18.405 — Advanced Complexity Theory

Class by Ryan Williams

Notes by Sanjana Das

Spring 2024

Lecture notes from the MIT class **18.405** (Advanced Complexity Theory), taught by Ryan Williams. All errors are my own.

Contents

1 An overview of complexity 6
- 1.1 Computation tasks 6
- 1.2 Resources 7
  - 1.2.1 An algorithmic model 7
  - 1.2.2 Time complexity 8
  - 1.2.3 Space complexity 8
  - 1.2.4 Nondeterminism 9
  - 1.2.5 Some relations between classes 10
- 1.3 Goals of complexity 10
- 1.4 Diagonalization 11
  - 1.4.1 The time hierarchy theorem 12
- 1.5 Boolean circuits 13
- 1.6 Restricted subsets of P/poly 14
- 1.7 Communication complexity 16
- 1.8 Query complexity 17
- 1.9 Connections 18
- 1.10 Some other topics 19

2 The class coNP and the polynomial hierarchy 20
- 2.1 Some properties of P 20
- 2.2 The class coNP 20
  - 2.2.1 Some coNP-complete problems 21
- 2.3 Time hierarchy for nondeterminism 21
  - 2.3.1 Some difficulties 22
  - 2.3.2 The proof idea 23
  - 2.3.3 Some simplifications 23
  - 2.3.4 The construction of f 23
  - 2.3.5 Proof of the upper bound 24
  - 2.3.6 Proof of the lower bound 24
- 2.4 Closure under composition and oracle classes 25
  - 2.4.1 Oracles 25
  - 2.4.2 Complexity classes 26
  - 2.4.3 Closure under composition 26

2.4.4 Some examples . . . 27
2.5 The polynomial hierarchy . . . 29
2.6 A logical characterization of PH . . . 30
2.6.1 The classes $\Sigma_k\mathsf{P}$ and $\Pi_k\mathsf{P}$ . . . 30
2.6.2 An equivalence . . . 32
2.6.3 Some consequences . . . 33

3 Time-space tradeoffs for SAT **35**
3.1 Time-space classes . . . 35
3.2 The proof ideas . . . 36
3.2.1 Some definitions . . . 36
3.2.2 Proof outline . . . 37
3.3 Step (3) — the slowdown . . . 38
3.4 Step (2) — the speedup . . . 38
3.5 Ideas behind the improvements . . . 40
3.5.1 Proof sketch of a better bound . . . 41

4 Oracles and relativization **42**
4.1 Some examples of relativization . . . 42
4.2 P vs. NP and the relativization barrier . . . 43
4.2.1 An oracle with $\mathsf{P}^A = \mathsf{NP}^A$ . . . 43
4.2.2 An oracle with $\mathsf{P}^B \neq \mathsf{NP}^B$ . . . 44
4.3 Some more examples of the relativization barrier . . . 46
4.4 Non-relativization of time-space lower bounds . . . 47

5 Space complexity **49**
5.1 Time vs. space . . . 49
5.1.1 Configuration graphs . . . 50
5.2 Nondeterministic vs. deterministic space . . . 51
5.3 The class NL . . . 52
5.3.1 Log-space reductions . . . 52
5.3.2 A NL-complete problem . . . 55
5.3.3 $\mathsf{NL} = \mathsf{coNL}$ . . . 56

6 Circuit complexity **59**
6.1 Boolean circuits . . . 59
6.2 Bounds on maximum possible circuit complexity . . . 60
6.3 An undecidable problem . . . 62
6.4 From algorithms to circuits . . . 63
6.4.1 A P-complete problem . . . 64
6.5 Algorithms with advice . . . 65
6.5.1 The class P/poly . . . 66
6.5.2 NP vs. P/poly . . . 67
6.6 The Karp–Lipton theorem . . . 67
6.6.1 The solution printer lemma . . . 68
6.6.2 Proof of the Karp–Lipton theorem . . . 68
6.7 Some circuit lower bounds . . . 69
6.7.1 Some extensions . . . 71

7 Randomized complexity **72**
7.1 Randomized computation . . . 72

7.2 Polynomial identity testing . . . 73
7.2.1 An extension to arithmetic circuits . . . 75
7.3 BPP vs. some other complexity classes . . . 76
7.4 $\mathsf{BPP} \subseteq \Sigma_2\mathsf{P}$ . . . 77
7.4.1 Overview of the proof . . . 78
7.4.2 Big set vs. small set . . . 79
7.4.3 An $\exists\forall$ characterization . . . 80
7.5 Some open problems about BPP . . . 80
7.6 Other versions of randomized computation . . . 81
7.6.1 The classes RP and coRP . . . 81
7.6.2 The class ZPP . . . 82
7.6.3 Some relationships and open questions . . . 83

8 Derandomization 84
8.1 The problem CAPP . . . 84
8.1.1 PromiseBPP . . . 85
8.2 Pseudorandom generators . . . 86
8.2.1 Definition of PRGs . . . 87
8.2.2 Existence of PRGs . . . 87
8.2.3 Derandomization from PRGs . . . 88
8.3 PRGs from circuit lower bounds . . . 90
8.3.1 Step 1 — worst-case to average-case hardness . . . 90
8.3.2 An overview of Step 2 — average-case hardness to PRGs . . . 91
8.3.3 A baby PRG . . . 92
8.3.4 NW-designs and the PRG construction . . . 93
8.3.5 A hybrid argument . . . 94
8.3.6 A circuit predicting f . . . 95

9 Counting complexity 97
9.1 The class #P . . . 97
9.1.1 Two classes of problems . . . 97
9.2 #P vs. FP . . . 98
9.3 #P-completeness . . . 99
9.3.1 #P-completeness for NP-complete problems . . . 100
9.3.2 #P-completeness of #Cycle . . . 101
9.4 The class PP — a decision version of counting . . . 103
9.5 The Valiant–Vazirani theorem . . . 105
9.5.1 The proof idea . . . 106
9.5.2 Pairwise independent hash families . . . 107
9.5.3 The isolation lemma . . . 108
9.5.4 Proof of the Valiant–Vazirani theorem . . . 109
9.6 Toda's theorem — $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$ . . . 110
9.6.1 Relativized SAT . . . 110
9.6.2 The class $\oplus\mathsf{P}$ . . . 111
9.6.3 A sequence of lemmas . . . 112
9.6.4 Proof of Toda's theorem from the lemmas . . . 113
9.6.5 Proof of Lemma 9.47 — $\mathsf{NP} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}}$ . . . 113
9.6.6 Proof of Lemma 9.49 — from $\mathsf{NP} \subseteq \mathsf{BPP}$ to $\mathsf{PH} \subseteq \mathsf{BPP}$ . . . 114
9.6.7 Proof of Lemma 9.48 — $\oplus\mathsf{P}$ is closed under composition . . . 116

10 Interactive proofs **118**
10.1 Setup of interactive proofs . . . 118
10.1.1 Modelling interaction . . . 118
10.1.2 The deterministic verifier setting . . . 119
10.1.3 The randomized verifier setting . . . 120
10.1.4 Some facts about IP . . . 122
10.2 An example — graph non-isomorphism . . . 122
10.3 Arthur–Merlin and Merlin–Arthur protocols . . . 124
10.3.1 Private-coin to public-coin protocols . . . 125
10.4 An interactive proof for Count-SAT . . . 126
10.4.1 Arithmetization . . . 127
10.4.2 The sum-check protocol . . . 127
10.4.3 Proof of completeness . . . 129
10.4.4 Proof of soundness . . . 129
10.5 $\mathsf{IP} = \mathsf{PSPACE}$ . . . 130
10.5.1 $\mathsf{IP} \subseteq \mathsf{PSPACE}$ . . . 130
10.5.2 Arithmetization of paths . . . 131
10.5.3 The interactive protocol . . . 132
10.6 Multiparty interactive proofs . . . 133
10.6.1 Probabilistic oracle machines . . . 135

11 Probabilistically checkable proofs **136**
11.1 The definition of PCPs . . . 136
11.2 The PCP theorem . . . 137
11.3 PCPs and hardness of approximation . . . 138
11.3.1 Constraint satisfaction problems . . . 138
11.3.2 PCPs and inapproximability . . . 139
11.3.3 Proof of the equivalence theorem . . . 140
11.3.4 Hardness of approximation for Max-Clique . . . 141
11.3.5 Hardness of approximation for Max-3SAT . . . 143
11.3.6 Vertex-Cover and the unique games conjecture . . . 143
11.3.7 Approximation in graph coloring . . . 145
11.4 A weaker PCP theorem . . . 145
11.4.1 The high-level idea . . . 145
11.4.2 Algebrization . . . 146
11.4.3 A sum-check . . . 148
11.4.4 The correct proof $\mathcal{P}$ . . . 148
11.4.5 Verifying the proof . . . 149

12 Communication complexity **151**
12.1 Setup and protocol trees . . . 151
12.2 Some examples . . . 152
12.3 The fooling set method . . . 153
12.4 A linear algebraic perspective . . . 155
12.4.1 Combinatorial rectangles . . . 155
12.4.2 Rank vs. communication complexity . . . 156
12.5 Randomized communication protocol . . . 157

13 Restricted circuit complexity **159**
13.1 The class AC^0 . . . 159
13.1.1 Random restrictions and Theorem 13.2 . . . 160

13.1.2 Theorem 13.3 for $d = 2$ . . . 161
13.1.3 A sketch for general d . . . 162
13.2 AC^0 with parity gates . . . 162
13.2.1 Approximating circuits with polynomials . . . 163
13.2.2 Two key theorems . . . 163
13.2.3 Theorem 13.12 and probabilistic polynomials . . . 164
13.3 Algorithmic methods for circuit lower bounds . . . 165
13.3.1 Proof idea for unrestricted circuits . . . 167

§1 An overview of complexity

In the first week of the class, we're going to give an overview of complexity theory — the motivation behind it, what we're studying, how we formalize things, and so on. (There'll be very few proofs here.)

In general, complexity theory asks the following question.

> **Question 1.1.** What resources are needed for computation tasks?

(Most of the words in this sentence need some explanation; we'll explain them soon.)

This is the main question of complexity theory. In contrast, the main question of algorithms is what resources are *sufficient* — with algorithms, we want to find an efficient way to solve some problem. In complexity theory, we ask what computational resources we really *need* — equivalently, what kinds of problems can we *not* solve with limited resources?

§1.1 Computation tasks

First, here's some examples of computation tasks that we'll look at. The most familiar type of task is a *decision problem* — i.e., a function $f\colon \{0,1\}^* \to \{0,1\}$, where we're given x, and we want to compute $f(x)$. (In general, we'll view all digital objects as bit-strings; we won't worry about more general alphabets.)

> **Remark 1.2.** In 18.404, we might have viewed decision problems as *languages* $L \subseteq \{0,1\}^*$, where we're given x and we want to figure out whether x is in L (and *accept* or *reject* accordingly). Here we think of f as the indicator function for L.

One familiar example of a decision problem is 3SAT.

Definition 1.3 (3SAT)

- **Input:** A 3CNF formula φ — i.e., a formula of the form $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ in Boolean variables $x_1, \ldots, x_n$ (taking values T or F), where each clause C is a **OR** of exactly 3 literals (e.g., $\overline{x_i} \vee x_j \vee x_k$).
- **Decide:** Does φ have a satisfying assignment — i.e., an assignment of Boolean values to the variables such that the formula evaluates to T?

The next class of problems we'll look at is *search problems*. We can view a search problem as a function $g\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$; we're given an input x, and we want to output some y such that $g(x,y) = 1$ (if such a y exists). Note that unlike with decision problems, here we're outputting many bits rather than 1.

We can think of g as a 'theorem-proof checker,' where x is a 'theorem' and y is a 'proof' (and g outputs 1 if y is a valid proof of x). Or in general, we can think of x as an instance and y as a solution, and g as a way of verifying solutions. Here's an example.

Definition 1.4 (Search-3SAT)

- **Input:** a 3CNF formula φ.
- **Output:** a satisfying assignment to φ (if one exists).

A third class of problems is *optimization problems*. One way to view an optimization problem is as a function $g\colon \{0,1\}^* \times \{0,1\}^* \to \mathbb{N}$; we're given an input x, and we want to output a string y which *maximizes* the value of $g(x,y)$. (We can think of this as a generalization of search problems — if the output of g only belongs to $\{0,1\}$, then maximizing $g(x,y)$ corresponds to finding y with $g(x,y) = 1$.)

Definition 1.5 (Max-3SAT)

- **Input:** a 3CNF formula φ.
- **Output:** an assignment which maximizes the number of satisfied clauses.

Here it might be that there's no assignment satisfying *all* the clauses, but we still want to find an assignment satisfying as many of them as we can.

Another class of problems is *counting* — we'll again have a function $g\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$. We're given an input x, and we want to output the *number* of strings y such that $g(x,y) = 1$ — so we want to count the *number* of solutions to a search problem.

Definition 1.6 (#3SAT)

- **Input:** a 3CNF formula φ.
- **Output:** the number of satisfying assignments to φ.

In this class, we'll study all of these types of problems (as well as others).

§1.2 Resources

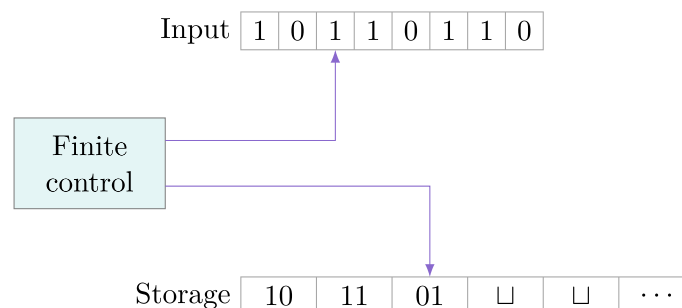
First, to talk about resources, we need some kind of computational model. We'll give one at a somewhat high level (without e.g., defining Turing machines as seven-tuples).

§1.2.1 An algorithmic model

In our algorithmic model, we'll have an input on a special tape, which we say is *read-only*; and our algorithm $\mathcal{M}$ will have a pointer to some position on this input tape.

Then we'll have some storage, which is *read-write*. This storage is divided into cells, each of which is initially blank (and each cell stores a constant number of bits).

And finally, we'll have a *finite control*. (Importantly, the finite control is *finite* — it doesn't depend on the length of the input.) This essentially consists of a finite list of instructions telling the machine what to do — it reads the bits it's looking at (both in the input and the storage tapes), writes to the storage tape (at the current cell it's on), and decides what to do next (e.g., moving left or right). So at each step, the machine reads a constant amount of information and makes a constant amount of change.



(We use $\sqcup$ to denote a blank cell.)

We can think of our machine as being a function $\mathcal{M}\colon \{0,1\}^* \to \{0,1\}^*$ defined by what it does on each input (here the output of the function corresponds to the storage tape when $\mathcal{M}$ halts — if we want a machine that just accepts or rejects, then we can look at only the first bit of the storage).

