) = \begin{cases} F(x)_i & i < |F(x)|, \sigma = 0 \\ 1 & i < |F(x)|, \sigma = 1 \\ 0 & i \geq |F(x)| \end{cases} \quad \blacksquare$$

## 8.6 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.

## 8.7 Further explorations

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

## 8.8 Acknowledgements



### Learning Objectives:

- Learn about the Turing Machine and  $\lambda$  calculus, which are important models of computation.
- See the equivalence between these models and NAND++ programs.
- See how many other models turn out to be “Turing complete”
- Understand the Church-Turing thesis.

9

## *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.

## 9.1 Turing machines

The “granddaddy” of all models of computation is the *Turing Machine*, which is the standard model of computation in most textbooks.<sup>1</sup> 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.

<sup>1</sup> 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.



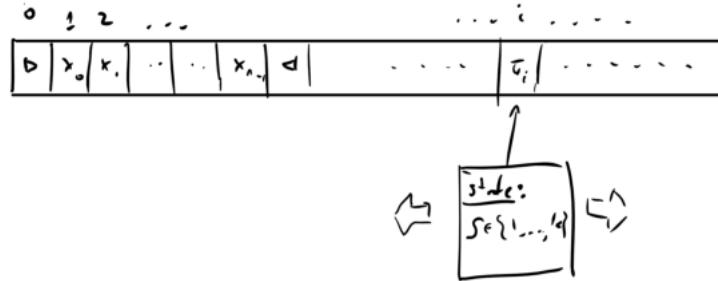
**Figure 9.1:** Until the advent of electronic computers, the word “computer” was used to describe a person, often female, that performed calculations. These human computers were absolutely essential to many achievements including mapping the stars, breaking the Enigma cipher, and the NASA space mission. 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

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  $\triangleright, x_0, \dots, x_{n-1}, \emptyset, \emptyset, \dots$ <sup>2</sup>
- 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 9.2:** A Turing machine has access to a *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.

3

<sup>3</sup> TODO: update figure to  $\{0, \dots, k - 1\}$ .

The formal definition of Turing machines is as follows:

**Definition 9.1 — Turing Machine.** A (one tape) *Turing machine* with  $k$  states and alphabet  $\Sigma \supseteq \{0,1,\triangleright,\emptyset\}$  is a function  $M : [k] \times \Sigma \rightarrow [k] \times \Sigma \times \{L,R\}$ . 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:  $T[0] = \triangleright,$

<sup>2</sup> We use the symbol  $\triangleright$  to denote the beginning of the tape, and the symbol  $\emptyset$  to denote an empty cell. Hence we will assume that  $\Sigma$  contains these symbols, along with 0 and 1.

$T[i] = x_i$  for  $i = 1, \dots, |x|$

- 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  $T[1], \dots, T[n - 1]$ . 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  $x$ .)

R

**Turing Machine configurations** Just as we did for NAND++ programs, we can also define the notion of *configurations* for Turing machines and define computation in terms of iterations of their *next step function*. However, we will not follow this approach in this course.

## 9.2 Turing Machines and NAND++ programs

As mentioned, Turing machines turn out to be equivalent to NAND++ programs:

**Theorem 9.1 — 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  $F$ .

### 9.2.1 Simulating Turing machines with NAND++ programs

We now prove the “if” direction of [Theorem 9.1](#), 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 $\ll$  and NAND++ are equivalent in power, it is sufficient to construct a NAND $\ll$  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  $M : [k] \times \Sigma \rightarrow \Sigma \times [k] \times \{L, R\}$  then there is a finite length NAND program ComputeM 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.
// we also assume we can index an array with a variable
// other than i,
// and with some simple expressions of it (e.g., foo_{j+1})
// this can be easily simulated in NAND<<
//
// we assume an encoding such that the default value for
// tape_j is "."

// below k is the number of states of the machine M
// ComputeM is a function that maps a symbol in Sigma and a
// state in [k]
// to the new state, symbol to write, and direction

tape_0 := ">"
j = 0
while (validx_j) { // copy input to tape
    tape_{j+1} := x_j
    j++
}

state := 0
head := 0 // location of the head
while NOT EQUAL(state,k-1) { // not ending state
    state', symbol, dir := ComputeM(tape_head,state)
    tape_head := symbols
    if EQUAL(dir,'L') AND NOT(EQUAL(tape_head,>)) {
        head--
    }
    if EQUAL(dir,'R') {
        head++
    }
}
```

```

state' := state
}

// after halting, we copy the contents of the tape
// to the output variables
// we stop when we see a non-boolean symbol
j := 1
while EQUAL(tape_j,0) OR EQUAL(tape_j,1) {
    y_{j-1} := tape_j
}

```

In addition to the standard syntactic sugar, we are also assuming in the above code that 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. Since these are *finite* functions (whose input and output length only depends on the number of states and symbols of the machine  $M$ , and not the input length), we can compute them using a `NAND` (and hence in particular a `NAND++` or `NAND«`) program.

### 9.2.2 Simulating `NAND++` programs with Turing machines

To prove the second direction of [Theorem 9.1](#), we need to show that for every `NAND++` program  $P$ , there is a Turing machine  $M$  that computes the same function as  $P$ . The idea behind the proof is that the TM  $M$  will *simulate* the program  $P$  as follows: (see also [Fig. 9.3](#))

- The *head position* of  $M$  will correspond to the position of the index  $i$  in the current execution of the program  $P$ .
- The *alphabet* of  $M$  will be large enough so that in position  $i$  of the tape will store the contents of all the variables of  $P$  of the form  $\text{foo}_{\langle i \rangle}$ . (In particular this means that the alphabet of  $M$  will have at least  $2^t$  symbols when  $t$  is the number of variables of  $P$ .)
- The *states* of  $M$  will be large enough to encode the current line number that is executed by  $P$ , as well as the contents of all variables that are indexed in the program by an absolute numerical index (e.g., variables of the form `foo` or `bar_17` that are not indexed with  $i$ .)

The key point is that the number of lines of  $P$  and the number of variables are constants that do not depend on the length of the input and so can be encoded in the alphabet and state size. More specifically, if  $V$  is the set of variables of  $P$ , then the alphabet of  $M$

will contain (in addition to the symbols  $\{0, 1, \triangleright, \emptyset\}$ ) the finite set of all functions from  $V$  to  $\{0, 1\}$ , with the semantics that if  $\sigma : V \rightarrow \{0, 1\}$  appears in the  $i$ -th position of the tape then for every variable  $v \in V$ , the value of  $v_{\langle i \rangle}$  equals  $\sigma(v)$ . Similarly, we will think of the state space of  $M$  as containing all pairs of the form  $(\ell, \tau)$  where  $\ell$  is a number between 0 and the number of lines in  $P$  and  $\tau$  is a function from  $V \times [c]$  to  $\{0, 1\}$  where  $c - 1$  is the largest absolute numerical index that appears in the program.<sup>4</sup> The semantics are that  $\ell$  encodes the line of  $P$  that is about to be executed, and  $\tau(v, j)$  encodes the value of the variable  $v_{\langle j \rangle}$ .

To simulate the execution of one step in  $P$ 's computation, the machine  $M$  will do the following:

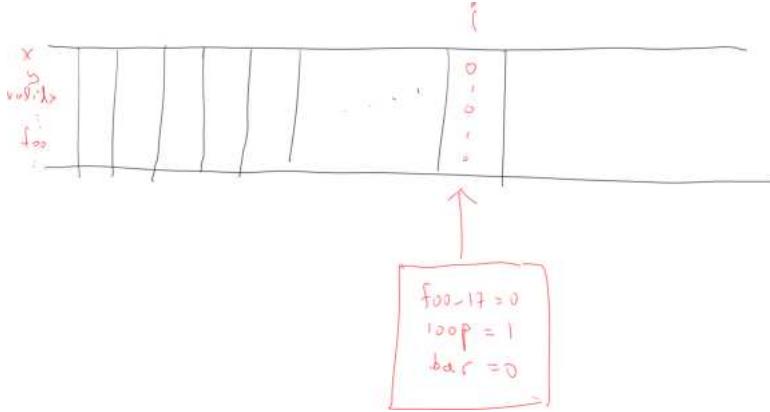
1. The contents of the symbol at the current head position and its state encode all information about the current line number to be executed, as well as the contents of all variables of the program of the form  $\text{foo}_{\langle i \rangle}$  where  $i$  is the current value of the index variable in the simulated program. Hence we can use that to compute the new line to be executed.
2.  $M$  will write to the tape and update its state to reflect the update to the variable that is assigned a new value in the execution of this line.
3. If the current line was the last one in the program, and the loop variable (which is encoded in  $M$ 's state) is equal to 1 then  $M$  will decide whether to move the head left or right based on whether the index is going to increase and decrease. We can ensure this is a function of the current variables of  $P$  by adding to the program the variables `breadcrumbs_i`, `atstart_i` and `indexincreasing` and the appropriate code so that `indexincreasing` is set to 1 if and only if the index will increase in the next iteration.<sup>5</sup>

The simulation will also contain an *initialization* and a *finalization* phases. In the *initialization phase*,  $M$  will scan the input and modify the tape so it contains the proper encoding of  $x_i$ 's  $\text{validx}_i$ 's variables as well as load into its state all references to these variables with absolute numerical indices. In the *finalization phase*,  $M$  will scan its tape to copy the contents of the  $y_i$ 's (i.e., output) variables to the beginning of the tape, and write a non  $\{0, 1\}$  value at their end. We leave verifying the (fairly straightforward) details of implementing these steps to the reader. Note that we can add a finite number of states to  $M$  or symbols to its alphabet to make this implementation easier. Writing down the full description of  $M$  from the above “pseu-

<sup>4</sup> While formally the state space of  $M$  is a number between 0 and  $k - 1$  for some  $k \in \mathbb{N}$ , by making  $k$  large enough, we can encode all pairs  $(\ell, \tau)$  of the form above as elements of the state space of the amchine, and so we can think of the state as having this form.

<sup>5</sup> While formally we did not allow the head of the Turing machine to stay in place, we can simulate this by adding a “dummy step” in which  $M$ 's head moves right and goes into a special state that will move left in the next step.

decode" is straightforward, even if somewhat painful, exercise, and hence this completes the proof of [Theorem 9.1](#).



**Figure 9.3:** To simulate a NAND++ program  $P$  using a machine  $M$  we introduce a large alphabet  $\Sigma$  such that each symbol in  $\Sigma$  can be thought of as a "mega symbol" that encodes the value of all the variables indexed at  $i$ , where  $i$  is the current tape location. Similarly each state of  $M$  can be thought of as a "mega state" that encodes the value of all variables of  $P$  that have an absolute numerical index, as well as the current line that is about to be executed.

**R** **Polynomial equivalence** If we examine the proof of [Theorem 9.1](#) then we can see that the equivalence between NAND++ programs and Turing machines is up to polynomial overhead in the number of steps. Specifically, our NAND« program to simulate a Turing Machine  $M$ , has two loops to copy the input and output and then one loop that iterates once per each step of the machine  $M$ . In particular this means that if the Turing machine  $M$  halts on every  $n$ -length input  $x$  within  $T(n)$  steps, then the NAND« program computes the same function within  $O(T(n) + n + m)$  steps. Moreover, the transformation of NAND« to NAND++ has polynomial overhead, and hence for any such machine  $M$  there is a NAND++ program  $P'$  that computes the same function within  $poly(T(n) + n + m)$  steps (where  $poly(f(n))$  is shorthand for  $f(n)^{O(1)}$  as defined in the mathematical background section). In the other direction, our Turing machine  $M$  simulates each step of a NAND++ program  $P$  in a constant number of steps. The initialization and finalization phases involve scanning over  $O(n + m)$  symbols and copying them around. Since the cost to move a symbol to a point that is of distance  $d$  in the tape can be  $O(d)$  steps, the total cost of these phases will be  $O((n + m)^2)$ . Thus, for every NAND++ program  $P$ , if  $P$  halts on input  $x$  after  $T$  steps, then the corresponding Turing machine  $M$  will halt after  $O(T + (n + m)^2)$  steps.

### 9.3 “Turing Completeness” and other Computational models

A computational model  $M$  is *Turing complete* if every partial function  $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$  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  $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$ ,  $F$  is computable by a Turing machine if and only if it is computable in  $M$ . Another way to state [Theorem 9.1](#) is that NAND++ is Turing equivalent. Since we can simulate any NAND« program by a NAND++ program (and vice versa), NAND« is Turing equivalent as well. It turns out that there are many other Turing-equivalent models of computation.

We now discuss a few examples.

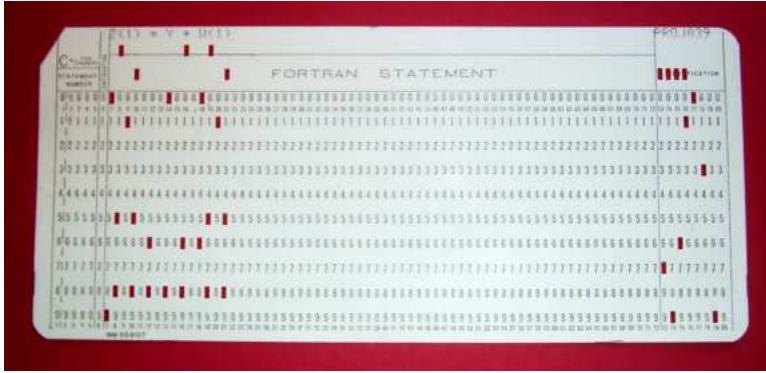
#### 9.3.1 RAM Machines

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*  $r_1, \dots, r_k$  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« programs. Every NAND« 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« program can simulate a RAM machine, using its variables as the registers.

#### 9.3.2 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



**Figure 9.4:** A punched card corresponding to a Fortran statement.

languages translates programs into machine languages that are quite similar to NAND $\ll$  programs or RAM machines. We can also translate NAND $\gg$  programs to such programming languages. Thus all these programming languages are Turing equivalent.<sup>6</sup>

#### 9.4 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

$$\text{square}(x) = x \cdot x \quad (9.1)$$

we write it as

$$\lambda x. x \cdot x \quad (9.2)$$

Generally, an expression of the form

$$\lambda x. e \quad (9.3)$$

<sup>6</sup> 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.

corresponds to the function that maps any expression  $z$  into the expression  $e[x \rightarrow z]$  which is obtained by replacing every occurrence of  $x$  in  $e$  with  $z$ .<sup>7</sup>

**Currying.** The expression  $e$  can itself involve  $\lambda$ , and so for example the function

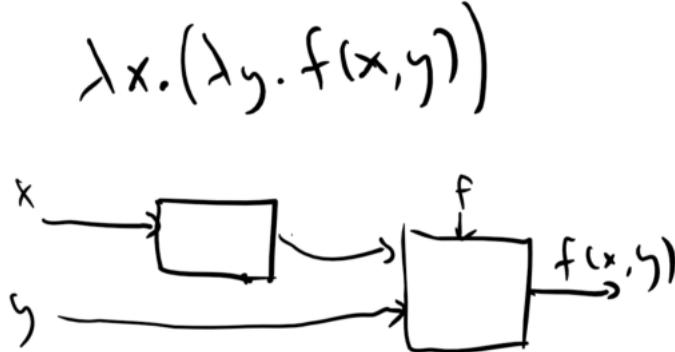
$$\lambda x.(\lambda y.x + y) \quad (9.4)$$

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  $\lambda x.(\lambda y.e)$ .<sup>8</sup>

<sup>7</sup> More accurately, we replace every expression of  $x$  that is *bound* 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  $(\lambda 7.x + 1)(7)$ . To avoid such annoyances, we can always ensure that every instance of  $\lambda var.e$  uses a unique variable identifier  $var$ . See the “logical operators” section in the math background lecture for more discussion on bound variables.

<sup>8</sup> This technique of simulating multiple-input functions with single-input functions is known as **Currying** and is named after the logician **Haskell Curry**.



**Figure 9.5:** 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  $fz$  instead of  $f(z)$  for the expression corresponding to invoking  $f$  on  $z$ .<sup>9</sup> That is, if  $f$  has the form  $\lambda x.e$  then  $fz$  is the same as  $f(z)$ , which corresponds to the expression  $e[x \rightarrow z]$  (i.e., the expression obtained by invoking  $f$  on  $z$  via replacing all copies of the  $x$  parameter with  $z$ ).

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

<sup>9</sup> When using identifiers with multiple letters for  $\lambda$  expressions, we’ll separate them with spaces or commas.

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  $DOUBLE$  is the  $\lambda$  expression  $\lambda f.(\lambda x.f(fx))$ , then for every function  $f$ ,  $DOUBLEf$  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  $DOUBLEf = \lambda x.(x + 2)$ .

**(Lack of) types.** Unlike most programming languages, 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.

#### 9.4.1 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:

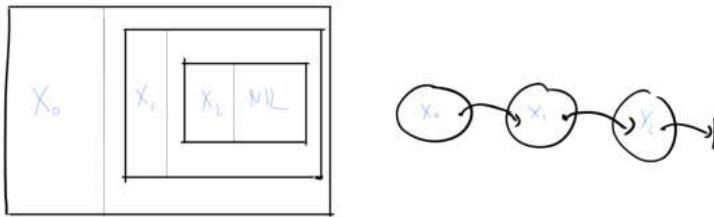
- **Boolean constants:** 0 and 1. We also have the  $IF(cond, a, b)$  functions that outputs  $a$  if  $cond = 1$  and  $b$  otherwise. Using  $IF$  we can also compute logical operations such as  $AND, OR, NOT, NAND$  etc.: can you see why?
- **The empty string:** The value  $NIL$  and the function  $ISNIL(x)$  that returns 1 iff  $x$  is  $NIL$ .
- **Strings/lists:** The function  $PAIR(x, y)$  that creates a pair from  $x$  and  $y$ . We will also have the function  $HEAD$  and  $TAIL$  to extract the first and second member of the pair. We can now create the list  $x, y, z$  by  $PAIR(x, PAIR(y, PAIR(z, NIL)))$ , see Fig. 9.6. A *string* is of course simply a list of bits.<sup>10</sup>
- **List operations:** The functions  $MAP, REDUCE, FILTER$ . Given a list  $L = (x_0, \dots, x_{n-1})$  and a function  $f$ ,  $MAP(L, f)$  applies  $f$  on every member of the list to obtain  $L = (f(x_0), \dots, f(x_{n-1}))$ . The function  $FILTER(L, f)$  returns the list of  $x_i$ ’s such that  $f(x_i) = 1$ ,

<sup>10</sup> Note that if  $L$  is a list, then  $HEAD(L)$  is its first element, but  $TAIL(L)$  is not the last element but rather all the elements except the first. We use  $NIL$  to denote the empty list and hence  $PAIR(x, NIL)$  denotes the list with the single element  $x$ .

and  $\text{REDUCE}(L, f)$  “combines” the list by outputting

$$f(x_0, f(x_1, \dots, f(x_{n-3}, f(x_{n-2}, f(x_{n-1}, \text{NIL})) \dots)) \quad (9.5)$$

For example  $\text{REDUCE}(L, +)$  would output the sum of all the elements of the list  $L$ . See Fig. 9.7 for an illustration of these three operations.



**Figure 9.6:** A list  $(x_0, x_1, x_2)$  in the  $\lambda$  calculus is constructed from the tail up, building the pair  $(x_2, \text{NIL})$ , then the pair  $(x_1, (x_2, \text{NIL}))$  and finally the pair  $(x_0, (x_1, (x_2, \text{NIL})))$ . 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 9.7:** Illustration of the *MAP*, *FILTER* and *REDUCE* operations.

Together these operations more or less amount to the Lisp/Scheme programming language.<sup>11</sup>

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:

**Theorem 9.2 — Lambda calculus and NAND++.** For every function  $F : \{0,1\}^* \rightarrow \{0,1\}^*$ ,  $F$  is computable in the  $\lambda$  calculus with the above basic operations if and only if it is computable by a NAND++ program.

<sup>11</sup> In Lisp, the *PAIR*, *HEAD* and *TAIL* functions are traditionally called *cons*, *car* and *cdr*.

*Proof.* We only sketch the proof. The “only if” direction is simple. As mentioned above, evaluating  $\lambda$  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 $\ll$  as well, which we can then transform to a NAND $++$  program.

For the “if” direction, we start by showing that for every normal-form NAND $++$  program  $P$ , we can compute the next-step function  $NEXT_P : \{0,1\}^* \rightarrow \{0,1\}^*$  using the above operations. It turns out not to be so hard. A configuration  $\sigma$  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  $\sigma = (\sigma^1, \dots, \sigma^T)$  of  $T$  lists of bits, each of length  $B$ . There is some constant index  $a \in [B]$  such that the  $i$ -th block is active if and only if  $\sigma_a^i = 1$ , and similarly there are also indices  $s, f$  that tell us if a block is the first or final block.

For every index  $c$ , we can extract from the configuration  $\sigma$  the  $B$  sized string corresponding to the block  $\sigma^i$  where  $\sigma_c^i = 1$  using a single *REDUCE* operation. Therefore, we can extract the first block  $\sigma^0$ , as well as the active block  $\sigma^i$ , and using similar ideas we can also extract a constant number of blocks that follow the first blocks ( $\sigma^1, \sigma^2, \dots, \sigma^{c'}$  where  $c'$  is the largest numerical index that appears in the program).<sup>12</sup>

Using the first blocks and active block, we can update the configuration, execute the corresponding line, and also tell if this is an operation where the index  $i$  stays the same, increases, or decreases. If it stays the same then we can compute  $NEXT_P$  via a *MAP* operation, using the function that on input  $C \in \{0,1\}^B$ , keeps  $C$  the same if  $C_a = 0$  (i.e.,  $C$  is not active) and otherwise updates it to the value in its next step. If  $i$  increases, then we can update  $\sigma$  by a *REDUCE* operation, with the function that on input a block  $C$  and a list  $S$ , we output  $PAIR(C, S)$  unless  $C_a = 1$  in which case we output  $PAIR(C', PAIR(C'', TAIL(S)))$  where  $(C', C'')$  are the new values of the blocks  $i$  and  $i + 1$ . The case for decreasing  $i$  is analogous.

Once we have a  $\lambda$  expression  $\varphi$  for computing  $NEXT_P$ , we can compute the final expression by defining

$$APPLY = \lambda f, \sigma. IF(HALT\sigma, \sigma, ff\varphi\sigma) \quad (9.6)$$

now for every configuration  $\sigma_0$ ,  $APPLY\ APPLY\sigma$  is the final configuration  $\sigma_t$  obtained after running the next-step function continuosly. Indeed, note that if  $\sigma_0$  is not halting, then  $APPLY\ APPLY\sigma_0$  outputs  $APPLY\ APPLY\varphi\sigma_0$  which (since  $\varphi$  computes the  $NEXT_P$ ) function

<sup>12</sup> It's also not hard to modify the program so that the largest numerical index is zero, without changing its functionality

is the same as  $\text{APPLY } \text{APPLY}\sigma_1$ . By the same reasoning we see that we will eventually get  $\text{APPLY } \text{APPLY}\sigma_t$  where  $\sigma_t$  is the halting configuration, but in this case we will get simply the output  $\sigma_t$ .<sup>13</sup> ■

#### 9.4.2 How basic is “basic”?

While the collection of “basic” functions we allowed for  $\lambda$  calculus 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 example, start by thinking how you could implement *MAP* using *REDUCE*, and then try to continue and minimize things further, trying to implement *REDUCE* with from *0, 1, IF, PAIR, HEAD, TAIL* together with the  $\lambda$  operations. Remember that your functions can take functions as input and return functions as output.

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  $\lambda$  formalism only, even without using 0’s and 1’s. It’s  $\lambda$ ’s all the way down! The idea is that we encode 0 and 1 themselves as  $\lambda$  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  $\lambda$  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  $1 = \lambda x. \lambda y. x$ .<sup>14</sup>
- The above implementation makes the *IF* function trivial:  $\text{IF}(\text{cond}, a, b)$  is simply  $\text{cond } a \ b$  since  $0ab = b$  and  $1ab = a$ . (We can write  $\text{IF} = \lambda x. x$  to achieve  $\text{IF cond } a \ b = \text{cond } a \ b$ .)
- 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  $\text{PAIR} = \lambda x, y. \lambda g. gxy$ . Note that now we can

<sup>13</sup> If this looks like recursion then this is not accidental- this is a special case of a general technique for simulating recursive functions in the  $\lambda$  calculus. See the discussion on the *Y* combinator below.

<sup>14</sup> We could of course have flipped the definitions of 0 and 1, but we use the above because it is the common convention in the  $\lambda$  calculus, where people think of 0 and 1 as “false” and “true”.

extract the first element of a pair  $p$  by writing  $p_1$  and the second element by writing  $p_0$ , and so  $\text{HEAD} = \lambda p.p_1$  and  $\text{TAIL} = \lambda p.p_0$ .

- We define  $\text{NIL}$  to be the function that ignores its input and always outputs 1. That is,  $\text{NIL} = \lambda x.1$ . The  $\text{ISNIL}$  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  $p_0_{x,y} = 0$  while  $\text{NIL}0_{x,y} = 1$ . Formally,  $\text{ISNIL} = \lambda p.p(\lambda x.y.0)$ .

#### 9.4.3 List processing and recursion without recursion

Now we come to the big hurdle, which is how to implement  $\text{MAP}$ ,  $\text{FILTER}$ , and  $\text{REDUCE}$  in the  $\lambda$  calculus. It turns out that we can build  $\text{MAP}$  and  $\text{FILTER}$  from  $\text{REDUCE}$ . For example  $\text{MAP}(L,f)$  is the same as  $\text{REDUCE}(L,g)$  where  $g$  is the operation that on input  $x$  and  $y$ , outputs  $\text{PAIR}(f(x), \text{NIL})$  if  $y$  is  $\text{NIL}$  and otherwise outputs  $\text{PAIR}(f(x), y)$ . (I leave checking this as a (recommended!) exercise for you, the reader.) So, it all boils down to implementing  $\text{REDUCE}$ . We can define  $\text{REDUCE}(L,g)$  recursively, by setting  $\text{REDUCE}(\text{NIL},g) = \text{NIL}$  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

$$\text{REDUCE} = \lambda L.g.\text{IF}(\text{ISNIL}(L), \text{NIL}, g\text{HEAD}(L)\text{REDUCE}(\text{TAIL}(L),g)) . \quad (9.7)$$

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 `eval` operation that given a string `code` and an input `x`, evaluates `code` on `x`. Then, if you have a program `P` that can print its own code, you can use `eval` as an alternative to recursion: instead of using a recursive call on some input `x`, the program will compute its own code, store it in some string variable `str` and then use `eval(str, x)`. You might find this confusing. I definitely

find this confusing. But hopefully the following will make things a little more concrete.

<sup>15</sup>

#### 9.4.4 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

$$\text{par}(x_0, \dots, x_n) = \begin{cases} 0 & |x| = 0 \\ x_0 \oplus \text{par}(x_1, \dots, x_n) & \text{otherwise} \end{cases} \quad (9.8)$$

where  $\oplus$  denotes the XOR operator.

Our key insight would be to recast Eq. (9.8) not as a *definition* of the parity function but rather as an *equation* on it. That is, we can think of Eq. (9.8) as stating that

$$\text{par} = \text{PAREQ}(\text{par}) \quad (9.9)$$

where  $\text{PAREQ}$  is a *non-recursive* function that takes a function  $p$  as input, and returns the function  $q$  defined as

$$q(x_0, \dots, x_n) = \begin{cases} 0 & |x| = 0 \\ x_0 \oplus p(x_1, \dots, x_n) & \text{otherwise} \end{cases} \quad (9.10)$$

In fact, it’s not hard to see that satisfying Eq. (9.9) is *equivalent* to satisfying Eq. (9.8), and hence  $\text{par}$  is the *unique* function that satisfies the condition Eq. (9.9). This means that to find a function  $\text{par}$  computing parity, all we need is a “magical function”  $\text{SOLVE}$  that given a function  $\text{PAREQ}$  finds “fixed point” of  $\text{PAREQ}$ : a function  $p$  such that  $\text{PAREQ}(p) = p$ . Given such a “magical function”, we could give a non-recursive definition for  $\text{par}$  by writing  $\text{par} = \text{SOLVE}(\text{PAREQ})$ .

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**.

<sup>15</sup> TODO: add a direct example how to implement  $\text{REDUCE}$  with  $\text{XOR}$  without using the  $\text{Y}$  combinator. Hopefully it can be done in a way that makes things more intuitive.

**Theorem 9.3 — Y combinator.** Let

$$Y = \lambda f.(\lambda x.f(xx))(\lambda y.f(yy)) \quad (9.11)$$

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

$$YF = (\lambda x.F(xx))(\lambda y.F(yy)) \quad (9.12)$$

But this is the same as applying  $F$  to  $gg$  where  $g = \lambda y.F(y,y)$ , or in other words

$$YF = F((\lambda y.F(y,y))(\lambda y.F(y,y))) \quad (9.13)$$

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 [Theorem 9.2](#).

For example, to compute parity we first give a recursive definition of parity using the  $\lambda$ -calculus as

$$\text{par}L = \text{IF}(\text{ISNIL}(L), 0, \text{XORHEAD}(L)\text{par}(\text{TAIL}(L))) \quad (9.14)$$

We then avoid the recursion by converting [Eq. \(9.14\)](#) to the operator  $\text{PAREQ}$  defined as

$$\text{PAREQ} = \lambda p.\lambda L.\text{IF}(\text{ISNIL}(L), 0, \text{XORHEAD}(L)p(\text{TAIL}(L))) \quad (9.15)$$

and then we can define  $\text{par}$  as  $\text{YPAREQ}$  since this will be the unique solution to  $p = \text{PAREQ}p$ .

**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

$$(\lambda x.fff)(\lambda x.fff) \quad (9.16)$$

If we try to evaluate it then the first step is to invoke the lefthand function on the righthand one and then obtain

$$(\lambda x. xxx)(\lambda x. xxx)(\lambda x. xxx) \quad (9.17)$$

To evaluate this, the next step would be to apply the second term on the third term,<sup>16</sup> which would result in

$$(\lambda x. xxx)(\lambda x. xxx)(\lambda x. xxx)(\lambda x. xxx) \quad (9.18)$$

We can see that continuing in this way we get longer and longer expressions, and this process never concludes.

## 9.5 Other models

There is a great variety of models that are computationally equivalent to Turing machines (and hence to NAND++/NAND« program). Chapter 8 of the book **The Nature of Computation** is a wonderful resource for some of those models. We briefly mention a few examples.

### 9.5.1 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  $mt$ .

### 9.5.2 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.

<sup>16</sup> This assumes we use the “call by value” evaluation ordering which states that to evaluate a  $\lambda$  expression  $fg$  we first evaluate the righthand expression  $g$  and then invoke  $f$  on it. The “Call by name” or “lazy evaluation” ordering would first evaluate the lefthand expression  $f$  and then invoke it on  $g$ . In this case both strategies would result in an infinite loop. There are examples though when “call by name” would not enter an infinite loop while “call by value” would. The SML and OCaml programming languages use “call by value” while Haskell uses (a close variant of) “call by name”.

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 diagonal 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++) 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++ program using the game of life (see [Fig. 9.8](#)). The [Wikipedia page](#) for the Game of Life contains 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. 9.9](#).

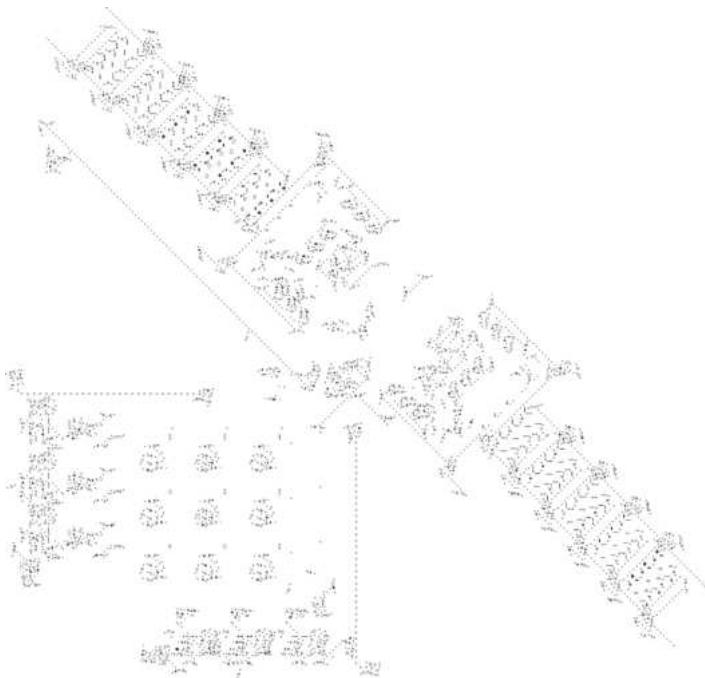
## 9.6 Our models vs standard texts

We can summarize the models we use versus those used in other texts in the following table:

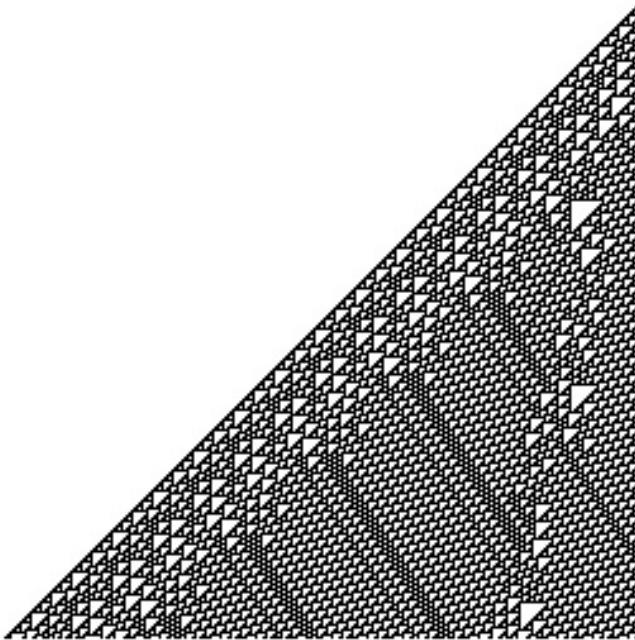
Model	These notes	Other texts
Nonuniform	NAND programs	Boolean circuits, straightline programs
Uniform (random access)	NAND« programs	RAM machines
Uniform (sequential access)	NAND++ programs	Oblivious one-tape Turing machines

\

Later on in this course we may 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



**Figure 9.8:** A Game-of-Life configuration simulating a Turing Machine. Figure by Paul Rendell.



**Figure 9.9:** Evolution of a one dimensional automata. Each row in the figure corresponds to the configuration. The initial configuration corresponds to the top row and contains only a single “live” cell. This figure corresponds to the “Rule 110” automaton of Stefan Wolfram which is Turing Complete. Figure taken from [Wolfram MathWorld](#).

“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*).

## 9.7 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,  $\lambda$  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.<sup>17</sup>

<sup>17</sup> The *extended* Church Turing thesis, which we'll discuss later in this course, is that NAND++ programs even capture the limit of what can be *efficiently* computable. Just like the PECTT, quantum computing presents the main challenge to this thesis.

## 9.8 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 $\ll$  but also Turing machines, RAM machines, the  $\lambda$ -calculus and many other models.
- Very simple models turn out to be “Turing complete” in the sense that they can simulate arbitrarily complex computation.

## 9.9 Exercises

<sup>18</sup>

**Exercise 9.1 — 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  $x, y, z$ .<sup>19</sup>

## 9.10 Bibliographical notes

<sup>20</sup>

<sup>18</sup> 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.

<sup>19</sup> Hint: You can reduce the number of variables a function takes by “pairing them up”. That is, define a  $\lambda$  expression *PAIR* such that for every  $x, y$   $\text{PAIR}xy$  is some function  $f$  such that  $f0 = x$  and  $f1 = y$ . Then use *PAIR* to iteratively reduce the number of variables used.

<sup>20</sup> TODO: Recommend Chapter 7 in the nature of computation

## 9.11 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

## 9.12 Acknowledgements



## 10

# Is every function computable?

### Learning Objectives:

- See a fundamental result in computer science and mathematics: the existence of uncomputable functions.
- See the canonical example for an uncomputable function: *the halting problem*.
- Introduction to the technique of *reductions* which will be used time and again in this course to show difficulty of computational tasks.
- Rice's Theorem, which is a starting point for much of research on compilers and programming languages, and marks the difference between *semantic* and *syntactic* properties of programs.

*"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  $x$ .<sup>1</sup> 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:

**Theorem 10.1 — Uncomputable functions.** There exists a function  $F^* : \{0,1\}^* \rightarrow \{0,1\}$  that is not computable by any NAND++ program.

*Proof.* The proof is illustrated in Fig. 10.1. 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  $F^*(x) := 1 - G(x)$ .

<sup>1</sup> In the 1800's, with the invention of the Fourier series and with the systematic study of continuity and differentiability, people have started 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 Poincaré 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. ■

<i>input programs</i>	0	1	01	10	...	$x$	...
0	$1 - P_0(0)$	$1 - P_0(1)$	$1 - P_0(01)$	$1 - P_0(10)$		$1 - P_0(x)$	
1	$1 - P_1(0)$	$1 - P_1(1)$	$1 - P_1(01)$	$1 - P_1(10)$		$1 - P_1(x)$	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$k$	$1 - P_k(0)$	$1 - P_k(1)$	$1 - P_k(01)$	$1 - P_k(10)$		$1 - P_k(x)$	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

**Figure 10.1:** 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 10.1](#) 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 10.1](#) is known as *diagonalization* since it can be described as defining a function based on the diagonal entries of a table as in Fig. 10.1.

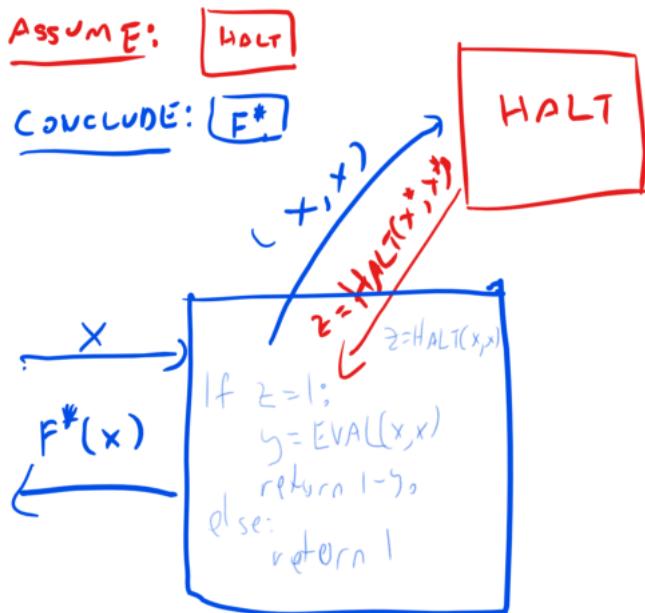
## 10.1 The Halting problem

Theorem 10.1 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 10.2 — Uncomputability of Halting function.** Let  $\text{HALT} : \{0,1\}^* \rightarrow \{0,1\}$  be the function such that  $\text{HALT}(P, x) = 1$  if the NAND++ program  $P$  halts on input  $x$  and equals to 0 if it does not. Then  $\text{HALT}$  is not computable.

Before turning to prove Theorem 10.2, we note that  $\text{HALT}$  is a very natural function to want to compute. For example, one can think of  $\text{HALT}$  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 10.2:** We prove that  $\text{HALT}$  is uncomputable using a *reduction* from computing the previously shown uncomputable function  $F^*$  to computing  $\text{HALT}$ . We assume that we had an algorithm that computes  $\text{HALT}$  and use that to obtain an algorithm that computes  $F^*$ .

*Proof.* The proof will use the previously established [Theorem 10.1](#), as illustrated in [Fig. 10.2](#). That is, we will assume, towards a contradiction, that there is NAND++ program  $P^*$  that can compute the  $\text{HALT}$  function, and use that to derive that there is some NAND++ program  $Q^*$  that computes the function  $F^*$  defined above, contradicting [Theorem 10.1](#). (This is known as a proof by *reduction*, since we reduce the task of computing  $F^*$  to the task of computing  $\text{HALT}$ . By the contrapositive, this means the uncomputability of  $F^*$  implies the uncomputability of  $\text{HALT}$ .)

Indeed, suppose that  $P^*$  was a NAND++ program that computes  $\text{HALT}$ . Then we can write a NAND++ program  $Q^*$  that does the following on input  $x \in \{0,1\}^*$ :<sup>2</sup>

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  $\text{EVAL}(x, x)$  (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  $P^*(x, x) = \text{HALT}(x, x)$  then the program  $Q^*(x) = F^*(x)$  where  $F^*$  is the function from the proof of [Theorem 10.1](#).

Note that the claim immediately implies that our assumption that  $P^*$  computes  $\text{HALT}$  contradicts [Theorem 10.1](#), 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  $z = \text{HALT}(x, x) = 1$ , 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  $Q^*(x) = 1 = F^*(x)$ . ■

<sup>2</sup> Note that we are using here a “high level” description of NAND++ programs. We know that we can implement the steps below, for example by first writing them in NAND« and then transforming the NAND« program to NAND++. Step 1 involves simply running the program  $P^*$  on some input.

### 10.1.1 Is the Halting problem really hard?

Many people's first instinct when they see the proof of [Theorem 10.2](#) 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).<sup>3</sup>

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:

```
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

<sup>3</sup> 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.

unclear that humans have any magical ability to say whether this (or other similar programs) will halt or not.



**Figure 10.3:** XKCD's take on solving the Halting problem, using the principle that "in the long run, we'll all be dead".

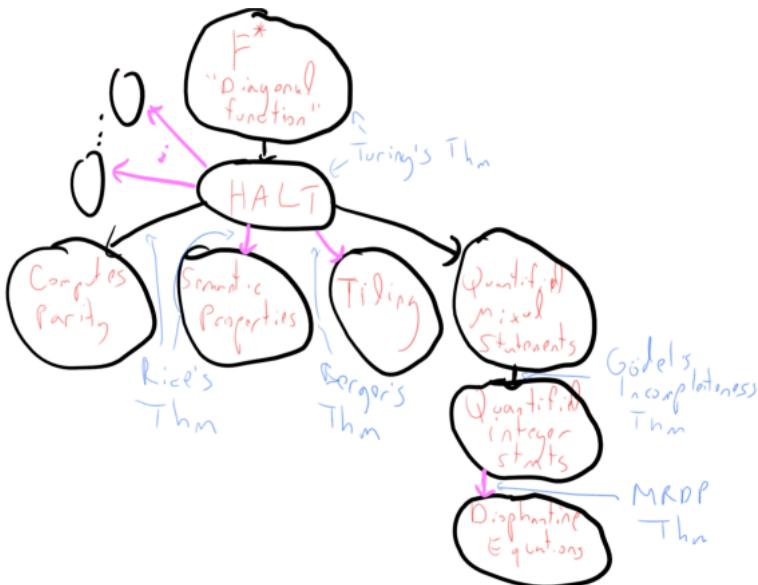
### 10.1.2 Reductions

The Halting problem turns out to be a linchpin of uncomputability, in the sense that [Theorem 10.2](#) 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. 10.4](#)).

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 showing this follows from the uncomputability of the function  $F^*$  from [Theorem 10.1](#).)

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 10.5](#).)



**Figure 10.4:** 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*.

4

<sup>4</sup> TODO: clean up this figure

## 10.2 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 properties” 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 10.3 — Halting without input.** Let  $\text{HALTONZERO} : \{0,1\}^* \rightarrow \{0,1\}$  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  $\text{HALTONZERO}$  is uncomputable.



The proof of [Theorem 10.3](#) is below, but before reading it you might want to pause for a couple of minutes and think how you would prove it yourself. In particular, try to think of what a reduction from  $\text{HALT}$  to  $\text{HALTONZERO}$  would look like. Doing so is an excellent way to get some initial comfort with the notion of proofs by *reduction*, which is a notion that will recur time and again in this course.

*Proof of Theorem 10.3.* The proof is by reduction to  $\text{HALT}$ . Suppose, towards the sake of contradiction, that  $\text{HALTONZERO}$  is computable. In other words, suppose towards the sake of contradiction that there exists 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  $\text{HALT}$ .

On input a program  $P$  and some input  $x$ , the algorithm  $B$  will construct a program  $P'$  such that  $P'(0) = P(x)$  and then feed this to  $A$ , returning  $A(P')$ . We will describe this algorithm in terms of how one can use the input  $x$  to modify the source code of  $P$  to obtain the source code of the program  $P'$ . However, it is clearly possible to do these modifications also on the level of the string representations of the programs  $P$  and  $P'$ .

Constructing the program  $P'$  is in fact rather simple. The algorithm  $B$  will obtain  $P'$  by modifying  $P$  to ignore its input and use  $x$  instead. In particular, for  $n = |x|$ , the program  $P'$  will have variables  $\text{myx\_}0, \dots, \text{my\_}\langle n-1 \rangle$  that are set to the constants zero or one based on the value of  $x$ . That is, it will contain lines of the form  $\text{myx\_}\langle i \rangle := \langle x_i \rangle$  for every  $i < n$ . Similarly,  $P'$  will have variables

`myvalidx_0, ..., myvalidx_{n - 1}` that are all set to one. Algorithm  $B$  will include in the program  $P'$  a copy of  $P$  modified to change any reference to  $x_{\langle i \rangle}$  to  $\text{myx}_{\langle i \rangle}$  and any reference to  $\text{validx}_{\langle i \rangle}$  to  $\text{myvalidx}_{\langle i \rangle}$ . Clearly, regardless of its input,  $P'$  always emulates the behavior of  $P$  on input  $x$ . In particular  $P'$  will halt on the input 0 if and only if  $P$  halts on the input  $x$ . Thus if the hypothetical algorithm  $A$  satisfies  $A(P') = \text{HALTONZERO}(P')$  for every  $P'$  then the algorithm  $B$  we construct satisfies  $B(P, x) = \text{HALT}(P, x)$  for every  $P, x$ , contradicting the uncomputability of  $\text{HALT}$ . ■

**R**

**The hardwiring technique** In the proof of [Theorem 10.3](#) we used the technique of “hardwiring” an input  $x$  to a program  $P$ . That is, modifying a program  $P$  that it uses “hardwired constants” for some of all of its input. This technique is quite common in reductions and elsewhere, and we will often use it again in this course.

Once we show the uncomputability of  $\text{HALTONZERO}$  we can extend to various other natural functions:

**Theorem 10.4 — Computing all zero function.** Let  $\text{ZEROFUNC} : \{0, 1\}^* \rightarrow \{0, 1\}$  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  $\text{ZEROFUNC}$  is uncomputable.

*Proof.* The proof is by reduction to  $\text{HALTONZERO}$ . 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  $\text{HALTONZERO}$ . Given a program  $P$ , 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  $\text{ZEROFUNC}$ . Hence,  $\text{ZEROFUNC}(P') = 1$  if and only if  $\text{HALTONZERO}(P) = 1$ . 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  $\text{ZEROFUNC}$  then  $B$  computes  $\text{HALTONZERO}$ , contradicting [Theorem 10.3](#). ■

We can simply prove the following:

**Theorem 10.5 — Uncomputability of verifying parity.** The following func-

tion is uncomputable

$$\text{COMPUTES-PARITY}(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{otherwise} \end{cases} \quad (10.1)$$

We leave the proof of [Theorem 10.5](#) as an exercise.

### 10.2.1 Rice's Theorem

[Theorem 10.5](#) can be generalized far beyond the parity function and in fact it rules out verifying any type of semantic specification on programs. We 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.

For example, consider the following two C programs

```
int First(int k) {
    return 2*k;
}

int Second(int n) {
    int i = 0;
    int j = 0
    while (j < n) {
        i = i + 2;
        j = j + 1;
    }
    return i;
}
```

`First` and `Second` are two distinct C programs, but they compute the same function. A *semantic* property, such as “computing a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  where  $f(m) \geq m$  for every  $m$ ”, would be either *true* for both programs or *false* for both programs, since it depends on the *function* the programs compute and not on their code. A *syntactic* property, such as “containing the variable `k`” or “using a `while` operation” might be true for one of the programs and false for the other, since it can depend on properties of the programs’ *code*.

Often the properties of programs that we are most interested in are the *semantic* ones, since we want to understand the programs’ functionality. Unfortunately, the following theorem shows that such properties are uncomputable in general:

**Theorem 10.6 — Rice's Theorem (slightly restricted version).** We say that two strings  $P$  and  $Q$  representing NAND++ programs *have the same functionality* if for every input  $x \in \{0,1\}^*$ , either the programs corresponding to both  $P$  and  $Q$  don't halt on  $x$ , or they both halt with the same output.

We say that a function  $F : \{0,1\}^* \rightarrow \{0,1\}$  is *semantic* if for every  $P$  and  $Q$  that have the same functionality,  $F(P) = F(Q)$ . Then the only semantic computable total functions  $F : \{0,1\}^* \rightarrow \{0,1\}$  are the constant zero function and the constant one function.

*Proof.* We will illustrate the proof idea by considering a particular semantic function  $F$ . Define  $\text{MONOTONE} : \{0,1\}^* \rightarrow \{0,1\}$  as follows:  $\text{MONOTONE}(P) = 1$  if there does not exist  $n \in \mathbb{N}$  and two inputs  $x, x' \in \{0,1\}^n$  such that for every  $i \in [n]$   $x_i \leq x'_i$  but  $P(x)$  outputs 1 and  $P(x') = 0$ . That is,  $\text{MONOTONE}(P) = 1$  if it's not possible to find an input  $x$  such that flipping some bits of  $x$  from 0 to 1 will change  $P$ 's output in the other direction from 1 to 0. We will prove that  $\text{MONOTONE}$  is uncomputable, but the proof will easily generalize to any semantic function. For starters we note that  $\text{MONOTONE}$  is not actually the all zeroes or all one function:

- The program  $\text{INF}$  that simply goes into an infinite loop satisfies  $\text{MONOTONE}(\text{INF}) = 1$ , since there are no inputs  $x, x'$  on which  $\text{INF}(x) = 1$  and  $\text{INF}(x') = 1$ .
- The program  $\text{PAR}$  that we've seen, which computes the XOR or parity of its input, is not monotone (e.g.,  $\text{PAR}(1,1,0,0,\dots,0) = 0$  but  $\text{PAR}(1,0,0,\dots,0) = 0$ ) and hence  $\text{MONOTONE}(\text{PAR}) = 0$ .

(It is important to note that in the above we talk about *programs*  $\text{INF}$  and  $\text{PAR}$  and not the corresponding functions that they compute.)

We will now give a reduction from  $\text{HALTONZERO}$  to  $\text{MONOTONE}$ . That is, we assume towards a contradiction that there exists an algorithm  $A$  that computes  $\text{MONOTONE}$  and we will build an algorithm  $B$  that computes  $\text{HALTONZERO}$ . Our algorithm  $B$  will work as follows:

1. On input a program  $P \in \{0,1\}^*$ ,  $B$  will construct the following program  $Q$ : "on input  $z \in \{0,1\}^*$  do: a. Run  $P(0)$ , b. Return  $\text{PAR}(z)$ ".
2.  $B$  will then return the value  $1 - A(Q)$ .

To complete the proof we need to show that  $B$  outputs

the correct answer, under our assumption that  $A$  computes  $\text{MONOTONE}$ . In other words, we need to show that  $\text{HALTONZERO}(P) = 1 - \text{MONOTONE}(Q)$ . However, note that if  $P$  does *not* halt on zero, then the program  $Q$  enters into an infinite loop in step a. and will never reach step b. Hence in this case the program  $Q$  has the same functionality as  $\text{INF}$ .<sup>5</sup> Thus,  $\text{MONOTONE}(Q) = \text{MONOTONE}(\text{INF}) = 1$ . If  $P$  *does* halt on zero, then step a. in  $Q$  will eventually conclude and  $Q$ 's output will be determined by step b., where it simply outputs the parity of its input. Hence in this case,  $Q$  computes the non-monotone parity function, and we get that  $\text{MONOTONE}(Q) = \text{MONOTONE}(\text{PAR}) = 0$ . In both cases we see that  $\text{MONOTONE}(Q) = 1 - \text{HALTONZERO}(P)$ , which is what we wanted to prove. An examination of this proof shows that we did not use anything about  $\text{MONOTONE}$  beyond the fact that it is semantic and non-trivial (in the sense that it is not the all zero, nor the all-ones function). ■

<sup>5</sup> Note that the program  $Q$  has different code than  $\text{INF}$ . It is not the same program, but it does have the same behavior (in this case) of never halting on any input.

### 10.3 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.

### 10.4 Exercises

**Exercise 10.1 — 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.<sup>6</sup> ■

<sup>6</sup> Hint: See Christopher Strachey's letter in the biographical notes.

## 10.5 Bibliographical notes

7

The diagonalization argument used to prove uncomputability of  $F^*$  is of course derived from Cantor's argument for the uncountability of the reals. In a twist of fate, using techniques originating from the works Gödel and Turing, Paul Cohen showed in 1963 that Cantor's 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". See [here](#) for recent progress on a related question.

<sup>7</sup> 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.

## 10.6 Further explorations

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

## 10.7 Acknowledgements



# 11

## Restricted computational models

*"Happy families are all alike; every unhappy family is unhappy in its own way", Leo Tolstoy (opening of the book "Anna Karenina").*

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  $\lambda$  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 11.1 — 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.



Once again, this is a good point for you to stop and try to prove the result yourself before reading the proof below.

*Proof.* We have seen in [Theorem 9.1](#) that for every NAND++ program  $P$  there is an equivalent Turing machine  $M_P$  such that for every  $x$ , that computes the same function. The machine  $M_P$  exactly simulated  $P$ , in the sense that  $M_P$  halts on  $x$  if and only  $P$  halts on  $x$  (and moreover if they both halt, they produce the same output). Going back to the proof of [Theorem 9.1](#), we can see that the transformation

### Learning Objectives:

- See that Turing completeness is not always a good thing
- Two important examples of non-Turing-complete, always-halting formalisms: *regular expressions* and *context-free grammars*.
- The pumping lemmas for both these formalisms, and examples of non regular and non context-free functions.
- Unrestricted grammars, and another example of an uncomputable function.

of the program  $P$  to the Turing machine  $M(P)$  was described in a *constructive* way.

Specifically, we gave explicit instructions how to build the Turing machine  $M(P)$  given the description of the program  $P$ . Thus, we can view the proof of [Theorem 9.1](#) as a high level description of an *algorithm* to obtain  $M_P$  from the program  $P$ , and using our “have your cake and eat it too” paradigm, this means that there exists also a NAND++ program  $R$  such that computes the map  $P \mapsto M_P$ . We see that

$$\text{HALT}(P, x) = \text{TMHALT}(M_P, x) = \text{TMHALT}(R(P), x) \quad (11.1)$$

and hence if we assume (towards the sake of a contradiction) that  $\text{TMHALT}$  is computable then [Eq. \(11.1\)](#) implies that  $\text{HALT}$  is computable, hence contradicting [Theorem 10.2](#). ■

The same proof carries over to other computational models such as the  $\lambda$  calculus, two dimensional (or even one-dimensional) automata etc. Hence for example, there is no algorithm to decide if a  $\lambda$  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. In this lecture we will discuss several such examples.

### 11.1 Turing completeness as a bug

We have seen that seemingly simple computational models or systems can turn out to be Turing complete. The [following webpage](#) lists several examples of formalisms that “accidentally” turned out to be Turing complete, including supposedly limited languages such as the C preprocessor, CCS, SQL, sendmail configuration, as well as games such as Minecraft, Super Mario, and the card game “Magic: The gathering”. This is not always a good thing, as it means that such formalisms can give rise to arbitrarily complex behavior. For example, the postscript format (a precursor of PDF) is a Turing-complete programming language meant to describe documents for printing. The expressive power of postscript can allow for short description of very complex images. But it also gives rise to some nasty surprises, such

as the attacks described in [this page](#) ranging from using infinite loops as a denial of service attack, to accessing the printer's file system.

An interesting recent example of the pitfalls of Turing-completeness arose in the context of the cryptocurrency [Ethereum](#). The distinguishing feature of this currency is the ability to design "smart contracts" using an expressive (and in particular Turing-complete) language.

In our current "human operated" economy, Alice and Bob might sign a contract to agree that if condition  $X$  happens then they will jointly invest in Charlie's company. Ethereum allows Alice and Bob to create a joint venture where Alice and Bob pool their funds together into an account that will be governed by some program  $P$  that decides under what conditions it disburses funds from it. For example, one could imagine a piece of code that interacts between Alice, Bob, and some program running on Bob's car that allows Alice to rent out Bob's car without any human intervention or overhead.

Specifically Ethereum uses the Turing-complete programming [solidity](#) which has a syntax similar to Javascript. The flagship of Ethereum was an experiment known as The "Decentralized Autonomous 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, where 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.

## 11.2 Regular expressions

One of the most common tasks in computing is to *search* for a piece of text. At its heart, the *search problem* is quite simple. The user gives out a function  $F : \{0,1\}^* \rightarrow \{0,1\}$ , and the system applies this function to a set of candidates  $\{x_0, \dots, x_k\}$ , returning all the  $x_i$ 's such that  $F(x_i) = 1$ . However, we typically do not want the system to get into an infinite loop just trying to evaluate this function! For this reason, such systems often do not allow the user to specify an *arbitrary* function using some Turing-complete formalism, but rather a function that is described by a restricted computational model, and in particular one in which all functions halt. One of the most popular models for this application is the model of **regular expressions**. You have probably come across regular expressions if you ever used an advanced text editor, a command line shell, or have done any kind of manipulations of text files.<sup>1</sup>

A *regular expression* over some alphabet  $\Sigma$  is obtained by combining elements of  $\Sigma$  with the operation of concatenation, as well as  $|$  (corresponding to *or*) and  $*$  (corresponding to repetition zero or more times).<sup>2</sup> For example, the following regular expression over the alphabet  $\{0,1\}$  corresponds to the set of all even length strings  $x \in \{0,1\}^*$  where the digit at location  $2i$  is the same as the one at location  $2i + 1$  for every  $i$ :

$$(00|11)* \quad (11.2)$$

The following regular expression over the alphabet  $\{a, b, c, d, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  corresponds to the set of all strings that consist of a sequence of one or more letters  $a$ - $b$  followed by a sequence of one or more digits (without a leading zero):

$$(a|b|c|d)(a|b|c|d)^*(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^* \quad (11.3)$$

<sup>1</sup> Sections 1.3 and 1.4 in [Sipser's book](#) are excellent resources for regular expressions. Sipser's book also discusses the equivalence of regular expressions with *finite automata*.

<sup>2</sup> Common implementations of regular expressions in programming languages and shells typically include some extra operations on top of  $|$  and  $*$ , but these can all be implemented as “syntactic sugar” using the operators  $|$  and  $*$ .

Formally, regular expressions are defined by the following recursive definition:<sup>3</sup>

**Definition 11.1 — Regular expression.** A *regular expression*  $\text{exp}$  over an alphabet  $\Sigma$  is a string over  $\Sigma \cup \{"(,)", "|", "\ast"\}$  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. (We often drop the parenthesis when there is no danger of confusion and so write this as  $\text{exp exp}'$ .)

4.  $\text{exp} = (\text{exp}')\ast$  where  $\text{exp}'$  is a regular expression.

Finally we also allow the following “edge cases”:  $\text{exp} = \emptyset$  and  $\text{exp} = ""$ .<sup>4</sup>

Every regular expression  $\text{exp}$  computes a function  $\Phi_{\text{exp}} : \Sigma^* \rightarrow \{0, 1\}$ , where  $\Phi_{\text{exp}}(x) = 1$  if  $x$  matches the regular expression. So, for example if  $\text{exp} = (00|11)^*$  then  $\Phi_{\text{exp}}(x) = 1$  if and only if  $x$  is of even length and  $x_{2i} = x_{2i+1}$  for every  $i < |x|/2$ . Formally, the function is 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  $\Phi_{\text{exp}}(x) = \Phi_{\text{exp}'}(x) \vee \Phi_{\text{exp}''}(x)$  where  $\vee$  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  $\text{exp} = (\text{exp}')\ast$  then  $\Phi_{\text{exp}}(x) = 1$  iff there are is  $k \in \mathbb{N}$  and some  $x_0, \dots, x_{k-1} \in \Sigma^*$  such that  $x$  is the concatenation  $x_0 \cdots x_{k-1}$  and  $\Phi_{\text{exp}'}(x_i) = 1$  for every  $i \in [k]$ .
5. Finally, for the edge cases  $\Phi_\emptyset$  is the constant zero function, and  $\Phi_{""}$  is the function that only outputs 1 on the constant string.

We say that a function  $F : \Sigma^* \rightarrow \{0, 1\}$  is *regular* if  $F = \Phi_{\text{exp}}$  for some regular expression  $\text{exp}$ . We say that a set  $L \subseteq \Sigma^*$  (also known as a *language*) is *regular* if the function  $F$  s.t.  $F(x) = 1$  iff  $x \in L$  is regular.

<sup>3</sup> Just like recursive functions, we can define a concept recursively. A definition of some class  $\mathcal{C}$  of objects can be thought of as defining a function that maps an object  $o$  to either *VALID* or *INVALID* depending on whether  $o \in \mathcal{C}$ . Thus we can think of **Definition 11.1** as defining a recursive function that maps a string  $\text{exp}$  over  $\Sigma \cup \{"(,)", "|", "\ast"\}$  to *VALID* or *INVALID* depending on whether  $\text{exp}$  describes a valid regular expression.

<sup>4</sup> These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.



The definitions above might not be easy to grasp in a first read, so you should probably pause here and go over it again until you understand why it corresponds to our intuitive notion of regular expressions. This is important not just for understanding regular

expressions themselves (which are used time and again in a great many applications) but also for getting better at understanding recursive definitions in general.

We can think of regular expressions as a type of “programming language” that defines functions  $\exp : \Sigma^* \rightarrow \{0, 1\}$ .<sup>5</sup> But it turns out that the “halting problem” for these functions is easy: they always halt.

**Theorem 11.2 — Regular expression always halt.** For every set  $\Sigma$  and  $\exp \in (\Sigma \cup \{"(", ")", "|", "\star"\})^*$ , 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 11.1 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  $\exp$  (i.e., the number of symbols in the string  $\exp$ , also denoted as  $|\exp|$ ). For  $m = 1$ ,  $\exp$  is a single alhpabet 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  $|\exp| = 1, \dots, m - 1$ . 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 11.2 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. One way to obtain such an algorithm is to replace this recursive algorithm with **dynamic programming**, using the technique of **memoization**.<sup>6</sup> It turns out that the resulting dynamic program only requires maintaining a finite (independent of the input length) amount of state, and hence it can be viewed as a **finite state machine** or finite automata. The relation of regular expressions with

<sup>5</sup> Regular expressions (and context free grammars, which we'll see below) are often thought of as *generative models* rather than computational ones, since their definition does not immediately give rise to a way to *decide* matches but rather to a way to generate matching strings by repeatedly choosing which rules to apply.

<sup>6</sup> If you haven't taken yet an algorithms course such as CS 124, you might not know these techniques. This is OK, as, while the more efficient algorithm is crucial for the many practical applications of regular expressions, is not of great importance to this course.

finite automate is a beautiful topic, and one we may return to later in this course.

The fact that functions computed by regular expressions always halt is of course 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:

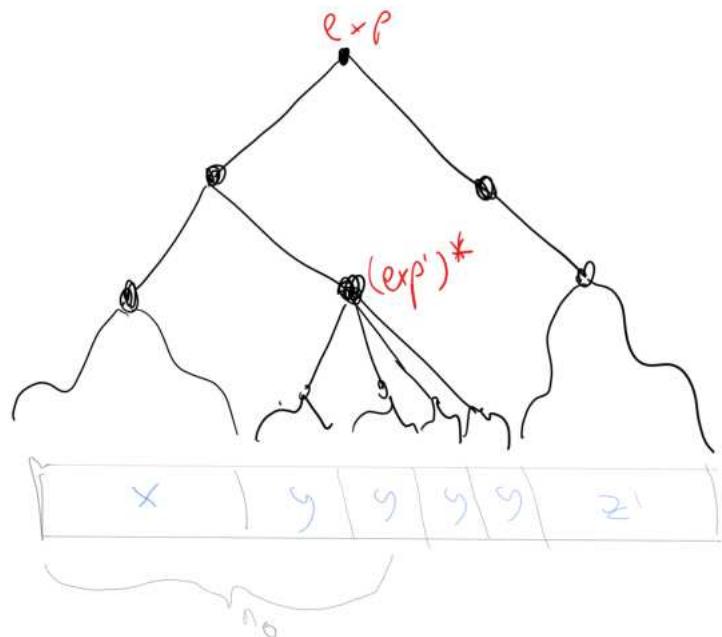
**Theorem 11.3 — Matching parenthesis.** Let  $\Sigma = \{\langle, \rangle\}$  and  $MATCHPAREN : \Sigma^* \rightarrow \{0, 1\}$  be the function that given a string of parenthesis, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over  $\Sigma$  that computes  $MATCHPAREN$ .

Theorem 11.3 is a consequence of the following result known as the *pumping lemma*:

**Theorem 11.4 — Pumping Lemma.** Let  $exp$  be a regular expression. Then there is some number  $n_0$  such that for every  $w \in \{0, 1\}^*$  with  $|w| > n_0$  and  $\Phi_{exp}(w) = 1$ , it holds that we can write  $w = xyz$  where  $|y| \geq 1$ ,  $|xy| \leq n_0$  and such that  $\Phi_{exp}(xy^kz) = 1$  for every  $k \in \mathbb{N}$ .

**Proof Idea:** The idea behind the proof is very simple (see Fig. 11.1). If we let  $n_0$  be, say, twice the number of symbols that are used in the expression  $exp$ , then the only way that there is some  $w$  with  $|w| > n_0$  and  $\Phi_{exp}(w) = 1$  is that  $exp$  contains the \* (i.e. star) operator and that there is a nonempty substring  $y$  of  $w$  that was matched by  $(exp')^*$  for some sub-expression  $exp'$  of  $exp$ . We can now repeat  $y$  any number of times and still get a matching string.

*Proof of Theorem 11.4.* To prove the lemma formally, we use induction on the length of the expression. Like all induction proofs, this is going to be somewhat lengthy, but at the end of the day it directly follows the intuition above that *somewhere* we must have used the star operation. Reading this proof, and in particular understanding how the formal proof below corresponds to the intuitive idea above,



**Figure 11.1:** To prove the “pumping lemma” we look at a word  $w$  that is much larger than the regular expression  $\text{exp}$  that matches it. In such a case, part of  $w$  must be matched by some sub-expression of the form  $(\text{exp}')^*$ , since this is the only operator that allows matching words longer than the expression. If we look at the “leftmost” such sub-expression and define  $y^k$  to be the string that is matched by it, we obtain the partition needed for the pumping lemma.

is a very good way to get more comfort with inductive proofs of this form.

Our inductive hypothesis is that for an  $n$  length expression,  $n_0 = 2n$  satisfies the conditions of the lemma. The base case is when the expression is a single symbol or that it is  $\emptyset$  or  $''$  in which case the condition is satisfied just because there is no matching string of length more than one. Otherwise,  $exp$  is of the form **(a)**  $exp'|exp''$ , **(b)**,  $(exp')(exp'')$ , **(c)** or  $(exp'')^*$  where in all these cases the subexpressions have fewer symbols than  $exp$  and hence satisfy the induction hypothesis.

In case **(a)**, every string  $w$  matching  $exp$  must match either  $exp'$  or  $exp''$ . In the former case, since  $exp'$  satisfies the induction hypothesis, if  $|w| > n_0$  then we can write  $w = xyz$  such that  $xy^kz$  matches  $exp'$  for every  $k$ , and hence this is matched by  $exp$  as well.

In case **(b)**, if  $w$  matches  $(exp')(exp'')$ , then we can write  $w = w'w''$  where  $w'$  matches  $exp'$  and  $w''$  matches  $exp''$ . Again we split to subcases. If  $|w'| > 2|exp'|$ , then by the induction hypothesis we can write  $w' = xyz$  of the form above such that  $xy^kz$  matches  $exp'$  for every  $k$  and then  $xy^kzw''$  matches  $(exp')(exp'')$ . This completes the proof since  $|xy| \leq 2|exp'|$  and so in particular  $|xy| \leq 2(|exp'| + |exp''|) \leq 2|exp|$ , and hence  $zw''$  can play the role of  $z$  in the proof. Otherwise, if  $|w'| \leq 2|exp'|$  then since  $|w|$  is larger than  $2|exp|$  and  $w = w'w''$  and  $exp = exp'exp''$ , we get that  $|w'| + |w''| > 2(|exp'| + |exp''|)$ . Thus, if  $|w'| \leq 2|exp'|$  it must be that  $|w''| > 2|exp''|$  and hence by the induction hypothesis we can write  $w'' = xyz$  such that  $xy^kz$  matches  $exp''$  for every  $k$  and  $|xy| \leq 2|exp''|$ . Therefore we get that  $w'xy^kz$  matches  $(exp')(exp'')$  for every  $k$  and since  $|w'| \leq 2|exp'|$ ,  $|w'xy| \leq 2(|exp'| + |exp'|)$  and this completes the proof since  $w'x$  can play the role of  $x$  in the statement.

Now in the case **(c)**, if  $w$  matches  $(exp')^*$  then  $w = w_0 \cdots w_t$  where  $w_i$  is a nonempty string that matches  $exp'$  for every  $i$ . If  $|w_0| > 2|exp'|$  then we can use the same approach as in the concatenation case above. Otherwise, we simply note that if  $x$  is the empty string,  $y = w_0$ , and  $z = w_1 \cdots w_t$  then  $xy^kz$  will match  $(exp')^*$  for every  $k$ . ■



**Recursive definitions and inductive proofs** When an object is *recursively defined* (as in the case of regular expressions) then it is natural to prove properties of such objects by *induction*. That is, if we want to prove that all objects of this type have property  $P$ , then it is natural to use an inductive steps that says

that if  $o', o'', o'''$  etc have property  $P$  then so is an object  $o$  that is obtained by composing them.