Now that we have a model, we can talk about resources.

§1.2.2 Time complexity

Definition 1.7. Given a function $t\colon \mathbb{N} \to \mathbb{N}$, we say an algorithm $\mathcal{A}$ computes f in time $O(t(n))$ if there is a constant c such that for all inputs $x \in \{0,1\}^*$, we have $\mathcal{A}(x) = f(x)$, and $\mathcal{A}$ on x runs for at most $c \cdot t(|x|)$ steps (before halting).

We measure time complexity as a function of the input length — as the input length grows, we generally allow our time bound to grow larger and larger. (This makes sense because otherwise we wouldn't even be able to read the whole input.)

We're now ready to define our first complexity class.

Definition 1.8. We define $\mathsf{TIME}[t(n)]$ as the set of decision problems $f\colon \{0,1\}^* \to \{0,1\}$ such that there exists an algorithm computing f in time $O(t(n))$.

This definition depends on how we define our model (e.g., whether we can only move left and right, or whether we can jump around) — for example, there's a notion of *random access machines*, where we can jump around on different registers, and the time classes you'll get for such machines are different from what you'll get from a standard one-tape Turing machine. But the next class *doesn't* depend on this.

Definition 1.9. We define $\mathsf{P} = \bigcup_c \mathsf{TIME}[n^c]$.

We think of P as the class of problems we can solve 'efficiently.' Why is this a good choice? In some sense, P is the right class to look at if you don't want to think about the details of how your machine works. But on the other hand, P includes problems which can be solved in time n^{10^9} (for example), which we probably don't think of as efficient. And in fact, there's an area of complexity theory called *fine-grained computation* that tries to address this.

But there's an idea that we can define inductively what it means for an algorithm to be efficient — where we say an algorithm is efficient if either it runs in $O(n)$ time, or it runs in $O(n)$ time but is able to call an efficient algorithm as a subroutine. (There's more to this than what we've described — for example, there has to be a constant number of 'layers.') It turns out that this way of thinking about things corresponds exactly to P — this is a result of Cobham from the 1960s. (Another way of saying this is that the product or composition of two polynomials is another polynomial, so P is closed under composition in a nice way — this notion will come up when we talk about P vs. NP.)

So that's roughly why we like P.

We can also consider a much bigger class.

Definition 1.10. We define $\mathsf{EXP} = \bigcup_c \mathsf{TIME}[2^{n^c}]$.

§1.2.3 Space complexity

The next resource we'll talk about is *space* (or memory).

Definition 1.11. We define $\mathsf{SPACE}[t(n)]$ as the set of decision problems f such that there exists an algorithm computing f in space $O(t(n))$.

When we say an algorithm runs in space $O(t(n))$, we're looking at how much of the storage tape it uses — so instead of measuring the number of steps we take, we're measuring the read-write storage.

Remark 1.12. To be more precise, we measure the space usage of an algorithm by looking at the rightmost cell on the storage tape that it uses. If we instead just counted the number of non-blank cells, machines which could jump around would have ways of 'cheating' by using cells really far out. But with this definition, $\mathsf{SPACE}[t(n)]$ is actually robust — it doesn't depend on the choice of model (as long as it's reasonable).

Because we've separated the input tape from the storage tape, it makes sense to even talk about space bounds that are smaller than linear.

Definition 1.13. We define LOGSPACE $= \mathsf{SPACE}[\log n]$.

This is an important space class. It lets us store a few pointers into the input, but we can't store a whole lot; but as far as we know, LOGSPACE is actually quite powerful.

Definition 1.14. We define PSPACE $= \bigcup_c \mathsf{SPACE}[n^c]$.

§1.2.4 Nondeterminism

Now we'll change how our finite control works — here we'll allow *multiple* possible moves. So the machine gets to see what's on the input tape and the storage (on its current cell), but now there's multiple moves it could make based on that definition.

We'll now define what it means for a nondeterministic algorithm to compute a function (we'll do this in a way that makes sense even for non-decision problems).

Definition 1.15. We say a nondeterministic algorithm $\mathcal{A}$ computes f in time $O(t(n))$ if there is a constant c such that for all $x \in \{0,1\}^*$:

- There exists a sequence of moves of $\mathcal{A}$ on x with at most $c \cdot t(|x|)$ steps on which $\mathcal{A}$ outputs $f(x)$.
- On *all* sequences of moves, $\mathcal{A}$ runs for at most $c \cdot t(|x|)$ steps and outputs either $f(x)$ or *reject*.

For a decision problem, if $f(x) = \textit{accept}$ then this says there's some sequence of moves on which $\mathcal{A}$ accepts, and on all other sequences it can either accept or reject. Meanwhile, if $f(x) = \textit{reject}$ then this says $\mathcal{A}$ rejects on all sequences. So this definition recovers the usual notion of a nondeterministic algorithm for decision problems, but is more general.

Definition 1.16. We define $\mathsf{NTIME}[t(n)]$ analogously to $\mathsf{TIME}[t(n)]$, and NP $= \bigcup_c \mathsf{NTIME}[n^c]$.

There's a useful characterization of NP, which brings us back to search problems.

Fact 1.17 — We have $f \in \mathsf{NP}$ if and only if there is a polynomial p and a function $g \in \mathsf{P}$ such that for all x, we have $f(x) = 1$ if and only if there exists y of length $p(|x|)$ such that $g(x,y) = 1$.

This is called the *verifier characterization* of NP — instead of thinking about nondeterministic machines, we think about a (deterministic) verifier g.

Conjecture 1.18 — We have $\mathsf{P} \neq \mathsf{NP}$.

We've probably seen some motivation for this question — for example, there's a wide variety of NP-complete problems, so that if any one of them had a polynomial-time algorithm, then P would equal NP. A canonical example is CircuitSAT.

Definition 1.19 (CircuitSAT)

- **Input:** a circuit C with one output wire and n input wires, made of **AND**, **OR**, and **NOT** gates.
- **Decide:** whether there exists y such that $C(y) = 1$.

The reason that CircuitSAT is NP-complete is essentially that Boolean logic is universal — given any verifier g, we can encode g as a Boolean circuit (where x is fixed, and y corresponds to the input wires).

Meanwhile, the obvious algorithm for CircuitSAT is exhaustive search — this would take time $2^n \cdot \mathsf{poly}(s)$, where s is the number of gates in the circuit. (We use $\mathsf{poly}(s)$ to denote an arbitrary but fixed polynomial in s.) If $\mathsf{P} = \mathsf{NP}$, then we'd actually get an algorithm which takes time $\mathsf{poly}(n, s)$, which seems too good to be true. But it turns out that even if we could beat exhaustive search by a *little* bit — e.g., if we could get an algorithm running in time $1.999^n \cdot \mathsf{poly}(s)$ — then we'd get a different breakthrough in complexity theory. (The fastest algorithm we know is basically $2^n \cdot s \cdot \mathsf{polylog}(s)$ — we don't know how to shave off any lower-order terms in n.)

Remark 1.20. There's a great survey by Scott Aaronson on P vs. NP.

§1.2.5 Some relations between classes

So far, we've defined a bunch of complexity classes; how do they relate to each other? We know that

$$\mathsf{LOGSPACE} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}.$$

And we know *some* of these inclusions are proper — we know $\mathsf{P} \subsetneq \mathsf{EXP}$ (by the time hierarchy theorem) and $\mathsf{LOGSPACE} \subsetneq \mathsf{PSPACE}$ (by the space hierarchy theorem). This lets us make some weird disjunctive statements — for example, either $\mathsf{LOGSPACE} \neq \mathsf{NP}$ or $\mathsf{NP} \neq \mathsf{PSPACE}$. But we don't know how to prove either of these two statements (and either one would be a major breakthrough). So there's a guarantee that *some* limitations exist, but we don't know exactly which ones. (Most people believe that *all* these inclusions are proper — i.e., these classes are all different.)

These classes all have interesting problems, and they have their own notions of completeness. (In fact, there's the complexity zoo, which contains hundreds of complexity classes.)

§1.3 Goals of complexity

What are the goals of complexity?

One goal is *impossibility results* — we want to show that some task can't be solved with limited resources. We think of these as *lower bounds* (as opposed to algorithms, which we can think of as upper bounds). Another goal is *trade-offs* or *connnections* or *reductions* — interesting ways in which different computing tasks relate to one another.

In some sense, the first goal tries to give a concrete answer to the question of what resources are needed to solve a problem. In the second, we ask questions such as 'if we restrict our algorithm to use only a certain amount of space, then how much time does it need?' (This is a space-time tradeoff.) Or we can ask what happens if one problem is easy — does this mean that some other problems are easy, or that others are hard? (There are actually connections in both directions — where one problem being easy implies another problem being easy, or one problem being easy implies another being hard.) This is most of what complexity theory results look like, because the first goal is quite hard (though we'll see some concrete examples).

Here's one dream scenario about impossibility results. In general, we might like to see some *non-asymptotic* limits on computing. For example, let Π be your favorite hard problem; one dream theorem you might

want to show is that every logical circuit solving Π on 1000-bit strings requires more than 10^{90} gates. (This bound is pretty reasonable — 10^{90} is pretty small compared to 2^{1000}, which is roughly what we'd need for exhaustive search.) The number 10^{90} is much greater than the number of atoms in the universe (as far as we know), so proving a statement like this would actually have physical implications — for example, we couldn't fit a computer for solving this problem in the known universe (it'd need too many atoms).

The question of P vs. NP is about asymptotics, so it doesn't really say anything about statements like this. To prove statements like this, we'll need to study something else, namely *circuit complexity* — this lets us look at a concrete tradeoff between the size of our input and the size of the circuits we need.

How far are we from proving statements like this? The answer is *very* far — the state of our bounds is even worse than what we know for P vs. NP. Here's a concrete example.

Definition 1.21 (MaxClique)

- **Input:** a graph, given as a $n \times n$ adjacency matrix.
- **Output:** a clique with the maximum number of nodes.

A *clique* is a subset of nodes in which all pairs have an edge between them. (Note that the input for a n-vertex graph is n^2 bits.)

And the following question is open.

Open question 1.22. Does MaxClique on n^2-bit input have a circuit of size at most $6n^2$?

In other words, is there a circuit just slightly larger than the input itself that would solve MaxClique? It's widely believed that the answer is 'absolutely not' — we expect to need a circuit of size *exponential* in n. But we don't know how to prove this. (Here 6 isn't a random number — if we replace it with 2, then we know the answer really is 'no,' i.e., we *can* prove a lower bound.)

Another open question, which bothers Ryan even more, is the following.

Open question 1.23. Does MaxClique have an algorithm that takes $O(n^2)$ time and $O(\log n)$ space?

(The funny thing is that we can prove theorems along these lines for SAT.)

Here we're requiring algorithms that run in basically linear time *and* logarithmic space. We think the answer is almost certainly 'no' — if the answer is 'yes,' this would change the world ten times more than everything else about computers combined. But we don't actually know — we think there's no way this could be true, but we don't know how to reason about it.

So complexity is a bit weird — there's a lot of conjectures people have made about things being hard, but not a whole lot of concrete results. There are some things we know and new programs that have emerged, but we're still in the dark in many ways.

§1.4 Diagonalization

The first method of proving lower bounds that we'll look at is diagonalization. Diagonalization is a method for proving impossibility that originated with Turing in terms of computing.

Theorem 1.24 (Turing)

There exists a function $f\colon \{0,1\}^* \to \{0,1\}$ which cannot be decided by *any* algorithm.

This means there's no algorithm whatsoever that always stops and gives the same answer as f. (Here we're not even putting a time bound on the algorithm.)

Proof. We'll have to 'go meta' — for an algorithm $\mathcal{M}$, we use $\langle \mathcal{M} \rangle$ to denote a *description* of $\mathcal{M}$ (i.e., a binary encoding of $\mathcal{M}$ that is 'easy to parse'). We'll then define our impossible function f such that $f(\langle \mathcal{M} \rangle)$ is 1 if and only if $\mathcal{M}$ run on $\langle \mathcal{M} \rangle$ doesn't accept (i.e., doesn't output 1).

(Here all we're using is the fact that algorithms have finite descriptions, and we're considering what happens when we take an algorithm and run it on its *own* description.)

Now assume there's some $\mathcal{M}^*$ that decides f. This means $\mathcal{M}^*$ accepts its own description $\langle \mathcal{M}^* \rangle$ if and only if $f(\langle \mathcal{M}^* \rangle) = 1$ (since $\mathcal{M}^*$ is supposed to be computing f).

But the definition of f tells us the complete opposite — $f(\langle \mathcal{M}^* \rangle)$ is supposed to be 1 if and only if $\mathcal{M}^*$ *doesn't* accept its own description! So that's a contradiction. □

At a really high level, the idea of diagonalization is that a program can't predict in advance what it's going to do — otherwise it could just do the opposite. Next, we'll see the *time hierarchy theorem*, which is in some sense an elaboration on this result.

§1.4.1 The time hierarchy theorem

First, the time hierarchy theorem needs the notion of a 'nice' time function — essentially one where our machine can compute the number of steps it's allowed to run for.

Definition 1.25. A function $t \colon \mathbb{N} \to \mathbb{N}$ is time-constructible if there exists an algorithm $\mathcal{M}$ such that $\mathcal{M}(1^n)$ outputs $t(n)$ (in binary) in $O(t(n))$ time.

(We use 1^n to denote the string of n 1's.) Most functions we'll ever think about — polynomials, exponentials, $n!$, and even funny functions such as $n^{\log n}$ (such a function is called *quasipolynomial*) — are all time-constructible, so this is a very broad definition of 'nice.'

Then the time hierarchy theorem essentially says that as we keep increasing the amount of time we allow, we get strictly more problems that we can solve.

Theorem 1.26 (Time hierarchy theorem)
There exists a constant $c > 1$ such that for all time-constructible functions t with $t(n) \geq n$ for all n, there is some decision problem $f \colon \{0,1\}^* \to \{0,1\}$ such that $f \in \mathsf{TIME}[t(n)^c]$ but $f \notin \mathsf{TIME}[t(n)]$.

The constant c that we'll prove this statement for depends on the computation model (e.g., Turing machines); but we'll see on the problem set that using this statement (for some arbitrary value of c) as a black box, we can actually make c arbitrarily close to 1.

The idea behind the proof of this theorem is *universal simulation* (and the reason it applies to so many models of computation is that so many models have a universal simulator).

Theorem 1.27 (Universal simulation theorem)
There is a *universal machine* $\mathcal{U}$ such that for all algorithms $\mathcal{M}$, all strings x, and all $t \in \mathbb{N}$:

- $\mathcal{U}(\langle \mathcal{M} \rangle, x, 1^t)$ runs in $\mathsf{poly}(|\mathcal{M}| \cdot t)$ time.
- $\mathcal{U}(\langle \mathcal{M} \rangle, x, 1^t)$ accepts if and only if $\mathcal{M}(x)$ accepts in time at most t.

What this means is essentially that there's some algorithm $\mathcal{U}$ that takes the description of *any* other algorithm $\mathcal{M}$ and an input x, simulates it step by step, and then outputs its answer. (We can also do the same thing for $\mathcal{M}$ which prints a string instead of just *accept* or *reject*.) This simulation has some overhead, but it's only polynomial. (We use $|\mathcal{M}|$ as shorthand for $|\langle\mathcal{M}\rangle|$.)

And we can use such a simulator to get the time hierarchy theorem.

Proof sketch of Theorem 1.26. Here's the main idea — let $\alpha\colon \mathbb{N} \to \mathbb{N}$ be any function which is unbounded but easy to compute (for example, we can take $\alpha(n) = n$). We then define f as follows — f takes every string it receives as input and interprets it as $\langle\mathcal{M}\rangle$, i.e., as the code of some algorithm $\mathcal{M}$. (If it gets a string that doesn't parse, it'll interpret it as the empty algorithm with no code.) And then we'll define $f(\langle\mathcal{M}\rangle)$ to be 1 if and only if $\mathcal{M}$ on $\langle\mathcal{M}\rangle$ doesn't accept in at most $\alpha(|\mathcal{M}|)t(|\mathcal{M}|)$ steps.

> **Remark 1.28.** if we just stopped without the condition on the number of steps, then we'd get exactly the hard problem from the proof of Theorem 1.24, and so our problem wouldn't even be decidable. Here we need an upper bound on f (i.e., that $f \in \mathsf{TIME}[t(n)^c]$), so we need to modify it a bit — and we do this by putting in the time limit.

In order to see that $f \in \mathsf{TIME}[t(n)^c]$, we can compute f by using the universal simulator — here x is $\langle\mathcal{M}\rangle$ and t is $\alpha(|\mathcal{M}|)t(|\mathcal{M}|)$, and the simulation only gives a polynomial overhead (which is where c comes from).

Now we'll check that $f \notin \mathsf{TIME}[t(n)]$. The idea is very similar to the proof of Theorem 1.24. We'll show that if $\mathcal{M}^*$ decides f, then there must be infinitely many inputs x for which $\mathcal{M}^*$ on x takes time greater than $\alpha(|\mathcal{M}|)t(|\mathcal{M}|)$. We'll do this by contradiction — assume that $\mathcal{M}^*$ decides f, and on all but finitely many inputs it takes time at most $\alpha(|\mathcal{M}|)t(|\mathcal{M}|)$. But then $\mathcal{M}^*$ on its own description is undefined by the same argument as in Theorem 1.24 (if $\mathcal{M}^*$ runs within this time bound, then the time limit in our definition of f isn't a problem).

> **Remark 1.29.** For this to work, we really need that there exist arbitrarily long descriptions of the same machine — but this can easily be done (e.g., by padding 0's to our descriptions).

And this means $\mathcal{M}^*$ doesn't run in time $O(t(n))$. (This is why we chose α to be an unbounded function — so that it drowns out any possible leading constant.) □

There's lots of theorems that can be proven using diagonalization-based arguments. You have to be careful — we might have seen that there are things like oracles that show diagonalization is limited in what it can do. But we can show several things — for example, we can actually prove limitations on SAT.

Theorem 1.30

For any $c < 2\cos\frac{\pi}{7}$, there is no algorithm for SAT that uses both time $O(n^c)$ and space $O(\log n)$.

(This can be thought of as very partial progress towards showing that $\mathsf{NP} \neq \mathsf{LOGSPACE}$.)

A nice feature of this proof is that it works for basically any model, including random access. (For MaxClique, we don't know how to rule out this kind of thing — as seen in Question 1.23 — but we *can* for SAT.)

§1.5 Boolean circuits

Next, we'll consider Boolean circuits. Here the setup is somewhat different from what we've seen with multi-tape Turing machines and algorithms and so on.

The idea is that we'd like to compute a function on a *fixed* number of inputs — i.e., a function $f\colon \{0,1\}^n \to \{0,1\}$ — using the minimum number of 'simple' functions. For example, one definition of 'simple' function

you could take is the collection of all functions $g\colon \{0,1\}^2 \to \{0,1\}$ — for example, you can imagine building up a program for computing f on all length-n inputs by using some number of **AND**s, **OR**s, and so on.

The notion of efficiency changes when we talk about using Boolean circuits to solve decision problems — i.e., functions $f\colon \{0,1\}^* \to \{0,1\}$ — and here the notion of efficient computation is captured by P/poly.

Definition 1.31. The class P/poly is the class of decision problems f such that there exists a *sequence* of circuits C_n (with one for each $n \in \mathbb{N}$) such that C_n has n inputs and at most cn^c gates for all n (for some constant c), and for every x, we have $f(x) = C_{|x|}(x)$.

In other words, to compute f on x, we first check the length of x, then find the corresponding circuit, and compute this circuit on x. This is a very weird kind of computer, where we essentially have a separate program C_n for each input length n. This means our circuit family will have an infinite description. And as a result, there are some undecidable problems in P/poly.

Definition 1.32

Let $\mathsf{Unary\text{-}Halt} = \{1^n \mid \text{the } n\text{th Turing machine halts on a blank tape}\}$ (for some computable enumeration of Turing machines).

This problem is undecidable (the binary version certainly is, and using a different encoding — namely, unary — doesn't change that). But it's actually in P/poly!

Why would that be? The point is that we just need to make a separate circuit for each input length. And if we consider what's going on for each individual input length, Unary-Halt restricted to strings of length n is either $\{1^n\}$ or $\emptyset$ (depending on whether the nth Turing machine halts or not). And there's a very small circuit computing either of these — in the first case we want to accept if and only if the input is all-1's, so we can take C_n to be the **AND** circuit $x_1 \wedge \cdots \wedge x_n$; and in the second case we want to never accept, so we can just take any contradictory circuit (e.g., $x_1 \wedge \overline{x_1}$).

So P/poly is in some sense a very strange complexity class. But it's the kind of class we'd have to understand in order to prove non-asymptotic results as mentioned in Subsection 1.3 — where we'd like to say e.g., that to compute a problem on 1000-bit instances we'd need a computer of size 10^{90}. (We need some model like this to get concrete tradeoffs between input length and computer size — standard things like P vs. NP don't work like this, since there you have a constant-sized program that's supposed to work regardless of how large the input is.)

Conjecture 1.33 — We have $\mathsf{NP} \not\subseteq \mathsf{P}/\mathsf{poly}$.

It's known that $\mathsf{P} \subseteq \mathsf{P}/\mathsf{poly}$ (and they're certainly not equal, because P/poly contains some undecidable problems), so if we could prove this conjecture, then we'd automatically get $\mathsf{P} \neq \mathsf{NP}$. But we're quite far away from something like this. In fact, it's open even whether $\mathsf{NTIME}[2^n] \subseteq \mathsf{P}/\mathsf{poly}$ — you could have problems whose proofs take time 2^n to check, but as far as we know, allowing different circuits for each length could potentially let you solve them.

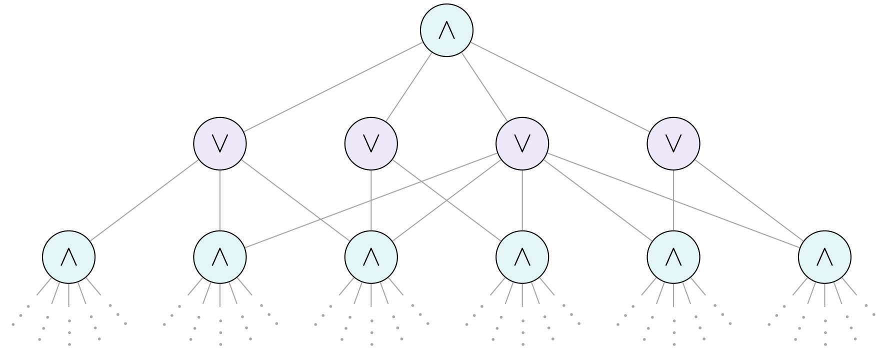
Remark 1.34. If we could get a 1.999^n time algorithm for CircuitSAT, this would imply not only that $\mathsf{NTIME}[2^n] \not\subseteq \mathsf{P}/\mathsf{poly}$ but in fact something much stronger. So somehow, tiny-looking progress over exhaustive search for CircuitSAT would result in huge breakthroughs here.

§1.6 Restricted subsets of P/poly

Because P/poly is so hard to understand, people have looked at several restricted subsets of it. We're not going to be able to talk about all of them — this course needs to be about more than just circuits. But

we'll see a story to motivate some of these restricted subsets.

One restricted subset people have looked at is AC^0 — this stands for *alternating circuits*. Here we again look at polynomial-sized circuits, but we put restrictions on what they look like — we allow **AND** and **OR** gates of any fan-in, and we allow **NOT** gates, but we only allow the circuit to have *constant* depth. (We don't count the depth of the **NOT** gates — we can push all of them down to the bottom using De Morgan's law, so they basically don't matter.) We can think of such a circuit as alternating between **AND**s and **OR**s.



(To see that this is a subset of $\mathsf{P/poly}$, we can imagine replacing each arbitrary fan-in **AND** or **OR** with a tree of **AND**s and **OR**s of fan-in 2.)

This looks kind of wild, because there's only a constant number of layers. But there's actually a lot of things you can do with this model — for example, you can add a small number of n-bit numbers, or you can compute a minimum.

But some limitations on this model were shown a long time ago.

Theorem 1.35 (FSS 1981)
For large n, the problem Parity_n does not have $\mathsf{poly}(n)$-size AC^0 circuits.

The problem Parity_2 just computes the sum of its inputs mod 2 — it takes n bits $x_1, \ldots, x_n$ and outputs $x_1 + \cdots + x_n$ mod 2.

This was the first of many lower bounds of its type — essentially ruling out massively parallel circuits. This is great, but what happens if we're given the parity function for free (so now we have gates **AND**, **OR**, and **PARITY**)? We call this new circuit model $\mathsf{AC}^0 + \mathsf{Parity}$. Now of course the above lower bound in Theorem 1.35 doesn't hold — we can compute Parity with just one gate. But we can still prove similar lower bounds.

Theorem 1.36 (RS 1987)
The Mod_3 function, defined as

$$\mathsf{Mod}_3(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } x_1 + \cdots + x_n \equiv 0 \pmod 3 \\ 0 & \text{otherwise,} \end{cases}$$

does not have $\mathsf{AC}^0 + \mathsf{Parity}$ circuits of polynomial size.

So we can't check divisibility by 3 using divisibility by 2 and **OR**s and **AND**s. More generally, we can show that Mod_p can't be done with Mod_q gates when p and q are powers of distinct primes.

Question 1.37. What if we get both Parity *and* Mod_3 gates for free?

This gives us the model $\mathsf{AC}^0 + \mathsf{Parity} + \mathsf{Mod}_3$; this is essentially equivalent to $\mathsf{AC}^0 + \mathsf{Mod}_6$ (if you can count mod 2 and 3, then by the Chinese remainder theorem you can count mod 6). And this is where complexity stopped for a long time — for example, the following question is still open.

Open question 1.38. Does EXP have $\mathsf{AC}^0 + \mathsf{Mod}_6$ circuits of polynomial size?

For all we know, it's still possible that EXP could be computed even with a depth-3 mod-6 circuit. (Why are we stuck? The main problem seems to be that there's no field of order 6.)

Remark 1.39. We *do* know such bounds for $\mathsf{NTIME}[n^{\mathsf{polylog}(n)}]$.

Remark 1.40. Do we know of any powerful things that we *can* do with $\mathsf{AC}^0 + \mathsf{Mod}_6$ circuits? The answer is yes — there's some weird things. For example, the $\mathsf{Majority}$ function turns out to have sub-exponential sized circuits. This is quite surprising — it doesn't seem like counting mod 2 and mod 3 should magically let you count up to $\frac{n}{2}$, but it turns out that it does.

§1.7 Communication complexity

The next thing we'll talk about is communication complexity. This is a very different kind of model. It's extremely well-understood compared to circuit complexity, but it's also extremely versatile.

In this setup, we have two parties, Alice and Bob. They have different inputs — we'll say Alice has a string $x \in \{0,1\}^n$ and Bob has $y \in \{0,1\}^n$ — and they share some communication channel. And they want to compute some target function $f(x,y)$ of both their inputs.

We'd like to set up a communication scheme or protocol for this — so Alice looks at x and sends some bit over, Bob looks at y and this bit and sends another bit over, and so on, and they hope to eventually compute $f(x,y)$.

Example 1.41

Consider the function $f(x,y) = \sum(x_i + y_i) \pmod 2$.

This function is easy to compute — Alice can simply compute her own parity $\sum x_i$ and send it to Bob, and then Bob can compute the entire parity and send it back (using the fact that $\sum(x_i + y_i) = \sum x_i + \sum y_i$). So this function takes constant communication complexity.

On the other hand, here's an example of a problem which is *difficult*.

Example 1.42

Consider the function $\mathsf{EQ}(x,y) = 1_{x=y}$ (i.e., $\mathsf{EQ}(x,y)$ is 1 if and only if $x = y$).

Theorem 1.43

The communication complexity of EQ is n.

In other words, this essentially says that someone has to send their entire input over — they can't do something more clever than that.

But this lower bound only holds for *deterministic* schemes. If we allow randomness (with some small probability of failure), then things change.

Theorem 1.44

With randomness, the communication complexity of EQ is $O(\log n)$.

So the communication complexity goes from n all the way down to $\log n$. The idea is hashing — Alice sends a $\log n$-bit hash of x to Bob, and Bob checks whether it matches the hash of y.

So randomness can be very powerful when we look at communication complexity.

On the other hand, here's a problem where randomness *doesn't* help.

Example 1.45

Consider the function $\mathsf{Disj}(x, y) = 1_{x \wedge y = 0}$ (equivalently, $\mathsf{Disj}(x, y)$ is 1 if and only if for every i, either $x_i = 0$ or $y_i = 0$).

(The name is because if we think of x and y as indicator variables for subsets of $[n]$ — where x_i is 1 if and only if i is in the set — then this problem corresponds to checking whether the two sets are disjoint.)

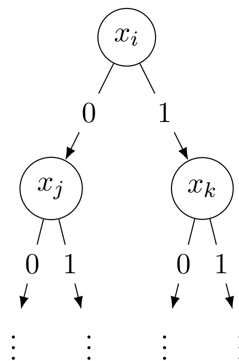
Theorem 1.46

Even with randomness, the communication complexity of Disj is $\Omega(n)$.

So randomness helps for some problems, but not for others.

§1.8 Query complexity

In query complexity, we want to compute a function $f\colon \{0,1\}^n \to \{0,1\}$ on some n-bit input $x \in \{0,1\}^n$ querying only a *small* number of bits of x. We can model this by a *decision tree* — we can imagine that we start by querying x_i; if x_i is 0 then we query some new bit x_j, while if it's 1 then we query some other bit x_k; and so on. And in the end, we reach some leaf that gives us either a 0 or 1 (representing our output).