Given the pumping lemma, we can easily prove [Theorem 11.3](#):

*Proof of Theorem 11.3.* Suppose, towards the sake of contradiction, that there is an expression  $\text{exp}$  such that  $\Phi_{\text{exp}} = \text{MATCHPAREN}$ . Let  $n_0$  be the number from [Theorem 11.3](#) and let  $w = \langle^{n_0} \rangle^{n_0}$  (i.e.,  $n_0$  left parenthesis followed by  $n_0$  right parenthesis). Then we see that if we write  $w = xyz$  as in [Theorem 11.3](#), the condition  $|xy| \leq n_0$  implies that  $y$  consists solely of left parenthesis. Hence the string  $xy^2z$  will contain more left parenthesis than right parenthesis. Hence  $\text{MATCHPAREN}(xy^2z) = 0$  but by the pumping lemma  $\Phi_{\text{exp}}(xy^2z) = 1$ , contradicting our assumption that  $\Phi_{\text{exp}} = \text{MATCHPAREN}$ . ■



**Regular expressions beyond searching** Regular expressions are widely used beyond just searching. First, they are typically used to define *tokens* in various formalisms such as programming data description languages. But they are also used beyond it. One nice example is the recent work on the [NetKAT network programming language](#). In recent years, the world of networking moved from fixed topologies to “software defined networks”, that are run by programmable switches that can implement policies such as “if packet is SSL then forward it to A, otherwise forward it to B”. By its nature, one would want to use a formalism for such policies that is guaranteed to always halt (and quickly!) and that where it is possible to answer semantic questions such as “does C see the packets moved from A to B” etc. The NetKAT language uses a variant of regular expressions to achieve that.

### 11.3 Context free grammars.

If you have ever written a program, you’ve experienced a *syntax error*. You might have had also the experience of your program entering into an *infinite loop*. What is less likely is that the compiler or interpreter entered an infinite loop when trying to figure out if your program has a syntax error. When a person designs a programming language, they need to come up with a function  $\text{VALID} : \{0, 1\}^* \rightarrow \{0, 1\}$  that determines the strings that correspond to valid programs in this language. The compiler or interpreter computes  $\text{VALID}$  on

the string corresponding to your source code to determine if there is a syntax error. To ensure that the compiler will always halt in this computation, language designers typically *don't* use a general Turing-complete mechanism to express the function *VALID*, but rather a restricted computational model. One of the most popular choices for such a model is *context free grammar*.

To explain context free grammars, let's begin with a canonical example. Let us try to define a function *ARITH* :

$\Sigma^* \rightarrow \{0,1\}$  that takes as input a string  $x$  over the alphabet  $\Sigma = \{(,), +, -, \times, \div, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$  and returns 1 if and only if the string  $x$  represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation to smaller expressions, or enclosing them in parenthesis, where the “base case” corresponds to expressions that are simply numbers. A bit more precisely, we can make the following definitions:

- A *number* is either 0 or a sequence of digits not starting with 0.
- An *operation* is one of  $+, -, \times, \div$
- An *expression* has either the form “*number*” or the form “*subexpression1 operation subexpression2*” or “*(subexpression)*”.

A context free grammar is a formal way of specifying such conditions.<sup>7</sup>

**Definition 11.2 — Context Free Grammar.** Let  $\Sigma$  be some finite set. A *context free grammar (CFG) over  $\Sigma$*  is a triple  $(V, R, s)$  where  $V$  is a set disjoint from  $\Sigma$  of *variables*,  $R$  is a set of *rules*, which are pairs  $(v, z)$  (which we will write as  $v \Rightarrow z$ ) where  $v \in V$  and  $z \in (\Sigma \cup V)^*$ , and  $s \in V$  is the starting rule.

If  $G = (V, R, s)$  is a context-free grammar over  $\Sigma$ , then for two strings  $\alpha, \beta \in (\Sigma \cup V)^*$  we say that  $\beta$  *can be derived in one step* from  $\alpha$ , denoted by  $\alpha \Rightarrow_G \beta$ , if we can obtain  $\beta$  from  $\alpha$  by applying one of the rules of  $G$ . That is, we obtain  $\beta$  by replacing in  $\alpha$  one occurrence of the variable  $v$  with the string  $z$ , where  $v \Rightarrow z$  is a rule of  $G$ . We say that  $\beta$  *can be derived* from  $\alpha$ , denoted by  $\alpha \Rightarrow_G^* \beta$ , if it can be derived by some finite number  $k$  of steps. That is, if there are  $\alpha_1, \dots, \alpha_{k-1} \in (\Sigma \cup V)^*$ , so that  $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \dots \Rightarrow_G \alpha_{k-1} \Rightarrow_G \beta$ .

We define the *function computed by  $(V, R, s)$*  to be the map  $\Phi_{V, R, s} : \Sigma^* \rightarrow \{0, 1\}$  such that  $\Phi_{V, R, s}(x) = 1$  iff  $s \Rightarrow_G^* x$ . We say that  $F : \Sigma^* \rightarrow \{0, 1\}$  is *context free* if  $F = \Phi_{V, R, s}$  for some CFG  $(V, R, s)$  we say that a set  $L \subseteq \Sigma^*$  (also known as a *language*) is *context free* if the function  $F$  such that  $F(x) = 1$  iff  $x \in L$  is context free.

<sup>7</sup> Sections 2.1 and 2.3 in Sipser's book are excellent resources for context free grammars.

The example above of well-formed arithmetic expressions can be captured formally by the following context free grammar:

- The alphabet  $\Sigma$  is  $\{((), +, -, \times, \div, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- The variables are  $V = \{expression, operation\}$ .
- The rules are the set  $R$  containing the following pairs:<sup>8</sup>
- $number \Rightarrow 0$  and  $number \Rightarrow 1number, \dots, number \Rightarrow 9number$
- $expression \Rightarrow expression + expression, expression \Rightarrow expression - expression, expression \Rightarrow expression \times expression, (expression \Rightarrow expression \div expression, expression \Rightarrow 0, (expression \Rightarrow 1number, \dots, expression \Rightarrow 9number.$
- The starting variable is  $expression$

There are various notations to write context free grammars in the literature, with one of the most common being **Backus–Naur form** where we write a rule of the form  $v \Rightarrow a$  (where  $v$  is a variable and  $a$  is a string) in the form  $\langle v \rangle := a$ . If we have several rules of the form  $v \mapsto a, v \mapsto b$ , and  $v \mapsto c$  then we can combine them as  $\langle v \rangle := a | b | c$  (and this similarly extends for the case of more rules).

A priori it might not be clear that the map  $\Phi_{V,R,s}$  is computable, but it turns out that we can in fact compute it. That is, the “halting problem” for context free grammars is trivial, or in other words, we have the following theorem:

**Theorem 11.5 — Context-free grammars always halt.** For every CFG  $(V, R, s)$  over  $\Sigma$ , the function  $\Phi_{V,R,s} : \Sigma^* \rightarrow \{0, 1\}$  is computable.<sup>9</sup>

*Proof.* We only sketch the proof. It turns out that we can convert every CFG to an equivalent version that has the so called *Chomsky normal form*, where all rules either have the form  $u \rightarrow vw$  for variables  $u, v, w$  or the form  $u \rightarrow \sigma$  for a variable  $u$  and symbol  $\sigma \in \Sigma$ , plus potentially the rule  $s \rightarrow " "$  where  $s$  is the starting variable. (The idea behind such a transformation is to simply add new variables as needed, and so for example we can translate a rule such as  $v \rightarrow u\sigma w$  into the three rules  $v \rightarrow ur, r \rightarrow tw$  and  $t \rightarrow \sigma$ .)

Using this form we get a natural recursive algorithm for computing whether  $s \Rightarrow_G^* x$  for a given grammar  $G$  and string  $x$ . We simply try all possible guesses for the first rule  $s \rightarrow uv$  that is used in such a derivation, and then all possible ways to partition  $x$  as a concatenation  $x = x'x''$ . If we guessed the rule and the partition correctly, then this reduces our task to checking whether  $u \Rightarrow_G^* x'$  and  $v \Rightarrow_G^* x''$ ,

<sup>8</sup> For the sake of clarity, we use quotation marks “..” to enclose the string which is the second pair of each rule. Also, note that our rules below, slightly differ from those illustrated above. The two sets of rules compute the same function, but the description below has been modified to be an equivalent form of the rules above that doesn’t have a rule whose right-hand side is only variables. We could have also relaxed the condition of containing an alphabet symbol by only requiring that rules with no alphabet symbols induce a *directed acyclic graph* over the variables.

<sup>9</sup> While formally we only defined computability of functions over  $\{0, 1\}^*$ , we can extend the definition to functions over any finite  $\Sigma^*$  by using any one-to-one encoding of  $\Sigma$  into  $\{0, 1\}^k$  for some finite  $k$ . It is a (good!) exercise to verify that if a function is computable with respect to one such encoding, then it is computable with respect to them all.

which (as it involves shorter strings) can be done recursively. The base cases are when  $x$  is empty or a single symbol, and can be easily handled. ■

While we can (and people do) talk about context free grammars over any alphabet  $\Sigma$ , in the following we will restrict ourselves to  $\Sigma = \{0, 1\}$ . This is of course not a big restriction, as any finite alphabet  $\Sigma$  can be encoded as strings of some finite size. It turns out that context free grammars can capture every regular expression:

**Theorem 11.6 — Context free grammars and regular expressions.** Let  $exp$  be a regular expression over  $\{0, 1\}$ , then there is a CFG  $(V, R, s)$  over  $\{0, 1\}$  such that  $\Phi_{V,R,s} = \Phi_{exp}$ .

*Proof.* We will prove this by induction on the length of  $exp$ . If  $exp$  is an expression of one bit length, then  $exp = 0$  or  $exp = 1$ , in which case we leave it to the reader to verify that there is a (trivial) CFG that computes it. Otherwise, we fall into one of the following case: **case 1:**  $exp = exp'exp''$ , **case 2:**  $exp = exp'|exp''$  or **case 3:**  $exp = (exp')^*$  where in all cases  $exp', exp''$  are shorter regular expressions. By the induction hypothesis have grammars  $(V', R', s')$  and  $(V'', R'', s'')$  that compute  $\Phi_{exp'}$  and  $\Phi_{exp''}$  respectively. By renaming of variables, we can also assume without loss of generality that  $V'$  and  $V''$  are disjoint.

In case 1, we can define the new grammar as follows: we add a new starting variable  $s \notin V \cup V'$  and the rule  $s \mapsto s's''$ . In case 2, we can define the new grammar as follows: we add a new starting variable  $s \notin V \cup V'$  and the rules  $s \mapsto s'$  and  $s \mapsto s''$ . Case 3 will be the only one that uses *recursion*. As before we add a new starting variable  $s \notin V \cup V'$ , but now add the rules  $s \mapsto ""$  (i.e., the empty string) and also add for every rule of the form  $(s', \alpha) \in R'$  the rule  $s \mapsto s\alpha$  to  $R$ .

We leave it to the reader as (again a very good!) exercise to verify that in all three cases the grammars we produce capture the same function as the original expression. ■

It turns out that CFG's are strictly more powerful than regular expressions. In particular, the “matching parenthesis” function can be computed by a context free grammar. Specifically, consider the grammar  $(V, R, s)$  where  $V = \{s\}$  and  $R$  is  $s \mapsto "", s \mapsto (s)s$ , and  $s \mapsto s(s)$ . It is not hard to see that it captures exactly the set of strings over  $\{(., )\}$  that correspond to matching parenthesis. However, there are some simple languages that are *not* captured by context free grammars, as can be shown via the following version of [Theorem 11.4](#).

**Theorem 11.7 — Context-free pumping lemma.** Let  $(V, R, s)$  be a CFG over  $\Sigma$ , then there is some  $n_0 \in \mathbb{N}$  such that for every  $x \in \Sigma^*$  with  $|\sigma| > n_0$ , if  $\Phi_{V,R,s}(x) = 1$  then  $x = abcde$  such that  $|b| + |c| + |d| \leq n_1$ ,  $|b| + |d| \geq 1$ , and  $\Phi_{V,R,s}(ab^kcd^ke) = 1$  for every  $k \in \mathbb{N}$ .

*Proof.* We only sketch the proof. The idea is that if the total number of symbols in the rules  $R$  is  $k_0$ , then the only way to get  $|x| > k_0$  with  $\Phi_{V,R,s}(x) = 1$  is to use *recursion*. That is there must be some  $v \in V$  such that by a sequence of rules we are able to derive from  $v$  the value  $bvd$  for some strings  $b, d \in \Sigma^*$  and then further on derive from  $v$  the string  $c \in \Sigma^*$  such that  $bcd$  is a substring of  $x$ . If try to take the minimal such  $v$  then we can ensure that  $|bcd|$  is at most some constant depending on  $k_0$  and we can set  $n_0$  to be that constant ( $n_0 = 10|R|k_0$  will do, since we will not need more than  $|R|$  applications of rules, and each such application can grow the string by at most  $k_0$  symbols). Thus by the definition of the grammar, we can repeat the derivation to replace the substring  $bcd$  in  $x$  with  $b^kcd^k$  for every  $k \in \mathbb{N}$  while retaining the property that the output of  $\Phi_{V,R,s}$  is still one. ■

Using Theorem 11.7 one can show that even the simple function  $F(x) = 1$  iff  $x = ww$  for some  $w \in \{0, 1\}^*$  is not context free. (In contrast, the function  $F(x) = 1$  iff  $x = ww^R$  for  $w \in \{0, 1\}^*$  where for  $w \in \{0, 1\}^n$ ,  $w^R = w_{n-1}w_{n-2} \cdots w_0$  is context free, can you see why?).



**Parse trees** While we present CFGs as merely deciding whether the syntax is correct or not, the algorithm to compute  $\Phi_{V,R,s}$  actually gives more information than that. That is, on input a string  $x$ , if  $\Phi_{V,R,s}(x) = 1$  then the algorithm yields the sequence of rules that one can apply from the starting vertex  $s$  to obtain the final string  $x$ . We can think of these rules as determining a connected directed acyclic graph (i.e., a *tree*) with  $s$  being a source (or *root*) vertex and the sinks (or *leaves*) corresponding to the substrings of  $x$  that are obtained by the rules that do not have a variable in their second element. This tree is known as the *parse tree* of  $x$ , and often yields very useful information about the structure of  $x$ . Often the first step in a compiler or interpreter for a programming language is a *parser* that transforms the source into the parse tree (often known in this context as the *abstract syntax tree*). There are also tools that can automatically convert a description of a context-free grammars into a *parser* algorithm that computes the parse tree of a given string. (Indeed,

the above recursive algorithm can be used to achieve this, but there are much more efficient versions, especially for grammars that have **particular forms**, and programming language designers often try to ensure their languages have these more efficient grammars.)

## 11.4 Unrestricted grammars

The reason we call context free grammars “context free” is because if we have a rule of the form  $v \mapsto a$  it means that we can always replace  $v$  with the string  $a$ , no matter the *context* in which  $v$  appears. More generally, we want to consider cases where our replacement rules depend on the context.<sup>10</sup> This gives rise to the notion of *general grammars* that allow rules of the form  $(a, b)$  where both  $a$  and  $b$  are strings over  $(V \cup \Sigma)^*$ . The idea is that if, for example, we wanted to enforce the condition that we only apply some rule such as  $v \mapsto 0w1$  when  $v$  is surrounded by three zeroes on both sides, then we could do so by adding a rule of the form  $000v000 \mapsto 0000w1000$  (and of course we can add much more general conditions).

<sup>10</sup> TODO: add example

TO BE CONTINUED, UNDECIDABILITY OF GENERAL GRAMMARS.

## 11.5 Lecture summary

- The uncomputability of the Halting problem for general models motivates the definition of restricted computational models.
- In restricted models we might be able to answer questions such as: does a given program terminate, do two programs compute the same function?

## 11.6 Exercises

## 11.7 Bibliographical notes

<sup>11</sup>

<sup>11</sup> 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.

### *11.8 Further explorations*

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

### *11.9 Acknowledgements*

### Learning Objectives:

- See more examples of uncomputable functions that are not as tied to computation.
- See Gödel's incompleteness theorem - a result that shook the world of mathematics in the early 20th century.

## 12

# 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.*

### 12.1 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 Babylonians 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.<sup>1</sup> 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:

**Theorem 12.1 — MRDP Theorem.** Let  $SOLVE : \{0,1\}^* \rightarrow \{0,1\}^*$  be the function that takes as input a multivariate polynomial with integer coefficients  $P : \mathbb{R}^k \rightarrow \mathbb{R}$  for  $k \leq 100$  and outputs either  $(x_1, \dots, x_k) \in \mathbb{N}^k$  s.t.  $P(x_1, \dots, x_k) = 0$  or the string `no solution` if no  $P$  does not have non-negative integer roots.<sup>2</sup> Then  $SOLVE$  is uncomputable. Moreover, this holds even for the easier function  $HASSOL : \{0,1\}^* \rightarrow \{0,1\}$  that given such a polynomial  $P$  outputs

<sup>1</sup> 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.

1 if there are  $x_1, \dots, x_k \in \mathbb{N}$  s.t.  $P(x_1, \dots, x_k) = 0$  and 0 otherwise.

The difficulty in finding a way to distinguish between “code” such as NAND++ programs, and “static content” such as polynomials is just another manifestation of the phenomenon that *code* is the same as *data*. While a fool-proof solution for distinguishing between the two is inherently impossible, finding heuristics that do a reasonable job keeps many firewall and anti-virus manufacturers very busy (and finding ways to bypass these tools keeps many hackers busy as well).

### 12.1.1 “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:<sup>3</sup>

$$\exists_{x_1, \dots, x_k \in \mathbb{N}} \text{s.t. } P(x_1, \dots, x_k) = 0. \quad (12.1)$$

**Theorem 12.1** states that there is no NAND++ program that can determine the truth of every statements of the form Eq. (12.1). 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 12.1** 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 12.1 — 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),  $\wedge$  (AND), and  $\vee$  (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 12.1** is interesting in its own right and not just as a “toy version” of **Theorem 12.1**. We often care deeply about determining the truth of quantified integer statements. For example, the statement

<sup>2</sup> As usual, we assume some standard way to express numbers and text as binary strings.

<sup>3</sup> Recall that  $\exists$  denotes the *existential quantifier*; that is, a statement of the form  $\exists_x \varphi(x)$  is true if there is *some* assignment for  $x$  that makes the Boolean function  $\varphi(x)$  true. The dual quantifier is the *universal quantifier*, denoted by  $\forall$ , where a statement  $\forall_x \varphi(x)$  is true if *every* assignment for  $x$  makes the Boolean function  $\varphi(x)$  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.

that Fermat's Last Theorem is true for  $n = 3$  can be phrased as the quantified integer statement

$$\neg \exists_{a \in \mathbb{N}} \exists_{b \in \mathbb{N}} \exists_{c \in \mathbb{N}} (a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (a \times a \times a + b \times b \times b = c \times c \times c) . \quad (12.2)$$

The twin prime conjecture, 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

$$\forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge \text{PRIME}(p) \wedge \text{PRIME}(p + 2) \quad (12.3)$$

where we replace an instance of  $\text{PRIME}(q)$  with the statement  $(q > 1) \wedge \forall_{a \in \mathbb{N}} \forall_{b \in \mathbb{N}} (a = 1) \vee (a = q) \vee \neg(a \times b = q)$ .

The claim (mentioned in Hilbert's quote above) that are infinitely many primes of the form  $p = 2^n + 1$  can be phrased as follows:

$$\begin{aligned} & \forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge \text{PRIME}(p) \wedge \\ & (\forall_{k \in \mathbb{N}} (k = 2) \vee \neg \text{PRIME}(k) \vee \neg \text{DIVIDES}(k, p - 1)) \end{aligned} \quad (12.4)$$

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 12.2 — Uncomputability of quantified integer statements.** Let  $QIS : \{0, 1\}^* \rightarrow \{0, 1\}$  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.<sup>4</sup> Then  $QIS$  is uncomputable.

Note that [Theorem 12.2](#) is an immediate corollary of [Theorem 12.1](#). Indeed, if you can compute  $QIS$  then you can compute  $HASSOL$  and hence if you *can't* compute  $HASSOL$  then you *can't* compute  $QIS$  either. But [Theorem 12.2](#) is easier (though not trivial) to prove, and we will provide the proof in the following section.

<sup>4</sup> 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  $\exists_{x \in \mathbb{N}} x = 1$ .

## 12.2 Proving the unsolvability of quantified integer statements.

In this section we will prove [Theorem 12.2](#). 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.

### 12.2.1 Quantified mixed statements and computation traces

As mentioned above, before proving [Theorem 12.2](#), 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 12.2 — 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),  $\wedge$  (AND), and  $\vee$  (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

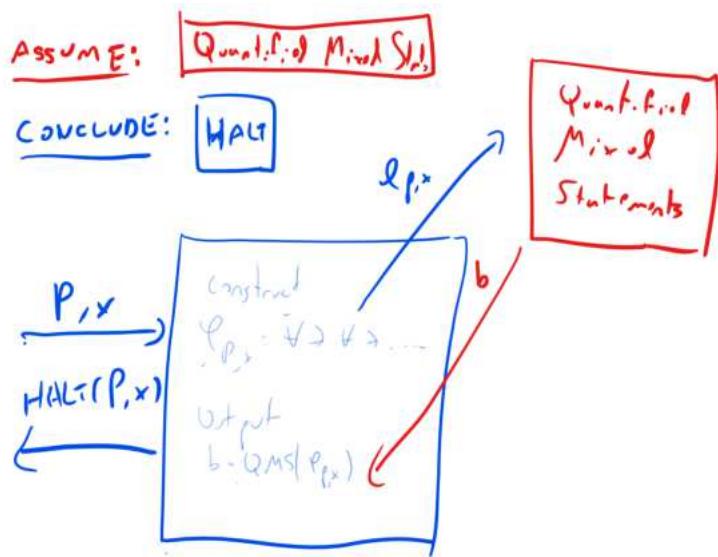
$$\forall_{a \in \{0,1\}^*} \exists_{b \in \{0,1\}^*} (|a| = |b|) \wedge (\forall_{i \in \mathbb{N}} (i > |a|) \vee (a_i \wedge b_{|a|-i}) \vee (\neg a_i \wedge \neg b_{|a|-i})). \quad (12.5)$$

Quantified mixed statements are a more general than quantified integer statements, and so the following theorem is potentially easier to prove than [Theorem 12.2](#):

**Theorem 12.3 — Uncomputability of quantified mixed statements.** Let  $\text{QMS} : \{0,1\}^* \rightarrow \{0,1\}$  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  $\text{QMS}$  is uncomputable.

### 12.2.2 “Unraveling” NAND++ programs and quantified mixed integer statements

We will first prove [Theorem 12.3](#) and then use it to prove [Theorem 12.2](#). The proof is again by reduction to  $\text{HALT}$  (see [Fig. 12.1](#)). That is, we do so by giving a program that transforms any NAND++ program  $P$  and input  $x$  into a quantified mixed statement  $\varphi_{P,x}$  such that  $\varphi_{P,x}$  is true if and only if  $P$  halts on input  $x$ . This will complete the proof, since it will imply that if  $\text{QMS}$  is computable then so is the  $\text{HALT}$  problem, which we have already shown is uncomputable.



**Figure 12.1:** We prove that  $\text{QMS}$  is uncomputable by giving a reduction that maps every pair  $(P, x)$  into a quantified mixed statements  $\varphi_{P,x}$  that is true if and only if  $P$  halts on  $x$ .

The idea behind the construction of the statement  $\varphi_{P,x}$  is the following. The statement will be true if and only if there exists a string  $\Delta \in \{0,1\}^*$  which corresponds to a summary of 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.

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 on [nandpl.org](http://nandpl.org) the parity program

```
tmp_1 := seen_i NAND seen_i
tmp_2 := x_i NAND tmp_1
val := tmp_2 NAND tmp_2
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 the following text (truncated here, since it is not the most riveting of reading material):

```
Executing command "tmp_1:=seen_iNANDseen_i", seen_0 has
    value 0, seen_0 has value 0, tmp_1 assigned value 1
Executing command "tmp_2:=x_iNANDtmp_1", x_0 has value
    0, tmp_1 has value 1, tmp_2 assigned value 1
Executing command "val_0:=tmp_2NANDtmp_2", tmp_2 has
    value 1, tmp_2 has value 1, val_0 assigned value 0
Executing command "ns_0:=s_0NANDs_0", s_0 has value 0,
    s_0 has value 0, ns_0 assigned value 1
Executing command "y_0:=ns_0NANDns_0", ns_0 has value
    1, ns_0 has value 1, y_0 assigned value 0
Executing command "u_0:=val_0NANDs_0", val_0 has value
    0, s_0 has value 0, u_0 assigned value 1
Executing command "v_0:=s_0NANDu_0", s_0 has value 0,
    u_0 has value 1, v_0 assigned value 1
Executing command "w_0:=val_0NANDu_0", val_0 has value
    0, u_0 has value 1, w_0 assigned value 1
Executing command "s_0:=v_0NANDw_0", v_0 has value 1,
    w_0 has value 1, s_0 assigned value 0
Executing command "seen_i:=zero_0NANDzero_0", zero_0
    has value 0, zero_0 has value 0, seen_0 assigned value 1
Executing command "stop_0:=validx_iNANDvalidx_i",
    validx_0 has value 1, validx_0 has value 1, stop_0
    assigned value 0
Executing command "loop:=stop_0NANDstop_0", stop_0 has
```

```

    value 0, stop_0 has value 0, loop assigned value 1
Entering new iteration
Executing command "tmp_1:=seen_i_NAND_seen_i", seen_1 has
    value 0, seen_1 has value 0, tmp_1 assigned value 1
...
...
...
Executing command "seen_i:=zero_0_NAND_zero_0", zero_0
    has value 0, zero_0 has value 0, seen_2 assigned value 1
Executing command "stop_0:=validx_i_NAND_validx_i",
    validx_2 has value 0, validx_2 has value 0, stop_0
    assigned value 1
Executing command "loop:=stop_0_NAND_stop_0", stop_0 has
    value 1, stop_0 has value 1, loop assigned value 0
Result: 1 (1)

```

The line by line execution trace is quite long and tedious, but note that it is very easy to locally check, without the need to redo the computation ourselves: we just need to see that each line computes the NAND correctly, and that the value that it claims for the variables on the righthand side of the assignment is the same value that was written to them in the previous line that accessed them.

More formally, we will use the notion of a *modification log* or “Deltas” of a NAND++ program, as presented in [Definition 7.4](#).<sup>5</sup> Recall that given a NAND++ program  $P$  and an input  $x \in \{0,1\}^n$ , if  $P$  has  $s$  lines and takes  $T$  iterations of its loop to halt on  $x$ , then the *modification log* of  $P$  on  $x$  is the string  $\Delta \in \{0,1\}^{sT+n}$  such that for every  $i \in [n]$ ,  $\Delta_i = x_i$  and for every  $\ell \in \{n, n+1, \dots, sT+n-1\}$ ,  $\Delta_\ell$  corresponds to the value that is assigned to a variable during step number  $(\ell - n)$  of the execution. Note that for every  $\ell \in \{n, n+1, \dots, sT+n-1\}$ ,  $\Delta_\ell$  is the NAND of  $\Delta_j$  and  $\Delta_k$  where  $j$  and  $k$  are the last lines in which the two variables referred to in the corresponding line are assigned a value.

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  $\Psi_{P,x}(\Delta)$  such that for every  $\Delta \in \{0,1\}^*$ ,  $\Psi_{P,x}(\Delta)$  is true if and only if  $\Delta$  is a consistent modification log 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 high level point is that we can express the fact that  $\Delta$  is consistent as the following conditions:

- For every  $i \in [n]$ ,  $\Delta_i = x_i$ . (Note that this is easily a mixed quantifier statement.)

<sup>5</sup> We could also have proven Gödel’s theorem using the sequence of all configurations, but the “deltas” have a simpler format.

- For every  $\ell \in \{n, \dots, Ts + n - 1\}$ ,  $\Delta_\ell = 1 - \Delta_j \Delta_k$  where  $j$  and  $k$  are the locations in the log corresponding to the last steps before step  $\ell - n$  in which the variables on the righthand side of the assignment were written to. (This is a mixed quantifier statement as long as we can compute  $j$  and  $k$  using some arithmetic function of  $i$ , or some integer quantifier statement using  $i$  as a parameter.)
- If  $t_0$  is the last step in which the variable `loop` is written to, then  $\Delta_{t_0+n} = 0$ . (Again this is a mixed quantifier statement if we can compute  $t_0$  from the length of  $\Delta$ .)

To ensure that we can implement the above as mixed quantifier statements we observe the following:

1. Since the *INDEX* function that maps the current step to the location of the value `i` was an explicit arithmetic function, 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  $i$ . (See [Exercise 12.2](#).)
2. We can come up with quantified integer statements  $PREV_1(t, t')$  and  $PREV_2(t, t'')$  that will satisfy the following. If at step  $t - n$  the operation invoked is `foo := bar NAND baz` then  $PREV_1(t, t')$  is true if and only if  $t'$  is  $n$  plus the last step before  $t$  in which `bar` was written to and  $PREV_2(t, t'')$  is true if and only if  $t''$  is  $n$  plus the last step before  $t$  in which `baz` was written to. (If one ore both of the righthand side variables are input variables, the  $t'$  and/or  $t''$  will correspond to the index in  $[n]$  of that variable.) Therefore, checking the validity of  $\Delta_t$  will amount to checing that  $\forall_{t' \in \mathbb{N}} \forall_{t'' \in \mathbb{N}} \neg PREV_1(t, t') \vee \neg PREV_2(t, t'') \vee (\Delta_t = 1 - \Delta_{t'} \Delta_{t''})$ . Note that these statements will themselves use *INDEX* because if `bar` and/or `baz` are indexed by `i` then part of the condition for  $PREV_1(t, t')$  and  $PREV_2(t, t'')$  will be to ensure that the `i` variable in these steps is the same as the variable in step  $t - n$ . Using this, plus the observation that the program has only a constant number of lines, we can create the statements  $PREV_1$  and  $PREV_2$ . (See [Exercise 12.3](#))
3. 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 `loop`.

Together these three steps enable the construction of the formula  $\Psi_{P,x}(\Delta)$ . (We omit the full details, which are messy but ultimately straightforward.) Given a construction of such a formula  $\Psi_{P,x}(\Delta)$  we can see that  $HALT(P, x) = 1$  if and only if the following quantified

mixed statement  $\varphi_{P,x}$  is true

$$\varphi_{P,x} = \exists_{\Delta \in \{0,1\}^*} \Psi_{P,x}(\Delta) \quad (12.6)$$

and hence we can write  $\text{HALT}(P, x) = \text{QMS}(\varphi_{P,x})$ . Since we can compute from  $P, x$  the statement  $\varphi_{P,x}$ , we see that if  $\text{QMS}$  is computable then so would have been  $\text{HALT}$ , yielding a proof by contradiction of [Theorem 12.3](#).

### 12.2.3 Reducing mixed statements to integer statements

We now show how to prove [Theorem 12.2](#) using [Theorem 12.3](#). 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  $\text{COORD}(X, i)$  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  $\forall_{X \in \mathbb{N}} \forall_{n \in \mathbb{N}}$  (and similarly replace existential quantifiers over strings). We can later replace all calls to  $|x|$  by  $n$  and all calls to  $x_i$  by  $\text{COORD}(X, i)$ . Hence an encoding of the form above yields a proof of [Theorem 12.2](#), 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 12.4 — Constructible prime sequence.** There is a sequence of prime numbers  $p_0 < p_1 < p_3 < \dots$  such that there is a quantified integer statement  $\text{PCOORD}(p, i)$  that is true if and only if  $p = p_i$ .

Using [Lemma 12.4](#) 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  $\text{COORD}(X, i)$  as

$$\forall_{p \in \mathbb{N}} \neg \text{PCOORD}(p, i) \vee \text{DIVIDES}(p, X) \quad (12.7)$$

where  $DIVIDES(a, b)$ , 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  $i < n$ ,  $COORD(X, i) = x_i$ , since  $p_i$  divides  $X$  if and only if  $x_i = 1$ .

Thus all that is left to conclude the proof of [Theorem 12.2](#) is to prove [Lemma 12.4](#), which we now proceed to do.

*Proof.* The sequence of prime numbers we consider is the following: We fix  $C$  to be a sufficiently large constant ( $C = 2^{2^{34}}$  will do) and define  $p_i$  to be the smallest prime number that is in the interval  $[(i + C)^3 + 1, (i + C + 1)^3 - 1]$ . It is known that there exists such a prime number for every  $i \in \mathbb{N}$ . Given this, the definition of  $PCOORD(p, i)$  is simple:

$$(p > (i + C) \times (i + C) \times (i + C)) \wedge (p < (i + C + 1) \times (i + C + 1) \times (i + C + 1)) \wedge \left( \forall p' \neg PRIME(p') \vee (p' \leq i) \vee (p' \geq p) \right) . \quad (12.8)$$

We leave it to the reader to verify that  $PCOORD(p, i)$  is true iff  $p = p_i$ . ■

### 12.3 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  $1 - 1 + 1 - 1 + 1 + \dots$ ) 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 Riemann, who eventually placed analysis on firmer

foundations, giving rise to the  $\epsilon$ '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 12.1](#) 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 study the existence of proofs, we need to make the notion of a *proof system* more precise. What axioms shall we use? What rules? Our idea will be to use an extremely general notion of proof. A *proof* will be simply a piece of text- a finite string- such that:

1. (*effectiveness*) Given a statement  $x$  and a proof  $w$  (both of which can be encoded as strings) we can verify that  $w$  is a valid proof for  $x$ . (For example, by going line by line and checking that each line does indeed follow from the preceding ones using one of the allowed inference rules.)
2. (*soundness*) If there is a valid proof  $w$  for  $x$  then  $x$  is true.

Those seem like rather minimal requirements that one would want from every proof system. Requirement 2 (soundness) is the very definition of a proof system: you shouldn't be able to prove things that are not true. Requirement 1 is also essential. If it there is no set of rules (i.e., an algorithm) to check that a proof is valid then in what sense is it a proof system? We could replace it with the system where the "proof" for a statement  $x$  would simply be "trust me: it's true".

Let us give a formal definition for this notion, specializing for the case of Diophantine equations:

**Definition 12.3 — Proof systems for diophantine equations.** A proof system for Diophantine equations is defined by a NAND++ program  $V$ . A *valid proof* in the system corresponding to  $V$  of the unsatisfiability of a diophantine equation " $P(\cdot, \dots, \cdot) = 0$ " is some string  $w \in \{0, 1\}^*$  such that  $V(P, w) = 1$ . The proof system corresponding to  $V$  is *sound* if there is no valid proof of a false statement. That is, for every diophantine equation " $P(\cdot, \dots, \cdot) = 0$ ", if there exists  $w \in \{0, 1\}^*$  such that  $V(p, w) = 1$  then for every

$x_1, \dots, x_t \in \mathbb{Z}, P(x_1, \dots, x_t) \neq 0.$

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. 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 12.5 — Gödel's Incompleteness Theorem.** For every sound proof system  $V$ , there exists a diophantine equation " $P(\cdot, \dots, \cdot) = 0$ " such that there is no  $x_1, \dots, x_t \in \mathbb{N}$  that satisfy it, but yet there is no proof in the system  $V$  for the statement that the equation is unsatisfiable.

*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 12.1](#). The algorithm will work as follows:

- On input a Diophantine equation  $P(x_1, \dots, x_t) = 0$ , for  $k = 1, 2, \dots$  do the following:
  1. Check for all  $x_1, \dots, x_t \in \{0, \dots, k\}$  whether  $x_1, \dots, x_t$  satisfies the equation. If so then halt and output 1.
  2. For all  $n \in \{1, \dots, k\}$  and all strings  $w$  of length at most  $k$ , check whether  $V(P, w) = 1$ . If so then halt and output 0.

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 12.1](#) ■

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 12.2](#). 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 12.5](#) shows that it yields a more general statement (see [Exercise 12.4](#)): for every uncomputable func-

tion  $F : \{0,1\}^* \rightarrow \{0,1\}$  and every sound proof system, there is some input  $x$  for which the proof system is not able to prove neither that  $F(x) = 0$  nor that  $F(x) \neq 0$  (see [Exercise 12.4](#)).

Also, the proof of [Theorem 12.5](#) 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 (i.e., if there is no statement  $x$  such that  $S$  can prove both  $x$  and  $\text{NOT}(x)$ ).
- 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.

### 12.3.1 The Gödel statement

One can extract from the proof of [Theorem 12.5](#) a procedure that for every proof system  $V$  (when thought of as a verification algorithm), yields a true statement  $x^*$  that cannot be proven in  $V$ . But Gödel's proof gave a very explicit description of such a statement  $x$  which is closely related to the “[Liar's paradox](#)”. That is, Gödel's statement  $x^*$  was designed to be true if and only if  $\forall_{w \in \{0,1\}^*} V(x, w) = 0$ . In other words, it satisfied the following property

$$x^* \text{ is true} \Leftrightarrow x^* \text{ does not have a proof in } V \quad (12.9)$$

One can see that if  $x^*$  is true, then it does not have a proof, but it is false then (assuming the proof system is sound) then it cannot have a proof, and hence  $x^*$  must be both true and unprovable. One might wonder how is it possible to come up with an  $x^*$  that satisfies a condition such as [Eq. \(12.9\)](#) where the same string  $x^*$  appears on both the righthand side and the lefthand side of the equation. The idea is that the proof of [Theorem 12.5](#) yields a way to transform every statement  $x$  into a statement  $F(x)$  that is true if and only if  $x$  does not have a proof in  $V$ . Thus  $x^*$  needs to be a *fixed point* of  $F$ : a sentence such that  $x^* = F(x^*)$ . It turns out that [we can always find](#) such a fixed point of  $F$ . We've already seen this phenomenon in the  $\lambda$  calculus, where the  $Y$  combinator maps every  $F$  into a fixed point  $YF$

of  $F$ . This is very related to the idea of programs that can print their own code. Indeed, Scott Aaronson likes to describe Gödel's statement as follows:

The following sentence repeated twice, the second time in quotes, is not provable in the formal system  $V$ . "The following sentence repeated twice, the second time in quotes, is not provable in the formal system  $V$ ."

In the argument above we actually showed that  $x^*$  is *true*, under the assumption that  $V$  is sound. Since  $x^*$  is true and does not have a proof in  $V$ , this means that we cannot carry the above argument in the system  $V$ , which means that  $V$  cannot prove its own soundness (or even consistency: that there is no proof of both a statement and its negation). Using this idea, it's not hard to get Gödel's second incompleteness theorem, which says that every sufficiently rich  $V$  cannot prove its own consistency. That is, if we formalize the statement  $c^*$  that is true if and only if  $V$  is consistent (i.e.,  $V$  cannot prove both a statement and the statement's negation), then  $c^*$  cannot be proven in  $V$ .

## 12.4 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 sound proof system (and in particular every 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.

## 12.5 Exercises

**Exercise 12.1 — Expression for floor.** Let  $FSQRT(n, m) = \forall_{j \in \mathbb{N}}((j \times j) > n) \vee (j < m) \vee (j = m)$ . Prove that  $FSQRT(m, n)$  is true if and only if  $m = \lfloor \sqrt{m} \rfloor$ . ■

**Exercise 12.2 — Expression for computing the index.** Recall that in Exercise 7.1 asked you to prove that at iteration  $t$  of a NAND++ program the variable  $i$  is equal to  $t - r(r + 1)$  if  $t \leq (r + 1)^2$  and equals

$(r+2)(r+1)t$  otherwise, where  $r = \lfloor \sqrt{t+1/4} - 1/2 \rfloor$ . Prove that there is a quantified integer statement *INDEX* with parameters  $t, i$  such that *INDEX*( $t, i$ ) is true if and  $i$  is the value of  $i$  after  $t$  iterations. ▀

**Exercise 12.3 — Expression for computing the previous line.** Give the following quantified integer expressions:

1.  $\text{MOD}(a, b, c)$  which is true if and only if  $c = a \bmod c$ . Note if a program has  $s$  lines then the line executed at step  $t$  is equal to  $t \bmod s$ .
2. Suppose that  $P$  is the three line NAND program listed below. Give a quantified integer statement  $\text{LAST}(n, t, t')$  such that  $\text{LAST}(t, t')$  is true if and only if  $t' - n$  is the largest step smaller than  $t - n$  in which the variable on the righthand side of the line executed at step  $t - n$  is written to. If this variable is an input variable  $x_i$  then let  $\text{LAST}(n, t, t')$  to be true if the current index location equals  $t'$  and  $t' < n$ . ▀