Then we can study the decision tree complexity of f — we want to minimize the number of bits that we ever read, which corresponds to minimizing the depth of this tree.

It turns out that randomized and deterministic query complexity are polynomially related. This is very surprising — you might think that randomness could help a whole lot (as it does in other settings, such as communication complexity), but it doesn't. On the other hand, *nondeterminism* does help a lot — there are problems that you only need a constant number of queries to solve nondeterministically, but a linear number to solve deterministically.

Example 1.47

Consider the function $\mathsf{NotEq}(x, y) = 1_{x \neq y}$ (this is a function $\{0,1\}^{2n} \to \{0,1\}$).

With nondeterminism, we can just guess the index i at which x and y differ, and check that they do differ at that index. But without nondeterminism, we need a linear number of queries to find that index.

Remark 1.48. This is in some sense a circuit version of sublinear time algorithms — it's similar to sublinear time algorithms in that if we have low tree depth then we're not reading the whole input, but here we have a different tree for each input length.

§1.9 Connections

There's many connections between different resources and models of computation.

One thing to consider is time vs. space. For example, you can think about P vs. PSPACE or P vs. $\mathsf{LOGSPACE}$ — could time be so powerful that you can 'reuse' it as well as you can reuse space, or is space so powerful that for any polynomial-time computation, you can get away with *logarithmic* space? We believe the answers to both questions are no (and we'll see completeness and similar things around them).

There's also the question of determinism vs. nondeterminism — for example, P vs. NP or $\mathsf{LOGSPACE}$ vs. $\mathsf{NLOGSPACE}$ and so on. We believe that $\mathsf{P} \neq \mathsf{NP}$. But the second question is a bit weird — for example, we do have $\mathsf{NSPACE}[\log n] \subseteq \mathsf{SPACE}[(\log n)^2]$.

Another question to consider is deterministic vs. randomized algorithms. Until now, all the classes we've seen whose names differed in even one letter are believed to be different. But here, the prevailing belief is that P is equal to BPP (this is the class of problems solvable in randomized polynomial time). There's also a version for this for logarithmic space — $\mathsf{LOGSPACE}$ vs. $\mathsf{BPLOGSPACE}$ — and it's also conjectured that these are equal. So we *believe* that randomized time can be simulated deterministically with a low overhead, but we're pretty far from *actually* understanding the power of BPP.

Open question 1.49. Is $\mathsf{NEXP} \neq \mathsf{BPP}$?

This is a really strange question — it'd be strange even if $\mathsf{EXP} = \mathsf{BPP}$ (this would mean that randomization *exponentially* speeds up determinism, which sounds insane). And there are some approaches to attacking this question, but right now it's open.

Given this, why on earth do people actually believe $\mathsf{P} = \mathsf{BPP}$? The reason is that it'd follow from circuit lower bounds that we *believe* to be true.

Theorem 1.50 (IW 1997)

If there exists $f \in \mathsf{TIME}[2^{O(n)}]$ such that f requires circuits of size at least 1.0001^n, then $\mathsf{P} = \mathsf{BPP}$.

(Recall that circuits are tricky — we have a separate circuit for each input length — so the hypothesis here doesn't follow from the time hierarchy theorem.)

In fact, this hypothesis would imply something even stronger — that PRGs exist.

This is an example of something that we call *hardness to randomness* — a framework for taking lower-bound statements like this and getting things like derandomization (i.e., removing the randomness from algorithms). We're essentially showing that deterministic time being 'hard' in some sense means that it's 'powerful' (in that randomized time can be simulated with deterministic time). We'll talk about this theorem in depth in the course.

There's also work in other directions — for example, if we can show $\mathsf{P} = \mathsf{BPP}$, then this will imply some circuit complexity lower bounds (which we will not state, because they're complicated).

Remark 1.51. A circuit complexity bound as in the hypothesis of Theorem 1.50 seems very hard to show — for all we know, $\mathsf{TIME}[2^{O(n)}]$ could have *linear*-sized circuits. But there do exist intermediate results — if we could prove certain weaker hypotheses, then we could also get certain weaker conclusions.

§1.10 Some other topics

Almost every limitation we've talked about so far has been a worst-case one. But sometimes we want *average-case* lower bounds — we want to say that no matter what computer we've got, the problem can't be solved on a decent *fraction* of instances. This is related to questions such as whether $\mathsf{P} \neq \mathsf{NP}$ implies cryptography (e.g., that one-way functions exist), since in order to get cryptographic constructions you need average-case hardness (where there's not just *one* bad input, but bad inputs everywhere you look).

Another thing we might not be able to cover in this course is randomized vs. quantum algorithms — there, the big question is BPP vs. BQP.

Finally, there's the topic of *probabilistically checkable proofs* (or PCPs). A PCP is essentially a proof system where you can check proofs by querying only a constant number of bits. This is somewhat weird, but it turns out that PCPs are inherently related to the hardness of optimization problems — specifically, to proving that some NP-hard optimization problems are hard to even *approximately* solve.

§2 The class coNP and the polynomial hierarchy

Today we'll introduce a number of concepts and theorems related to the P vs. NP problem.

§2.1 Some properties of P

Early on, people approached the problem of P vs. NP by noting that P has lots of very nice properties, and if we could show that NP *doesn't* have some of these properties, then we could separate the two. So we'd like to find some property of P that NP doesn't have.

Last week, we mentioned P has very nice closure properties. Here's two properties we'll look at today:

(1) P is closed under complement — if we're given a function $f \in \mathsf{P}$, then its negation $\neg f = 1 - f$ (which flips the output) is also in P. (As a reminder, when we talk about decision problems, we think of them as functions $\{0,1\}^* \to \{0,1\}$ — we take arbitrary binary strings as input, and output either *accept* or *reject*.) This is because with a deterministic Turing machine, we can simply flip the answer.

(2) Another property, which is a bit more complicated, is closure under *composition* — suppose we take some function f, and suppose f has a polynomial time algorithm that sometimes calls some other function $g \in \mathsf{P}$ as a subroutine. Then f is also in P — the proof boils down to the fact that if $p(n)$ and $q(n)$ are polynomials, so is $p(q(n))$.

So in other words, if we have a polynomial-time function calling a polynomial-time oracle, then we can rewrite it as a polynomial-time function.

Question 2.1. Does NP have these properties?

In other words, is NP closed under complement — if we flip the bit of a NP function, is it still in NP? And if we have a NP function that makes calls to another function in NP, is it still in NP?

We can define these meaningfully, and they'll lead to interesting classes.

Remark 2.2. We do not believe that NP is closed under complement or under composition (in the way we'll define, or really in any reasonable definition). But these are open problems. When trying to understand whether NP has these nice properties, we'll end up defining different classes that capture *apparently* different problems, but we don't know how to prove that they really are different.

§2.2 The class coNP

First we'll consider the property of closure under complement; this leads to the definition of coNP.

Definition 2.3. We define coNP $= \{f\colon \{0,1\}^* \to \{0,1\} \mid \neg f \in \mathsf{NP}\}$.

We may have also seen this in terms of languages — then coNP $= \{L \subseteq \{0,1\}^* \mid \{0,1\}^* \setminus L \in \mathsf{NP}\}$.

Then NP is closed under complement if and only if NP = coNP. Whether this is true is an open question. But because we know P is closed under complement, if we prove NP $\neq$ coNP, then you will also have shown P $\neq$ NP (because NP is not closed under complement, but P is).

There's a different way to characterize coNP, in terms of polynomial-time verifiers:

Fact 2.4 — We can view coNP as the set of functions $f\colon \{0,1\}^* \to \{0,1\}$ such that there exists a 'verifier' $g\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ such that for all inputs x, we have $f(x) = 1$ if and only if for *all* possible witnesses y of length $|y| = \mathsf{poly}(|x|)$, we have $g(x, y) = 1$.

Let's think about what this means — if we think about the verifier characterization of NP, our function f should output 1 if and only if there *exists* some y of polynomial length such that the verifier, given our input x and the witness y, outputs 1. With coNP, what we're saying is that we just replace *exists* with *for all*. You can think of 'co-nondeterministic computation' as some other mode of computation where instead of magically guessing the correct witness that leads you to accept, you verify over *all* possible witnesses in parallel that all lead to accept — if a *single* witness tells you to reject, then you reject.

This is a very important way of thinking about coNP. It's sort of the complement of nondeterministic computation — where instead of existentially guessing a witness that leads to an answer, we're going over *all* possible witnesses (and accepting if all lead to acceptance).

§2.2.1 Some coNP-complete problems

There are lots of coNP-complete problems (problems which are in coNP and are coNP-hard) which quite naturally have algorithms that work like this (i.e., by quantifying over all possible witnesses). Here are some examples.

Definition 2.5 (UNSAT)

- **Input:** a Boolean formula φ.
- **Decide:** whether φ has *no* satisfying assignments.

Definition 2.6 (Tautology)

- **Input:** a Boolean formula φ.
- **Decide:** whether φ is *always* true (regardless of what the variables are set to).

Similarly, problems like NoHamPath (checking whether a graph *doesn't* have a Hamiltonian path) and NoClique (checking whether a graph *doesn't* have a clique of a given size) are coNP-complete (they're essentially the flipped versions of the NP-complete problems HamPath and Clique).

We can think of each of these as defined over a 'for all' quantifier (as in the definition with g) — we have $\mathsf{Unsat}(\varphi) = 1$ if and only if for *all* assignments a, we have $\varphi(a) = 0$. Similarly, if we look at Tautology, we're simply checking whether for all assignments a, we have $\varphi(a) = 1$.

Remark 2.7. The reason for this is just that when you complement an 'exists,' you get a 'for all.'

So coNP is a way of classifying the *complements* of NP problems. The question of NP = coNP is essentially asking whether given a Boolean formula that's a tautology, is there an efficient *proof* that it is a tautology? Do we really have to try all the possible assignments and check that they output `T`, or can we do this checking more efficiently (in polynomial time)?

§2.3 Time hierarchy for nondeterminism

The NP vs. coNP question has bearing on a theorem we saw last time, namely the *time hierarchy theorem*.

Theorem 1.26 (Time hierarchy theorem)

There is some $c > 1$ such that for all 'nice' $t\colon \mathbb{N} \to \mathbb{N}$, there is a function $f \in \mathsf{TIME}[t(n)^c] \setminus \mathsf{TIME}[t(n)]$.

(Here 'nice' means time-constructible.)

Remark 2.8. On the problem set, we'll see that c can be made arbitrarily close to 1.

Another way of saying this, which we may have seen before, is that $\mathsf{TIME}[t(n)] \subsetneq \mathsf{TIME}[t(n)^c]$ — so as we strictly increase the time we allow to solve problems, there exist more problems we can solve (though the problems provided by the time hierarchy theorem are a bit unnatural — they involve taking the code of some machine and then simulating it and doing the opposite — and it's another issue to find natural problems in the in-between space).

We'd like to extend this to nondeterminism — we'd like to say there's something solvable in $\mathsf{NTIME}[t(n)^c]$ but not $\mathsf{NTIME}[t(n)]$ (e.g., a problem that has proofs of size $O(n^{100})$ but not $O(n)$).

§2.3.1 Some difficulties

In the proof of the time hierarchy theorem, we defined our 'bad' function f as $f(\langle \mathcal{M} \rangle) = 0$ if and only if $\mathcal{M}(\langle \mathcal{M} \rangle)$ (i.e., $\mathcal{M}$ run on its own description) outputs 1 in at most $|\mathcal{M}|\, t(|\mathcal{M}|)$ steps. (We use $\langle \mathcal{M} \rangle$ to denote the 'code' of $\mathcal{M}$ — some binary description of our machine.)

Suppose we want to carry this argument out for nondeterminism. Then we have a problem — we need to first simulate a machine $\mathcal{M}$ on itself, and then output the *opposite* answer. (In order to output 0, we need to check whether $\mathcal{M}$ on $\langle \mathcal{M} \rangle$ outputs 1.) And this seems to require taking the complement, which looks problematic.

Just doing the simulation itself can be done — there is a universal simulator for nondeterministic algorithms. In other words, there's one algorithm that can simulate them all, with the usual properties you want — it takes in the code of a nondeterministic algorithm, an input, and a time bound; and then it simulates step by step and does whatever the given machine does. Explicitly, there is a nondeterministic $\mathcal{U}$ such that $\mathcal{U}(\langle \mathcal{M} \rangle, x, 1^t)$ will simulate $\langle \mathcal{M} \rangle$ on x for t steps, and give the *same* answer. But with nondeterminism, we can't just flip the bit to get the *opposite* answer.

So this might make us wonder, is there *no* time hierarchy theorem for nondeterminism? If there were no nondeterminstic time hierarchy theorem — i.e., if $\mathsf{NTIME}(n) = \mathsf{NTIME}(n^{1+\varepsilon})$ for some ε — then by an analogous argument to the one on the problem set, we would obtain that $\mathsf{NP} = \mathsf{NTIME}(n)$. And if you show this, then nondeterminism has *no* time hierarchy — n^{10^9} witnesses can always just be made linear size (and verifiable in linear time). This would be so far from what P is that it would already imply $\mathsf{P} \neq \mathsf{NP}$ (intuitively, the idea of the proof is that 'the time hierarchy theorem is true for P but not NP').

Unsurprisingly (given this), there *is* a nondeterministic time hierarchy — if not, we'd have already proven $\mathsf{P} \neq \mathsf{NP}$ (it'd mean nondeterminism is so powerful and weird that we can already separate them). But the point of all this discussion is to illustrate that the proof is going to have to be somewhat weird — we've got to do something other than simulate the given code and do the opposite.

Theorem 2.9 (Nondeterministic time hierarchy theorem)

For all time-constructible functions $s, t \colon \mathbb{N} \to \mathbb{N}$ such that $t(n+1) = o(s(n))$, we have $\mathsf{NTIME}(t(n)) \subsetneq \mathsf{NTIME}(s(n))$.

Intuitively, the given hypothesis states that s grows at least a little bit faster than t — for example, we could take them to be 1.9^n and 2^n, or one polynomial and another polynomial of slightly higher degree. And for any such s and t, there's some problem that can be solved with a proof verifiable in $s(n)$ time, but not with s replaced by t.

This turns out to be very useful — there's an entire industry of lower bounds proved by assuming that we have a nice algorithm, and somehow using it to cook up algorithms that contradict the hierarchy theorem. (We somehow do magic guessing to take an algorithm that runs in $s(n)$ time and get it to run in $t(n)$ time.) So this theorem is very important from the point of view of lower bounds.

§2.3.2 The proof idea

The idea of the proof is essentially 'procrastination diagonalization.' We're going to put off taking the complement — we get an input $\langle \mathcal{M} \rangle$, and we just say, you know what, I'm going to simulate you on a slightly larger input and do whatever you do on that slightly larger input. And on longer input, we just add one more and do whatever you do. And we keep doing this until our input length is so long that brute force simulation of nondeterminism actually takes linear time in the input length. So we essentially put off taking the complement until the input is *so* long that we actually can.

So the idea is that we procrastinate — we acknowledge that we can't actually take the complement until we have 'exponential time' (we don't know how to complement because we don't know if $\mathsf{NP} = \mathsf{coNP}$), but we do know how to procrastinate, so we'll do that.

§2.3.3 Some simplifications

First, we're going to prove a slightly modified statement, to make the proof simpler — we'll construct a function f that's in $\mathsf{NTIME}(t(n+1)^c)$ but not in nondeterministic time $t(n)$, where c is the overhead we have for our universal simulator. It turns out that with nondeterminism, you can get a universal simulator with $c = 1$ (i.e., you can simulate with only constant overhead), but we won't prove this. Also, what we really need to show is that f is not in nondeterministic time $O(t(n))$; the proof can be modified to get this, but we won't do so, and for simplicity we'll just consider time exactly $t(n)$ (without the leading constants).

The next simplification we'll make is to assume that every nondeterministic time-t algorithm has at most 2^t computation paths — every time we make a nondeterministic step, we'll say we have 2 choices. (We could theoretically have any constant number of choices — the number of choices at each step is bounded by the length of the code, but it could be an arbitrary constant — but for simplicity we'll make this assumption. The proof can be modified to deal with the general case of having c^t computation paths.)

§2.3.4 The construction of f

We'll now give the pseudocode of f — we'll give a nondeterministic algorithm for f that takes time $O(t(n+1)^c)$, and then show that this function can't be implemented in time $t(n)$.

Algorithm 2.10 (f) — On input $x = \langle \mathcal{N} \rangle 10^k$ (where $\mathcal{N}$ is a nondeterministic machine and $k \geq 0$ — if x isn't of this form, *reject*), let $y = \langle \mathcal{N} \rangle 1$ be x without the padded 0's, and let $s = 2^{2t(|y|)}$.

(1) First suppose $k < s$. This is the 'procrastination' step — we simulate $\mathcal{N}$ on the input $x0$ for $t(|x|+1)$ steps, and output whatever answer it gives (if it doesn't answer in the given number of steps, then we cut it off and *reject*).

(2) On the other hand, if $k \geq s$, then we simulate $\mathcal{N}$ on y by brute-force — by trying all the $2^{t(|y|)}$ possible paths of length $t(|y|)$ — and output 1 if and only if $\mathcal{N}$ rejects all paths.

The idea is that (1) is what we mean by procrastination — we're given an input x of length n, and we simulate $\mathcal{N}$ on an input of length $n+1$ (using a universal simulator). Meanwhile, (2) is the interesting step where we do something devious — the condition $k \geq s$ means that the number of 0's is much, much bigger than the length of y. The point is that this is the step where we're taking a complement, but we're not trying to complement $\mathcal{N}$ on the input we're given — instead, we're complementing $\mathcal{N}$ on a much *smaller* input (which allows us to brute-force).

Remark 2.11. The reason we toss in a 1 (i.e., we take $y = \langle \mathcal{N} \rangle 1$ rather than just $\langle \mathcal{N} \rangle$) is just to get an unambiguous encoding — it tells us where to chop off the 0's.

§2.3.5 Proof of the upper bound

We'll now show that this function does what we want. First, we'll prove the upper bound.

> **Claim 2.12** — There exists c such that $f \in \mathsf{NTIME}(t(n+1)^c)$.

Proof. For this, we need to verify that both cases can be handled in the time bound. For (1), as we've said before, there exists a universal simulator for a nondeterministic algorithm that works the way you think it would (it simulates step by step; it makes a guess if it's able to guess; and so on). (The reason we have $t(n+1)$ instead of $t(n)$ is that we're running the simulator on an input of length $n+1$.) So we set c to accomodate whatever overhead we need for the universal simulator.

Meanwhile, we've set things up so that the second case — in particular, the part where we're actually brute-forcing over all possible paths — takes linear time. The point is that we need to take $2^{t(|y|)}$ time in order to try all possible paths (with an extra factor of $\mathsf{poly}(t(|y|))$ from the simulation overhead, but this isn't important). This is less than $s = 2^{2t(|y|)}$, and we assumed that $k \geq s$; so this is less than our input length (which is at least k). Intuitively, the point is that we don't run this step unless the input is so long that simulating without the 0's is actually quick (relative to the input length, which is huge). □

(The upper bound is the easier part; the lower bound is the tricky one.)

> **Remark 2.13.** The simulations in the two cases are very different — in the first case we simulate step-by-step, making guesses whenever $\mathcal{N}$ does (using a universal simulator). In the second case, we're instead brute-forcing and doing the opposite (on some much smaller input).

§2.3.6 Proof of the lower bound

So far, we've proven why our function f can be solved nondeterministically (with polynomial overhead in our runtime t). Now we want to show that no nondeterministic algorithm running in time t can compute f.

> **Claim 2.14** — No nondeterministic $t(n)$-time algorithm computes f.

Proof. Let's look at what the two steps (1) and (2) are doing in a more succinct way. In (1), we're saying that for all $k = 0, 1, \dots, s-1$, we define

$$f(\langle \mathcal{N} \rangle 10^k) = \mathcal{N}(\langle \mathcal{N} \rangle 10^{k+1}).$$

(This is because to compute f, we just add one 0 to our input, then simulate $\mathcal{N}$, and output whatever it does — to be more precise, this is only true if $\mathcal{N}$ runs in time $t(n)$, but this will be true for our purposes.) Meanwhile, (2) says that

$$f(\langle \mathcal{N} \rangle 10^s) = \neg \mathcal{N}(\langle \mathcal{N} \rangle 1).$$

We want to get a contradiction — suppose that there is a nondeterministic algorithm $\mathcal{M}$ computing f which runs in $t(n)$ time (so then our simulator has enough time to simulate $\mathcal{M}$).

Now we ask, what happens when we run $\mathcal{M}$ on its own code? (This is what we generally do in diagonalization proofs.) First, since $\mathcal{M}$ is equivalent to f, we can stick in $\mathcal{M}$ everywhere in place of $\mathcal{N}$ and f. Then (1) tells us that

$$\mathcal{M}(\langle \mathcal{M} \rangle 10^k) = \mathcal{M}(\langle \mathcal{M} \rangle 10^{k+1})$$

for all $0 \leq k \leq s-1$. (This is because $\mathcal{M}$ runs in time $t(n)$, so when we simulate in (1), we never cut it off early.) So $\mathcal{M}$ does the same thing on its own code padded with k 0's as it does with one extra 0.

But then (2) tells us that $\mathcal{M}(\langle\mathcal{M}\rangle 10^s) = \neg\mathcal{M}(\langle\mathcal{M}\rangle 1)$ — if we run $\mathcal{M}$ on its own code with enough 0's, then we do the opposite of what we'd do if we had no 0's at all.

But this is a contradiction — these equalities cannot all simultaneously hold. Explicitly, (1) tells us

$$\mathcal{M}(\langle\mathcal{M}\rangle 1) = \mathcal{M}(\langle\mathcal{M}\rangle 10) = \mathcal{M}(\langle\mathcal{M}\rangle 100) = \cdots = \mathcal{M}(\langle\mathcal{M}\rangle 10^s).$$

But then from (2), we get that $\mathcal{M}(\langle\mathcal{M}\rangle 10^s) = \neg\mathcal{M}(\langle\mathcal{M}\rangle 1)$. So this is a contradiction. □

Remark 2.15. The reason we need $\mathcal{M}$ to run in $t(n)$ time for this to work is that in (1), we're only simulating the machine we're given on $x0$ for $t(|x|+1)$ steps. (If $\mathcal{M}$ took longer, then we'd cut it off and reject before we got its answer, and we wouldn't have the equalities from (1).) To get the stronger statement with $O(t(n))$ instead of just $t(n)$, we'd need to multiply $t(|x|+1)$ by some unbounded function and simulate for that amount of time instead.

Remark 2.16. In the strong version of the theorem (stated at the beginning), we just require $t(n+1) = o(s(n))$. The point is that s is big enough that we can simulate $t(n+1)$ times any constant we like — so that way we can diagonalize against any machine running in $O(t(n))$ time. The reason we don't have c in this statement is that c is the constant coming up in whatever nondeterministic universal simulator you like; and there exists one where $c = 1$. So if you stick in the right universal simulator into the same argument (where we simulate for $s(n)$ steps instead of $t(n+1)$), then you get this statement. (But we're abstracting this away and not worrying about it; we're sort of trying to emphasize that this argument is robust to the model you use.)

Remark 2.17. This is a kind of weird proof. One analogy is it's kind of like a baldness paradox — no one with more than 100000 hairs on their head is bald. Removing one hair doesn't baldify a non-bald person. But a person with 0 hairs is bald. This is kind of similar — we add one more hair (one more bit to our input length), and eventually we add so many bits that actually we can just do the opposite.

As a corollary, last class, we talked about NEXP, the class of problems solvable in nondeterministic exponential time (this is a huge class).

Corollary 2.18

We have NEXP $\neq$ NP.

(This is similar to the statement P $\neq$ EXP.)

§2.4 Closure under composition and oracle classes

Now we're done talking about closure under complement, and we'll move to composition.

Question 2.19. Is NP closed under composition?

There's a way to make this very formal, using the concept of *oracles*.

§2.4.1 Oracles

Definition 2.20. An oracle is a function $A\colon \Sigma^* \to \{0,1\}$, where Σ is a finite alphabet.

(We may have seen this as a language — a subset $A \subseteq \Sigma^*$ — but the two formulations are equivalent, since for every language we can define a function telling us whether a string is in the language or not.)

For every oracle, we can define an *oracle Turing machine* (or *oracle algorithm*).

Definition 2.21 (Oracle Turing machine). A Turing machine with oracle A, written $\mathcal{M}^A$, is a Turing machine with a read-only input tape (where it reads the input), a read-write storage (where it does its thinking), and an *oracle tape*. The machine is allowed to query the oracle — it can write some query z on its oracle tape, and enter a special state to ask the oracle a question; and we define the transition function to enter q_{yes} if $z \in A$ (equivalently $A(z) = 1$) and q_{no} otherwise.

So this is an extension of the Turing machine model where we get to write to A and get answers. (We don't care what A is, or how the oracle query steps are computed — for example, A doesn't even have to be a computable function.)

Remark 2.22. If you don't like to think about Turing machines, you can think of an algorithm with oracle A as an algorithm with a new kind of if-then statement — for some previously-defined z, we can have statements of the form 'if $A(z) = 1$ then — else —' (everything we'll write will use this if-then phrasing anyways).

§2.4.2 Complexity classes

When we think about complexity classes with oracles, we're usually thinking about the following question.

Question 2.23. Given the power to solve A, what *other* problems can we solve?

Definition 2.24. We define P^A as the set of decision problems f such that f is decidable in polynomial time with an oracle A.

Importantly, we can often raise one complexity class to another.

Definition 2.25. We define P^{NP} as the set of f such that there exists some $A \in \mathsf{NP}$ such that $f \in \mathsf{P}^A$.

Example 2.26
We have $\mathsf{Tautology} \in \mathsf{P}^{\mathsf{SAT}}$.

Proof. Given φ, how do we figure out whether φ is a tautology (i.e., something that's always true) using a SAT oracle? We can simply run SAT on $\neg\varphi$ (the negation of φ) — if $\neg\varphi$ is satisfiable then we know φ is *not* a tautology, so we *reject*; and otherwise we *accept*. □

So with just one oracle call, we're solving a coNP problem. (The point is essentially that with a NP oracle we always get back a 'yes' or a 'no' answer, and we can flip the answer ourselves, so the difference between NP and coNP disappears.)

§2.4.3 Closure under composition

Using oracles, there's a nice way to formalize the statement that P is closed under composition.

Theorem 2.27
For all $A \in \mathsf{P}$, we have $\mathsf{P}^A = \mathsf{P}$. (In other words, $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$.)

This describes closure under composition — we have a class, and whenever we have something in this class calling a subroutine in the class as an oracle, we're still in the class.

Question 2.28. Is NP closed under composition in the same way?

We don't know the answer to this, but it leads to the definition of a new class.

Definition 2.29. For any oracle A, we define NP^A as the set of functions decidable in nondeterministic polynomial time with oracle A.

Definition 2.30. We define $\mathsf{NP}^{\mathsf{NP}}$ as the set of functions f such that there exists an oracle $A \in \mathsf{NP}$ for which we have $f \in \mathsf{NP}^A$.

How can we make sense of this? When thinking about NP, it's often useful to think in terms of the verifier formulation (Fact 1.17). Essentially, for f to be in NP means that it has a 'nifty proof' — i.e., there's a polynomial-time verifier $\mathcal{V}$ such that $f(x) = 1$ if and only if there exists y such that $\mathcal{V}(x, y) = 1$.

And it's possible to show that f being in NP^A means that there's a verifier *with oracle* A such that $f(x) = 1$ if and only if there exists y such that the verifier accepts (x, y) — so now our verifier is allowed to call the A-oracle as well. Then for $\mathsf{NP}^{\mathsf{NP}}$, we essentially get to guess y and then call a NP oracle to check.

Then we can formulate Question 2.28 in the following way.

Question 2.31. Is $\mathsf{NP}^A = \mathsf{NP}$ for all $A \in \mathsf{NP}$?

Next class, we're going to define a class PH — called the *polynomial hierarchy* — such that $\mathsf{NP} \subseteq \mathsf{PH}$ and $\mathsf{PH}^{\mathsf{PH}} = \mathsf{PH}$ (PH is essentially the 'closure of NP under composition'). This class almost certainly contains problems not in NP. But we'll show that if $\mathsf{P} = \mathsf{NP}$, then the *entire* polynomial hierarchy collapses to P — i.e., $\mathsf{P} = \mathsf{PH}$. This means if $\mathsf{P} = \mathsf{NP}$, then not only can we solve SAT in polynomial time, but we can solve all kinds of weird logic optimization problems that we don't even believe to be in NP.

§2.4.4 Some examples

First, we'll make a few observations regarding these oracle classes.

Fact 2.32 — We have $\mathsf{P}^{\mathsf{SAT}} = \mathsf{P}^{\mathsf{NP}}$ and $\mathsf{NP}^{\mathsf{SAT}} = \mathsf{NP}^{\mathsf{NP}}$.

Proof. The fact that $\mathsf{P}^{\mathsf{SAT}} \subseteq \mathsf{P}^{\mathsf{NP}}$ is immediate, since $\mathsf{SAT} \in \mathsf{NP}$. For the reverse direction, the point is that SAT is NP-complete. So if we consider any problem which can be solved with an A-oracle for some $A \in \mathsf{NP}$, then we can instead reduce A to SAT in polynomial time and run our SAT-oracle on this SAT problem — this lets us implement the A-oracle with a SAT-oracle instead. (The same proof shows $\mathsf{NP}^{\mathsf{SAT}} = \mathsf{NP}^{\mathsf{NP}}$.) □

Fact 2.33 — For any oracle A, we have $\mathsf{P}^A = \mathsf{P}^{\neg A}$ (where $\neg A$ is the oracle $\neg A(x) = 1 - A(x)$).