```
y_0 := foo_i NAND foo_i
foo_i := x_i NAND x_i
loop := validx_i NAND validx_i
```

**Exercise 12.4 — axiomatic proof systems.** For every representation of logical statements as strings, we can define an axiomatic proof system to consist of a finite set of strings  $A$  and a finite set of rules  $I_0, \dots, I_{m-1}$  with  $I_j : (\{0,1\}^*)^{k_j} \rightarrow \{0,1\}^*$  such that a proof  $(s_1, \dots, s_n)$  that  $s_n$  is true is valid if for every  $i$ , either  $s_i \in A$  or is some  $j \in [m]$  and are  $i_1, \dots, i_{k_j} < i$  such that  $s_i = I_j(s_{i_1}, \dots, s_{i_{k_j}})$ . 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  $F(x) \neq 0$ . ▀

6

<sup>6</sup> TODO: Maybe add an exercise to give a MIS that corresponds to any regular expression.

## 12.6 Bibliographical notes

## 12.7 Further explorations

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

### 12.8 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 before Wiles' work.)



### Learning Objectives:

- Describe at a high level some interesting computational problems.
- The difference between polynomial and exponential time.
- Examples of techniques for obtaining efficient algorithms
- Examples of how seemingly small differences in problems can make (at least apparent) huge differences in their computational complexity.

13

## 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 Twoness. . . 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 a “**threshold phenomenon**” or “zero one law” for running time, where for many natural problems, they either can be solved in *polynomial* running time (e.s., 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 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 $\llcorner$  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.

### 13.1 Problems on graphs

We now present a few examples of computational problems that people are interesting in solving. 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)  $\{u, v\}$  which we sometimes denote as  $\overline{u v}$  or  $u \sim v$ .<sup>1</sup> We will assume graphs are undirected and *simple* (i.e., containing no parallel edges or self-loops) unless stated otherwise.

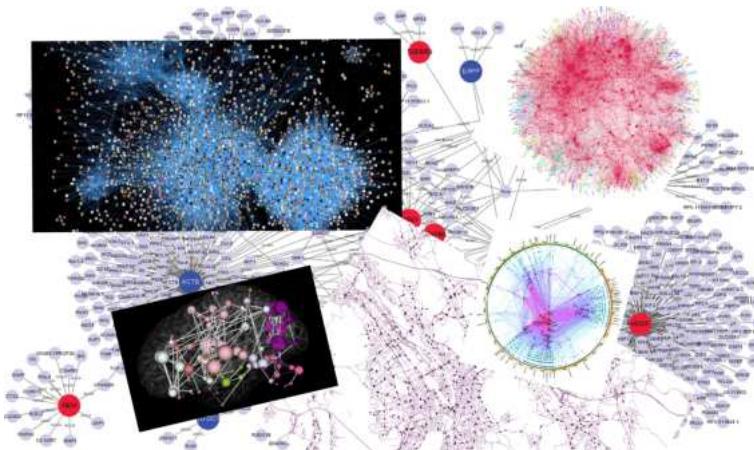
We typically will think of the vertices in a graph as simply the set  $[n]$  of the numbers from 0 till  $n - 1$ . Graphs can be represented either in the *adjacency list* representation, which is a list of  $n$  lists, with the  $i^{\text{th}}$  list corresponding to the neighbors of the  $i^{\text{th}}$  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.<sup>2</sup> We can transform between these two representations using  $O(n^2)$  operations, and hence for our purposes we will mostly consider them

<sup>1</sup> 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  $\overrightarrow{v u}$ .

<sup>2</sup> In an undirected graph, the adjacency matrix  $A$  is *symmetric*, in the sense that it satisfies  $A_{i,j} = A_{j,i}$ .

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 undirected graph.

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), causal 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 13.1:** Some examples of graphs found on 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.

### 13.1.1 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  $t$  (if such a path exists). That is, we want to find the smallest number  $k$  such that there are vertices  $v_0, v_1, \dots, v_k$  with  $v_0 = s$ ,  $v_k = t$  and for every  $i \in \{0, \dots, k-1\}$  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.<sup>3</sup>

More formally, the algorithm for shortest path can be described as follows:

**Algorithm** *SHORTESTPATH*:

- **Input:** Graph  $G = (V, E)$ , vertices  $s, t$
- **Goal:** Find the shortest path  $v_0, v_1, \dots, v_k$  such that  $v_0 = s$ ,  $v_k = t$  and  $\{v_i, v_{i+1}\} \in E$  for every  $i \in [k]$ , if such a path exists.
- **Operation:**
  1. We will maintain a label  $L[v]$  for every vertex  $v$ . Initially no vertex is labeled except for  $s$  that is labeled with “start”.
  2. We maintain a *queue*  $Q$  of vertices, initially  $Q$  contains only  $s$ .
  3. While  $Q$  is not empty do the following:
    - a. Pop the vertex  $v$  from the top of the queue.
    - b. If  $v = t$  exit output the path which is the reverse order of  $v, L[v], L[L[v]], L[L[L[v]]], \dots, s$ .
    - c. Otherwise, label all the unlabeled neighbors of  $v$  with  $v$  and add them to  $Q$
  4. Output “no path”

Since we only add to the queue unlabeled vertices, we never push to the queue a vertex more than once, and hence the algorithm takes  $n$  “push” and “pop” operations. It returns the correct answer since add the vertices to the queue in the order of their distance from  $s$ , and hence we will reach  $t$  after we have explored all the vertices that are closer to  $s$  than  $t$ .

<sup>3</sup> Since we assume  $m \geq n - 1$ ,  $O(m)$  is the same as  $O(n + m)$ . **Dijkstra's algorithm** is a well-known generalization of BFS to *weighted* graphs.

### 13.1.2 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  $t$ . If the graph is a road network, then the longest path might seem less motivated than the shortest path, but of course graphs can be and are used to model a variety of phenomena, and in many such cases the longest path (and some of its variants) are highly motivated. In particular, finding the longest path is a generalization of the famous **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 path visiting all vertices 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 that tries all paths.

Specifically, in a graph of degree at most  $d$ , we can enumerate over all paths of length  $k$  by going over the (at most  $d$ ) neighbors of each vertex. This would take about  $O(d^k)$  steps, and since the longest simple path can't have length more than the number of vertices, this means that the brute force algorithms runs in  $O(d^n)$  time (which we can bound by  $O(n^n)$  since the maximum degree is  $n$ ). The best algorithm for the longest path improves on this, but not by much: it takes  $\Omega(c^n)$  time for some constant  $c > 1$ .<sup>4</sup>

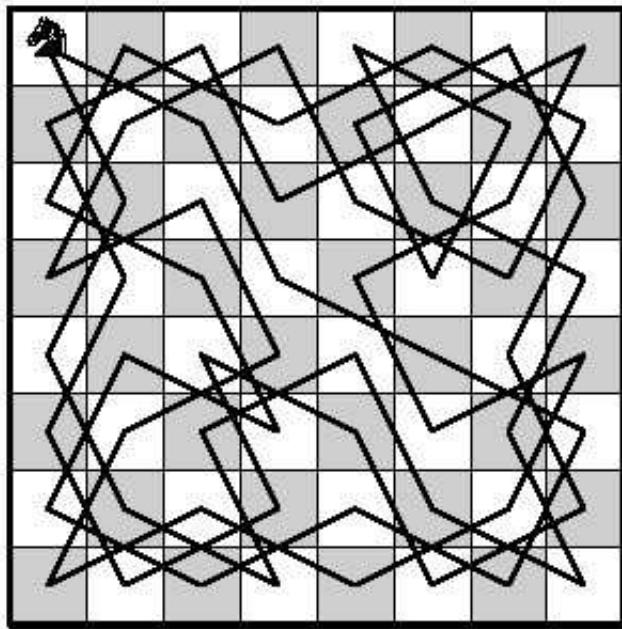
### 13.1.3 Finding the minimum cut in a graph

Given a graph  $G = (V, E)$ , a *cut* is a subset  $S$  of  $V$  such that  $S$  is neither empty nor is it all of  $V$ . The edges cut by  $S$  are those edges where one of their endpoints is in  $S$  and the other is in  $\bar{S} = V \setminus S$ . We denote this set of edges by  $E(S, \bar{S})$ . If  $s, t \in V$  then an  $s, t$  *cut* is a cut such that  $s \in S$  and  $t \in \bar{S}$ . (See Fig. 13.3.) The *minimum  $s, t$  cut problem* is the task of finding, given  $s$  and  $t$ , the minimum number  $k$  such that there is an  $s, t$  cut cutting  $k$  edges (the problem is also sometimes phrased as finding the set that achieves this minimum).<sup>5</sup>

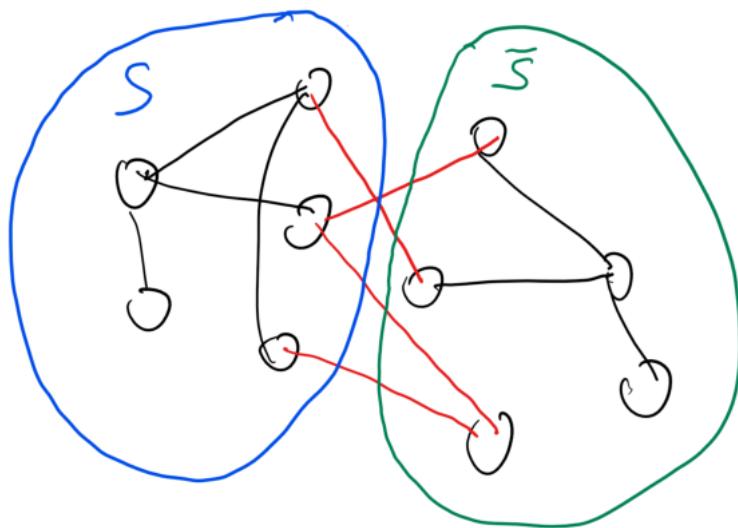
The minimum  $s, t$  cut problem appears in many applications. Minimum cuts often correspond to *bottlenecks*. For example, in a com-

<sup>4</sup> At the moment the best record is  $c \sim 1.65$  or so. Even obtaining an  $O(2^n)$  time bound is not that simple, see Exercise 13.1.

<sup>5</sup> One can also define the problem of finding the *global minimum cut* (i.e., the non-empty and non-everything set  $S$  that minimizes the number of edges cut). One can verify that a polynomial time algorithm for the minimum  $s, t$  cut can be used to solve the global cut in polynomial time as well (can you see why?).



**Figure 13.2:** A knight's tour can be thought of as a maximally long path on the graph corresponding to a chessboard where we put an edge between any two squares that can be reached by one step via a legal knight move.



**Figure 13.3:** A cut in a graph  $G = (V, E)$  is simply a subset  $S$  of its vertices. The edges that are cut by  $S$  are all those whose one endpoint is in  $S$  and the other one is in  $\bar{S} = V \setminus S$ . The cut edges are colored red in this figure.

munication 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. In the setting of **image segmentation**, one 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.

**Solving the minimum cut problem:** Here is an algorithm to solve the minimum cut problem:

**Algorithm MINCUTNAIVE:**

- **Input:** Graph  $G = (V, E)$  and two distinct vertices  $s, t \in V$
- **Goal:** Return  $S$  s.t.  $s \in S$  and  $t \notin S$  that minimizes  $|E(S, \bar{S})|$ .
- **Operation:**
  1. If  $V = \{s, t\}$  then return  $\{s\}$ .
  2. Otherwise choose  $v \in V \setminus \{s, t\}$ , and define  $G', G''$  to be two graphs with vertex set  $V \setminus \{v\}$ , where in  $G'$  the edge set  $E'$  is obtained by making all of  $v$ 's neighbors be neighbors of  $s$ , and in  $G''$  the edge set  $E''$  is obtained by making all of  $v$ 's neighbors be neighbors of  $t$ . That is,  $E'$  has the same edges as  $E$  except that in edges involving  $v$  we replace  $v$  with  $s$ , and  $E''$  has the same edges as  $E$  except that in edges involving  $v$  we replace  $v$  with  $t$ .
  3. Compute recursively  $S' = \text{MINCUTNAIVE}(G', s, t)$  and  $S'' = \text{MINCUTNAIVE}(G'', s, t)$ .
  4. If  $S' \cup \{v\}$  cuts fewer edges in  $G$  than  $S''$  then return  $S' \cup \{v\}$ . Otherwise return  $S''$ .



It is an excellent exercise for you to pause at this point and verify: (1) that you understand what this algorithm does, (2) that you understand why this algorithm will in fact return the minimum cut in the graph, and (3) that you can analyze the running time of this algorithm.

We can prove by induction that Algorithm *MINCUTNAIVE* does indeed return the minimum cut. Indeed, it definitely does so for graphs of two vertices. Now we assume by induction that *MINCUTNAIVE* solves the minimum cut problem for graphs of at most  $n - 1$  vertices, and we will prove that it does so for graphs of  $n$

vertices. Indeed, under the inductive hypothesis, our recursive calls in step 3 return the minimum cuts  $S'$  and  $S''$  of the  $n - 1$  vertex graphs  $G'$  and  $G''$  respectively. But if  $v$  is the vertex we choose in step 2, we can think of  $G'$  as simply a graph where we “merged”  $s$  and  $v$  to be a single vertex with the neighbors of both  $s$  and  $v$ , and  $G''$  as the graph where we merged  $t$  and  $v$ . So the  $s, t$  cuts in  $G'$  correspond to  $s, t$  cuts in  $G$  that don’t separate  $s$  and  $v$ , while  $s, t$  cuts in  $G''$  correspond to  $s, t$  cuts in  $G$  that don’t separate  $t$  and  $v$ . Since in the minimum cut  $S$ , either  $s$  or  $t$  will be in the same side as  $v$ , the best one out of the minimal cuts from  $G'$  and  $G''$  will be the minimum cut in  $G$ .

The running time  $T(n)$  of *MINCUTNAIVE* on  $n$  vertex graphs can be described by the recursive equation  $T(n) = 2T(n - 1) + f(n)$  where  $f(n)$  is the time to execute the non recursive steps 1,2 and 4. In an algorithms course we might want to worry about the exact data structures we use to implement these steps, but for this course it is enough that we can do these steps in polynomial time (which is not hard to see). Nevertheless, this running time bound is terrible: even if  $f(n)$  was equal to 1 we would still get that  $T(n) \geq 2^n!$  Indeed, this recursive algorithm is nothing but a fancy description of the trivial algorithm that enumerates over all the roughly  $2^n$  (in fact, precisely  $2^{n-2}$ ) sets  $S$  that contain  $s$  and don’t contain  $t$ .

Since minimum cut is a problem we want to solve, this seems like bad news. Luckily however we are able to find much faster algorithms that run in *polynomial time* (which, as mentioned in the mathematical background lecture, we denote by  $\text{poly}(n)$ ) for this problem. There are several algorithms to do so, but many of them rely on the **Max Flow Min Cut Theorem** 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.<sup>6</sup> For example, this directly implies that the value of the minimum cut problem is the solution for the following **linear program**:<sup>7</sup>

$$\begin{aligned} & \max_{x \in \mathbb{R}^m} F_s(x) - F_t(x) \text{ s.t.} \\ & \quad \forall_{u \notin \{s, t\}} F_u(x) = 0 \end{aligned} \tag{13.1}$$

where for every vertex  $u$  and  $x \in \mathbb{R}^m$ ,  $F_u(x) = \sum_{e \in E[s, t, u] \in e} x_e$ .

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  $O(\min\{m^{10/7}, m\sqrt{n}\})$ .<sup>8</sup>

<sup>6</sup> A *flow* of capacity  $c$  from a *source*  $s$  to a *sink*  $t$  in a graph can be thought of as describing how one would send some quantity from  $s$  to  $t$  in the graph (e.g., sending  $c$  liters of water (or any other matter one can partition arbitrarily), on pipes described by the edges). Mathematically, a flow is captured by assigning numbers to edges, and requiring that on any vertex apart from  $s$  and  $t$ , the amount flowing in is equal to the amount flowing out, while in  $s$  there are  $c$  units flowing out and in  $t$  there are  $c$  units flowing in.

<sup>7</sup> A *linear program* is the task of maximizing or minimizing a linear function of  $n$  real variables  $x_0, \dots, x_{n-1}$  subject to certain linear equalities and inequalities on the variables.

<sup>8</sup> TODO: add references in bibliographical notes: Madry, Lee-Sidford

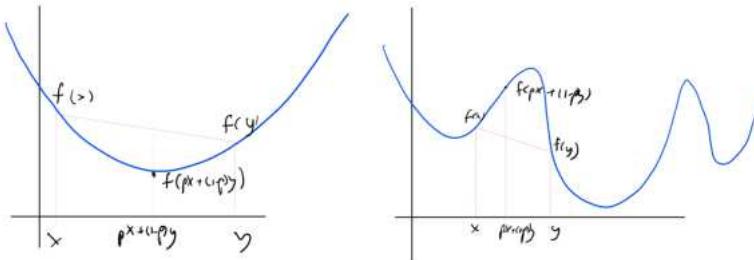
### 13.1.4 Finding the maximum cut in a graph

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  $S$ .<sup>9</sup> 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.

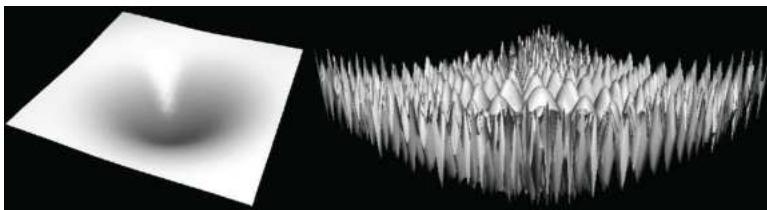
Once again, a priori it might not be clear that the maximum cut problem should be harder than minimum cut but this turns out to be the case. We do not know of an algorithm that solves this problem much faster than the trivial “brute force” algorithm that tries all  $2^n$  possibilities for the set  $S$ .

<sup>9</sup> We can also consider the variant where one is given  $s, t$  and looks for the  $s, t$ -cut that maximizes the number of edges cut. The two variants are equivalent up to  $O(n^2)$  factors in the running time, but we use the global max cut formulation since it is more common in the literature.

### 13.1.5 A note on convexity



**Figure 13.4:** In a *convex* function  $f$  (left figure), for every  $x$  and  $y$  and  $p \in [0, 1]$  it holds that  $f(px + (1 - p)y) \leq p \cdot f(x) + (1 - p) \cdot f(y)$ . In particular this means that every *local minimum* of  $f$  is also a *global minimum*. In contrast in a *non convex* function there can be many local minima.



**Figure 13.5:** In the high dimensional case, if  $f$  is a *convex* function (left 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  $f : D \rightarrow \mathbb{R}$  is *convex* if for every  $x, y \in D$  and  $p \in [0, 1]$   $f(px + (1 - p)y) \leq pf(x) + (1 - p)f(y)$ .

$p)f(y)$ . 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.<sup>10</sup> 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. A *linear* function is both convex and concave, which is the reason both the maximization and minimization problems for linear functions can be done efficiently.

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 (linear) 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  $(x_1, y_1), \dots, (x_m, y_m)$ , where each  $x_i \in \{0, 1\}^n$  and  $y_i \in \{0, 1\}$ , we want to find the function  $h : \{0, 1\}^n \rightarrow \{0, 1\}$  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*  $\ell(h(x_i), y_i)$ .<sup>11</sup> 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

<sup>10</sup> 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.

<sup>11</sup> 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  $R : H \rightarrow \mathbb{R}$  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”.

even a local) minima, this continuous embedding can still help us. In particular, when running a local improvement algorithm such as 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.<sup>12</sup>

## 13.2 Beyond graphs

Not all computational problems arise from graphs. We now list some other examples of computational problems that of great interest.

### 13.2.1 The 2SAT problem

A *propositional formula*  $\varphi$  involves  $n$  variables  $x_1, \dots, x_n$  and the logical operators AND ( $\wedge$ ), OR ( $\vee$ ), and 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 AND of ORs 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

$$(x_7 \vee \bar{x}_{22} \vee x_{15}) \wedge (x_{37} \vee x_{22}) \wedge (x_{55} \vee \bar{x}_7) \quad (13.2)$$

We say that a formula is a  $k$ -CNF it is an AND of 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 13.2](#)). This reduces 2SAT to the (efficiently solvable) problem of determining connectivity in directed graphs.

<sup>12</sup> In machine learning parlance, this task is known as *supervised learning*. The set of examples  $(x_1, y_1), \dots, (x_m, y_m)$  is known as the *training set*, and the error on additional samples from the same distribution is known as the *generalization error*, and can be measured by checking  $h$  against a *test set* that was not used in training it.

### 13.2.2 The 3SAT problem

The 3SAT problem is the task of determining satisfiability for 3CNFs. One might think that changing from two to 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 later 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.

### 13.2.3 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

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} &= b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} &= b_1 \\ \vdots + \vdots + \vdots + \vdots &= \vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned} \tag{13.3}$$

where  $\{a_{i,j}\}_{i,j \in [n]}$  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).<sup>13</sup> 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. In contrast, if we insist on *integer* solutions, the task of solving for linear

<sup>13</sup> 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 . 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.

equalities or inequalities is known as **integer programming**, and the best known algorithms are exponential time in the worst case.

#### 13.2.4 Solving quadratic equations

Suppose that we want to solve not just *linear* but also equations involving *quadratic* terms of the form  $a_{i,j,k}x_jx_k$ . That is, suppose that we are given a set of quadratic polynomials  $p_1, \dots, p_m$  and consider the equations  $\{p_i(x) = 0\}$ . To avoid issues with bit representations, we will always assume that the equations contain the constraints  $\{x_i^2 - x_i = 0\}_{i \in [n]}$ . Since only 0 and 1 satisfy the equation  $a^2 - a$ , this assumption means that we can restrict attention to solutions in  $\{0, 1\}^n$ . Solving quadratic equations in several variable is a classical and extremely well motivated problem. This is the generalization of the classical case of single-variable quadratic equations that generations of high school students grapple with. It also generalizes the **quadratic assignment problem**, introduced in the 1950's as a way to optimize assignment of economic activities. Once again, we do not know a much better algorithm for this problem than the one that enumerates over all the  $2^n$  possibilities.

### 13.3 More advanced examples

We now list a few more examples of interesting problems that are a little more advanced but are of significant interest in areas such as physics, economics, number theory, and cryptography.

#### 13.3.1 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  $perm_2(A)$  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  $perm_2(A)$  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:

$$\sum_{\pi \in S_n} \text{sign}(\pi) \prod_{i=0}^{n-1} A_{i,\pi(i)} \quad (13.4)$$

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.<sup>14</sup>

From a first look, Eq. (13.4) does not seem like it makes much progress. After all, all we did is replace one formula involving a sum over  $n!$  terms with an even more complicated formula involving a sum over  $n!$  terms. But fortunately Eq. (13.4) 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.

<sup>14</sup> 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.

### 13.3.2 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. In a similar “two to three” type of a phenomenon, we do not know of a much better than brute force algorithm to even compute the permanent modulo 3.

### 13.3.3 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

$$\max_{p \in \Delta_n} \min_{q \in \Delta_n} p^\top A q = \min_{q \in \Delta_n} \max_{p \in \Delta_n} p^\top A q \quad (13.5)$$

The min-max theorem turns out to be a corollary of linear programming duality, and indeed the value of Eq. (13.5) can be computed efficiently by a linear program.

### 13.3.4 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 does not necessarily qual 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.

### 13.3.5 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  $2, \dots, N - 1$ , 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  $\sqrt{N}$ . But this is still quite terrible. If  $N$  is a 1024 bit integer,  $\sqrt{N}$  is about  $2^{512}$ , and so running this algorithm on such an input would take much more than the lifetime of the universe.

Luckily, 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.

### 13.3.6 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 roughly  $2^{O(\sqrt{n})}$  and also algorithms that are widely believed (though not fully rigorously analyzed) to run in time roughly  $2^{O(n^{1/3})}$ .<sup>15</sup>

<sup>15</sup> The “roughly” adjective above refers to neglecting factors that are polylogarithmic in  $n$ .

## 13.4 Our current knowledge



**Figure 13.6:** The current computational status of several interesting problems. For all of them we either know a polynomial-time algorithm or the known algorithms require at least  $2^{n^c}$  for some  $c > 0$ . In fact for all except the *factoring* problem, we either know an  $O(n^3)$  time algorithm or the best known algorithm require at least  $2^{\Omega(n)}$  time where  $n$  is a natural parameter such that there is a brute force algorithm taking roughly  $2^n$  or  $n!$  time. Whether this “cliff” between the easy and hard problem is a real phenomenon or a reflection of our ignorane is still an open question.

The difference between an exponential and polynomial time algorithm 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.

### 13.5 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.

### 13.6 Exercises

**Exercise 13.1 — exponential time algorithm for longest path.** Give a  $\text{poly}(n)2^n$  time algorithm for the longest path problem in  $n$  vertex graphs.<sup>16</sup> ■

**Exercise 13.2 — 2SAT algorithm.** For every 2CNF  $\varphi$ , define the graph  $G_\varphi$  on  $2n$  vertices corresponding to the literals  $x_1, \dots, x_n, \bar{x}_1, \dots, \bar{x}_n$ , such that there is an edge  $\overrightarrow{\ell_i \ell_j}$  iff the constraint  $\ell_i \vee \ell_j$  is in  $\varphi$ . Prove that  $\varphi$  is unsatisfiable if and only if there is some  $i$  such that there is a path

<sup>16</sup> Hint: Use dynamic programming to compute for every  $s, t \in [n]$  and  $S \subseteq [n]$  the value  $P(s, t, S)$  which equals to 1 if there is a simple path from  $s$  to  $t$  that uses exactly the vertices in  $S$ . Do this iteratively for  $S$ ’s of growing sizes.

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 13.3 — Regular expressions.** A *regular expression* over the binary alphabet is a string consisting of the symbols  $\{0, 1, \emptyset, (,), *, |\}$ . An expression  $exp$  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  $exp, exp'$  are expressions corresponding to the functions,  $f, f'$  then  $(exp)|(exp')$  corresponds to the function  $f \vee f'$ ,  $(exp)(exp')$  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  $f(x_0, \dots, x_i) = 1$  and  $f(x_{i+1}, \dots, x_{n-1}) = 1$ , and  $(exp)*$  corresponds to the function  $g$  such that  $g(x) = 1$  if either  $x = \emptyset$  or there are strings  $x_1, \dots, x_k$  such that  $x$  is the concatenation of  $x_1, \dots, x_k$  and  $f(x_i) = 1$  for every  $i \in [k]$ .

Prove that for every regular expression  $exp$ , the function corresponding to  $exp$  is computable in polynomial time. Can you show that it is computable in  $O(n)$  time? ■

### 13.7 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.

<sup>17</sup>

<sup>17</sup> TODO: add reference to best algorithm for longest path - probably the Bjorklund algorithm

### 13.8 Further explorations

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

### 13.9 Acknowledgements

### Learning Objectives:

- Formally modeling running time, and in particular notions such as  $O(n)$  or  $O(n^3)$  time algorithms.
- The classes **P** and **EXP** modelling polynomial and exponential time respectively.
- The *time hierarchy theorem*, that in particular says that for every  $k \geq 1$  there are functions we *can* compute in  $O(n^{k+1})$  time but *can not* compute in  $O(n^k)$  time.

14

## 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, using the NAND++ and NAND« models we have seen before. Since we think of programs that can take as input a string of arbitrary length, their running time is not a fixed number but rather what we are interested in is measuring the *dependence* of the number of steps the program takes on the length of the input. That is, for any program  $P$ , we will be interested in being interested in the maximum number of steps that  $P$  takes on inputs of length  $n$  (which we often denote as  $T(n)$ ).<sup>1</sup> For example, if a function  $F$  can be computed by

<sup>1</sup> Because we are interested in the *maximum* number of steps for inputs of a given length, this concept is often known as *worst case complexity*. The *minimum* number of steps (or "best case" complexity) to compute a function on length  $n$  inputs is typically not a meaningful quantity since essentially every natural problem will have some trivially easy instances. However, the *average case complexity* (i.e., complexity on a "typical" or "random" input) is an interesting concept which we'll return to when we discuss *cryptography*. That said, worst-case complexity is the most standard and basic of the complexity measures, and will be our focus in most of this course.

a NAND« (or NAND++) program that on inputs of length  $n$  takes  $O(n)$  steps then we will think of  $F$  as “efficiently computable”, while if any such program requires  $2^{\Omega(n)}$  steps to compute  $F$  then we consider  $F$  “intractable”. Formally, we define running time as follows:

**Definition 14.1 — Running time.** Let  $T : \mathbb{N} \rightarrow \mathbb{N}$  be some function mapping natural numbers to natural numbers. We say that a function  $F : \{0,1\}^* \rightarrow \{0,1\}$  is *computable in  $T(n)$  NAND« time* if there is a NAND« program  $P$  computing  $F$  such that for every sufficiently large  $n$  and every  $x \in \{0,1\}^n$ , on input  $x$ ,  $P$  runs for at most  $T(n)$  steps. Similarly, we say that  $F$  is *computable in  $T(n)$  NAND++ time* if there is a NAND++ program  $P$  computing  $F$  such that on every sufficiently large  $n$  and  $x \in \{0,1\}^n$ , on input  $x$ ,  $P$  runs for at most  $T(n)$  steps.

We let  $\text{TIME}_{<<}(T(n))$  denote the set of Boolean functions that are computable in  $T(n)$  NAND« time, and define  $\text{TIME}_{++}(T(n))$  analogously.<sup>2</sup>

**Definition 14.1** naturally extend to non Boolean and to partial functions as well, and so we will talk about the time complexity of these functions.

**Which model to choose?** Unlike the notion of computability, the exact running time can be a function of the model we use. However, it turns out that if we care about “coarse enough” resolution (as we will in this course) then the choice of the model, whether it is NAND«, NAND++, or Turing or RAM machines of various flavors, does not matter. (This is known as the *extended Church-Turing Thesis*). Nevertheless, to be concrete, we will 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.

**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 shorter inputs) or other degenerate cases such as functions that can be computed without reading the input or cases where the running time bound itself is hard to compute. Thus we make the following definition:

<sup>2</sup> The relaxation of considering only “sufficiently large”  $n$ ’s is not very important but it is convenient since it allows us to avoid dealing explicitly with un-interesting “edge cases”. In most cases we will anyway be interested in determining running time only up to constant and even polynomial factors. Note that we can always compute a function on a finite number of inputs using a lookup table.

**Definition 14.2 — 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  $n \geq n'$

\* There is a NAND<sup>3</sup> program that on input numbers  $n, i$ , given in binary, can compute in  $O(T(n))$  steps the  $i$ -th bit of a prefix-free representation of the  $i$ -th bit of  $T(n)$  (represented as a string in some prefix-free way).

All the functions mentioned above are “nice” per Definition 14.2, and from now on we will only care about the class  $\text{TIME}(T(n))$  when  $T$  is a “nice” function. The last condition simply means that we can compute the binary representation of  $T(n)$  in time which itself is roughly  $T(n)$ . This condition is typically easily satisfied. For example, for arithmetic functions such as  $T(n) = n^3$  or  $T(n) = \lfloor n \rfloor^{1.2} \log n$  we can typically compute the binary representation of  $T(n)$  in time which is polynomial in the *number of bits* in this representation. Since the number of bits is  $O(\log T(n))$ , any quantity that is polynomial in this number will be much smaller than  $T(n)$  for large enough  $n$ .

The two main time complexity classes we will be interested in are the following:

- **Polynomial time:** We say that a function  $F : \{0,1\}^* \rightarrow \{0,1\}$  is *computable in polynomial time* if it is in the class  $\mathbf{P} = \bigcup_{c \in \mathbb{N}} \text{TIME}(n^c)$ .
- **Exponential time:** We say that function  $F : \{0,1\}^* \rightarrow \{0,1\}$  is *computable in exponential time* if it is in the class  $\mathbf{EXP} = \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{n^c})$ .

Since exponential time is much larger than polynomial time, clearly  $\mathbf{P} \subseteq \mathbf{EXP}$ . All of the problems we listed in the last lecture are in  $\mathbf{EXP}$ ,<sup>3</sup> but as we’ve seen, for some of them there are much better algorithms that demonstrate that they are in fact in  $\mathbf{P}$ .

A table of the examples from the previous lecture. All these problems are in  $\mathbf{EXP}$  but the only the ones on the left column are currently known to be in  $\mathbf{P}$  (i.e., have a polynomial-time algorithm).

<sup>3</sup> Strictly speaking, many of these problems correspond to *non Boolean* functions, but we will sometimes “abuse notation” and refer to non Boolean functions as belonging to  $\mathbf{P}$  or  $\mathbf{EXP}$ . We can easily extend the definitions of these classes to non Boolean and partial functions. Also, for every non-Boolean function  $F : \{0,1\}^* \rightarrow \{0,1\}^*$ , we can define a Boolean variant  $\text{Bool}(F)$  such that  $F$  can be computed in polynomial time if and only if  $\text{Bool}(F)$  is.

P	EXP (but not known to be in P)
Shortest path	Longest Path
Min cut	Max cut
2SAT	3SAT
Linear eqs	Quad. eqs
Zerosum	Nash
Determinant	Permanent
Primality	Factoring

### 14.1 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 14.1 — Efficient simulation of NAND« with NAND++.** There are absolute constants  $a, b$  such that for every function  $F$  and nice function  $T : \mathbb{N} \rightarrow \mathbb{N}$ , if  $F \in \text{TIME}_{\ll}(T(n))$  then there is a NAND++ program  $P'$  that computes  $F$  in  $T'(n) = a \cdot T(n)^b$ . That is,  $\text{TIME}_{\ll}(T(n)) \subseteq \text{TIME}_{++}(aT(n)^b)$

The constant  $b$  can be easily shown to be at most five, and with more effort can be optimized further. [Theorem 14.1](#) means that the definition of the classes **P** and **EXP** are robust to the choice of model, and will not make a difference whether we use NAND++ or NAND«. In fact, similar results are known for Turing Machines, RAM machines, C programs, and a great many other models, which justifies the choice of **P** as capturing a technology-independent notion of tractability. As we discussed before, the equivalence between NAND++ and NAND« (as well as other models) allows us to pick our favorite one depending on the task at hand (i.e., “have our cake and eat it too”). 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 or prove a *negative result*, we can restrict attention to NAND++ programs.

**Proof Idea:** We have seen in [Theorem 8.1](#) that every function  $F$  that is computable by a NAND« program  $P$  is computable by a NAND++ program  $P'$ . To prove [Theorem 14.1](#), we follow the exact same proof but just check that the overhead of the simulation of  $P$  by  $P'$  is polynomial.

*Proof of Theorem 14.1.* As mentioned above, we follow the proof of Theorem 8.1 (simulation of NAND $\ll$  programs using NAND++ programs) and use the exact same simulation, but with a more careful accounting of the number of steps that the simulation costs. Recall, that the simulation of NAND $\ll$  works by “peeling off” features of NAND $\ll$  one by one, until we are left with NAND++. We now sketch the main observations we use to show that this “peeling off” costs at most a polynomial overhead:

1. If  $P$  is a NAND $\ll$  program that computes  $F$  in  $T(n)$  time, then on inputs of length  $n$ , all integers used by  $P$  are of magnitude at most  $T(n)$ . This means that the largest value  $i$  can ever reach is at most  $T(n)$  and so each one of  $P$ 's variables can be thought of as an array of at most  $T(n)$  indices, each of which holds a natural number of magnitude at most  $T(n)$  (and hence one that can be encoded using  $O(\log T(n))$  bits). Such an array can be encoded by a bit array of length  $O(T(n) \log T(n))$ .
2. All the arithmetic operations on integers use the gradeschool algorithms, that take time that is polynomial in the number of bits of the integers, which is  $\text{poly}(\log T(n))$  in our case.
3. Using the `i++` and `i--` operations we can load an integer (represented in binary) from the variable `foo` into the index `i` using a cost of  $O(T(n)^2)$ . The idea is that we create an array `marker` that contains a single 1 coordinate and all the rest are zeroes. We will repeat the following for at most  $T(n)$  steps: at each step we decrease `foo` by one (at a cost of  $O(\log T(n))$ ) and move the 1 in `marker` one step to the right (at a cost of  $O(T(n))$ ). We stop when `foo` reaches 0, at which point `marker` has 1 in the location encoded by the number that was in `foo`, and so if we move `i` until `marker_i` equals to 1 then we reach our desired location.
4. Once that is done, all that is left is to simulate `i++` and `i--` in NAND++ using our “breadcrumbs” and “wait for the bus” technique. To simulate  $T$  steps of increasing and decreasing the index, we will need at most  $O(T^2)$  steps of NAND++ (see Fig. 14.1). In the worst case for every increasing or decreasing step we will need to wait a full round until `i` reaches 0 and gets back to the same location, in which case the total cost will be  $O(1 + 2 + 3 + 4 + \dots + T) = O(T^2)$  steps.

Together these observations imply that the simulation of  $T$  steps of NAND $\ll$  can be done in  $\text{poly}(T)$  step. (In fact the cost is  $O(T^4 \text{polylog}(T)) = O(T^5)$  steps, and can even be improved further though this does not matter much.) ■

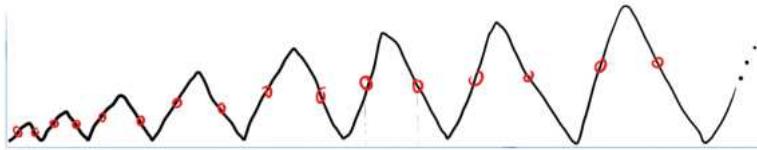


Figure 14.1: The path an index variable takes in a NAND++ program

(R)

**Turing machines and other models** If we follow the equivalence results between NAND++/NAND« and other models, including Turing machines, RAM machines, Game of life,  $\lambda$  calculus, and many others, then we can see that these results also have at most a polynomial overhead in the simulation in each way.<sup>4</sup> It is a good exercise to go through, for example, the proof of [Theorem 9.1](#) and verify that it establishes that Turing machines and NAND++ programs are equivalent up to polynomial overhead.

<sup>4</sup> One technical point is that for  $\lambda$  calculus, one needs to be careful about the order of application of the reduction steps, which can matter for computational efficiency. Counting running time for  $\lambda$  calculus is somewhat delicate, see [this paper](#).

## 14.2 Efficient universal machine: a NAND« interpreter in NAND«

We have seen in [Theorem 8.2](#) the “universal program” or “interpreter”  $U$  for NAND++. Examining that proof, and combining it with [Theorem 14.1](#), we can see that the program  $U$  has a *polynomial* overhead, in the sense that it can simulate  $T$  steps of a given NAND++ (or NAND«) program  $P$  on an input  $x$  in  $O(T^a)$  steps for some constant  $a$ . But in fact, by directly simulating NAND« programs, we can do better with only a *constant* multiplicative overhead:

**Theorem 14.2 — Efficient universality of NAND«.** There is a NAND« program  $U$  that computes the partial function  $\text{TIMEDEVAL} : \{0,1\}^* \rightarrow \{0,1\}^*$  defined as follows:

$$\text{TIMEDEVAL}(P, x, 1^T) = P(x) \quad (14.1)$$

if  $P$  is a valid representation of a NAND« program which produces an output on  $x$  within at most  $T$  steps. Moreover, for every program  $P$ , the running time of  $U$  on input  $P, x, 1^T$  is  $O(T)$ . (The hidden constant in the Oh notation can depend on the program  $P$  but is at most polynomial in the length of  $P$ 's description as a string.).

**P** Before reading the proof of [Theorem 14.2](#), try to think how you would compute *TIMEDEVAL* using your favorite programming language. That is, how you would write a program *TIMEDEVAL*(*P*, *x*, *T*) that gets a NAND« program *P* (represented in some convenient form), a string *x*, and an integer *T*, and simulates *P* for *T* steps. You will likely find that your program requires  $O(T)$  steps to perform this simulation.

*Proof of Theorem 14.2.* To present a universal NAND« program in full we need to describe a precise representation scheme, as well as the full NAND« instructions for the program. While this can be done, it is more important to focus on the main ideas, and so we just sketch the proof here. A complete specification for NAND« is given in the Appendix, and for the purposes of this simulation, we can simply use the representation of the code NAND« as an ASCII string.

The program *U* gets as input a NAND« program *P*, an input *x*, and a time bound *T* (given in the form  $1^T$ ) and needs to simulate the execution of *P* for *T* steps. To do so, *U* will do the following:

1. *U* will maintain variables *icP*, *lcP*, and *iP* for the iteration counter, line counter, and index variable of *P*.
2. *U* will maintain an array *varsP* for all other variables of *P*. If *P* has *s* lines then it uses at most  $3s$  variable identifiers. *U* will associate each identifier with a number in  $[3s]$ . It will encode the contents of the variable with identifier corresponding to *a* and index *j* at the location  $\text{varsP}_{\langle 3s \cdot j + a \rangle}$ .
3. *U* will maintain an array *LinesP* of  $O(s)$  size that will encode the lines of *P* in some canonical encoding.
4. To simulate a single step of *P*, the program *U* will recover the line corresponding to *lcP* from the *LinesP* and execute it. Since NAND« has a constant number of arithmetic operations, we can simulate choosing which operation to execute with a sequence of a constantly many if-then-else's.<sup>5</sup> When executing these operations, *U* will use the variable *icP* that keeps track of the iteration counter of *P*.

Simulating a single step of *P* will take *U*  $O(s)$  steps, , and hence the simulation will be  $O(sT)$  which is  $O(T)$  when suppressing constants such as *s* that depend on the program *P*. ■

<sup>5</sup> While NAND« does not formally have if/then/else, we can easily add this as syntactic sugar.

### 14.3 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 *can* be computed in time  $2^n$ , but *cannot* be computed in time  $2^{0.9n}$ ? It turns out that the answer is **Yes**:

**Theorem 14.3 — Time Hierarchy Theorem.** For every nice function  $T$ , there is a function  $F : \{0,1\}^* \rightarrow \{0,1\}$  in  $\text{TIME}(T(n) \log n) \setminus \text{TIME}(T(n))$ .<sup>6</sup>

Note that in particular this means that  $\mathbf{P} \neq \mathbf{EXP}$ .

**Proof Idea:** In the proof of [Theorem 10.2](#) (the uncomputability of the Halting problem), we have shown that the function  $\text{HALT}$  cannot be computed in any finite time. An examination of the proof shows that it gives something stronger. Namely, the proof shows that if we fix our computational budget to be  $T$  steps, then the proof shows that not only we can't distinguish between programs that halt and those that do not, but cannot even distinguish between programs that halt within at most  $T'$  steps and those that take more than that (where  $T'$  is some number depending on  $T$ ). Therefore, the proof of [Theorem 14.3](#) follows the ideas of the uncomputability of the halting problem, but again with a more careful accounting of the running time.

*Proof of Theorem 14.3.* Recall the Halting function  $\text{HALT} : \{0,1\}^* \rightarrow \{0,1\}$  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 *Bounded Halting* function  $\text{HALT}_T(P, x)$  to equal 1 for every NAND<sup>«</sup> program  $P$  such that  $|P| \leq \log \log |x|$ , and such that  $P$  halts on the input  $x$  within  $100T(|x|)$  steps.  $\text{HALT}_T$  equals 0 on all other inputs.<sup>7</sup>

[Theorem 14.3](#) is an immediate consequence of the following two claims:

**Claim 1:**  $\text{HALT}_T \in \text{TIME}(T(n) \log n)$

and

**Claim 2:**  $\text{HALT}_T \notin \text{TIME}(T(n))$ .

<sup>6</sup> There is nothing special about  $\log n$ , and we could have used any other efficiently computable function that tends to infinity with  $n$ .

<sup>7</sup> The constant 100 and the function  $\log \log n$  are rather arbitrary, and are chosen for convenience in this proof.

We now turn to proving the two claims.

**Proof of claim 1:** We can easily check whether an input has the form  $P, x$  where  $|P| \leq \log \log |x|$  in linear time. Since  $T(\cdot)$  is a nice function, we can evaluate it in  $O(T(n))$  time. Then, using the universal NAND $\llcorner$  program of [Theorem 14.2](#), we can evaluate  $\text{HALT}_T$  in at most  $\text{poly}(|P|)T(n)$  steps.<sup>8</sup>

Since  $(\log \log n)^a = o(\log n)$  for every  $a$ , this will be smaller than  $T(n) \log n$  for every sufficiently large  $n$ .

**Proof of claim 2:** The proof is very reminiscent of the proof that  $\text{HALT}$  is not computable. Assume, toward the sake of contradiction, that there is some NAND $\llcorner$  program  $P^*$  that computes  $\text{HALT}_T(P)$  within  $T(|P|)$  steps. We are going to show a contradiction by creating a program  $Q$  and showing that under our assumptions, if  $Q$  runs for less than  $T(n)$  steps when given (a padded version of) its own code as input then it actually runs for more than  $T(n)$  steps and vice versa. (It is worth re-reading the last sentence twice or thrice to make sure you understand this logic. It is very similar to the direct proof of the uncomputability of the halting problem where we obtained a contradiction by using an assumed “halting solver” to construct a program that, given its own code as input, halts if and only if it does not halt.)

We will define  $Q$  to be the program that on input a string  $z$  does the following:

1. If  $z$  does not have the form  $z = P1^m$  where  $P$  represents a NAND $\llcorner$  program and  $|P| < 0.1 \log \log m$  then return 0.
2. Compute  $b = P^*(P, z)$  (at a cost of at most  $T(|P| + |z|)$  steps, under our assumptions).
3. If  $b = 1$  then  $Q$  goes into an infinite loop, otherwise it halts.

We chose  $m$  sufficiently large so that  $|Q| < 0.001 \log \log m$  where  $|Q|$  denotes the length of the description of  $Q$  as a string. We will reach a contradiction by splitting into cases according to whether or not  $\text{HALT}_T(Q, Q1^m)$  equals 0 or 1.

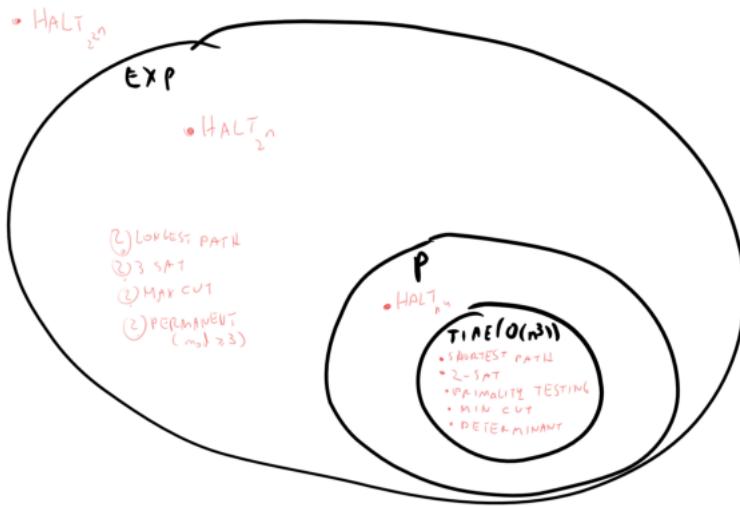
On the one hand, if  $\text{HALT}_T(Q, Q1^m) = 1$ , then under our assumption that  $P^*$  computes  $\text{HALT}_T$ ,  $Q$  will go into an infinite loop on input  $z = Q1^m$ , and hence in particular  $Q$  does *not* halt within  $100T(|Q| + m)$  steps on the input  $z$ . But this contradicts our assumption that  $\text{HALT}_T(Q, Q1^m) = 1$ .

This means that it must hold that  $\text{HALT}_T(Q, Q1^m) = 0$ . But in this case, since we assume  $P^*$  computes  $\text{HALT}_T$ ,  $Q$  does not do

<sup>8</sup> Recall that we use  $\text{poly}(m)$  to denote a quantity that is bounded by  $am^b$  for some constants  $a, b$  and every sufficiently large  $m$ .

anything in phase 3 of its computation, and so the only computation costs come in phases 1 and 2 of the computation. It is not hard to verify that Phase 1 can be done in linear and in fact less than  $5|z|$  steps. Phase 2 involves executing  $P^*$ , which under our assumption requires  $T(|Q| + m)$  steps. In total we can perform both phases in less than  $10T(|Q| + m)$  in steps, which by definition means that  $\text{HALT}_T(Q, Q1^m) = 1$ , but this is of course 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.



**Figure 14.2:** Some complexity classes and some of the functions we know (or conjecture) to be contained in them.

#### 14.4 Simulating NAND $\ll$ or NAND $\ll+$ 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 14.4 — Nonuniform computation contains uniform computa-**

**tion.** There is some  $c \in \mathbb{N}$  s.t. for every  $F : \{0,1\}^* \rightarrow \{0,1\}$  in  $\text{TIME}_{++}(T(n))$  and every sufficiently large  $n \in \mathbb{N}$ ,  $F_n$  is in  $\text{SIZE}(cT(n))$ .

*Proof.* The proof follows by the “unraveling” argument that we’ve already seen in the proof of [Theorem 7.2](#). Given a NAND++ program  $P$  and some function  $T(n)$ , we can transform NAND++ to be “simple” in the sense of [Definition 7.2](#), and by direct examination this transformation costs at most a factor of 4 in the running time. Thus we can construct a NAND program on  $n$  inputs and with less than  $2T(n)$  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. While the original NAND++ program  $P$  might have ended on some inputs *before*  $T(n)$  iterations have passed, by transforming it to be simple we ensure that there is no harm in “extra” iterations, since all assignments to the output are “guarded” by ensuring they make no difference if the program should have already halted before. ■

Note that by combining it with [Theorem 14.1](#), [Theorem 14.4](#) implies that if  $F \in \text{TIME}(T(n))$  then there are some constants  $a, b$  such that for every large enough  $n$ ,  $F_n \in \text{SIZE}(aT(n)^b)$ . (In fact, by direct inspection of the proofs we can see that  $a = 1$  and  $b = 5$  would work.)

**Algorithmic version: the “NAND++ to NAND compiler”:** The transformation of the NAND++ program  $P$  to the NAND program  $Q_P$  is itself algorithmic. (Indeed it can be done in about 5 lines of Python.) Thus we can also phrase this result as follows:

**Theorem 14.5 — NAND++ to NAND compiler.** There is an  $O(n)$ -time

NAND« program *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  $O(T)$  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.

#### 14.5 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 **no**. Indeed, consider the following “unary halting function”  $UH : \{0,1\}^* \rightarrow \{0,1\}$  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 *constant* number of lines.

The issue here is *uniformity*. For a function  $F : \{0,1\}^* \rightarrow \{0,1\}$ , if  $F$  is in  $\text{TIME}(T(n))$  then we have a *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, for “natural” functions, if you pretend that  $\text{TIME}(T(n))$  is roughly the same as  $\text{SIZE}(T(n))$ , you will be right more often than wrong.

#### 14.5.1 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  $F : \{0,1\}^* \rightarrow \{0,1\}$  and some nice time bound  $T : \mathbb{N} \rightarrow \mathbb{N}$ , 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  $F : \{0,1\}^n \rightarrow \{0,1\}$  such that  $F_n$  can be computed by even a constant size NAND program but  $F$  is uncomputable.

Note that the *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 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  $\mathbf{P}/\text{poly}$ , 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.

#### 14.6 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 thesis is the statement that the same holds for *efficiently computable* functions, which is typically

interpreted as saying that NAND++ programs can simulate every physically realizable computing device with polynomial overhead.

In other words, the extended Church Turing thesis says that for every *scalable computing device*  $C$  (which has a finite description but can be in principle used to run computation on arbitrarily large inputs), there are some constants  $a, b$  such that for every function  $F : \{0, 1\}^* \rightarrow \{0, 1\}$  that  $C$  can compute on  $n$  length inputs using an  $S(n)$  amount of physical resources,  $F$  is in  $\text{TIME}(aS(n)^b)$ .

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, which would be experimentally testable.

In the last hundred+ years of studying and mechanizing computation, no one has yet constructed a scalable computing device (or even gave a convincing blueprint) that violates the extended Church Turing Thesis. That said, as we mentioned before *quantum computing*, if realized, does pose a serious challenge to this thesis. However, even if the promises of quantum computing are fully realized, it still seems that the extended Church-Turing thesis is fundamentally or “morally” correct, in the sense that, while we do need to adapt the thesis to account for the possibility of quantum computing, its broad outline remains unchanged. We are still able to model computation mathematically, we can still treat programs as strings and have a universal program, and we still have hierarchy and uncomputability results.<sup>9</sup> Moreover, for most (though not all!) concrete problems we care about, the prospect of quantum computing does not seem to change their time complexity. 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.

<sup>9</sup> Note that indeed, quantum computing is *not* a challenge to the Church Turing itself, as a function is computable by a quantum computer if and only if it is computable by a “classical” computer or a NAND++ program. It is only the running time of computing the function that can be affected by moving to the quantum model.

#### 14.7 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.

## 14.8 Exercises

**Exercise 14.1 — Composition of polynomial time.** Prove that if  $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$  are in  $\overline{\mathbf{P}}$  then their *composition*  $F \circ G$ , which is the function  $H$  s.t.  $H(x) = F(G(x))$ , is also in  $\overline{\mathbf{P}}$ . ■

**Exercise 14.2 — Non composition of exponential time.** Prove that there is some  $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$  s.t.  $F, G \in \overline{\mathbf{EXP}}$  but  $F \circ G$  is not in  $\mathbf{EXP}$ .<sup>10</sup>

**Exercise 14.3 — 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  $t \in \{1, \dots, T(n)\}$ , after  $P$  executes the  $t^{th}$  step of the execution the value of the index  $i$  is equal to  $t(n, i)$ . In particular this value does *not* depend on  $x$  but only on its length.<sup>11</sup> 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  $O(T^2(n) \log T(n))$ . ■

<sup>12</sup>

**Exercise 14.4 — 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  $NANDEVAL \in \overline{\text{TIME}}(\tilde{O}(T(n)^2))$ . For extra credit prove that  $NANDEVAL \in \overline{\text{TIME}}(\tilde{O}(T(n)))$ .

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  $COMPILE \in \overline{\text{TIME}}(\tilde{O}(n))$ . Use that to show that  $TIMEDEVAL \in \overline{\text{TIME}}(\tilde{O}(n))$ . ■

<sup>10</sup> TODO: check that this works, idea is that we can do bounded halting.

<sup>11</sup> 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.

<sup>12</sup> TODO: Add exercise showing NAND is like NAND++ with advice. Mention the definition of  $\mathbf{P}_{/poly}$ .

### 14.9 *Bibliographical notes*

<sup>13</sup>

<sup>13</sup> TODO: add reference to best algorithm for longest path - probably the Bjorklund algorithm

### 14.10 *Further explorations*

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

### 14.11 *Acknowledgements*

### Learning Objectives:

- Introduce the notion of *polynomial-time reductions* as a way to relate the complexity of problems to one another.
- See several examples of such reductions.
- 3SAT as a basic starting point for reductions.

15

## *Polynomial-time reductions*

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.
- Each one of these problems is a *search* problem, whereby we search for a solution that is “good” in some easy to define sense (e.g., a long path, a satisfying assignment, etc..).
- Each of these problems has trivial exponential time algorithms that involve enumerating all possible solutions.
- At the moment, for all these problems the best known algorithms are not much better than the trivial one in the worst-case.

In this lecture and the next one we will see that, despite their apparent differences, we can relate their complexity. In fact, it turns out that 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.

### 15.0.1 Decision problems

For reasons of technical conditions rather than anything substantial, we will concern ourselves with *decision problems* (i.e., Yes/No questions) or in other words *Boolean* (i.e., one-bit output) functions. Thus, we will model all the problems as functions mapping  $\{0,1\}^*$  to  $\{0,1\}$ :

- The  $3SAT$  problem can be phrased as the function  $3SAT : \{0,1\}^* \rightarrow \{0,1\}$  that maps a  $3CNF$  formula  $\varphi$  to 1 if there exists some assignment  $x$  that satisfies it and to 0 otherwise.<sup>1</sup>
- The *quadratic equations* problem corresponds to the function  $QUADEQ : \{0,1\}^* \rightarrow \{0,1\}$  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  $LONGPATH : \{0,1\}^* \rightarrow \{0,1\}$  that maps a graph  $G$  and a number  $k$  to 1 if there is a simple<sup>2</sup> path in  $G$  of length at least  $k$  and maps  $(G, k)$  to 0 otherwise. The longest path problem is a generalization of the well known **Hamiltonian Path Problem** of determining whether a path of length  $n$  exists in a given  $n$  vertex graph.
- The *maximum cut* problem corresponds to the function  $MAXCUT : \{0,1\}^* \rightarrow \{0,1\}$  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.

<sup>1</sup> We assume some representation of formulas as strings, and define the function to output 0 if its input is not a valid representation. We will use the same convention for all the other functions below.

<sup>2</sup> Recall that a *simple* path in a graph is one that does not visit any vertex more than once. For the *shortest path problem* we can assume that a path is simple without loss of generality since removing a loop (a portion of the path that starts from the same vertex and returns to it) only makes the path shorter. For the *longest path problem* we need to make this restriction to avoid “degenerate” paths such as paths that repeat endlessly the same loop.

## 15.1 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*:<sup>3</sup>

**Definition 15.1 — 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  $x \in \{0,1\}^*$ ,

$$F(x) = G(R(x)) . \quad (15.1)$$

We say that  $F$  and  $G$  have *equivalent complexity* if  $F \leq_p G$  and  $G \leq_p F$ .

<sup>3</sup> Several notions of reductions are defined in the literature. The notion defined in [Definition 15.1](#) is often known as a *mapping reduction*, *many-to-one reduction* or a *Karp reduction*.

**F.**

If  $F \leq_p G$  and  $G$  is computable in polynomial time (i.e.,  $G \in \mathbf{P}$ ), then  $F$  is computable in polynomial time as well. Indeed, Eq. (15.1) shows a way how to compute  $F$  by applying the polynomial-time reduction  $R$  and then the polynomial-time algorithm for computing  $F$ . One can think of  $F \leq_p G$  as saying that (as far as polynomial-time computation is concerned)  $F$  is “easier or equal in difficulty to”  $G$ . With this interpretation, we would expect that if  $F \leq_p G$  and  $G \leq_p H$  then it would hold that  $F \leq_p H$  and indeed this is the case:

**Lemma 15.1** For every  $F, G, H : \{0, 1\}^* \rightarrow \{0, 1\}$ , if  $F \leq_p G$  and  $G \leq_p H$  then  $F \leq_p H$ .

**P**

We leave the proof of Lemma 15.1 as Exercise 15.2. Pausing now and doing this exercise is an excellent way to verify that you understood the definition of reductions.

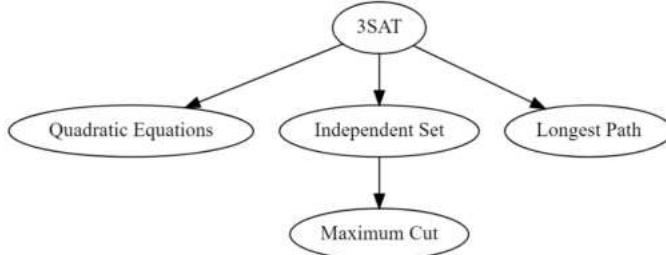
**R**

**Polynomial reductions** We have seen reductions before in the context of proving uncomputability of problems such as *HALTONZERO* and others. The most crucial difference between the notion in Definition 15.1 and previously occurring notions is that in the context of relating the time complexity of problems, we need the reduction to be computable in *polynomial time*, as opposed to merely computable. Definition 15.1 also restricts reductions to have a very specific format. That is, to show that  $F \leq_p G$ , rather than allowing a general algorithm for  $F$  that uses a “magic box” that computes  $G$ , we only allow an algorithm that computes  $F(x)$  by outputting  $G(R(x))$ . This restricted form is convenient for us, but people have defined and used more general reductions as well.

## 15.2 Some example reductions

We will now use reductions to relate the computational complexity of the problems mentioned above- 3SAT, Quadratic Equations, Maximum Cut, and Longest Path. We start by reducing 3SAT to the latter three problems, demonstrating that solving either of them will solve it 3SAT. Along the way we will introduce one more problem: the *independent set* problem. Like the others it shares the characteristics that it is an important and well motivated computational problem,

and that the best known algorithm for it takes exponential time. In the next lecture we will show the other direction: reducing each one of these problems to 3SAT in one fell swoop.



**Figure 15.1:** Our first stage in showing equivalence is to reduce 3SAT to the three other problems

### 15.2.1 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 15.2 — Hardness of quadratic equations.**

$$3SAT \leq_p QUADEQ \quad (15.2)$$

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.

**Proof Idea:** At the end of the day, a 3SAT formula can be thought of as a list of equations on some variables  $x_0, \dots, x_{n-1}$ . Namely the equations are that each of the  $x_i$ 's should be equal to either 0 or 1, and that the variables should satisfy some set of constraints which corresponds to the OR of three variables or their negation. To show that  $3SAT \leq_p QUADEQ$  we need to give a polynomial-time reduction that maps a 3SAT formula  $\varphi$  into a set of quadratic equations  $E$  such that  $E$  has a solution if and only if  $\varphi$  is satisfiable. The idea is that we can transform a 3SAT formula  $\varphi$  first to a set of *cubic* equations by mapping every constraint of the form  $(x_{12} \vee \bar{x}_{15} \vee x_{24})$  into an equation of the form  $(1 - x_{12})x_{15}(1 - x_{24}) = 0$ . We can then turn this into a *quadratic equation* by mapping any cubic equation of the form  $x_i x_j x_k = 0$  into the two quadratic equations  $y_{i,j} = x_i x_j$  and  $y_{i,j} x_k = 0$ .

*Proof of Theorem 15.2.* To prove Theorem 15.2 we need to give a polynomial-time transformation of every 3SAT formula  $\varphi$  into a set of quadratic equations  $E$ , and prove that  $\text{3SAT}(\varphi) = \text{QUADEQ}(E)$ .

We now describe the transformation of a formula  $\varphi$  to equations  $E$  and show the completeness and soundness conditions. Recall that a 3SAT formula  $\varphi$  is a formula such as  $(x_{17} \vee \bar{x}_{101} \vee x_{57}) \wedge (x_{18} \vee \bar{x}_{19} \vee \bar{x}_{101}) \wedge \dots$ . 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  $x_{19}x_{52} - x_{12} + 2x_{33} = 2$ , etc.. We will include the constraints  $x_i^2 - x_i = 0$  for every  $i \in [n]$  in our equations, which means that we can restrict attention to assignments where  $x_i \in \{0, 1\}$  for every  $i$ .

There is a natural way to map a 3SAT instance into a set of *cubic* equations, and that is to map a clause such as  $(x_{17} \vee \bar{x}_{101} \vee x_{57})$  to the equation  $(1 - x_{17})x_{101}(1 - x_{57}) = 0$ . 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  $E$ . To make the equations *quadratic* we introduce for every  $i, j \in [n]$  a variable  $y_{i,j}$  and include the constraint  $y_{i,j} - x_i x_j = 0$  in the equations. This is a quadratic equation that ensures that  $y_{i,j} = x_i x_j$  for every  $i, j \in [n]$ . Now we can turn any cubic equation in the  $x$ 's into a quadratic equation in the  $x$  and  $y$  variables. For example, we can “open up the parenthesis” of an equation such as  $(1 - x_{17})x_{101}(1 - x_{57}) = 0$  to  $x_{101} - x_{17}x_{101} - x_{101}x_{57} + x_{17}x_{101}x_{57} = 0$ . We can now replace the cubic term  $x_{17}x_{101}x_{57}$  with the quadratic term  $y_{17,101}x_{57}$ . This can be done for every cubic equation in the same way, replacing any cubic term  $x_i x_j x_k$  with the term  $y_{i,j}x_k$ . The bottom line is that we get a set  $E$  of quadratic equations in the variables  $x_0, \dots, x_{n-1}, y_{0,0}, \dots, y_{n-1,n-1}$  such that the 3SAT formula  $\varphi$  is satisfiable if and only if the equations  $E$  are satisfiable. ■

### 15.3 The independent set problem

For a graph  $G = (V, E)$ , an **independent set** (also known as a *stable set*) is a subset  $S \subseteq V$  such that there are no edges with both endpoints in  $S$  (in other words,  $E(S, S) = \emptyset$ ). Every “singleton” (set consisting of a single vertex) is trivially an independent set, but finding larger independent sets can be challenging. The *maximum independent set* problem (hencefore simply “independent set”) is the task of finding the largest independent set in the graph.<sup>4</sup> The inde-

<sup>4</sup> While we will not consider it here, people have also looked at the *maximal* (as opposed to *maximum*) independent set, which is the task of finding a “local maximum” of an independent set: an independent set  $S$  such that one cannot add a vertex to it without losing the independence property (such a set is known as a *vertex cover*). Finding a maximal independent set can be done efficiently by a greedy algorithm, but this local maximum can be much smaller than the global maximum.

pendent set problem is naturally related to *scheduling problems*: if we put an edge between two conflicting tasks, then an independent set corresponds to a set of tasks that can all be scheduled together without conflicts. But it also arises in very different settings, including trying to find structure in **protein-protein interaction graphs**.<sup>5</sup>

To phrase independent set as a decision problem, we think of it as a function  $ISET : \{0, 1\}^* \rightarrow \{0, 1\}$  that on input a graph  $G$  and a number  $k$  outputs 1 if and only if the graph  $G$  contains an independent set of size at least  $k$ . We will now reduce 3SAT to Independent set.

**Theorem 15.3 — Hardness of Independent Set.**  $3SAT \leq_p ISET$ .

<sup>5</sup> In the molecular biology literature, people often refer to the computationally equivalent **clique problem**.

**Proof Idea:** The idea is that finding a satisfying assignment to a 3SAT formula corresponds to satisfying many local constraints without creating any conflicts. One can think of “ $x_{17} = 0$ ” and “ $x_{17} = 1$ ” as two conflicting events, and of the constraints  $x_{17} \vee \bar{x}_5 \vee x_9$  as creating a conflict between the events “ $x_{17} = 0$ ”, “ $x_5 = 0$ ” and “ $x_9 = 0$ ”, saying that these three cannot simultaneously co-occur. Using these ideas, we can think of solving a 3SAT problem as trying to schedule non conflicting events, though the devil is, as usual, in the details.

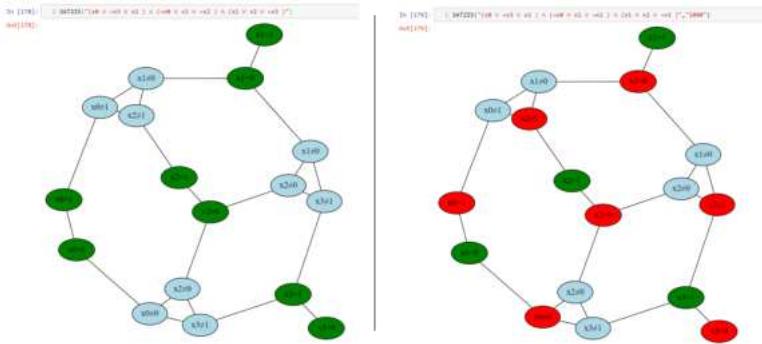
*Proof of Theorem 15.3.* Given a 3SAT formula  $\varphi$  on  $n$  variables and with  $m$  clauses, we will create a graph  $G$  with  $2n + 3m$  vertices as follows: (see Fig. 15.2 for an example)

- For every variable  $x_i$  of  $\varphi$ , we create a pair of vertices that are labeled “ $x_i = 0$ ” and “ $x_i = 1$ ”, and put an edge between them. Note that this means that every independent set  $S$  in the graph can contain at most one of those vertices.
- For every clause in  $\varphi$  involving the variables  $x_i, x_j, x_k$ , note that the clause is the OR of these three variables or their negations and so there are some  $a, b, c \in \{0, 1\}$  such that the clause is not satisfied if and only if  $x_i = a$ ,  $x_j = b$ , and  $x_k = c$ . We add three vertices to the graph with the labels “ $x_i \neq a$ ”, “ $x_j \neq b$ ” and “ $x_k \neq c$ ” and connect them with a triangle. This means that every independent set  $S$  in the graph can contain at most one of the members of this triangle. Moreover, we put an edge between the vertex labeled “ $x_i \neq a$ ” and the vertex we labeled “ $x_i = a$ ” which means that no independent set in the graph can contain both of them, and add analogous edges connecting “ $x_j \neq b$ ” with “ $x_j = b$ ” and “ $x_k \neq c$ ” with “ $x_k = c$ ”.

The construction of  $G$  based on  $\varphi$  can clearly be carried out in polynomial time. Hence to prove the theorem we need to show that  $\varphi$  is satisfiable if and only if  $G$  contains an independent set of  $n + m$  vertices. We now show both directions of this equivalence:

- **Completeness:** The “completeness” direction is to show that if  $\varphi$  has a satisfying assignment  $x^*$  then  $G$  has an independent set  $S^*$  of  $n + m$  vertices. Let us now show this. Indeed, suppose that  $\varphi$  has a satisfying assignment  $x^* \in \{0,1\}^n$ . Then there exists an  $n + m$  vertex independent set  $S^*$  in  $G$  that is constructed as follows: for every  $i$ , we include in  $S^*$  the vertex labeled “ $x_i = x_i^*$ ” and for every clause we choose one of the vertices in the clause whose label agrees with  $x^*$  (i.e., a vertex of the form  $x_j = b$  where  $b \neq x_j^*$ ) and add it to  $S^*$ . There must exist such a vertex as otherwise it would hold that  $x^*$  does not satisfy this clause (can you see why?). Moreover, such a vertex would not have a neighbor in  $S^*$  since we don’t add any other vertex from the triangle and in the case above the vertex “ $x_j = b$ ” is not a member of  $S^*$  since  $b \neq x^*$ . Since we added one vertex per variable and one vertex per clause, we get that  $S^*$  has  $n + m$  vertices, and by the reasoning above it is an independent set.
- **Soundness:** The “soundness” direction is to show that if  $G$  has an independent set  $S^*$  of  $n + m$  vertices, then  $\varphi$  has a satisfying assignment  $x^* \in \{0,1\}^n$ . Let us now show this. Indeed, suppose that  $G$  has an independent set  $S^*$  with  $n + m$  vertices. Out of the  $2n$  vertices corresponding to variables and their negation,  $S^*$  can contain at most  $n$ , since otherwise it would contain a neighboring pair of vertices of the form “ $x_i = 0$ ” and “ $x_i = 1$ ”. Out of the  $3m$  vertices corresponding to clauses,  $S^*$  can contain at most  $m$  since otherwise it would contain a pair of vertices inside the same triangle. Hence the only way  $S^*$  has  $n + m$  vertices is if it contains exactly  $n$  vertices out of those corresponding to variables and exactly  $m$  vertices out of those corresponding to clauses. Now define  $x^* \in \{0,1\}^n$  as follows: for every  $i \in [n]$ , we set  $x_i^* = a$  if the vertex “ $x_i = a$ ” is in  $S^*$  (since  $S^*$  is an independent set and contains  $n$  of these vertices, it will contain exactly one vertex of this form). We claim that  $x^*$  is a satisfying assignment for  $\varphi$ . Indeed, for every clause of  $\varphi$ , let  $x_i = a, x_j = b, x_k = c$  be the assignment that is “forbidden” by this clause. Then since  $S^*$  contains one vertex out of “ $x_i \neq a$ ”, “ $x_j \neq b$ ” and “ $x_k \neq c$ ” it must be that  $x^*$  does not match this assignment, as otherwise  $S^*$  would not be an independent set.

This completes the proof of Theorem 15.3 ■



**Figure 15.2:** An example of the reduction of 3SAT to IS. We reduce the formula of 4 variables and 3 clauses into a graph of  $8 + 3 \cdot 3 = 17$  vertices. On the lefthand side is the resulting graph, where the green vertices correspond to the variable and the light blue vertices correspond to the clauses. On the righthand side is the 7 vertex independent set corresponding to a particular satisfying assignment. This example is taken from a jupyter notebook, the code of which is available.

#### 15.4 Reducing Independent Set to Maximum Cut

**Theorem 15.4 — Hardness of Max Cut.**  $ISET \leq_p MAXCUT$

**Proof Idea:** We will map a graph  $G$  into a graph  $H$  such that a large independent set in  $G$  becomes a partition cutting many edges in  $H$ . We can think of a cut in  $H$  as coloring each vertex either “blue” or “red”. We will add a special “source” vertex  $s$ , connect it to all other vertices, and assume without loss of generality that it is colored blue. Hence the more vertices we color red, the more edges from  $x$  we cut. Now, for every edge  $u, v$  in the original graph  $G$  we will add a special “gadget” which will be a small subgraph that involves  $u, v$ , the source  $x$ , and two other additional vertices. We design the gadget in a way so that if the red vertices are not an independent set in  $G$  then the corresponding cut in  $H$  will be “penalized” in the sense that it would not cut as many edges. Once we set for ourselves this objective, it is not hard to find a gadget that achieves it, see the proof below.

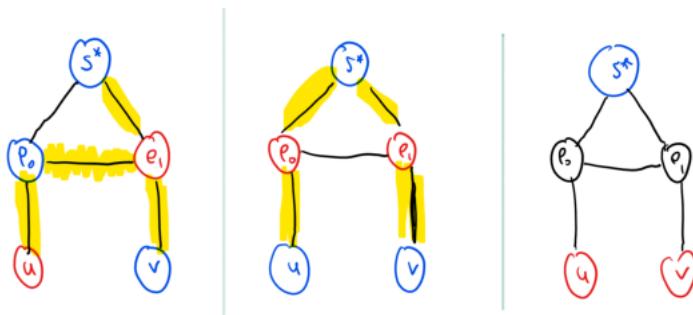
*Proof of Theorem 15.4.* We will transform a graph  $G$  of  $n$  vertices and  $m$  edges into a graph  $H$  of  $n + 1 + 2m$  vertices and  $n + 5m$  edges in the following way. The graph  $H$  will contain all vertices of  $G$  (though not the edges between them!) and in addition to that will contain:

- \* A special vertex  $s^*$  that is connected to all the vertices of  $G$
- \* For every edge  $e = \{u, v\} \in E(G)$ , two vertices  $e_0, e_1$  such that  $e_0$  is connected to  $u$  and  $e_1$  is connected to  $v$ , and moreover we add the edges  $\{e_0, e_1\}, \{e_0, s^*\}, \{e_1, s^*\}$  to  $H$ .

Theorem 15.4 will follow by showing that  $G$  contains an independent set of size at least  $k$  if and only if  $H$  has a cut cutting at least  $k + 4m$  edges. We now prove both directions of this equivalence:

- **Completeness:** If  $I$  is an independent  $k$ -sized set in  $G$ , then we can define  $S$  to be a cut in  $H$  of the following form. We let  $S$  contain all the vertices of  $I$  and for every edge  $e = \{u, v\} \in E(G)$ , if  $u \in I$  and  $v \notin I$  then we add  $e_1$  to  $S$ , if  $u \notin I$  and  $v \in I$  then we add  $e_0$  to  $S$ , and if  $u \notin I$  and  $v \notin I$  then we add both  $e_0$  and  $e_1$  to  $S$ . (We don't need to worry about the case that both  $u$  and  $v$  are in  $I$  since it is an independent set.) We can verify that in all cases the number of edges in the gadget corresponding to  $e$  from  $S$  to its complement will be four (see Fig. 15.3). Since  $s^*$  is not in  $S$ , we also have  $k$  edges from  $s^*$  to  $I$ , for a total of  $k + 4m$  edges.
- **Soundness:** Suppose that  $S$  is a cut in  $H$  that cuts at least  $C = k + 4m$  edges. By changing to  $\bar{S}$  if needed (which does not change the size of the cut), we can assume that  $x^*$  is not in  $S$ . Now let  $I$  be the set of vertices of  $G$  that are in  $H$ , and let  $m_{in} = |E(I, I)|$  be the set of edges in  $G$  that are contained in  $I$  and let  $m_{out} = m - m_{in}$ . By the properties of our gadget we know that we can cut at most three edges when both vertices are in  $S$ , and at most four edges otherwise. Hence  $C \leq |I| + 3m_{in} + 4m_{out} = |I| + 3m_{in} + 4(m - m_{in}) = |I| + 4m - m_{in}$ . Since  $C = k + 4m$  we get that  $|I| - m_{in} \geq k$ . Now we can transform  $I$  into an independent set  $I' \subseteq I$  of size  $|I| - m_{in}$  by going over every edge  $e$  inside  $I$  and removing one of the endpoints of  $e$  from  $I$ . Therefore we get that  $G$  contains an independent set  $I'$  of at least  $k$  vertices.

■



**Figure 15.3:** In the reduction of independent set to max cut, we have a “gadget” corresponding to every edge  $e = \{u, v\}$  in the original graph. If we think of the side of the cut containing the special source vertex  $s^*$  as “blue” and the other side as “red”, then the leftmost and center figures show that if  $u$  and  $v$  are not both red then we can cut four edges from the gadget. In contrast, by enumerating all possibilities one can verify that if both  $u$  and  $v$  are red, then no matter how we color the intermediate vertices  $e_0, e_1$ , we will cut at most three edges from the gadget.

## 15.5 Reducing 3SAT to Longest Path

6

One of the most basic algorithms in Computer Science is Dijkstra's algorithm to find the *shortest path* between two vertices. We now show that in contrast, an efficient algorithm for the *longest path* problem would imply a polynomial-time algorithm for 3SAT.

<sup>6</sup> This section is still a little messy, feel free to skip it or just read it without going into the proof details

**Theorem 15.5 — Hardness of longest path.**

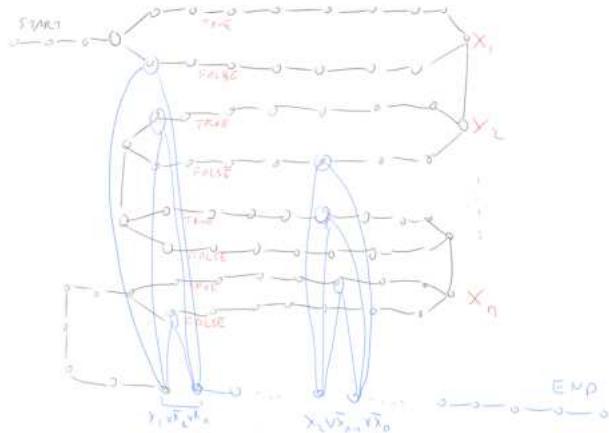
$$3SAT \leq_p LONGPATH \quad (15.3)$$

**Proof Idea:** To prove Theorem 15.5 need to show how to transform a 3CNF formula  $\varphi$  into a graph  $G$  and two vertices  $s, t$  such that  $G$  has a path of length at least  $k$  if and only if  $\varphi$  is satisfiable. The idea of the reduction is sketched in Fig. 15.4 and Fig. 15.5. We will construct a graph that contains a potentially long “snaking path” that corresponds to all variables in the formula. We will add a “gadget” corresponding to each clause of  $\varphi$  in a way that we would only be able to use the gadgets if we have a satisfying assignment.

*Proof of Theorem 15.5.* 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  $x_{17} \vee \bar{x}_{55} \vee x_{72}$  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  $x_{55}$  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 15.4:** 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  $x_{17} \vee \bar{x}_{32} \vee x_{57}$ , the corresponding vertices would have to have been not traversed before.

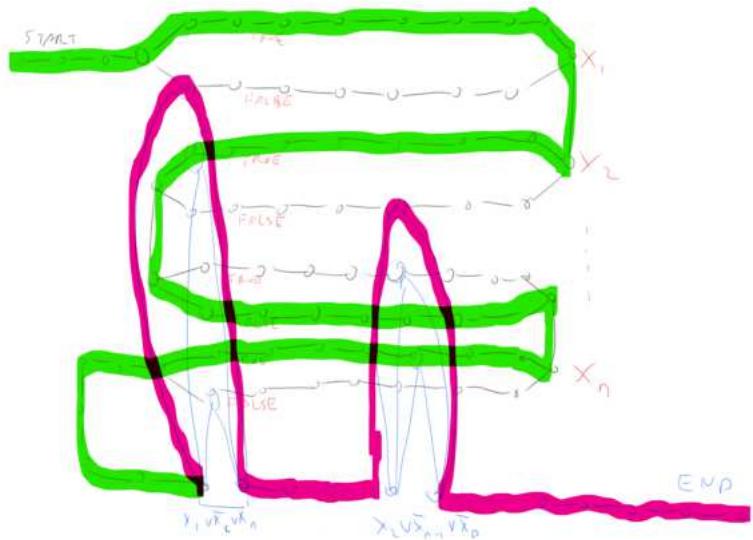
## 15.6 Exercises

7

**Exercise 15.1** Prove ?? ■

**Exercise 15.2 — Transitivity of reductions.** Prove that if  $F \leq_p G$  and  $G \leq_p H$  then  $F \leq_p H$ . ■

<sup>7</sup> 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.



**Figure 15.5:** The graph above with the longest path marked on it, the part of the path corresponding to variables is in green and part corresponding to the clauses is in pink.

### 15.7 Bibliographical notes

### 15.8 Further explorations

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

### 15.9 Acknowledgements

### Learning Objectives:

- Introduce the class NP capturing a great many important computational problems
- NP-completeness: evidence that a problem might be intractable.
- The P vs NP problem.

16

## NP, NP completeness, and the Cook-Levin Theorem

*"In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually"*, Richard Karp, 1972

*"It is not the verifier who counts; not the man who points out how the solver of problems stumbles, or where the doer of deeds could have done them better. The credit belongs to the man who actually searches over the exponential space of possibilities, whose face is marred by dust and sweat and blood; who strives valiantly; who errs, who comes short again and again . . . who at the best knows in the end the triumph of high achievement, and who at the worst, if he fails, at least fails while daring greatly, so that his place shall never be with those cold and timid souls who neither know victory nor defeat."*, paraphrasing Theodore Roosevelt (1910).

### 16.1 The class NP

So far we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path. But to show that these problems are *computationally 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. All of the problems we discussed in the previous lecture, and a great many other problems, share the same commonality: they are all

*search* problems, where the goal is to decide, given an instance  $x$  whether there exists a *solution*  $y$  that satisfies some condition that can be verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable, in Max-Cut the instance is a graph and the solution is a cut in the graph, and so on and so forth. It turns out that *every* such search problem can be reduced to 3SAT.

To make this precise, we make the following mathematical definition. We define the class **NP** to contain all Boolean functions that correspond to a *search problem* of the form above. That is, functions that output 1 on  $x$  if and only if there exists a solution  $w$  such that the pair  $(x, w)$  satisfies some polynomial-time checkable condition. Formally, **NP** is defined as follows:

**Definition 16.1 — 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  $G \in \mathbf{P}$  and for every  $x \in \{0, 1\}^n$

$$F(x) = 1 \Leftrightarrow \exists_{w \in \{0, 1\}^{an^b}} \text{s.t. } G(xw) = 1 \quad (16.1)$$

The name **NP** stands for “nondeterministic polynomial time” and is used for historical reasons, see the bibliographical notes. The string  $w$  in Eq. (16.1) is sometimes known as a *solution*, *certificate*, or *witness* for the instance  $x$ .

### 16.1.1 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.<sup>1</sup>
- QUADEQ is in **NP** since for every  $\ell$  variable instance of quadratic equations  $E$ ,  $QUADEQ(E) = 1$  if and only if there exists an assignment  $x \in \{0, 1\}^\ell$  that satisfies  $E$  and we can check this condition in polynomial time.
- ISET is in **NP** since for every graph  $G$  and integer  $k$ ,  $ISET(G, k) = 1$  if and only if there exists a set  $S$  of  $k$  vertices that contains no pair of neighbors in  $G$ .
- LONGPATH is in **NP** since for every graph  $G$  and integer  $k$ ,  $LONGPATH(G, k) = 1$  if and only if there exists a simple path

<sup>1</sup> 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. (16.1) as simply stipulating that the “solution”  $y$  to the problem  $x$  is of size at most  $poly(|x|)$ .

$P$  in  $G$  that is of length at least  $k$ , and we can check this condition in polynomial time.

- $\text{MAXCUT}$  is in  $\text{NP}$  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.

### 16.1.2 From $\text{NP}$ to $3\text{SAT}$

There are many, many, *many*, more examples of interesting functions we would like to compute that are easily shown to be in  $\text{NP}$ . What is quite amazing is that if we can solve  $3\text{SAT}$  then we can solve all of them!

The following is one of the most fundamental theorems in Computer Science:

**Theorem 16.1 — Cook-Levin Theorem.** For every  $F \in \text{NP}$ ,  $F \leq_p 3\text{SAT}$ .

We will soon show the proof of [Theorem 16.1](#), but note that it immediately implies that  $\text{QUADEQ}$ ,  $\text{LONGPATH}$ , and  $\text{MAXCUT}$  all reduce to  $3\text{SAT}$ . In fact, combining it with the reductions we've seen, it implies that all these problems are *equivalent!* For example, to reduce  $\text{QUADEQ}$  to  $\text{LONGPATH}$ , we can first reduce  $\text{QUADEQ}$  to  $3\text{SAT}$  using [Theorem 16.1](#) and use the reduction we've seen from  $3\text{SAT}$  to  $\text{LONGPATH}$ . There is of course nothing special about  $\text{QUADEQ}$  here- by combining [Theorem 16.1](#) with the reduction we saw, we see that just like  $3\text{SAT}$ , *every*  $F \in \text{NP}$  reduces to  $\text{LONGPATH}$ , and the same is true for  $\text{QUADEQ}$  and  $\text{MAXCUT}$ . All these problems are in some sense “the hardest in  $\text{NP}$ ” since an efficient algorithm for any one of them would imply an efficient algorithm for *all* the problems in  $\text{NP}$ . This motivates the following definition

**Definition 16.2 — NP completeness.** We say that  $G : \{0, 1\}^* \rightarrow \{0, 1\}$  is  $\text{NP}$  hard if for every  $F \in \text{NP}$ ,  $F \leq_p G$ . We say that  $G$  is  $\text{NP}$  complete if  $G$  is  $\text{NP}$  hard and it is in  $\text{NP}$ .

[Theorem 16.1](#) and the reductions we've seen in the last lecture show that despite their superficial differences,  $3\text{SAT}$ , quadratic equations, longest path, independent set, and maximum cut, are all  $\text{NP}$  complete. Many thousands of additional problems have been shown to be  $\text{NP}$  complete, arising from all the sciences, mathematics, eco-

nomics, engineering and many other fields.<sup>2</sup>

### 16.1.3 What does this mean?

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. (However, it is still an excellent idea for you to pause here and verify that you see why every  $F \in \text{P}$  will be in  $\text{NP}$  as per [Definition 16.1](#).) 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. (Again, please verify that you understand why this follows from the definition.)

The most famous conjecture in Computer Science is that  $\text{P} \neq \text{NP}$ . One way to refute this conjecture is to give a polynomial-time algorithm for even a single one of the  $\text{NP}$ -complete problems such as 3SAT, Max Cut, or the thousands of others that have been studied in all fields of human endeavors. The fact that these problems have been studied by so many people, and yet not a single polynomial-time algorithm for them was found, supports that conjecture that indeed  $\text{P} \neq \text{NP}$ . In fact, for many of these problems (including all the ones we mentioned above), we don't even know of a  $2^{o(n)}$  time algorithm! However, to the frustration of computer scientists, we have not yet been able to prove that  $\text{P} \neq \text{NP}$  or even rule out the existence of an  $O(n)$  time algorithm for 3SAT. Resolving whether or not  $\text{P} = \text{NP}$  is known as the [P vs NP problem](#). A million dollar prize has been offered for the solution of this problem, a [popular book](#) was written, and every year a new paper comes out claiming a proof of  $\text{P} = \text{NP}$  or  $\text{P} \neq \text{NP}$ , only to wither under the scrutiny.<sup>3</sup> The following [120 page survey of Aaronson](#), as well as [chapter 3 in Wigderson's upcoming book](#) are excellent sources for summarizing what is known about this problem.

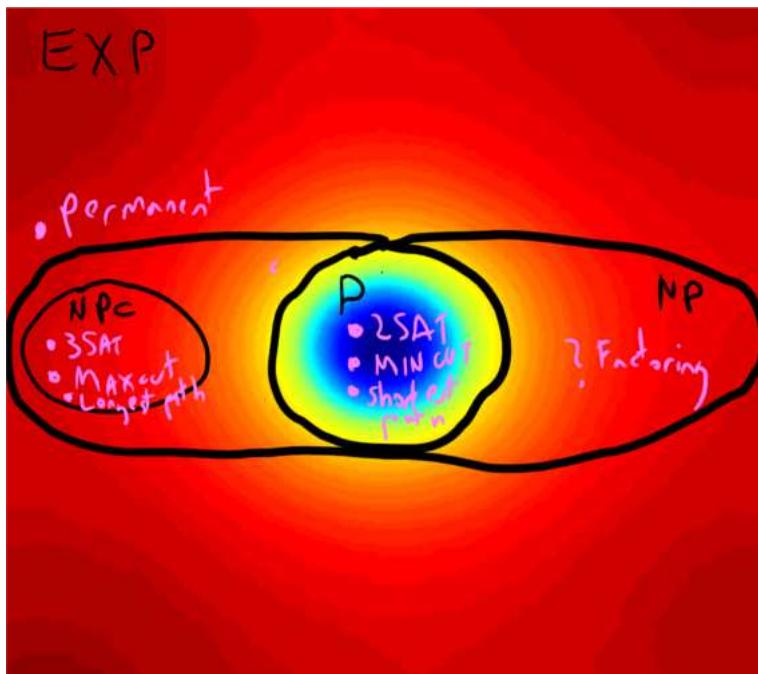
One of the mysteries of computation is that people have observed a certain empirical “zero one law” or “dichotomy” in the computational complexity of natural problems, in the sense that many natural problems are either in  $\text{P}$  (often in  $\text{TIME}(O(n))$  or  $\text{TIME}(O(n^2))$ ), or they are are  $\text{NP}$  hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, with not too many examples where the best running time is some strange intermediate complexity such as  $2^{2^{\sqrt{\log n}}}$ . However, it is believed that there exist problems in  $\text{NP}$  that are neither in  $\text{P}$  nor 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 (see also [Exercise 16.1](#)).

<sup>4</sup>

<sup>2</sup> For some partial lists, see [this Wikipedia page](#) and [this website](#).

<sup>3</sup> The following [web page](#) keeps a catalog of these failed attempts. At the time of the writing it lists about 110 papers claiming to resolve the question, of which about 60 claim to prove that  $\text{P} = \text{NP}$  and about 50 claim to prove that  $\text{P} \neq \text{NP}$ .

<sup>4</sup> TODO: maybe add examples of  $\text{NP}$  hard problems as a barrier to understanding - problems from economics, physics, etc.. that prevent having a closed-form solutions



**Figure 16.1:** 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 in NP. It is conjectured that all the NP complete problems require at least  $\exp(n^\epsilon)$  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.

## 16.2 The Cook-Levin Theorem

We will now prove the Cook-Levin Theorem, which 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**.

To prove [Theorem 16.1](#) we need to show that  $F \leq_p 3SAT$  for every  $F \in \mathbf{NP}$ . 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 16.1](#) will follow from the following three lemmas:

**Lemma 16.2** NANDSAT is **NP**-hard.

**Lemma 16.3** NANDSAT  $\leq_p$  3NAND.

**Lemma 16.4** 3NAND  $\leq_p$  3SAT.

From the transitivity of reductions, [Lemma 16.2](#), [Lemma 16.3](#), and [Lemma 16.4](#) together immediately imply that 3SAT is **NP**-hard, hence establishing [Theorem 16.1](#). (Can you see why?) We now prove these three lemmas one by one, providing the requisite definitions as we go along.

### 16.2.1 The NANDSAT Problem, and why it is **NP** hard.

We define the NANDSAT problem as follows. On input a string  $Q \in \{0,1\}^*$ , we define  $NANDSAT(Q) = 1$  if and only if  $Q$  is a valid representation of an  $n$ -input and single-output NAND program and there exists some  $w \in \{0,1\}^n$  such that  $Q(w) = 1$ . While we don't need this to prove [Lemma 16.2](#), note that NANDSAT is in **NP** since we can verify that  $Q(w) = 1$  using the polynomial-time algorithm for evaluating NAND programs.<sup>6</sup> We now present the proof of [Lemma 16.2](#).

<sup>6</sup>  $Q$  is a NAND program and not a NAND++ program, and hence it is only defined on inputs of some particular size  $n$ . Evaluating  $Q$  on any input  $w \in \{0,1\}^n$  can be done in time polynomial in the number of lines of  $Q$ .

**Proof Idea:** To prove Lemma 16.2 we need to show that for every  $F \in \mathbf{NP}$ ,  $F \leq_p \text{NANDSAT}$ . The high level idea is that by the definition of **NP**, there is some NAND++ program  $P^*$  and some polynomial  $T(\cdot)$  such that  $F(x) = 1$  if and only if there exists some  $w$  such that  $P^*(xw)$  outputs 1 within  $T(|x|)$  steps. Now by “unrolling the loop” of the NAND++  $P^*$  we can convert it into a NAND program  $Q$  that on input  $w$  will simulate  $P^*(xw)$  for  $T(|x|)$  steps. We will then get that  $\text{NANDSAT}(Q) = 1$  if and only if  $F(x) = 1$ .

*Proof of Lemma 16.2.* We now present the details. Let  $F \in \mathbf{NP}$ . By Definition 16.1 there exists  $G \in \mathbf{P}$  and  $a, b \in \mathbb{N}$  such that for every  $x \in \{0,1\}^*$ ,  $F(x) = 1$  if and only if there exists  $w \in \{0,1\}^{a|x|^b}$  such that  $G(xw) = 1$ . Since  $G \in \mathbf{P}$  there is some NAND++ program  $P^*$  that computes  $G$  in at most  $n'^c$  time for some constant  $c$  where  $n'$  is the size of its input. Moreover, as shown in Theorem 7.1, we can assume without loss of generality that  $P^*$  is simple in the sense of Definition 7.2.

To prove Lemma 16.2 we need to give a polynomial-time computable map of every  $x^* \in \{0,1\}^*$  to a NAND program  $Q$  such that  $F(x^*) = \text{NANDSAT}(Q)$ . Let  $x^* \in \{0,1\}^*$  be such a string and let  $n = |x^*|$  be its length. In time polynomial in  $n$ , we can obtain a NAND program  $Q^*$  of  $n + an^b$  inputs and  $|P^*| \cdot (n + an^b)^c$  lines (where  $|P^*|$  denotes the number of lines of  $P^*$ ) such that  $Q^*(xw) = P^*(xw)$  for every  $x \in \{0,1\}^n$  and  $w \in \{0,1\}^{an^b}$ . Indeed, we can do this by simply copying and pasting  $(n + an^b)^c$  times the code of  $P^*$  one after the other, and replacing all references to  $i$  in the  $j$ -th copy with  $\text{INDEX}(j)$ .<sup>7</sup> We also replace references to  $\text{valid}_k$  with one if  $k < an^b$  and zero otherwise. By the definition of NAND++ and the fact that the original program  $P^*$  was simple and halted within at most  $(n + an^b)^c$  steps, the NAND program  $Q^*$  agrees with  $P^*$  on every input of the form  $xw \in \{0,1\}^{n+an^b}$ .<sup>8</sup>

Now we transform  $Q^*$  into  $Q$  by “hardwiring” its first  $n$  inputs to correspond to  $x^*$ . That is, we obtain a program  $Q$  such that  $Q(w) = Q^*(x^*w)$  for every  $w \in \{0,1\}^{an^b}$  by replacing all references to the variables  $x_{-}\langle j \rangle$  for  $j < n$  with either one or zero depending on the value of  $x_j^*$ . (We also map  $x_{-}\langle n \rangle, \dots, x_{-}\langle n + an^b \rangle$  to  $x_{-}\langle 0 \rangle, \dots, x_{-}\langle an^b - 1 \rangle$  so that the number of inputs is reduced from  $n + an^b$  to  $an^b$ .) You can verify that by this construction  $Q$  has  $an^b$  inputs and for every  $w \in \{0,1\}^{an^b}$ ,  $Q(w) = Q^*(x^*w)$ . We now claim that  $\text{NANDSAT}(Q) = F(x^*)$ . Indeed note that  $F(x^*) = 1$  if and only if there exists  $w \in \{0,1\}^{an^b}$  s.t.  $P^*(x^*w) = 1$ . But since  $Q^*(xw) = P^*(xw)$  for every  $x, w$  of these lengths, and

<sup>7</sup> Recall that  $\text{INDEX}(j)$  is the value of the  $i$  index variable in the  $j$ -th iteration. The particular formula for  $\text{INDEX}(j)$  was given in Eq. (7.4) but all we care is that it is computable in time polynomial in  $j$ .

<sup>8</sup> We only used the fact that  $P^*$  is simple to ensure that we have access to the one and zero variables, and that assignments to the output variable  $y_{-}0$  are “guarded” in the sense that adding extra copies of  $P^*$  after it already halted will not change the output. It is not hard to ensure these properties as shown in Theorem 7.1.