The point is that the oracle A gives us a single bit as its answer, and we can just flip that bit. (The same statement is true if we replace P with NP — this is really a property of oracles, that they give a yes-no answer and we can just flip the answer.)

Now let's look at a few interesting problems in $\mathsf{NP}^{\mathsf{NP}}$ (or $\mathsf{coNP}^{\mathsf{NP}}$).

Definition 2.34 (MinFormula)
- **Input:** a Boolean formula φ.
- **Decide:** is φ minimal? (We say φ is *minimal* if there is no equivalent formula ψ with $|\psi| < |\varphi|$.)

Here φ is a Boolean formula (e.g., with **AND**, **OR**, and **NOT**), and we fix some encoding of such formulas in binary. And the question we're asking is, given φ, is there no formula ψ whose length (in this encoding) is strictly smaller than that of φ, and which computes the same function? In other words, for φ to be minimal means that for every ψ with $|\psi| < |\varphi|$, there is some x with $\varphi(x) \neq \psi(x)$.

Checking minimality of formulas is important in circuit design and logic design, so this is a reasonably natural problem. But it seems a bit strange — we have a universal quantifier over ψ, as well as an existential quantifier over x. So in some sense, this seems more general than either NP or coNP.

Claim 2.35 — We have $\mathsf{MinFormula} \in \mathsf{coNP}^{\mathsf{NP}}$.

Proof. First, how can we reason about something like this? In the pseudocode for a coNP algorithm, we get to do some sort of 'try all' over polynomial-length strings y, then run some polynomial-time procedure on both x and y, and accept if and only if *all* choices of y lead to acceptance. (We can think of this as a *co-nondeterministic* machine model — in contrast, with a nondeterministic model, we'd try all y and accept if there's *some* guess that leads to acceptance.)

And for this problem, we can just write the following pseudocode: given φ, we do a 'for all' over ψ with $|\psi| < |\varphi|$. If $\psi \not\equiv \varphi$ then we *accept*; otherwise we *reject*.

The coNP part is the 'for all.' Meanwhile, checking that $\psi \not\equiv \varphi$ is a NP query — we want to know whether there exists x such that $\varphi(x) \neq \psi(x)$, and this is a NP problem. So this gives a coNP algorithm making some oracle call to NP. $\square$

Open question 2.36. Is MinFormula $\mathsf{coNP}^{\mathsf{NP}}$-complete?

This is at least a reasonable question — MinFormula certainly *looks* much harder than NP or coNP, and it's got both a coNP part (where we quantify over all ψ) and a NP part (where we call the oracle).

Remark 2.37. In this algorithm, we just called the oracle once. But in general we're allowed to call it any polynomial number of times.

We don't know whether MinFormula is $\mathsf{coNP}^{\mathsf{NP}}$-complete, but in fact we *do* know of $\mathsf{coNP}^{\mathsf{NP}}$-complete problems, and we'll get to one later.

Today we'll define the polynomial hierarchy, which we can informally think of as the closure of NP under composition — it'll contain NP, coNP, P^{NP}, $\mathsf{NP}^{\mathsf{NP}}$, $\mathsf{coNP}^{\mathsf{NP}}$, and so on. So in particular, MinFormula is one example of a (reasonably natural) problem in the polynomial hierarchy. Here's another such problem.

Definition 2.38
Let $\mathsf{MaxClique} = \{\langle G, s\rangle \mid s \text{ is the maximum size of a clique in } G\}$.

In the problem Clique, we just want to know whether there *exists* a clique of size s in G. But here we want to know not just this, but also that there's no clique of size $s + 1$.

How could we solve $\mathsf{MaxClique}$ using an oracle? Given input $\langle G, s\rangle$, we want to *accept* if $\langle G, s\rangle \in \mathsf{Clique}$ and $\langle G, s+1\rangle \notin \mathsf{Clique}$, and *reject* otherwise. This shows $\mathsf{MaxClique} \in \mathsf{P}^{\mathsf{NP}}$ — we're just deterministically making two oracle queries, and accepting if both give the right answer.

Finally, here's a third example.

Definition 2.39 ($\exists\forall$-SAT)

- **Input:** a statement $\varphi = (\exists\, x_1, \ldots, x_n)(\forall\, y_1, \ldots, y_n)[F(x, y)]$, where F is a Boolean formula.
- **Decide:** whether this statement is true or false.

This is sort of a generalization of SAT and Tautology — SAT is what we get when we don't have y, and Tautology is what we get when we don't have x.

Theorem 2.40

The problem $\exists\forall$-SAT is in $\mathsf{NP}^{\mathsf{NP}}$, and in fact is $\mathsf{NP}^{\mathsf{NP}}$-complete.

We won't show it's complete right now, but to see that it's in $\mathsf{NP}^{\mathsf{NP}}$, we can again write down pseudocode: given φ, we first guess an assignment a for x. Then we want to check whether $F(a, y)$ is a tautology, or equivalently $\neg F(a, y)$ is unsatisfiable — so we use the oracle to check whether $\neg F(a, y) \in \mathsf{SAT}$. If the answer is yes, then we *reject*; otherwise we *accept*.

Remark 2.41. How does this compare to the problem TQBF? In TQBF, we're given a statement of this form, but with an arbitrary number of $\exists$'s and $\forall$'s — i.e., we're given

$$\varphi = (\exists\, x_1)(\forall\, x_2)(\exists\, x_3)\cdots[F(x_1, x_2, x_3, \ldots)],$$

where the quantifiers can be in any order you want. But with $\exists\forall$-SAT, we only have a bunch of $\exists$'s followed by a bunch of $\forall$'s.

But this does in particular show that $\exists\forall$-SAT somehow sits between NP and PSPACE (it sits above NP and coNP, but it stays in PSPACE since we've still got a QBF).

§2.5 The polynomial hierarchy

Now we'll define PH, which will contain NP and all these other classes (and which we can think of in some sense as the closure of NP under composition).

Definition 2.42. We define $\mathsf{PH} = \bigcup_{k\geq 1} \mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}^{\cdots}}}$, where the kth term (called the kth level) has k NP's.

The point is that in the kth level of this definition, we take the complexity class defined in the $(k-1)$th level, make that into an oracle, and give ourselves NP-access to that oracle. (Note that $\mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$ should be read as $\mathsf{NP}^{(\mathsf{NP}^{\mathsf{NP}})}$, as with repeated exponentiation of numbers.)

Conjecture 2.43 — For all k, the kth level of the polynomial hierarchy is not equal to PH.

In other words, this conjecture states that as we increase the number of levels, we should be able to solve more problems. (We can sort of think of this as being able to add more quantifier power to our QBFs — for example, $\mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$ would let us take QBFs with $\exists\forall\exists$ quantifiers.) This conjecture is often called 'PH doesn't collapse,' and it's open whether it's true or false (most people believe it's true, but it's pretty hard to reason about things involving e.g. more than three NP's).

If $\mathsf{P} = \mathsf{PH}$, then certainly $\mathsf{P} = \mathsf{NP}$. But in fact, the converse is true — if $\mathsf{P} = \mathsf{NP}$, then all of this hierarchy collapses to P. (We'll prove this shortly.) However, we *don't* know whether $\mathsf{NP} = \mathsf{PH}$ implies $\mathsf{P} = \mathsf{NP}$. (It *does* imply $\mathsf{NP} = \mathsf{coNP}$, since $\mathsf{coNP} \subseteq \mathsf{NP}^{\mathsf{NP}}$ — you can simply call the oracle for the complement problem and then negate its answer.)

Theorem 2.44

We have $\mathsf{P} = \mathsf{NP}$ if and only if $\mathsf{P} = \mathsf{PH}$.

This is one of the main reasons for studying PH — to separate P from NP, it suffices to separate P from one of these bigger classes (i.e., one of these higher levels of PH). So this opens up a whole new space of possible problems you might want to prove lower bounds on (e.g., all these problems like $\mathsf{MinFormula}$).

Proof. One direction is obvious — we have $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PH}$, so if $\mathsf{P} = \mathsf{PH}$ then of course $\mathsf{P} = \mathsf{NP}$.

For the reverse direction, we need the following fact.

Fact 2.45 — For any classes $\mathcal{C}$ and $\mathcal{D}$, if $\mathcal{C} = \mathcal{D}$ then $\mathsf{NP}^{\mathcal{C}} = \mathsf{NP}^{\mathcal{D}}$.

This is true essentially by definition — we defined $\mathsf{NP}^{\mathcal{C}}$ as the set of functions which can be solved in NP with an oracle $A \in \mathcal{C}$, and if $\mathcal{C} = \mathcal{D}$ then $A \in \mathcal{C}$ if and only if $A \in \mathcal{D}$.

Now we want to show that for all k, if $\mathsf{P} = \mathsf{NP}$ then $\mathsf{P} = \mathsf{NP} \uparrow\uparrow k$ (where we use this notation to denote $\mathsf{NP}^{\mathsf{NP}^{\cdots}}$ with k copies of NP). This suffices because PH is just the union of $\mathsf{NP} \uparrow\uparrow k$ over all k, so if we show that each one of these is P, then PH would just be the union of a bunch of copies of P.

And we can show this by simple induction — the base case $k = 1$ is our assumption. For the inductive step, suppose that $\mathsf{P} = \mathsf{NP}$ and that we've already shown $\mathsf{P} = \mathsf{NP} \uparrow\uparrow k$. Then by Fact 2.45 we have

$$\mathsf{NP} \uparrow\uparrow (k+1) = \mathsf{NP}^{\mathsf{NP} \uparrow\uparrow k} = \mathsf{NP}^{\mathsf{P}}.$$

And calling a P oracle doesn't really do anything (we could just solve the problem ourselves), so NP^{P} is just NP. And we assumed that $\mathsf{NP} = \mathsf{P}$, so this gives $\mathsf{NP} \uparrow\uparrow (k+1) = \mathsf{P}$, and we're done. □

§2.6 A logical characterization of PH

Next, we'll talk about a logical characterization of PH. It's kind of annoying to write out things like $\mathsf{NP}^{\mathsf{NP}^{\mathsf{NP}}}$. So we might wonder, do we really need all these exponentials, or is there a nice, clean way of explaining what's going on when we raise NP to a $\mathsf{NP}^{\mathsf{NP}}$ oracle and so on? It turns out that there *is* a nice, clean way, and this is the way that people usually think about PH.

§2.6.1 The classes $\Sigma_k\mathsf{P}$ and $\Pi_k\mathsf{P}$

First, we'll use a few convenient pieces of notation.

Notation 2.46. For a statement $\pi(x)$ (which is either true or false), we write $f(x) = \pi(x)$ to mean that $f(x)$ is 1 if $\pi(x)$ is true and 0 if $\pi(x)$ is false.

(The usual notation for this is $f(x) = 1_{\pi(x)}$, but we'll simply write $f(x) = \pi(x)$ for convenience.)

Notation 2.47. We write $\exists^{\mathsf{P}}\, y$ as an abbreviation for $(\exists\, y \mid |y| \leq \mathsf{poly}(|x|))$ (where x will denote our input), and $\forall^{\mathsf{P}}\, y$ as an abbreviation for $(\forall\, y \mid |y| \leq \mathsf{poly}(|x|))$.

Here's an example of this notation (used to state the verifier characterizations of NP and coNP mentioned in Facts 1.17 and 2.4).

Example 2.48

- We have $f \in \mathsf{NP}$ if and only if there exists $g \in \mathsf{P}$ with $f(x) = (\exists^{\mathsf{P}}\, y)[g(x, y) = 1]$.
- We have $f \in \mathsf{coNP}$ if and only if there exists $g \in \mathsf{P}$ with $f(x) = (\forall^{\mathsf{P}}\, y)[g(x, y) = 1]$.

Based on this, we can define classes with more quantifiers.

Definition 2.49. We define $\Sigma_2\mathsf{P}$ as the class of decision problems f for which there exists $g \in \mathsf{P}$ with

$$f(x) = (\exists^{\mathsf{P}}\, y)(\forall^{\mathsf{P}}\, z)[g(x, y, z) = 1].$$

With NP, we had a verifier g that took in an input x and a witness y, and then accepted or rejected. Here we can imagine we have a 'judge' g that takes in inputs from *two* players — y from the first player and z from the second — and accepts or rejects based on what they are (where the second player gets to choose z based on y).

It's easy to see that $\mathsf{NP} \subseteq \Sigma_2\mathsf{P}$, because we could just ignore the $\forall$ quantifier — i.e., we could just ignore z and run the NP verifier. Similarly, we have $\mathsf{coNP} \subseteq \Sigma_2\mathsf{P}$, because we could just ignore the $\exists$ quantifier — i.e., we could just ignore y and run the coNP verifier.

Definition 2.50. We define $\Pi_2\mathsf{P}$ as the set of decision problems f whose *complements* are in $\Sigma_2\mathsf{P}$ — equivalently, such that there exists $g \in \mathsf{P}$ with

$$f(x) = (\forall^{\mathsf{P}}\, y)(\exists^{\mathsf{P}}\, z)[g(x, y, z) = 1].$$

Here are some examples, to get an idea of what these classes mean.

Example 2.51

We have $\mathsf{MinFormula} \in \Pi_2\mathsf{P}$.

Proof. As seen in the proof of Claim 2.35, we can write MinFormula using logical quantifiers as

$$\mathsf{MinFormula}(\varphi) = (\forall\, \psi \mid |\psi| < |\varphi|)(\exists\, a)[\varphi(a) \neq \psi(a)]$$

(we read this as 'for all ψ such that $|\psi| < |\varphi|$, there exists a such that $\varphi(a) \neq \psi(a)$'). This is exactly a $\forall\exists$ statement as in Definition 2.50 (the definition of $\Pi_2\mathsf{P}$). □

And we can extend 2 to arbitrary k, where we have k (alternating) quantifiers instead of just 2.

Definition 2.52. We define $\Sigma_k\mathsf{P}$ as the set of decision problems f such that there exists $g \in \mathsf{P}$ with

$$f(x) = (\exists^{\mathsf{P}}\, y_1)(\forall^{\mathsf{P}}\, y_2) \cdots (Q_k^{\mathsf{P}}\, y_k)[g(x, y_1, \ldots, y_k) = 1]$$

(where the kth quantifier Q_k is $\forall$ if k is even and $\exists$ if k is odd).

(Note that here $y_1, \ldots, y_k$ are all polynomial-length strings, not bits.)