$Q(w) = Q^*(x^*w)$  it follows that this holds if and only if there exists  $w \in \{0,1\}^{an^b}$  such that  $Q(w) = 1$ . But the latter condition holds exactly when  $NANDSAT(Q) = 1$ . ■

**P**

The proof above is a little bit technical but ultimately follows quite directly from the definition of NP, as well as NAND and NAND++ programs. If you find it confusing, try to pause here and work out the proof yourself from these definitions, using the idea of “unrolling the loop” of a NAND++ program. It might also be useful for you to think how you would implement in your favorite programming language the function `expand` which on input a NAND++ program  $P$  and numbers  $T, n$  would output an  $n$ -input NAND program  $Q$  of  $O(|T|)$  lines such that for every input  $x \in \{0,1\}^n$ , if  $P$  halts on  $x$  within at most  $T$  steps and outputs  $y$ , then  $Q(x) = y$ .

### 16.2.2 The 3NAND problem

The 3NAND problem is defined as follows: the input is a logical formula  $\varphi$  on a set of variables  $z_0, \dots, z_{r-1}$  which is an AND of constraints of the form  $z_i = NAND(z_j, z_k)$ . For example, the following is a 3NAND formula with 5 variables and 3 constraints:

$$(z_3 = NAND(z_0, z_2)) \wedge (z_1 = NAND(z_0, z_2)) \wedge (z_4 = NAND(z_3, z_1)) \quad (16.2)$$

The output of 3NAND on input  $\varphi$  is 1 if and only if there is an assignment to the variables of  $\varphi$  that makes it evaluate to “true” (that is, there is some assignment  $z \in \{0,1\}^r$  satisfying all of the constraints of  $\varphi$ ). As usual, we can represent  $\varphi$  as a string, and so think of 3NAND as a function mapping  $\{0,1\}^*$  to  $\{0,1\}$ . We now prove [Lemma 16.3](#).

**Proof Idea:** To prove [Lemma 16.3](#) we need to give a polynomial-time map from every NAND program  $Q$  to a 3NAND formula  $\varphi$  such that there exists  $w$  such that  $Q(w) = 1$  if and only if there exists  $z$  satisfying  $\varphi$ . This will actually follow directly from our notion of “modification logs” or “deltas” of NAND++ programs (see [Definition 7.4](#)). We will have a variable of  $\varphi$  corresponding to every line of  $Q$ , with a constraint ensuring that if line  $i$  has the form `foo := bar` `NAND` `blah` then the variable corresponding to line  $i$  should be the

NAND of the variables corresponding to the lines in which `bar` and `blah` were just written to. We will also have variables associated with the input  $w$ , and use them in lines such as `foo := x_17 NAND x_33` or `foo := bar NAND x_55`. Finally we add a constraint that requires the last assignment to  $y_0$  to equal 1. By construction satisfying assignments to our formula  $\varphi$  will correspond to valid modification logs of executions of  $Q$  that end with it outputting 1. Hence in particular there exists a satisfying assignment to  $\varphi$  if and only if there is some input  $w \in \{0,1\}^n$  on which the execution of  $Q$  on  $w$  ends in 1.

*Proof of Lemma 16.3.* To prove Lemma 16.3 we need to give a reduction from NANDSAT to 3NAND. Let  $Q$  be a NAND program with  $n$  inputs, one output, and  $m$  lines. We can assume without loss of generality that  $Q$  contains the variables one and zero by adding the following lines in its beginning if needed:

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

We map  $Q$  to a 3NAND formula  $\varphi$  as follows:

- $\varphi$  has  $m + n$  variables  $z_0, \dots, z_{m+n-1}$
- For every  $\ell \in \{n, n+1, \dots, n+m\}$ , if the  $\ell - n$ -th line of the program  $Q$  is `foo := bar NAND blah` then we add to  $\varphi$  the constraint  $z_\ell = \text{NAND}(z_j, z_k)$  where  $j - n$  and  $k - n$  correspond to the last lines in which the variables `bar` and `blah` (respectively) were written to. If one or both of `bar` and `blah` was not written to before then we use  $z_{\ell_0}$  instead of the corresponding value  $z_j$  or  $z_k$  in the constraint, where  $\ell_0 - n$  is the line in which `zero` is assigned a value. If one or both of `bar` and `blah` is an input variable `x_i` then we use  $z_i$  in the constraint.
- Let  $\ell^*$  be the last line in which the output  $y_0$  is assigned a value. Then we add the constraint  $z_{\ell^*} = \text{NAND}(z_{\ell_0}, z_{\ell_0})$  where  $\ell_0 - n$  is as above the last line in which `zero` is assigned a value. Note that this is effectively the constraint  $z_{\ell^*} = \text{NAND}(0, 0) = 1$ .

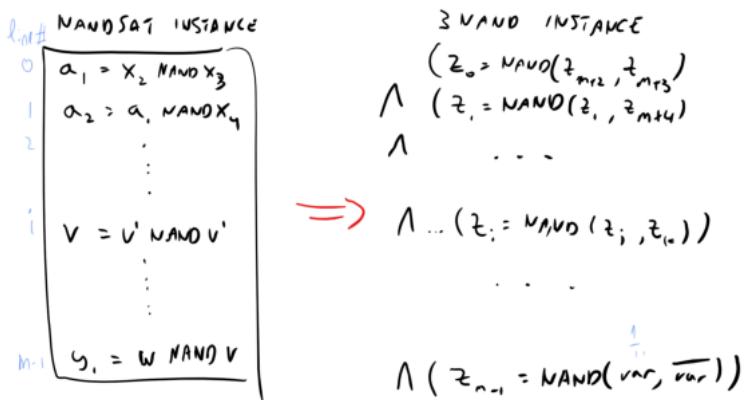
To complete the proof we need to show exists  $w \in \{0,1\}^n$  s.t.  $Q(w) = 1$  if and only if there exists  $z \in \{0,1\}^{n+m}$  that satisfies all constraints in  $\varphi$ . We now show both sides of this equivalence.

- **Completeness:** Suppose that there is  $w \in \{0,1\}^n$  s.t.  $Q(w) = 1$ . Let  $z \in \{0,1\}^{n+m}$  be defined as follows. For  $i \in [n]$ ,  $z_i = w_i$  and for  $i \in \{n, n+1, \dots, n+m\}$   $z_i$  equals the value that is assigned in the  $(i - n)$ -th line of  $Q$  when executed on  $w$ . Then by construction

$z$  satisfies all of the constraints of  $\varphi$  (including the constraint that  $z_{\ell^*} = \text{NAND}(0, 0) = 1$  since  $Q(w) = 1$ .)

- **Soundness:** Suppose that there exists  $z \in \{0, 1\}^{n+m}$  satisfying  $\varphi$ . Soundness will follow by showing that  $Q(z_0, \dots, z_{n-1}) = 1$  (and hence in particular there exists  $w \in \{0, 1\}^n$ , namesly  $w = z_0 \dots z_{n-1}$ , such taht  $Q(w) = 1$ ). To do this we will prove the following claim (\*): for every  $\ell \in [m]$ ,  $z_{\ell+n}$  equals the value assigned in the  $\ell$ -th step of the execution of the program  $Q$  on  $z_0, \dots, z_{n-1}$ . Note that because  $z$  satisfies the constraints of  $\varphi$ , (\*) is sufficient to prove the soundness condition since these constraints imply that the last value assigned to the variable  $y_{-0}$  in the execution of  $Q$  on  $z_0 \dots z_{n-1}$  is equal to 1. To prove (\*) suppose, towards a contradiction, that it is false, and let  $\ell$  be the smallest number such that  $z_{\ell+n}$  is *not* equal to the value assigned in the  $\ell$ -th step of the execution of  $Q$  on  $z_0, \dots, z_{n-1}$ . But since  $z$  satisfies the constraints of  $\varphi$ , we get that  $z_{\ell+n} = \text{NAND}(z_i, z_j)$  where (by the assumption above that  $\ell$  is *smallest* with this property) these values *do* correspond to the values last assigned by to the variables on the righthand side of the assignment operator in the  $\ell$ -th line of the program. But this means that the value assigned in the  $\ell$ -th step is indeed simply the NAND of  $z_i$  and  $z_j$ , contradicting our assumption on the choice of  $\ell$ .

■



**Figure 16.2:** We reduce *NANDSAT* to *3NAND* 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  $\psi$ .

### 16.2.3 From 3NAND to 3SAT

To conclude the proof of [Theorem 16.1](#), we need to show [Lemma 16.4](#) and show that  $3NAND \leq_p 3SAT$ . We now do so.

**Proof Idea:** To prove [Lemma 16.4](#) we need to map a 3NAND formula  $\varphi$  into a 3SAT formula  $\psi$  such that  $\varphi$  is satisfiable if and only if  $\psi$  is. The idea is that we can transform every NAND constraint of the form  $a = NAND(b, c)$  into the AND of ORs involving the variables  $a, b, c$  and their negations, where each of the ORs contains at most three terms. The construction is fairly straightforward, and the details are given below.



It is a good exercise for you to try to find a 3CNF formula  $\xi$  on three variables  $a, b, c$  such that  $\xi(a, b, c)$  is true if and only if  $a = NAND(b, c)$ . Once you do so, try to see why this implies a reduction from  $3NAND$  to  $3SAT$ , and hence completes the proof of [Lemma 16.4](#)

*Proof of Lemma 16.4.* The constraint

$$z_i = NAND(z_j, z_k) \quad (16.3)$$

is satisfied if  $z_i = 1$  whenever  $(z_j, z_k) \neq (1, 1)$ . By going through all cases, we can verify that [Eq. \(16.3\)](#) is equivalent to the constraint

$$(\overline{z_i} \vee \overline{z_j} \vee \overline{z_k}) \wedge (z_i \vee z_j) \wedge (z_i \vee z_k) . \quad (16.4)$$

Indeed if  $z_j = z_k = 1$  then the first constraint of [Eq. \(16.4\)](#) is only true if  $z_i = 0$ . On the other hand, if either of  $z_j$  or  $z_k$  equals 0 then unless  $z_i = 1$  either the second or third constraints will fail. This means that, given any 3NAND formula  $\varphi$  over  $n$  variables  $z_0, \dots, z_{n-1}$ , we can obtain a 3SAT formula  $\psi$  over the same variables by replacing every 3NAND constraint of  $\varphi$  with three 3OR constraints as in [Eq. \(16.4\)](#).<sup>9</sup> Because of the equivalence of [Eq. \(16.3\)](#) and [Eq. \(16.4\)](#), the formula  $\psi$  satisfies that  $\psi(z_0, \dots, z_{n-1}) = \varphi(z_0, \dots, z_{n-1})$  for every assignment  $z_0, \dots, z_{n-1} \in \{0, 1\}^n$  to the variables. In particular  $\psi$  is satisfiable if and only if  $\varphi$  is, thus completing the proof. ■

<sup>9</sup> 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"  $z_{n+m}$  and include it in all the OR's involving only two variables, and add a constraint requiring this dummy variable to be zero.

### 16.2.4 Wrapping up

We have shown that for every function  $F$  in **NP**,  $F \leq_p NANDSAT \leq_p 3NAND \leq_p 3SAT$ , and so  $3SAT$  is **NP-hard**. Since in the previous

lecture we have seen that  $3SAT \leq_p QUADEQ$ ,  $3SAT \leq_p ISET$ ,  $3SAT \leq_p MAXCUT$  and  $3SAT \leq_p LONGPATH$ , all these problems are **NP**-hard as well. Finally, since all the aforementioned problems are in **NP**, they are all in fact **NP**-complete and have equivalent complexity. There are thousands of other natural problems that are **NP** complete as well. Finding a polynomial time algorithm for one of them will imply a polynomial-time algorithm for all of them.

### 16.3 Lecture summary

- Many of the problems which we don't know polynomial-time algorithms for are **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  $\mathbf{NP} \neq \mathbf{P}$  which means that we believe that polynomial-time algorithms for these problems are not merely *unknown* but are *nonexistent*.
- While an **NP** hardness result means for example that a full fledged "textbook" solution to a problem such as MAX-CUT that is as 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*.

### 16.4 Exercises

**Exercise 16.1 — 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  $F \in \mathbf{NP}$  such that  $F \notin \mathbf{P}$  and  $F$  is not **NP** complete.<sup>10</sup>

<sup>10</sup> **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  $t = |\varphi|^{\log|\varphi|}$ .

### 16.5 Bibliographical notes

### 16.6 Further explorations

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

### 16.7 Acknowledgements



# 17

## *Advanced hardness reductions (advanced lecture)*

PLAN: show NP hardness of the permanent (maybe also mention Tutte polynomial?), maybe others such as the Dichotmoy theorem. Perhaps assume hardness of unique-SAT, which we will show in the randomness section. Maybe say something about PCP reductions?

### *17.1 Lecture summary*

### *17.2 Exercises*

### *17.3 Bibliographical notes*

### *17.4 Further explorations*

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

### *17.5 Acknowledgements*



## *What if P equals NP?*

*"There should be 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 ≠ 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 another way to phrase this question is as saying "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 God decided to make 3SAT easy then 3SAT will have an  $10^6 \cdot n$  (or at worst  $10^6 n^2$ ) time algorithm (i.e., 3SAT will be in  $\text{TIME}(cn)$  or  $\text{TIME}(cn^2)$  for a not-too-large constant  $c$ ). If she decided to make 3SAT hard, then for every  $n \in \mathbb{N}$ , 3SAT on  $n$  variables cannot be solved by a NAND program of fewer than  $2^{10^{-6}n}$  lines (which, through the relations between  $\text{SIZE}(T(n))$ ,  $\text{TIME}_{++}(T(n))$  and  $\text{TIME}(T(n))$  that we've seen in [Theorem 14.1](#) and [Theorem 14.4](#), implies that  $3SAT \notin \text{TIME}(2^{o(n)})$ ).

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.

### 18.1 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 18.1 — Search vs Decision.** Suppose that  $P = \text{NP}$ . 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  $y \in \{0, 1\}^{an^b}$  satisfying  $P(xy) = 1$ , then  $FIND(x)$  finds some string  $y'$  satisfying this condition.

**P** To understand what the statement of [Theorem 18.1](#) means, let us look at the special case of the *MAXCUT* problem. It is not hard to see that there is a polynomial time algorithm *VERIFYCUT* such that  $\text{VERIFYCUT}(G, k, S) = 1$  if and only if  $S$  is a subset of  $G$ 's vertices that cuts at least  $k$  edges. [Theorem 18.1](#) implies that if  $\mathbf{P} = \mathbf{NP}$  then there is a polynomial time algorithm *FINDCUT* that on input  $G, k$  outputs a set  $S$  such that  $\text{VERIFYCUT}(G, k, S) = 1$  if such a set exists. This means that if  $\mathbf{P} = \mathbf{NP}$ , by trying all values of  $k$  we can find in polynomial time that maximum cut in any given graph. We can use a similar argument to show that if  $\mathbf{P} = \mathbf{NP}$  then we can find a satisfying assignment for every satisfiable 3CNF formula, find the longest path in a graph, solve integer programming, and so on and so forth.

**Proof Idea:** The idea behind the proof of [Theorem 18.1](#) is simple. Let us demonstrate it for the particular case of 3SAT. Suppose that  $\mathbf{P} = \mathbf{NP}$  and we are given a satisfiable 3CNF formula  $\varphi$ , and we want to find a satisfying assignment  $y$  for  $\varphi$ . Define  $3SAT_0(\varphi)$  to output 1 if there is a satisfying assignment  $y$  for  $\varphi$  such that its first bit is 0, and similarly define  $3SAT_1(\varphi) = 1$  if there is a satisfying assignment  $y$  with  $y_0 = 1$ . The key observation is that both  $3SAT_0$  and  $3SAT_1$  are in  $\mathbf{NP}$ , and so if  $\mathbf{P} = \mathbf{NP}$  then we can compute them in polynomial time as well. Thus we can use this to find the first bit of the satisfying assignment. We can continue in this way to recover all the bits, see the full proof below.

*Proof of Theorem 18.1.* If  $\mathbf{P} = \mathbf{NP}$  then for every polynomial-time NAND++ program  $P$  and  $a, b \in \mathbb{N}$ , there is a polynomial-time algorithm *STARTSWITH* that on input  $x \in \{0, 1\}^*$  and  $z \in \{0, 1\}^\ell$ , outputs 1 if and only if there exists some  $y \in \{0, 1\}^{an^b}$  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 *STARTSWITH* function is in  $\mathbf{NP}$  and hence can be solved in polynomial time if  $\mathbf{P} = \mathbf{NP}$ .

Now for any program  $P$  and  $a, b \in \mathbb{N}$ , we can implement *FIND*( $x$ ) as follows:

1. For  $\ell = 0, \dots, an^b - 1$  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  $z_0, \dots, z_{an^b-1}$

To analyze the *FIND* algorithm note that it makes  $2an^{b-1}$  invocations to *STARTSWITH* and hence if the latter is polynomial-time then so is *FIND*. Now suppose that  $x$  is such that there exists *some*  $y$  satisfying  $P(xy) = 1$ . We claim that at every step  $\ell = 0, \dots, an^b - 1$ , we maintain the invariant that there exists  $y \in \{0, 1\}^{an^b}$  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  $\ell = an^b - 1$ ,  $z$  satisfies  $P(xz) = 1$ .

We prove the claim by induction. For  $\ell = 0$  this holds vacuously. Now for every  $\ell$ , if the call  $\text{STARTSWITH}(xz_0 \dots z_{\ell-1}0)$  returns 1 then we are guaranteed the invariant by definition of *STARTSWITH*. Now under our inductive hypothesis, there is  $y_\ell, \dots, y_{an^b-1}$  such that  $P(xz_0, \dots, z_{\ell-1}y_\ell, \dots, y_{an^b-1}) = 1$ . If the call to  $\text{STARTSWITH}(xz_0 \dots z_{\ell-1}0)$  returns 0 then it must be the case that  $y_\ell = 1$ , and hence when we set  $z_\ell = 1$  we maintain the invariant.  $\blacksquare$

## 18.2 Quantifier elimination

So, if  $\mathbf{P} = \mathbf{NP}$  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

$$\exists_x P(x) \tag{18.1}$$

for some NAND program  $P$ . But we can think of more general statements such as

$$\exists_x \forall_y P(x, y) \tag{18.2}$$

or

$$\exists_x \forall_y \exists_z P(x, y, z). \tag{18.3}$$

For example, given an  $n$ -input NAND program  $P$ , we might want to find the *smallest* NAND program  $P'$  that is computes the same function as  $P$ . The question of whether there is such  $P'$  of size at most  $k$  can be phrased as

$$\exists_{P'} \forall_x |P'| \leq k \wedge P(x) = P'(x). \tag{18.4}$$

It turns out that if  $\mathbf{P} = \mathbf{NP}$  then we can solve these kinds of problems as well.<sup>1</sup>

<sup>1</sup> Since NAND programs are equivalent to Boolean circuits, this is known as the **circuit minimization problem** and is widely studied in Engineering.

**Theorem 18.2 — 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  $a$  inputs, returns 1 if and only if

$$\exists_{x_1 \in \{0,1\}^n} \forall_{x_2 \in \{0,1\}^n} \cdots Q_{x_a \in \{0,1\}^n} P(x_1, \dots, x_a) \quad (18.5)$$

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  $SOLVE_{a-1}$  that can solve the problem Eq. (18.5) 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  $1 - SOLVE_{a-1}(1 - P_{x_1})$  where  $P_{x_1}$  is a NAND program that on input  $x_2, \dots, x_a \in \{0,1\}^n$  outputs  $P(x_1, \dots, x_n)$ . Now note that by the definition of  $SOLVE$

$$\begin{aligned} & \exists_{x_1 \in \{0,1\}^n} S_P(x_1) = \\ & \exists_{x_1} \overline{SOLVE_{a-1}(\overline{P_{x_1}})} = \\ & \exists_{x_1} \exists_{x_2} \cdots \overline{Q'_{x_a} \overline{P(x_1, \dots, x_a)}} = \\ & \exists_{x_1} \forall_{x_2} \cdots Q_{x_a} P(x_1, \dots, x_a) \end{aligned} \quad (18.6)$$

Hence we see that if we can solve the satisfiability problem for  $S_P$  then we can solve Eq. (18.5). ■

This algorithm can also solve the search problem as well: find the value  $x_1$  that certifies the truth of Eq. (18.5). 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  $\Omega(n^{2^a})$ .<sup>2</sup>

### 18.2.1 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*.

<sup>2</sup> 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 **PSPACE**-hardness which is even stronger than **NP**-hardness.

### 18.3 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  $P = NP$  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 3SAT 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.<sup>3</sup>

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, and little wheels, and twiddles that we can turn. We have to learn how to make these jumps. And it is a tall order. And that’s why things are difficult.”*

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 obtain-

<sup>3</sup> Some people might claim that, if it indeed holds  $P = NP$ , then evolution should have already discovered the efficient 3SAT 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 3SAT and other **NP**-hard problems.

ing “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.

#### 18.4 Can $P \neq NP$ 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.<sup>4</sup> 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).<sup>5</sup>

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  $P \neq NP$ ?

In short, the answer is No. For example, suppose that we are trying to decide between the “3SAT is easy” conjecture (there is an  $10^6 n$  time algorithm for 3SAT) and the “3SAT is hard” conjecture (for every  $n$ , any NAND program that solves  $n$  variable 3SAT takes  $2^{10^{-6}n}$  lines), then, since for  $n = 10^8$ ,  $2^{10^{-6}n} > 10^6 n$ , 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.<sup>6</sup> 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

<sup>4</sup> One way to phrase it is that for every infinite subset  $S$  of the real numbers  $\mathbb{R}$ , either there is a one-to-one and onto function  $f : S \rightarrow \mathbb{R}$  or there is a one-to-one and onto function  $f : S \rightarrow \mathbb{N}$ .

<sup>5</sup> Formally, what he proved is that if ZFC is consistent, then so is ZFC when we assume either the continuum hypothesis or its negation.

<sup>6</sup> This inefficiency is not necessarily inherent. Later in this course we will discuss results in program checking, interactive proofs, and average-case complexity, that can be used for efficient verification of proofs of related statements. In contrast, the inefficiency of verifying *failure* of all programs could well be inherent.

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.

### 18.5 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, 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.

7

### 18.6 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 *can not* 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 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

<sup>7</sup> Talk more about coping with NP hardness. Main two approaches are *heuristics* such as SAT solvers that succeed on *some* instances, and *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.

relativity, the pursuit of impossibility results has often been one of the most fruitful enterprises of mankind.

### *18.7 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.

### *18.8 Exercises*

### *18.9 Bibliographical notes*

8

<sup>8</sup> TODO: Scott’s two surveys

### *18.10 Further explorations*

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

- Polynomial hierarchy hardness for circuit minimization and related problems, see for example [this paper](#).

### *18.11 Acknowledgements*



# 19

## 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.<sup>1</sup> 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.

<sup>1</sup> Harvard’s STAT 110 class (whose lectures are available on [youtube](#)) is a highly recommended introduction to probability. See also these [lecture notes](#) from MIT’s “Mathematics for Computer Science” course.

### 19.1 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  $A = \{x : \sum_i x_i = 0 \pmod{2}\}$ . Let us calculate this probability:

**Lemma 19.1**

$$\mathbb{P}_{x \sim \{0,1\}^n}[\sum x_i \text{ is even}] = 1/2 \quad (19.1)$$

*Proof.* Let  $A = \{x \in \{0,1\}^n : \sum_i x_i = 0 \pmod{2}\}$ . 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  $x_0, \dots, x_{n-2}$ , if  $\sum_{i=0}^{n-2} x_i$  is even then  $(x_0, \dots, x_{n-1}, 0) \in A$  and  $(x_0, \dots, x_{n-1}, 1) \notin A$ . Similarly, if  $\sum_{i=0}^{n-2} x_i$  is odd then  $(x_0, \dots, x_{n-1}, 1) \in A$  and  $(x_0, \dots, x_{n-1}, 0) \notin A$ . Hence, for every one of the  $2^{n-1}$  prefixes  $(x_0, \dots, x_{n-2})$ , there is exactly a single continuation of  $(x_0, \dots, x_{n-2})$  that places it in  $A$ . ■

We can also use the *intersection* ( $\cap$ ) and *union* ( $\cup$ ) operators to talk about the probability of both event  $A$  and event  $B$  happening, or the probability of event  $A$  or event  $B$  happening. For example, the probability  $p$  that  $x$  has an *even* number of ones and  $x_0 = 1$  is the same as  $\mathbb{P}[A \cap B]$  where  $A = \{x \in \{0,1\}^n : \sum_i x_i = 0 \pmod{2}\}$  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

$$\mathbb{P}_{x \sim \{0,1\}^n} \left[ \sum_i x_i = 0 \pmod{2} \wedge x_0 = 1 \right]. \quad (19.2)$$

If  $A \subseteq \{0,1\}^n$  is an event, then  $\bar{A} = \{0,1\}^n \setminus A$  corresponds to the event that  $A$  does *not* happen. Note that  $\mathbb{P}[\bar{A}] = 1 - \mathbb{P}[A]$ : can you see why?

### 19.1.1 Random variables

A *random variable* is a function  $X : \{0,1\}^n \rightarrow \mathbb{R}$  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,

$$\mathbb{E}[X] = \sum_{x \in \{0,1\}^n} 2^{-n} X(x) . \quad (19.3)$$

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 19.2 — Linearity of expectation.**

$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] \quad (19.4)$$

*Proof.*

$$\begin{aligned} \mathbb{E}[X + Y] &= \sum_{x \in \{0,1\}^n} 2^{-n} (X(x) + Y(x)) = \\ &\sum_{x \in \{0,1\}^n} 2^{-n} X(x) + \sum_{x \in \{0,1\}^n} 2^{-n} Y(x) = \quad (19.5) \\ &\mathbb{E}[X] + \mathbb{E}[Y] \end{aligned}$$

■

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  $X = X_0 + \dots + X_{n-1}$  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 19.3 — 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.

### 19.1.2 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  $\mu : S \rightarrow [0, 1]$  such that  $\sum_{x \in S} \mu(x) = 1$ . We think of this as the experiment where we obtain every  $x \in S$  with probability  $\mu(x)$ , 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  $X : S \rightarrow \mathbb{R}$ , where the probability that  $X = y$  is equal to  $\sum_{x \in S : X(x)=y} \mu(x)$ .

2

<sup>2</sup> TODO: add exercise on simulating die tosses and choosing a random number in  $[m]$  by coin tosses

## 19.2 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](#)".<sup>3</sup>

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

<sup>3</sup> 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  $\mathbb{P}[A \cap B] = \mathbb{P}[A]\mathbb{P}[B]$ . If  $\mathbb{P}[A \cap B] > \mathbb{P}[A]\mathbb{P}[B]$  then we say that  $A$  and  $B$  are *positively correlated*, while if  $\mathbb{P}[A \cap B] < \mathbb{P}[A]\mathbb{P}[B]$  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

$$\begin{aligned}\mathbb{P}[x_0 = 1] &= 1/2 \\ \mathbb{P}[x_0 + x_1 + x_2 \geq 2] &= \mathbb{P}[\{011, 101, 110, 111\}] = 4/8 = 1/2\end{aligned}\quad (19.6)$$

but

$$\mathbb{P}[x_0 = 1 \wedge x_0 + x_1 + x_2 \geq 2] = \mathbb{P}[\{101, 110, 111\}] = 3/8 > (1/2)(1/2) \quad (19.7)$$

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  $\mathbb{P}[x_0 = 1 \wedge x_1 = 1] = \mathbb{P}[\{110, 111\}] = 2/8 = (1/2)(1/2)$  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  $\mathbb{P}[B|A] = \mathbb{P}[A \cap B]/\mathbb{P}[A]$ . 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  $\mathbb{P}[B|A] = \mathbb{P}[B]$ .

**More than two events:** We can generalize this definition to more than two events. We say that events  $A_1, \dots, A_k$  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  $I \subseteq [k]$ ,

$$\mathbb{P}[\bigwedge_{i \in I} A_i] = \prod_{i \in I} \mathbb{P}[A_i] \quad (19.8)$$

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  $\{x_0 + x_1 = 0 \pmod 2\}$  are *not* mutually independent even though every pair of these events is independent (can you see why?).

### 19.2.1 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  $\{Y = v\}$  are independent. That is,  $\mathbb{P}[X = u \wedge Y = v] = \mathbb{P}[X = u]\mathbb{P}[Y = v]$ . For example, if two random variables depend on the result of tossing different coins then they are independent:

**Lemma 19.4** Suppose that  $S = \{s_1, \dots, s_k\}$  and  $T = \{t_1, \dots, t_m\}$  are disjoint subsets of  $\{0, \dots, n-1\}$  and let  $X, Y : \{0, 1\}^n \rightarrow \mathbb{R}$  be random variables such that  $X = F(x_{s_1}, \dots, x_{s_k})$  and  $Y = G(x_{t_1}, \dots, x_{t_m})$  for some functions  $F : \{0, 1\}^k \rightarrow \mathbb{R}$  and  $G : \{0, 1\}^m \rightarrow \mathbb{R}$ . 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  $B = \{x \in \{0, 1\}^m : G(x) = b\}$ . Since  $S$  and  $T$  are disjoint, we can reorder the indices so that  $S = \{0, \dots, k-1\}$  and  $T = \{k, \dots, k+m-1\}$  without affecting any of the probabilities. Hence we can write  $\mathbb{P}[X = a \wedge Y = b] = |C|/2^n$  where  $C = \{x_0, \dots, x_{n-1} : (x_0, \dots, x_{k-1}) \in A \wedge (x_k, \dots, x_{k+m-1}) \in B\}$ . 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

$$\frac{|C|}{2^n} = \frac{|A|}{2^k} \frac{|B|}{2^m} \frac{2^{n-k-m}}{2^{n-k-m}} = \mathbb{P}[X = a]\mathbb{P}[Y = b] \quad (19.9)$$

■

Note that if  $X$  and  $Y$  are independent then

$$\begin{aligned} \mathbb{E}[XY] &= \sum_{a,b} \mathbb{P}[X = a \wedge Y = b]ab = \sum_a \mathbb{P}[X = a]\mathbb{P}[Y = b]ab = \\ &\quad \left( \sum_a \mathbb{P}[X = a]a \right) \left( \sum_b \mathbb{P}[Y = b]b \right) = \quad (19.10) \\ &\quad \mathbb{E}[X]\mathbb{E}[Y] \end{aligned}$$

(This is not an “if and only if”, see [Exercise 19.2](#).)

If  $X$  and  $Y$  are independent random variables then so are  $F(X)$  and  $G(Y)$  for every functions  $F, G : \mathbb{R} \rightarrow \mathbb{R}$ . 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

$$\begin{aligned}
 \mathbb{P}[F(X) = a \wedge G(Y) = b] &= \sum_{x \text{ s.t. } F(x)=a, y \text{ s.t. } G(y)=b} \mathbb{P}[X = x \wedge Y = y] = \\
 &\quad \sum_{x \text{ s.t. } F(x)=a, y \text{ s.t. } G(y)=b} \mathbb{P}[X = x] \mathbb{P}[Y = y] = \\
 &\quad \left( \sum_{x \text{ s.t. } F(x)=a} \mathbb{P}[X = x] \right) \cdot \left( \sum_{y \text{ s.t. } G(y)=b} \mathbb{P}[Y = y] \right) = \\
 &\quad \mathbb{P}[F(X) = a] \mathbb{P}[G(Y) = b]
 \end{aligned} \tag{19.11}$$

### 19.2.2 Collections of independent random variables.

We can extend the notions of independence to more than two random variables. We say that the random variables  $X_0, \dots, X_{n-1}$  are *mutually independent* if for every  $a_0, \dots, a_{n-1}$  then

$$\mathbb{P}[X_0 = a_0 \wedge \dots \wedge X_{n-1} = a_{n-1}] = \mathbb{P}[X_0 = a_0] \cdots \mathbb{P}[X_{n-1} = a_{n-1}] \tag{19.12}$$

and similarly we have that

**Lemma 19.5 — Expectation of product of independent random variables.** If  $X_0, \dots, X_{n-1}$  are mutually independent then

$$\mathbb{E}\left[\prod_{i=0}^{n-1} X_i\right] = \prod_{i=0}^{n-1} \mathbb{E}[X_i] \tag{19.13}$$

**Lemma 19.6 — Functions preserve independence.** If  $X_0, \dots, X_{n-1}$  are mutually independent, and  $Y_0, \dots, Y_{n-1}$  are defined as  $Y_i = F_i(X_i)$  for some functions  $F_0, \dots, F_{n-1} : \mathbb{R} \rightarrow \mathbb{R}$ , then  $Y_0, \dots, Y_{n-1}$  are mutually independent as well.

We leave proving Lemma 19.5 and Lemma 19.6 as Exercise 19.3  
**Exercise 19.4.** 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.

## 19.3 Concentration

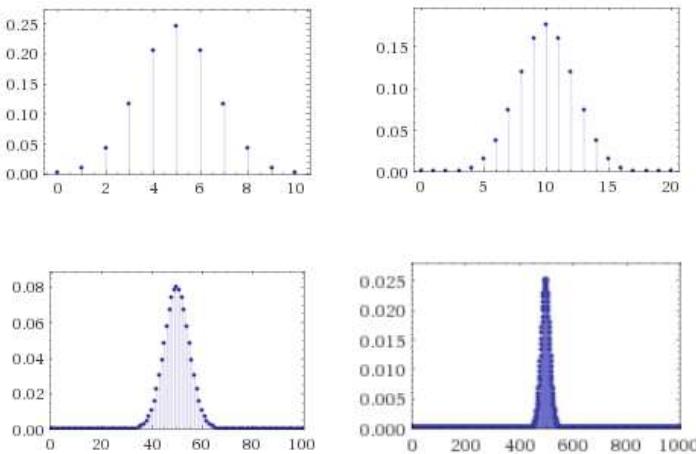
The name “expectation” is somewhat misleading. For example, suppose that I place a bet on the outcome of 10 coin tosses, where if they all come out to be 1’s then I pay you 100000 dollars and

otherwise you pay me 10 dollars. If we let  $X : \{0,1\}^{10} \rightarrow \mathbb{R}$  be the random variable denoting your gain, then we see that

$$\mathbb{E}[X] = 2^{-10} \cdot 100000 - (1 - 2^{-10})10 \sim 90 \quad (19.14)$$

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. 19.1).



**Figure 19.1:** 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 19.7 — Markov's inequality.** If  $X$  is a non-negative random variable then  $\mathbb{P}[X \geq k \mathbb{E}[X]] \leq 1/k$ .

*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.

#### 19.4 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 19.8 — 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  $X = X_1 + \dots + X_n$  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  $X_1, \dots, X_n$  are independent,

$$\text{Var}[X_1 + \dots + X_n] = \text{Var}[X_1] + \dots + \text{Var}[X_n]. \quad (19.15)$$

(We leave showing this to the reader as [Exercise 19.5](#).)

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. \(19.15\)](#) 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  $1/k^2$ .

## 19.5 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,  $X = X_1 + \dots + X_n$  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 19.9 — Chernoff bound.** If  $X_1, \dots, X_n$  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  $\epsilon > 0$

$$\mathbb{P}\left[\left|\sum_{i=0}^{n-1} X_i - pn\right| > \epsilon n\right] \leq 2 \exp(-\epsilon^2 n/2) \quad (19.16)$$

We omit the proof, which appears in many texts, and uses Markov's inequality on i.i.d random variables  $Y_0, \dots, Y_n$  that are of the form  $Y_i = e^{\lambda X_i}$  for some carefully chosen parameter  $\lambda$ . See [Exercise 19.8](#) 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 19.9](#) for a generalization.)

<sup>4</sup> TODO: maybe add an example application of Chernoff. Perhaps a probabilistic method proof using Chernoff+Union bound.

## 19.6 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.

## 19.7 Exercises

**Exercise 19.1** Give an example of random variables  $X, Y : \{0, 1\}^3 \rightarrow \mathbb{R}$  such that  $\mathbb{E}[XY] \neq \mathbb{E}[X]\mathbb{E}[Y]$ . ■

**Exercise 19.2** Give an example of random variables  $X, Y : \{0, 1\}^3 \rightarrow \mathbb{R}$  such that  $X$  and  $Y$  are *not* independent but  $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ . ■

**Exercise 19.3 — Product of expectations.** Prove Lemma 19.5 ■

**Exercise 19.4 — Transformations preserve independence.** Prove Lemma 19.6 ■

**Exercise 19.5 — Variance of independent random variances.** Prove that if  $X_0, \dots, X_{n-1}$  are independent random variables then  $\text{Var}[X_0 + \dots + X_{n-1}] = \sum_{i=0}^{n-1} \text{Var}[X_i]$ . ■

**Exercise 19.6 — 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  $F : S \rightarrow \{0, 1\}^*$ ,  $\mathbb{E}_{x \sim \mu} |F(x)| \geq k$ .

2. Prove that for every  $\epsilon$ , there is some  $n$  and a one-to-one function  $F : S^n \rightarrow \{0, 1\}^*$ , such that  $\mathbb{E}_{x \sim \mu^n} |F(x)| \leq n(k + \epsilon)$ , where  $x \sim \mu$  denotes the experiments of choosing  $x_0, \dots, x_{n-1}$  each independently from  $S$  using the distribution  $\mu$ . ■

**Exercise 19.7 — Entropy approximation to binomial.** Let  $H(p) = p \log(1/p) + (1-p) \log(1/(1-p))$ .<sup>5</sup> Prove that for every  $p \in (0, 1)$

<sup>5</sup> While you don't need this to solve this exercise, this is the function that maps  $p$  to the entropy (as defined in Exercise 19.6) of the  $p$ -biased coin distribution over  $\{0, 1\}$ , which is the function  $\mu : \{0, 1\} \rightarrow [0, 1]$  s.y.  $\mu(0) = 1 - p$  and  $\mu(1) = p$ .

and  $\epsilon > 0$ , if  $n$  is large enough then<sup>6</sup>

$$2^{(H(p)-\epsilon)n} \binom{n}{pn} \leq 2^{(H(p)+\epsilon)n} \quad (19.17)$$

where  $\binom{n}{k}$  is the binomial coefficient  $\frac{n!}{k!(n-k)!}$  which is equal to the number of  $k$ -size subsets of  $\{0, \dots, n-1\}$ . ■

**Exercise 19.8 — Chernoff using Stirling.** 1. Prove that  $\mathbb{P}_{x \sim \{0,1\}^n} [\sum x_i = k] = \binom{n}{k} 2^{-n}$ .

2. Use this and [Exercise 19.7](#) to prove the Chernoff bound for the case that  $X_0, \dots, X_n$  are i.i.d. random variables over  $\{0, 1\}$  each equaling 0 and 1 with probability 1/2. ■

**Exercise 19.9 — Poor man's Chernoff.** Let  $X_0, \dots, X_n$  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  $j_1, \dots, j_n \in \mathbb{N}$ , if there exists one  $i$  such that  $j_i$  is odd then  $\mathbb{E}[\prod_{i=0}^{n-1} Y_i^{j_i}] = 0$ .
- 2. Prove that for every  $k$ ,  $\mathbb{E}[(\sum_{i=0}^{n-1} Y_i)^k] \leq (10kn)^{k/2}$ .<sup>7</sup>
- 3. Prove that for every  $\epsilon > 0$ ,  $\mathbb{P}[|\sum_i Y_i| \geq \epsilon n] \geq 2^{-\epsilon^2 n / (10000 \log 1/\epsilon)}$ .<sup>8</sup> ■

**Exercise 19.10 — 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  $\{0, \dots, M-1\}$  for some  $M$ . Fortunately, we can simulate this with an exponentially small probability of error: prove that for every  $M$ , if  $n > k \lceil \log M \rceil$ , then there is a function  $F : \{0, 1\}^n \rightarrow \{0, \dots, M-1\} \cup \{\perp\}$  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  $\{0, \dots, M-1\}$ .<sup>9</sup> ■

**Exercise 19.11 — 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 19.12** Would the answer to [Exercise 19.11](#) change if the country had 300,000,000,000 citizens? ■

**Exercise 19.13 — Sampling (2).** Under the same assumptions as [Exercise 19.11](#), what is the smallest value  $n$  among the following choices

<sup>6</sup> Hint: Use Stirling’s formula for approximating the factorial function.

<sup>7</sup> Hint: Bound the number of tuples  $j_0, \dots, j_{n-1}$  such that every  $j_i$  is even and  $\sum j_i = k$ .

<sup>8</sup> Hint: Set  $k = 2 \lceil \epsilon^2 n / 1000 \rceil$  and then show that if the event  $|\sum_i Y_i| \geq \epsilon n$  happens then the random variable  $(\sum_i Y_i)^k$  is a factor of  $\epsilon^{-k}$  larger than its expectation.

<sup>9</sup> Hint: Think of  $x \in \{0, 1\}^n$  as choosing  $k$  numbers  $y_1, \dots, y_k \in \{0, \dots, 2^{\lceil \log M \rceil} - 1\}$ . Output the first such number that is in  $\{0, \dots, M-1\}$ .

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.

<sup>10</sup>

<sup>10</sup> TODO: add some exercise about the probabilistic method

### 19.8 *Bibliographical notes*

### 19.9 *Further explorations*

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

### 19.10 *Acknowledgements*



## 20

### *Probabilistic computation*

*"in 1946 .. (I asked myself) what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method ... might not be to lay it out say one hundred times and simply observe and count", Stanislaw Ulam, 1983*

*"The salient features of our method are that it is probabilistic ... and with a controllable minuscule probability of error.", Michael Rabin, 1977*

In early computer systems, much effort was taken to drive out randomness and noise. Hardware components were prone to non-deterministic behavior from a number of causes, whether it is vacuum tubes overheating or actual physical bugs causing short circuits (see Fig. 20.1). 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



**Figure 20.1:** A 1947 entry in the **log book** of the Harvard MARK II computer containing an actual bug that caused a hardware malfunction. By Courtesy of the Naval Surface Warfare Center.

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.<sup>1</sup>

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”.

## 20.1 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:

<sup>1</sup> 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 20.1 — 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  $\mathbb{E}[X] = m(1/2) = m/2$ .  $\blacksquare$

### 20.1.1 Amplification

Theorem 20.1 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

$$pm + (1-p)(m/2 - 1) \leq pm + m/2 - 1 = m/2 + pm - 1 < m/2. \quad (20.1)$$

So, if we repeat this experiment, for example, 1000 $m$  times, then the probability that we will never be able to cut at least  $m/2$  edges is at most

$$(1 - 1/m)^{1000m} \leq 2^{-1000} \quad (20.2)$$

(using the inequality  $(1 - 1/m)^m \leq 1/e \leq 1/2$ ).

### 20.1.2 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.<sup>2</sup> For example, we will see in [Exercise 20.1](#) 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.

### 20.1.3 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:

<sup>2</sup> 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:
  1. If  $x$  satisfies  $\varphi$  then output  $x$ .
  2. Otherwise, choose a random clause  $(\ell_i \vee \ell_j \vee \ell_k)$  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 20.3](#)), 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.<sup>3</sup>

**Theorem 20.2 — 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  $1/2$ .

<sup>3</sup> At the time of this writing, the best known **randomized** algorithms for 3SAT run in time roughly  $O(1.308^n)$  and the best known **deterministic** algorithms run in time  $O(1.3303^n)$  in the worst case. As mentioned above, the simple WalkSAT algorithm takes  $\tilde{O}((4/3)^n) = \tilde{O}(1.333..^n)$  time.

*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 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.

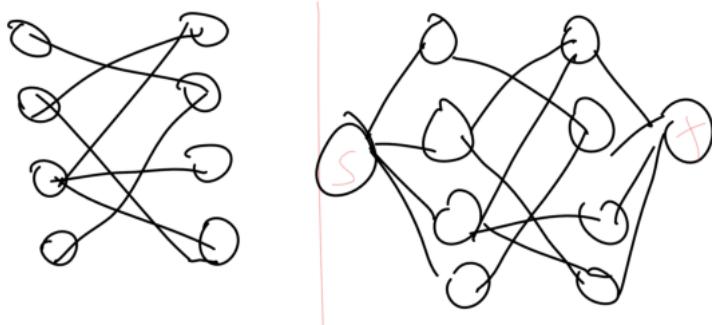
We now prove our earlier claim that with probability  $1/2$  over

$x \in \{0,1\}^n$ ,  $\Delta(x, x^*) \leq n/2$ . Indeed, consider the map  $FLIP : \{0,1\}^n \rightarrow \{0,1\}^n$  where  $FLIP(x_0, \dots, x_{n-1}) = (1-x_0, \dots, 1-x_{n-1})$ . 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  $\overline{A}$  to  $A$ , implying that  $|A| \geq |\overline{A}|$  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  $(1 - \frac{1}{2\sqrt{3}^n})^{100\sqrt{3}^n} \leq (1/e)^{50}$ .  $\blacksquare$

#### 20.1.4 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  $R$ .



**Figure 20.2:** The bipartite matching problem in the graph  $G = (L \cup R, E)$  can be reduced to the minimum  $s, t$  cut problem in the graph  $G'$  obtained by adding vertices  $s, t$  to  $G$ , connecting  $s$  with  $L$  and connecting  $t$  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. 20.2). However, we will see a different probabilistic algorithm to determine whether a graph contains such a matching.

Let us label  $G$ 's vertices as  $L = \{\ell_0, \dots, \ell_{n-1}\}$  and

$R = \{r_0, \dots, r_{n-1}\}$ . A matching  $M$  corresponds to a *permutation*  $\pi \in S_n$  (i.e., one-to-one and onto function  $\pi : [n] \rightarrow [n]$ ) where for every  $i \in [n]$ , we define  $\pi(i)$  to be the unique  $j$  such that  $M$  contains the edge  $\{\ell_i, r_j\}$ . Define an  $n \times n$  matrix  $A = A(G)$  where  $A_{i,j} = 1$  if and only if the edge  $\{\ell_i, r_j\}$  is present and  $A_{i,j} = 0$  otherwise. The correspondence between matchings and permutations implies the following claim:

**Lemma 20.3 — Matching polynomial.** Define  $P = P(G)$  to be the polynomial mapping  $\mathbb{R}^{n^2}$  to  $\mathbb{R}$  where

$$P(x_{0,0}, \dots, x_{n-1,n-1}) = \sum_{\pi \in S_n} \left( \prod_{i=0}^{n-1} \text{sign}(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_{i,\pi(i)} \quad (20.3)$$

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  $P(x) \neq 0$ .

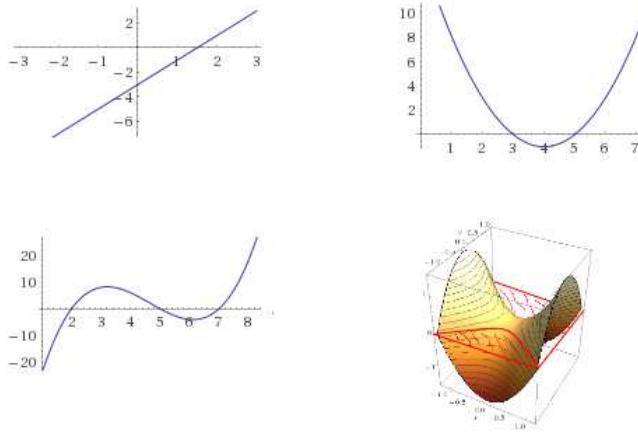
*Proof.* If  $G$  has a perfect matching  $M^*$ , then let  $\pi^*$  be the permutation corresponding to  $M$  and let  $x^* \in \mathbb{Z}^{n^2}$  defined as follows:  $x_{i,j} = 1$  if  $j = \pi^*(i)$  and  $x_{i,j} = 0$ . Note that for every  $\pi \neq \pi^*$ ,  $\prod_{i=0}^{n-1} x_{i,\pi(i)} = 0$  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. (20.3), 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



**Figure 20.3:** 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 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 20.4 — Schwartz–Zippel lemma.** For every integer  $q$ , and polynomial  $P : \mathbb{R}^m \rightarrow \mathbb{R}$  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  $[q]^n = \{(x_0, \dots, x_{m-1}) : x_i \in \{0, \dots, q-1\}\}$ .

We omit the (not too complicated) proof of [Theorem 20.4](#). We remark that it holds not just over the real numbers but over any field

as well. Since the matching polynomial  $P$  of [Lemma 20.3](#) has degree at most  $n$ , [Theorem 20.4](#) leads directly to a simple algorithm for testing if it is nonzero:

**Algorithm Perfect-Matching:**

1. **Input:** Bipartite graph  $G$  on  $2n$  vertices  $\{\ell_0, \dots, \ell_{n-1}, r_0, \dots, r_{n-1}\}$ .
2. For every  $i, j \in [n]$ , choose  $x_{i,j}$  independently at random from  $[2n] = \{0, \dots, 2n - 1\}$ .
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 20.4](#)). 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.

## 20.2 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.

## 20.3 Exercises

**Exercise 20.1 — Deterministic max cut algorithm.** <sup>4</sup> ■

**Exercise 20.2 — 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  $\{0, \dots, M - 1\}$  for some  $M$ . Fortunately, we can simulate

<sup>4</sup> 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  $F : \{0,1\}^n \rightarrow \{0, \dots, M-1\} \cup \{\perp\}$  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  $\{0, \dots, M-1\}$ .<sup>5</sup>

**Exercise 20.3 — Better walksat analysis.** 1. Prove that for every  $\epsilon > 0$ , if

$n$  is large enough then for every  $x^* \in \{0,1\}^n$   $\mathbb{P}_{x \sim \{0,1\}^n} [\Delta(x, x^*) \leq n/3] \leq 2^{-(1-H(1/3)-\epsilon)n}$  where  $H(p) = p \log(1/p) + (1-p) \log(1/(1-p))$  is the same function as in [Exercise 19.7](#).

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 20.4 — Faster bipartite matching (challenge).**<sup>6</sup>

<sup>5</sup> Hint: Think of  $x \in \{0,1\}^n$  as choosing  $k$  numbers  $y_1, \dots, y_k \in \{0, \dots, 2^{\lceil \log M \rceil} - 1\}$ . Output the first such number that is in  $\{0, \dots, M-1\}$ .

<sup>6</sup> TODO: add exercise to improve the matching algorithm by working modulo a prime

## 20.4 Bibliographical notes

monte carlo history: <http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068>

## 20.5 Further explorations

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

## 20.6 Acknowledgements

## 21

# 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.*

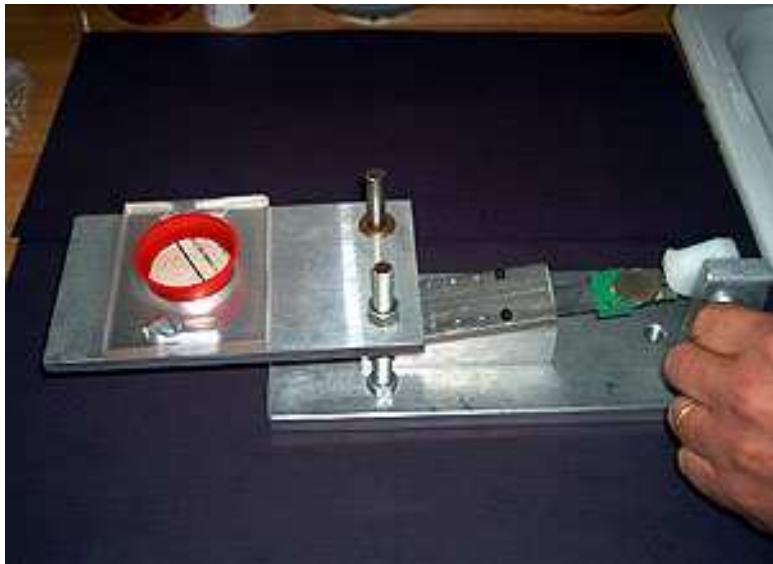
So far we have described probabilistic algorithms in an informal way, assuming that an operation such as “pick a string  $x \in \{0,1\}^n$ ” can be done efficiently. We have neglected to address two questions:

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. 21.1. 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  $\{0,1\}^n$ .<sup>1</sup> 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

<sup>1</sup> Indeed, as [this paper](#) shows, even (real-world) coin tosses do not have exactly the distribution of a uniformly random string.



**Figure 21.1:** A mechanical coin tosser built for Percy Diaconis by Harvard technicians Steve Sansone and Rick Haggerty

very easy. We define the RNAND programming language to include all the operations of NAND plus the following additional operation:

```
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 21.1 — 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  $F \in \text{BPSIZE}(T)$  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  $T : \mathbb{N} \rightarrow \mathbb{N}$  be a nice function. We say that  $F \in \text{BPTIME}(T(n))$  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  $BPTIME(T(n))$  for the subset of  $\overline{BPTIME}(T(n))$  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 21.1 — 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  $10m/\epsilon^2$  times more lines (or runs in at most  $m/\epsilon^2$  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  $t = 10m/\epsilon^2$  times, using fresh randomness each one, to compute outputs  $y_0, \dots, y_{t-1}$ . 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  $X_0, \dots, X_{t-1}$  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  $\exp(-\epsilon^2 t/4) \leq 2^{-m}$ . ■

### 21.1 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  $BPSIZE(T) \subseteq SIZE(T^c)$  for every  $T \in \mathbb{N}$ ?
2. Is there some constant  $c$  such that  $\overline{BPTIME}(T(n)) \subseteq \overline{TIME}(T(n)^c)$  for every nice  $T : \mathbb{N} \rightarrow \mathbb{N}$ ? Specifically, is it the case that  $\overline{\text{BPP}} = \overline{\text{P}}$  where  $\overline{\text{BPP}} = \cup_{a \in \mathbb{N}} \overline{BPTIME}(n^a)$  stands for the class of functions that can be computed in probabilistic polynomial time.