Definition 2.53. We define $\Pi_2\mathsf{P}$ as the set of decision problems f with $\neg f \in \Sigma_k\mathsf{P}$ — equivalently, there exists $g \in \mathsf{P}$ with

$$g(x) = (\forall^{\mathsf{P}}\, y_1)(\exists^{\mathsf{P}}\, y_2)\cdots(Q_k^{\mathsf{P}} y_k)[g(x, y_1, \ldots, y_k) = 1],$$

where Q_k is $\exists$ if k is even and $\forall$ if k is odd.

(When we negate, an $\exists$ turns into a $\forall$ and vice versa.)

Exercise 2.54. We have $\Sigma_k\mathsf{P} \cup \Pi_k\mathsf{P} \subseteq \Sigma_{k+1}\mathsf{P} \cap \Pi_{k+1}\mathsf{P}$.

Proof. The idea is that if we compare $\Sigma_k\mathsf{P}$ or $\Pi_k\mathsf{P}$ to $\Sigma_{k+1}\mathsf{P}$ or $\Pi_{k+1}\mathsf{P}$, we've got k alternating quantifiers vs. $k+1$. So we can always ignore a quantifier on one of the ends and things will work out — for example, if we want k quantifiers starting with an $\exists$ from $k+1$ starting with $\forall$, then we can strip off the first quantifier; while if we want the same thing from $k+1$ starting with $\exists$, then we can strip off the *last* quantifier instead. □

§2.6.2 An equivalence

We've now defined a bunch of classes involving towers of NP, and a bunch of other classes involving $\exists$'s and $\forall$'s. And it turns out that these are the *same* classes!

Theorem 2.55

We have $\mathsf{NP} \uparrow\uparrow k = \Sigma_k\mathsf{P}$ and $\mathsf{coNP} \uparrow\uparrow k = \Pi_k\mathsf{P}$ for all k.

(We could also have said $\mathsf{coNP}^{\mathsf{NP}^{\mathsf{NP}^{\cdots}}} = \Pi_k\mathsf{P}$ instead of $\mathsf{coNP}^{\mathsf{coNP}^{\mathsf{coNP}^{\cdots}}}$ — it doesn't make a difference, since we can take the complements of our oracles.)

Proof. We'll only show the proof for $k = 2$ — specifically, we'll only show that $\Sigma_2\mathsf{P} = \mathsf{NP}^{\mathsf{NP}}$. The statement for larger values of k can be shown by induction (but we'll not do it because it's kind of messy).

One direction is not hard — if $f \in \Sigma_2\mathsf{P}$, then we can show it's in $\mathsf{NP}^{\mathsf{NP}}$ in a pretty uniform way. By the definition of $\Sigma_2\mathsf{P}$, we have that there exists $g \in \mathsf{P}$ such that

$$f(x) = (\exists^{\mathsf{P}}\, y)(\forall^{\mathsf{P}}\, z)[g(x, y, z) = 1].$$

We'll now pull out the quantifier over y, and call the resulting function $h(x, y)$ — so we define

$$h(x, y) = (\forall^{\mathsf{P}}\, z)[g(x, y, z) = 1].$$

From this we can see that $h \in \mathsf{coNP}$, so $\neg h \in \mathsf{NP}$. And now we're trying to construct a NP algorithm for f that calls a NP oracle, and we can simply use $\neg h$ as our oracle — we define a nondeterministic algorithm $\mathcal{N}^{\neg h}$ which, on input x, first guesses y of polynomial length, then calls the oracle to find $\neg h(x, y)$, and *accepts* if $\neg h(x, y) = 0$ and *rejects* if $\neg h(x, y) = 1$. (The reason we're accepting on 0 and rejecting on 1 is because of the complement.) This is a nondeterministic algorithm that calls a NP oracle and decides f — the point is that we're guessing the $\exists$ part ourselves, and then using the oracle to check the $\forall$ part.

The other direction — that $\mathsf{NP}^{\mathsf{NP}} \subseteq \Sigma_2\mathsf{P}$ — is harder. The difficulty is that a $\mathsf{NP}^{\mathsf{NP}}$ machine could call its oracle many times, potentially adaptively; meanwhile, with $\Sigma_2\mathsf{P}$ we've got to have one $\exists$ followed by one $\forall$, which essentially corresponds to making *one* guess and then *one* oracle call. So we're in some sense saying that no matter how many oracle calls you make, it's possible to replace all of them with just *one* call.

Let's suppose we start with some nondeterministic oracle machine $\mathcal{N}^{\mathsf{SAT}}$ (by Fact 2.32 we can replace any NP oracle with a SAT one). We want to somehow turn the behavior of this algorithm into an $\exists\forall$ statement,

where we existentially guess some things, universally guess some more things, and check some polynomial-time computable predicate.

First, let $p(n)$ be some upper bound on the runtime of $\mathcal{N}^{\mathsf{SAT}}$, the length of all SAT queries, and the number of SAT queries that $\mathcal{N}^{\mathsf{SAT}}$ can make (on any thread) on inputs of length n. (It's supposed to run in nondeterministic polynomial time, so there's certainly *some* polynomial upper bound on these quantities.)

Now the idea is as follows. Given input x, we first imagine guessing a computation path P of $\mathcal{N}^{\mathsf{SAT}}$ on x (i.e., a sequence of transitions from the start state to the accept state, in the usual Turing machine model). And we also guess the queries it makes along the way — so we guess a bunch of SAT instances $F_1, \ldots, F_{p(n)}$ (each of which is a Boolean formula of length at most $p(n)$). And we *also* guess the *answers* to those SAT queries — so we guess $a_1, \ldots, a_{p(n)} \in \{0, 1\}$ (where a_i represents our guess for the answer to F_i — i.e., whether F_i is satisfiable or not).

Now we need some way of proving that our guesses for the answers are actually correct. So we also guess satisfying assignments $A_1, \ldots, A_{p(n)}$. This handles the queries that are satisfiable, but what about the queries that are *unsatisfiable*? To show that F_i is unsatisfiable, we really need to go through *all* possible assignments and prove that those assignments all make F_i false. So for this, we'll do a $\forall$ over possible assignments $A'_1, \ldots, A'_{p(n)}$.

So our quantified statement is going to look like

$$(\exists \,\text{path } P, \text{ queries } F_i, \text{ answers } a_i, \text{ assignments } A_i)(\forall \,\text{assignments } A'_i)[\cdots],$$

where $\cdots$ will be our polynomial-time checkable predicate. Now to define this predicate, we need to check that everything really works as described:

- First, we check that our computation path is valid assuming that our answers to the queries are. This means we imagine running $\mathcal{N}^{\mathsf{SAT}}(x)$ on the computation path P, and when it makes its ith query F_i, we answer it with a_i. (In other words, we're trying to run a specific path of $\mathcal{N}^{\mathsf{SAT}}(x)$ *assuming* that we've guessed all the right answers of the oracle.) If we find that this computation path is invalid (it contains an invalid transition or F_i is not the ith query that $\mathcal{N}^{\mathsf{SAT}}$ makes as we follow this path), then *reject*. If we find that $\mathcal{N}^{\mathsf{SAT}}$ rejects, then we also *reject*.
- Next, we need to check that all the answers are correct, or more precisely, that the guessed assignments A_i match the answers.

  First, we'll check the answers $a_i = 1$ (i.e., the answers where we claim F_i is satisfiable). To do so, we go through each i with $a_i = 1$ (there's only polynomially many such i), and check that $F_i(A_i) = 1$ for all such i. If there is any i for which this is *not* true, then we *reject*.
- Similarly, for each of the answers $a_i = 0$, we check that $F_i(A'_i) = 0$. If there is any i for which this is not true, then we *reject*.
- Finally, if we haven't yet rejected, then we *accept*. □

§2.6.3 Some consequences

Here are some consequences of this.

Corollary 2.56
We have $\Sigma_2\mathsf{P} = \mathsf{coNP}^{\mathsf{NP}}$, and $\mathsf{PH} = \bigcup_k \Sigma_k\mathsf{P} = \bigcup_k \Pi_k\mathsf{P}$.

This also gives us a natural way to define complete problems for each of these levels.

Definition 2.57 (Σ_kSAT)

- **Input:** a quantified Boolean formula of the form

$$\varphi = (\exists\, x_1)(\forall\, x_2)\cdots(Q_k\, x_k)[F(x_1, \ldots, x_k) = 1]$$

  (where each of $x_1, \ldots, x_k$ is a vector — consisting of many variables).
- **Decide:** whether φ is true or false.

We can show that Σ_kSAT is Σ_k-complete (in the same way that we show SAT is NP-complete, Tautology is coNP-complete, and so on).

Corollary 2.58
We have PH $\subseteq$ PSPACE.

Proof. Each Σ_kSAT problem is an instance of TQBF, which is in PSPACE. □

Question 2.59. Do there exist PH-complete problems?

By a PH-complete problem we mean a problem $f \in$ PH such that for all $g \in$ PH we have $g \leq_{\mathsf{p}} f$. The answer is yes if and only if PH collapses — if there's a complete problem f, then it has to lie in *some* level in PH, and this would cause the entire hierarchy to collapse to that level.

Remark 2.60. There's also a way of thinking about PH with respect to an oracle, under which we have $\mathsf{PH}^{\mathsf{PH}} = \mathsf{PH}$. (So PH is closed under composition in this sense.)

§3 Time-space tradeoffs for SAT

Last class, we defined the *polynomial hierarchy* PH, and we saw two ways of characterizing it — one in terms of a generalization of both nondeterminism and conondeterminism called $\Sigma_k\mathsf{P}$, and one in terms of oracle calls (as $\bigcup_k \mathsf{NP}^{\mathsf{NP}^{\cdots}}$). We can use PH to classify problems that seem harder than NP and coNP. But today, we'll see a really practical use of PH. You can think of machines in PH as generalizations of both nondeterministic and co-nondeterministic machines. But it turns out that you can use PH to prove concrete bounds (specifically, for SAT) on *deterministic* machines! In some sense, we're trying to prove things about realistic algorithms, and somehow we can do this by going through PH, which is quite unrealistic.

§3.1 Time-space classes

Today we're going to talk about progress on separating LOGSPACE and NP (we do believe they're different — for example, we don't think SAT can be solved in logarithmic space). You might wonder how we can measure how well we're doing in separating them; one way to quantify this is by looking at *time-space classes*.

Definition 3.1. For a time bound $t\colon \mathbb{N} \to \mathbb{N}$ and a space bound $s\colon \mathbb{N} \to \mathbb{N}$, we define $\mathsf{TS}[t, s]$ as the set of decision problems f which can be decided by an algorithm simultaneously using $O(t(n))$ time and $O(s(n))$ space.

For things like logarithmic space bounds to make sense, we need to be precise about the model we're using — we use the model described in Subsubsection 1.2.1, where the input x is given to us on a read-only tape and our algorithm gets to access some read-write storage, and only the storage counts for the space bound. (For example, logarithmic space bounds let us store a few pointers or indices into the read-only input.)

Example 3.2

We have $\mathsf{TS}[n, 1] = \mathsf{REG}$ (where REG is the set of regular languages — i.e., languages which can be decided by a finite automaton).

Proof. If we have a constant amount of space, then we might as well put it into the finite control (since there's only a constant number of possibilities for what it could store). Then we've got a machine that just has a finite control and moves back and forth on a read-only tape. This is essentially a two-way finite automaton, and two-way finite automata are equivalent to one-way ones. □

One way to think about the question of LOGSPACE vs. NP is that it's actually equivalent to the following conjecture. (The equivalence is a theorem we won't prove; it's not obvious.)

Conjecture 3.3 — For all k, we have $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^k, \log n]$.

So if we want to talk about partial progress on NP vs. LOGSPACE, maybe instead of trying to prove this for *all* constants k, we can try proving it for the largest k we possibly can.

Question 3.4. For what k can we prove $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^k, \log n]$?

Unfortunately, the best-known k is still less than 2.

Theorem 3.5 (Williams 2007)

For all $c < 2\cos\frac{\pi}{7} \approx 1.801$, we have $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^c, \log n]$.

One nice thing about this statement is that it doesn't depend on the machine model — it's really exploiting something fundamental about nondeterminism (and the fixed-polynomial runtime with a small space bound).

Today we're going to discuss how we prove theorems like this. We're not going to prove the whole theorem, but we'll prove a somewhat weaker bound that illustrates the main ideas.

Remark 3.6. There's a sense in which in order to get a better bound, you have to do something beyond the techniques in this lecture — there's a no-go result that if you use only the techniques in this lecture, then the best bound you can get is $2\cos\frac{\pi}{7}$. So in some sense, this bound is optimal for the current techniques.

What does this have to do with SAT? It turns out that there are proofs of the Cook–Levin theorem with nice corollaries like the following.

Theorem 3.7

If $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^c, \log n]$ (for some c), then $\mathsf{SAT} \notin \mathsf{TS}[n^c/(\log n)^2, \log n]$.

Essentially, the idea behind this is that there's a really souped-up version of the Cook–Levin reduction where you take a nondeterministic computation that runs in time $t(n)$, and represent it as a 3CNF instance of size $t(n) \cdot \mathsf{polylog}(t(n))$ — and moreover, you can compute different bits of this 3CNF on the fly as you need them, in a very space-efficient way. (So if you could solve SAT in low time and space, then you could solve any problem in $\mathsf{NTIME}[n]$ as well, with a $\mathsf{polylog}(n)$ loss.)

§3.2 The proof ideas

In the rest of the class, we'll prove Theorem 3.5 (with a different bound on c). The main idea in the proof is something called *alternation-trading*. At a high level, we're going to give a proof by contradiction. We start by assuming that

$$\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^c, \log n], \tag{$*$}$$

which means there's some sort of 'super SAT algorithm' — an algorithm for SAT that runs in very low time and space. And we can use ($*$) to get 'super-duper PH algorithms' (though with some loss in the runtime, since the proof of Theorem 2.44 — that $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{P} = \mathsf{PH}$ — involves applying the assumption $\mathsf{P} = \mathsf{NP}$ multiple times).

And then stuff happens, and eventually we profit (i.e., get a contradiction). More specifically, we're going to contradict the nondeterministic time hierarchy theorem (Theorem 2.9) — we'll show that we can take any problem in $\mathsf{NTIME}[n^2]$ and simulate it in nondeterministic time faster than n^2, which will contradict the time hierarchy theorem.

§3.2.1 Some definitions

First, we're going to define more specific time-bounded classes for Σ_2. (We've defined classes for deterministic time and nondeterministic time, and we can do this for Σ_2 too.)

Definition 3.8. We define $\Sigma_2\mathsf{TIME}[t(n)]$ as the set of decision problems f for which there exists a *linear*-time algorithm $\mathcal{A}$ such that for all x, letting $n = |x|$, we have

$$f(x) = (\exists\, y \mid |y| = O(t(n)))(\forall\, z \mid |z| = O(t(n)))[\mathcal{A}(x, y, z) \text{ accepts}].$$

(In words, this says that $f(x) = 1$ if and only if there exists some $|y| = O(t(n))$ such that for all $|z| = O(t(n))$, we have that $\mathcal{A}(x, y, z)$ accepts. Note that the runtime of $\mathcal{A}$ is linear in its input length, which consists of x, y, and z — so its runtime is in particular $O(t(n))$, as long as $t(n) \geq n$.)