### 21.2 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 21.2 — Simulating RNAND with NAND.** For every  $T \in \mathbb{N}$  and  $F : \{0, 1\}^n \rightarrow \{0, 1\}^m$ , if  $F \in RSIZE(T)$  then  $F \in SIZE(100nT)$ .

*Proof.* Suppose that  $P$  is a  $T$ -line RNAND program such that  $\mathbb{P}[P(x) = F(x)] \geq 2/3$ . We use [Theorem 21.1](#) to obtain an  $50nT$  line program  $P'$  such that

$$\mathbb{P}[P'(x) = F(x)] \geq 1 - 0.9 \cdot 2^{-n}. \quad (21.1)$$

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  $i \in \{0, \dots, R-1\}$ . 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. \(21.1\)](#) as follows: for every  $x \in \{0,1\}^n$

$$\sum_{r \in \{0,1\}^R} B_{x,r} \leq 0.1 \cdot 2^{R-n}. \quad (21.2)$$

It follows that the total number of 1's in this matrix is at most  $0.12^R$ , 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.)<sup>2</sup> Let  $P^*$  be the following NAND program which is obtained by taking  $P'$  and replacing the  $i^{th}$  line of the form `var := RAND` with the line `var := ri*`. That is, we replace this with `var := one NAND one or var := 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  $x \in \{0,1\}^n$ . ■

<sup>2</sup> In fact, the same reasoning shows that at least a 0.9 fraction of the rows have this property.

**Note:** [Theorem 21.2](#) 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  $x \in \{0,1\}^n$ .

### 21.3 Derandomizing uniform computation

The proof of [Theorem 21.2](#) 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

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  $r^* \in \{0,1\}^{poly(n)}$  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.<sup>3</sup>

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 21.3 — Can’t replace tape deterministically.** There is a linear time probabilistic algorithm  $A$  such that for every  $x \in \{0,1\}^*$ ,  $\mathbb{P}[A(x) = 1] < 1/10$  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  $r$ .

<sup>3</sup> 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  $r_1, \dots, r_n$  and outputs 1 if and only if  $x_0 = r_0, x_1 = r_1, \dots, x_9 = r_9$  (if  $n < 10$  then  $A$  always outputs 0). Clearly  $A$  runs in  $O(n)$  steps and for every  $x \in \{0,1\}^*$ ,  $\mathbb{P}[A(x) = 1] \leq 2^{-10} < 0.1$ . However, by definition, for every fixed string  $r$  of length at least 10,  $A(r;r) = 1$ . ■

The proof of [Lemma 21.3](#) 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.

#### 21.4 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.)

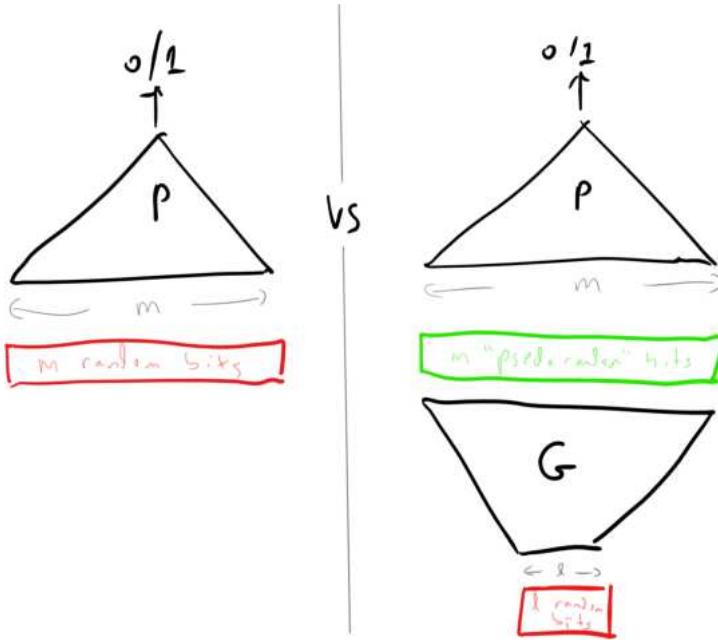
We make the following definition:

**Definition 21.2 — 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,

$$\left| \mathbb{P}_{s \sim \{0,1\}^\ell} [P(G(s)) = 1] - \mathbb{P}_{r \sim \{0,1\}^m} [P(r) = 1] \right| < \epsilon \quad (21.3)$$

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 21.2:** A pseudorandom generator  $G$  maps a short string  $s \in \{0, 1\}^\ell$  into a long string  $r \in \{0, 1\}^m$  such that an 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  $s_0, \dots, s_{m-1}$ . In this case  $\mathbb{P}_{s \sim \{0, 1\}^\ell}[P(G(s)) = 1]$  will simply equal  $\mathbb{P}_{r \in \{0, 1\}^m}[P(r) = 1]$ , 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. (21.3) 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 21.2 does not involve RNAND++ programs but deterministic NAND programs with no randomness and no loops.

We will now (partially) answer both questions.

#### 21.4.1 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 21.4](#), there are pseudorandom generators where  $m$  is *exponentially larger* than  $\ell$ .

**Lemma 21.4 — 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.<sup>4</sup> 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.

5

*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. \(21.3\)](#). 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. \(21.3\)](#) 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  $P_1, \dots, P_M$  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  $2^{O(T \log T)} < 2^{T^2}$  (for large enough  $T$ ).<sup>6</sup> Thus if we let  $E_i$  be the event [Eq. \(21.3\)](#) is vio-

<sup>4</sup> There is a whole (highly recommended) [book](#) by Alon and Spencer devoted to this method.

<sup>5</sup> TODO: if we end up discussing the probabilistic method before this proof, then move this discussion to that point.

<sup>6</sup> Recall that we showed this by arguing that we can translate all variables in a  $\leq T$ -line program to have the form  $x_{\langle j \rangle}, y_{\langle j \rangle}$  or  $\text{work}_{\langle j \rangle}$  (where  $j$  is a number between 0 and  $3T$ ) without changing the function that the program computes and hence without affecting the event [Eq. \(21.3\)](#). For programs in this form, every line can be described by at most  $4 \log T$  ASCII characters which means that the program can be described by at most  $10T \log T$  characters or  $12T \log T$  bits, which means there are at most  $2^{12T \log T}$  of them.

lated for program  $P_i$  with respect to the random  $G$ , then by setting  $C$  large enough we can ensure that  $\mathbb{P}[E_i] < 1/(10M)$  which means that by the union bound with probability 0.9, Eq. (21.3) 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 21.4, 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  $y_0, \dots, y_{L-1} \in \{0,1\}^m$  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

$$\left| \frac{1}{L} \sum_{i=0}^{L-1} P(y_s) - \mathbb{P}_{s \sim \{0,1\}^m} [P(s) = 1] \right| > \epsilon \quad (21.4)$$

is at most  $2^{-T^2}$ . Eq. (21.4) 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  $y_0, \dots, y_{L-1}$  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  $2 \cdot 2^{-\epsilon^2 L/2}$ . ■

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  $\overline{\text{TIME}}(m(n)^c)$  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)

### 21.4.2 Usefulness of pseudorandom generators

TO BE CONTINUED (show that optimal pseudorandom generators imply that  $\overline{\text{BPP}} = \overline{\text{P}}$ )

## 21.5 Lecture summary

## 21.6 Exercises

## 21.7 Bibliographical notes

## 21.8 Further explorations

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

## 21.9 Acknowledgements

## 22

# *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.

*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*



## 23

# *Derandomization from worst-case assumptions (advanced lecture)*

PLAN: Sketch the proof that BPP is in QuasiP if the permanent 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.

### *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

# 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".

### 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



## 25

### *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)

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*



## 26

# *Compression, coding, and information*

PLAN: Define entropy, talk about compression, Huffman coding, talk about channel capacity and error correcting codes.

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

## *Space bounded computation*

PLAN: Example of space bounded algorithms, importance of preserving space. The classes L and PSPACE, space hierarchy theorem.

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*



# 28

## *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.

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*



# 29

## *Proofs and algorithms*

PLAN: Talk about proofs and algorithms, zero knowledge proofs, correspondence between proofs and algorithms, Coq or other proof assistants, SAT solvers.

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*



## 30

# *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

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*



# 31

## *Data structure lower bounds*

PLAN: Lower bounds for data structures, cell-probe lower bounds using communication complexity.

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*



## 32

# *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"

### *32.1 Lecture summary*

### *32.2 Exercises*

### *32.3 Bibliographical notes*

### *32.4 Further explorations*

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

### *32.5 Acknowledgements*



# 33

## *Shor's Algorithm (advanced lecture)*

PLAN: Give proof of Shor's Algorithm.

33.1 *Lecture summary*

33.2 *Exercises*

33.3 *Bibliographical notes*

33.4 *Further explorations*

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

33.5 *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.

## *33.6 NAND programming language specification*

### *33.6.1 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  $:=$  operator and variables of the form  $y$  or  $y_{\langle i \rangle}$  can only appear on the lefthand side of the  $:=$  operator. The *number of inputs* of a NAND program  $P$  equals one plus the largest number  $i$  such that a variable of the form  $x_{\langle i \rangle}$  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 is always smaller 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  $i < j$ . Similarly, we require that if a variable of the form  $x_{\langle j \rangle}$  appears in the program, then  $x_{\langle i \rangle}$  must appear in it for all  $i < j$ .<sup>1</sup>

### 33.6.2 Semantics of NAND programs

To evaluate a NAND program  $P$  with  $n$  inputs and  $m$  outputs on input  $x_0, \dots, x_{n-1}$  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  $:=$  operator the value of the NAND of the variables on the righthand side. In this evaluation, we identify `foo` with `foo_0` and `bar_079` with `bar_79`. 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  $y_0, \dots, y_{\langle m-1 \rangle}$ . (Recall that all variables of the form  $y_{\langle i \rangle}$  must be assigned some value.)

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  $F$ .

<sup>1</sup> I've been going back and forth on whether we should make this requirement. The main advantage in it is that it ensures that  $\text{Size}(s)$  is a finite set, as the inputs and outputs are always smaller than  $s$ . It does however mean that we may need to add "dummy lines" of the form `one := x_i NAND zero` to the program to ensure that every input variable is touched. We could also simply require that  $s \geq n$ .

## 33.7 NAND++ programming language specification

The NAND++ programming language adds to NAND the ability for loops and to access an unbounded amount of memory.

### 33.7.1 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.

<sup>2</sup>

<sup>2</sup> Add discussion of *invalidy* array for NAND++ and NAND«, see the loops lecture

### 33.7.2 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  $x \in \{0,1\}^*$  we do the following:

1. Set  $ic = 0, r = 0, m = 0, i = 0, inc = +1$  (*ic* stands for “iteration counter” and *r* is the current “rounds” of the index variable).
2. For every line in *P* of the form *vara* := *varb* NAND *varc*, assign to the variable denoted by *vvara* 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\_<i>* where  $\langle i \rangle$  denotes the current value of *i*. If a variable has the form *x\_<j>* then if  $j < |x|$  it gets the value  $x_j$  and otherwise it gets the value 0. If a variable has the form *validx\_<j>* then if  $j < |x|$  it gets the value 1 and otherwise it gets the value 0. If a variable on the lefthandside has the form *y\_<j>* then we let  $m = \max\{m, j + 1\}$ .
3. If the variable *loop* has the value 0 then halt with output  $y_0, \dots, y_{m-1}$  where  $y_j$  equals the value of *y\_<j>* 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  $r \leftarrow r + 1$  (in this case we have completed a “round”) and  $inc = +1$ .
  - If  $i = r$  then set  $inc = -1$ .
  - Then set  $i \leftarrow i + inc$ ,  $ic \leftarrow ic + 1$ , 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 `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  $(ic + 1)\ell$  where  $ic$  is the value of the iteration counter  $ic$  at the time when the program halts and  $\ell$  is the number of lines in  $P$ . If  $F : \{0,1\}^* \rightarrow \{0,1\}^*$  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  $T : \mathbb{N} \rightarrow \mathbb{N}$  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  $F(x)$ .

### 33.7.3 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> 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:

```
# Converts a NAND program from text to the list of tuples
# representation
# prog: code of program
# n: number of inputs
# m: number of outputs
# t: number of variables
def triples(prog,n,m,t):

    varsidx = {}

    def varindex(var): # index of variable with name var
        if var[:2]=='x_': return int(var[2:])
        if var[:2]=='y_': return t-m+int(var[2:])
        if var in varsidx: return varsidx[var]
        return varsidx.setdefault(var,len(varsidx)+n)

    result = []

    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()
result.append([varindex(var1),varindex(var2),varindex(
    var3)])

return result

```

The function above assumes we know some parameters of the program, such as its input and output length, and the number of distinct variables, but this is easy to get as well.

```

# compute the parameters of a given program code
# returns: n = number of inputs, m = number of outputs, t =
#           number of distinct variables
def params(prog):

    varnames = set()
    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()
        varnames.update([var1,var2,var3])

    t = len(varnames)
    n = len([var for var in varnames if var[:2]=='x_'])
    m = len([var for var in varnames if var[:2]=='y_'])

    return [n,m,t]

```

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 L on input x
# L is given in the list of tuples representation
def EVAL(L,n,m,x):
    t = max([max(triple) for triple in L])+1 # num of vars in
        L
    vars = [0]*t # initialize variable array to zeroes
    vars[:n] = x # set first n vars to x

    for (a,b,c) in L: # evaluate each triple
        vars[a] = 1-vars[b]*vars[c]

    return vars[t-m:] # output last m variables

```

Moreover, if we want to *compile* NAND programs, we can easily transform them to C code using the following NAND2C function:<sup>3</sup>

```
# Transforms a NAND program to a C function
# prog: string of program code
# n: number of inputs
# m: number of outputs
# code has not been tested
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
        vname = vname.split('_')
        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) ];
// See Stackoverflow answer https://stackoverflow.com/
    questions/2525310/how-to-define-and-work-with-an-array
    -of-bits-in-c

unsigned long getval(bfield_t vars, int idx) {{
    return 1 & (vars[idx / (8 * sizeof(long)) ] >> (idx %
```

<sup>3</sup> The function is somewhat more complex than the minimum needed, since it uses an array of *bits* to store the variables.

```

        (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(avars))

Cprog = Cprog + main + '''
    for(j=0;j<{{m}};++j) {{
        y[j] = getval(vars,{n}+j)
    }}
    return y;
}
''''.format(n=n,m=m)

return Cprog

```

### 33.8 NAND« programming language specification

The NAND« (pronounced “NAND shift”) 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++ can not (and indeed any NAND« program can be “compiled to a NAND++ program) but it can be polynomially faster.

#### 33.8.1 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:  $idx \leftarrow \lfloor foo2^{-bar} \rfloor$ )
- `foo := bar « baz` (left shift:  $idx \leftarrow foo2^{bar}$ )
- `foo := bar % baz` (modular reduction)
- `foo := bar * baz` (multiplication)
- `foo := bar / baz` (integer division:  $idx \leftarrow \lfloor \frac{foo}{bar} \rfloor$ )
- `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.

### 33.8.2 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 iteration 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 iteration counter, then we assign the value of the iteration 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  $x_0, x_1, x_2, \dots$  (which we will typically assume to be bits, but could be integers as well) we do the following:

1. Set  $ic = 0, m = 0, i = 0$ , ( $ic$  stands for “iteration counter” and  $r$  is the current “rounds” of the index variable).
2. For every line in  $P$ , we do the following:
  - (a) If the line has 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` (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 `foo` without an index then we treat it as if it was `foo_0`.
    - If a variable has the form `foo_i` then we treat it as if it is `foo_{j}` where  $j$  denotes the current value of the index variable `i`.
    - If a variable has the form `x_{j}` then if  $j < |x|$  it gets the value  $x_j$  and otherwise it gets the value 0.
    - If a variable has the form `validx_{j}` 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_{j}` then we let  $m = \max\{m, j + 1\}$ .
  - (b) If a line correspond to an index operation of the form `foo := bar OP baz` where `OP` is one of the operations listed above then we evaluate `foo` and `bar` as in the step above and compute the value  $v$  to be the operation `OP` applied to the values of `foo` and `bar`. We assign to the index variable corresponding to `idx` the value  $\max\{0, \min\{ic, v\}\}$ .

If the variable `loop` has the value 0 then halt with output  $y_0, \dots, y_{m-1}$  where  $y_j$  equals the value of `y_{j}` 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. Set

$px \leftarrow ic + 1$ , set  $i = INDEX(ic)$  where  $INDEX$  is the function that maps the iteration 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  $(ic + 1)\ell$  where  $ic$  is the value of the iteration 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.

### 33.9 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.

#### 33.9.1 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

```
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:

```
def fool,...,fook := Func(bar1,...,barl) {
    function_code
}
```

denotes code for a function that takes  $l$  inputs bar<sub>1</sub>,...,bar<sub>l</sub> and returns  $k$  outputs fool,...,fook. We can then invoke such a function by writing

```
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

```
foo := XOR(bar,baz)
```

or

```
if AND(foo,bar) {
    code
}
```

where XOR,AND have been defined above.

**NAND for loops:** We can introduce syntactic sugar for loops in NAND, as long as the number of times the loop executes is fixed and independent of the input size. Hence we will use

More generally, we will replace code of the form

```
for i in RANGE do {
    code
}
```

where RANGE specifies a finite set  $I = \{i_0, \dots, i_{k-1}\}$  of natural numbers, as syntactic sugar for  $|R|$  copies of code, where for  $j \in [k]$ , we replace all occurrences of \_<expr(i)> in the  $j$ -th copy with \_<expr( $i_j$ )> where expr(i) denotes an arithmetic expression in i (involving i, constants, parenthesis, and the operators +, -, \*, mod, /) and for every  $x \in \mathbb{N}$ , expr(c) denotes the result of applying this expression to the value c.

We specify the set  $I = \{i_0, \dots, i_{k-1}\}$  by simply writing  $[\langle i_0 \rangle, \langle i_1 \rangle, \dots, \langle i_{k-1} \rangle]$ . We will also use the  $\langle \text{beg} \rangle : \langle \text{end} \rangle$  notation so specify the interval  $\{\text{beg}, \text{beg} + 1, \dots, \text{end} - 1\}$ . For example,  $[2:4, 10:13]$  specifies the set  $\{2, 3, 10, 11, 12\}$ .

**Operator overloading and infix notation:** For convenience of notation, we can define functions corresponding to standard operators  $*, +, -, \dots$ , etc.. and then code such as `foo * bar` will correspond to `operator*(foo, bar)`.

4

<sup>4</sup> TODO: potentially add lists and lists operations

### 33.9.2 Encoding of integers, strings, lists.

While NAND/NAND++/NAND« natively only supports values of 0 and 1 for variables (and integers for indices in NAND++/NAND«), in our standard library We will use the following default 8-symbol alphabet for specifying constants and strings in NAND/NAND++/NAND«:

Symbol	Encoding	Default semantics
.	000	End of string/integer/list marker
0	001	The bit 0
1	010	The bit 1
[	011	Begin list
]	100	End list
,	101	Separate items
:	110	Define mapping
-	111	Space / "no op" / to be ignored

When we write code such as `foo := "str"` where `str` is a length  $n$  sequence from this alphabet that doesn't contain `.`, then we mean that we store in  $\text{var}_{\langle 0 \rangle}$  till  $\text{var}_{\langle 3n - 1 \rangle}$  the encoding for the  $n$  symbols of `str`, while we store in  $\text{var}_{\langle 3n \rangle}, \text{var}_{\langle 3n + 1 \rangle}, \text{var}_{\langle 3n + 2 \rangle}$  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 `foo := 13` and `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

```
foo := 12
bar := 1598
```

```
baz := foo * bar
```

Using multidimensional arrays we can also use code such as

```
foo_0 := 124
foo_1 := 57
foo_2 := 5459
```

**Lists:** We store lists and nested lists using the separators [ , ], with the binary encoding of each element. Thus code such as

```
foo := [ 13 , 1, 4 ]
```

will be the same as

```
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

```
foo := [17,8] + [24,9]
```

will be the same as

```
foo := [17,8,24,9]
```

we will use `in` as shorthand for the operation that scans a list to see if it contains an element and so in the code

```
foo := [12,127,8]
bar := 8 in foo
```

`bar` is assigned the value 1.

The `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 `length(bar)` times where each time `foo` will get the current element.

5

6

<sup>5</sup> TODO: check if we should add map and filter for lists

### 33.9.3 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.

7

<sup>7</sup> TODO: potentially have a loop that iterates over a list

### *3. Bibliography*