When we define NTIME we omit z and just have one existential quantifier; similarly, to define $\mathsf{coNTIME}$ we'd omit y and just have one universal quantifier.

Then we have $\Sigma_2\mathsf{P} = \bigcup_k \Sigma_2\mathsf{TIME}[n^k]$ (this is the same as our definition from last class).

We're going to do some sort of algorithm design using Σ_2 (and more generally, you can do the same with Σ_3, Σ_4, and so on) — we're going to design an algorithm that does a small amount of existential and universal guessing to eventually get some answer. We'll in particular see an interesting example of space-bounded computation in Σ_2 time, which will be critical to the proof.

§3.2.2 Proof outline

We'll first state the steps of the proof at a high level; then in the rest of the class we'll fill in the details.

We're assuming $(*)$ — that $\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^c, \log n]$. And our goal is to contradict the nondeterministic time hierarchy theorem — we're going to start with $\mathsf{NTIME}[n^2]$ and get a sequence of inclusions that eventually ends up giving $\mathsf{NTIME}[n^{2-\varepsilon}]$.

(1) First, we're assuming that $\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^c, \log n]$, and from this we can get that $\mathsf{NTIME}[n^2] \subseteq \mathsf{TS}[n^{2c}, \log n]$. We saw how to do things like this on the problem set — if we've got a containment of two classes with small time bounds, we can lift to a containment with larger time bounds by *padding*. (This is fairly standard, so we won't go over it.)

(2) Next, we show that $\mathsf{TS}[n^{2c}, \log n] \subseteq \Sigma_2\mathsf{TIME}[n^c \log n]$. Essentially, this states that if we're allowed to guess and universally verify, then we can speed up our runtime by (approximately) a square root. For that reason, it's called the *speedup* step — we start off with some space-bounded computation, and we can substantially speed it up by adding in two quantifiers. (You can add in more quantifiers and get a faster speedup as well, but we'll just use two for simplicity.) With this step we're going outside the bounds of what we might consider a 'reasonable' model of computation, but we'll eventually end up back with nondeterministic time.

    This step is actually unconditionally true (i.e., it doesn't involve the assumption $(*)$).

(3) Now we have two quantifiers, and we need to remove one of them to get back to NTIME — we'll show that $\Sigma_2\mathsf{TIME}[n^c \log n] \subseteq \mathsf{NTIME}[n^{c^2} \log^c n]$. This is called the *slowdown* step, because we're going back to NTIME at the cost of slowing our algorithm down. (This step does involve the assumption $(*)$.)

If this all works out, then we get a contradiction when $c^2 < 2$ (because we'll have shown that $\mathsf{NTIME}[n^2] \subseteq \mathsf{NTIME}[n^{c^2} \log^c n]$, and if $c^2 < 2$ then we've shrunk the exponent, so this contradicts the nondeterministic time hierarchy theorem). This gives the following theorem.

Theorem 3.9 (LV 1999)
We have $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^{\sqrt{2}-\varepsilon}, \log n]$ for all $\varepsilon > 0$.

So this gives a $n^{1.4}$ lower bound. To improve this bound, there's a whole bunch of other things involving more complicated sequences of speedups and slowdowns (or deriving intermediate inclusions and applying those in such sequences). This is how you eventually get the bound of $2 \cos \frac{\pi}{7}$, and we've shown that this is the best possible bound we can get with such techniques.

§3.3 Step (3) — the slowdown

We'll start with step (3) — here we assume $(*)$, i.e., that we have a fast low-space simulation of nondeterministic time, and we want to use this to remove quantifiers from our Σ_2 time algorithm.

Lemma 3.10

For all $a \geq 1$, the assumption $(*)$ (that $\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^c, \log n]$) implies $\Sigma_2\mathsf{TIME}[n^a] \subseteq \mathsf{NTIME}[n^{ac}]$.

(We'll eventually take a to be c, but this in fact holds for all a.)

The idea is that Σ_2 involves two quantifiers, and using the assumption that NTIME has a fast algorithm, we can remove the universal one (this has the cost of blowing up the runtime a bit, which is why we get the slowdown). The proof is actually very similar to our proof that $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{P} = \mathsf{PH}$; we just have to be more careful about tracking the time bounds.

Proof. Suppose that $f \in \Sigma_2\mathsf{TIME}[n^a]$ — this means there is some linear-time algorithm $\mathcal{M}$ such that $f(x) = 1$ if and only if

$$(\exists\, y \mid |y| \leq |x|^a)(\forall\, z \mid |z| \leq |x|^a)[\mathcal{M}(x, y, z) \text{ accepts}].$$

And the idea is that we'll isolate the part $(\forall\, z \mid |z| \leq |x|^a)[\mathcal{M}(x, y, z) \text{ accepts}]$ — we define $g(x, y)$ as the function which is 1 if and only if $|y| = |x|^a$ and for all $|z| = |x|^a$ we have that $\mathcal{M}(x, y, z)$ accepts. (We can imagine taking some weird Σ_2 machine that does existential and then universal guessing; and we cut it off after its existential guess and put the $\forall$ into g.)

Then we have $g \in \mathsf{coNTIME}[n]$ (this is because we have a conondeterministic description of g, where the check runs in time $O(|x|^a)$ — and we only run this check when $|y| = |x|^a$, so since y is part of the input to g this means g runs in linear time).

And $(*)$ implies $\mathsf{coNTIME}[n] \subseteq \mathsf{TS}[n^c, \log n]$ as well (since if we've got a simulation for nondeterministic problems in a deterministic class, we can get one for their complements by just flipping the answer of the deterministic machine). So in particular, this means $g \in \mathsf{TS}[n^c, \log n]$.

And the rest of the proof is just a matter of interpreting what that means — let $\mathcal{A}$ be an algorithm for g which shows it's in $\mathsf{TS}[n^c, \log n]$. Then we get a new nondeterministic algorithm for f, as follows — on input x, we guess y of length $|x|^a$, and then run $\mathcal{A}(x, y)$ and output its answer.

This is a nondeterministic algorithm, and running $\mathcal{A}(x, y)$ takes time $O((|x| + |y|)^c) = O(|x|^{ac})$ (since $\mathcal{A}$ runs in time n^c in its input length, and $|y| = |x|^a$). So we've got a nondeterministic algorithm running in time n^{ac} on inputs of length n, which means $f \in \mathsf{NTIME}[n^{ac}]$. □

The idea behind this proof is sort of that if we have a fast SAT algorithm, then if we have some quantified statement

$$(\exists\, y_1)(\forall\, y_2) \cdots (Q_k\, y_k)[\cdots]$$

(where the predicate is some deterministic thing), then every time we see a quantifier next to some deterministic predicate, we can remove it at the cost of blowing up the runtime of the deterministic algorithm a bit (coming from the runtime of this SAT algorithm). More generally, you can imagine using this same argument to go from $\Sigma_k\mathsf{TIME}$ to $\Sigma_{k-1}\mathsf{TIME}$ (blowing up the exponent by a factor of c).

§3.4 Step (2) — the speedup

Now we'll get to the more interesting part — the speedup step, which is actually unconditionally true.

Lemma 3.11

For any $t(n) \geq n$ and $s(n) \geq \log n$, we have $\mathsf{TS}[t, s] \subseteq \Sigma_2\mathsf{TIME}[\sqrt{ts}]$.

In our application, s will be $\log n$, which is really small. But even if s were e.g. $(\log n)^{100}$, the extra factor we get from s would be negligible here and all the arguments would go through. So Theorem 3.9 is actually true for any $s(n) = n^{o(1)}$; we just used $s(n) = \log n$ for concreteness.

To prove Lemma 3.11, we want to take some (deterministic) $\mathcal{M}$ that runs in time t and space s, and simulate it such that we existentially guess something, universally guess something, and run in time $\sqrt{ts}$.

And the main idea is to think of $\mathsf{TS}[t, s]$ computation in terms of a table (sometimes called a *tableau*). Imagine a table where we're writing down the configurations of what's happening in our machine $\mathcal{M}$ at every step — so our first row is the initial configuration c_0, then the next row is c_1, and so on, up to c_t.

— c_0 —
— c_1 —
— c_2 —
$\vdots$
— c_t —

What's in a configuration? We want to store all the information we need to start up the machine at a given point in time, but we *don't* want to store the input (all our simulations have access to the input, which is sitting on their input tape). So we'll have $O(s(n))$ bits encoding what the read-write storage looks like, $O(1)$ bits to store a program counter or state (something saying where you are in the program — e.g., if we model our finite control as a bunch of states with transitions, then we store the current state), and $O(\log n)$ bits to store which index of the input the machine is currently reading. We assumed $s(n) \geq \log n$, so in total, this takes $O(s(n))$ bits.

And the idea is that we're going to guess configurations spaced out by roughly $\sqrt{t}$ steps each — i.e., we'll guess c_0, c_k, c_{2k}, and so on (for some k we'll specify later — for now, think of k as roughly $\sqrt{t}$). We then want to check that these configurations really are correct — equivalently, that for every i, if we start $\mathcal{M}$ in the configuration c_{ik} and simulate it for k steps, then we end up in $c_{(i+1)k}$. So we'll use our universal quantifier to pick which i to look at, and the deterministic part of our algorithm will be the simulation of $\mathcal{M}$ from c_{ik}.

More precisely, here's how the algorithm works.

Algorithm 3.12 — On input x (letting k be some quantity depending on t and s we'll set later):

(1) Existentially guess configurations c_0, c_k, c_{2k}, $\ldots$, c_t (i.e., we guess the sublist of configurations in the table spaced out by roughly k steps).

(2) Universally guess an index i (with $0 \leq i \leq \frac{t}{k}$).

(3) Simulate $\mathcal{M}$ starting from c_{ik} for k steps, and check that it reaches c_{i+1}. *Reject* if not.

(4) Check that c_0 is the starting configuration and c_t is an accepting configuration; *reject* if not.

(5) If we haven't rejected (meaning that $\mathcal{M}$ starting in c_{ik} really does go to $c_{(i+1)k}$ in k moves and the sequence starts and ends correctly), then we *accept*.

(Importantly, this works because $\mathcal{M}$ is *deterministic*, so there's only one valid computation path.)

Now we need to set the parameter k appropriately and analyze the runtime of this algorithm. First, we're guessing $\ell \approx \frac{t}{k}$ configurations, which means we need to guess $O(\ell s)$ bits (for the existential quantifier). Next, the universal quantifier is actually really short — we're choosing an index from 0 to ℓ, which only takes $O(\log \ell)$ bits. Finally, we need to deterministically simulate $\mathcal{M}$ for k steps. The amount of time this takes could depend on the machine model (in some weak models, it's possible that it might take time $O(s)$ to simulate a single step, in which case we'd get a runtime of $O(ks)$), but in many models (e.g., random access or multitape models), we'll only need time $O(k)$. (The dependence on s isn't actually important, though, because s is only $\log n$ — so even if we did collect a few extra factors of s in the runtime, they wouldn't hurt the proof.)

Finally, we can set $k = \sqrt{ts}$ (so ℓs is also $\sqrt{ts}$). This gives us a runtime of $O(\sqrt{ts})$, as desired.

We can also set up the quantifiers slightly differently to prove statements like the following.

Corollary 3.13

For $t(n) \geq n$ and $s(n) \geq \log n$, we have $\mathsf{TS}[t, s] \subseteq \Pi_2\mathsf{TIME}[\sqrt{ts}]$.

Here we have a $\forall\exists$ instead of $\exists\forall$. So the idea is that we'll try all lists of configurations c_0, c_k, ... from the starting configuration to a *rejecting* configuration and show that they're *incorrect* (i.e., there exists some step at which the simulation is wrong).

§3.5 Ideas behind the improvements

We'll now discuss the ideas that go into improving this argument to get the better bound of $2\cos\frac{\pi}{7}$. This is a weird-looking number, but where it comes from is that it's the positive root of a cubic that you get from two double-recursions of speedups and slowdowns.

One thing we can try to do is to do a bunch of speedups and slowdowns right after each other — we can use the fast SAT algorithm that comes out of one step of the proof to (conditionally) improve the speedup step, by very carefully changing how we guess the configurations (in the Σ_2 computation).

But here we're only using two quantifiers. It turns out that you can actually get more mileage out of using more quantifiers — i.e., using Σ_3, Σ_4, and so on. (There's also a Σ_k-time hierarchy theorem, which we can try to contradict for any k.) This is where the double-recursion comes from — we have one recurrence that creates a conditional speedup, and another that removes quantifiers.

First, there's a naive way to use Σ_k for larger values of k — in our proof of the speedup step (2), we reduced the problem of checking whether a configuration c_0 leads to another configuration c_t into a bunch of smaller problems of checking whether c_{ik} leads to $c_{(i+1)k}$, using an $\exists\forall$. The most naive thing to do is to use another $\exists\forall$ to check that c_{ik} leads to $c_{(i+1)k}$ (in the same way). This would give us a Σ_4 simulation and a cube root.

But in fact, we can actually do better. As discussed in Corollary 3.13, there are two ways to do the simulation. On one hand, we can check that there *is* a way to get from the start state to an *accept* state (by existentially guessing a sequence leading to an accept state and then universally verifying that it works). But on the other hand, as with Corollary 3.13, we can check that there is *no* way to get from the start state to a *reject* state (by going over all sequences leading to a reject state and then existentially finding a bug in them — if all lists ending in reject have a bug, then the machine must accept).

We can use this idea to check whether c_{ik} leads to $c_{(i+1)k}$ with a $\forall\exists$ instead — then this gives us an $\exists\forall\forall\exists$ simulation, letting us say that

$$\mathsf{TS}[t, s] \subseteq \Sigma_3[t^{1/3}s].$$

(The precise dependence on s doesn't matter, as s is tiny — specifically $\mathsf{polylog}(n)$.) And more generally, doing this for more levels (alternating between $\exists\forall$ and $\forall\exists$ simulations) gives

$$\mathsf{TS}[t, s] \subseteq \Sigma_k[t^{1/k}s].$$

So adding in more quantifiers does allow us to get better and better speedups.

§3.5.1 Proof sketch of a better bound

Finally, we'll give a brief sketch of how to get a more sophisticated lower bound.

We've already showed that $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^{\sqrt{2}-\varepsilon}, \log n]$. Now we want to suppose $\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^c, \log n]$ for some $c \geq \sqrt{2}$, and we want to still get some contradiction (for some new range of c).

This time, instead of starting with $\mathsf{NTIME}[n^2]$, we're going to start with $\Pi_2\mathsf{TIME}[n]$. First, similarly to the slowdown lemma, we can simulate $\Pi_2\mathsf{TIME}$ with $\mathsf{coNTIME}$, using the assumption to remove a quantifier ($\Pi_2\mathsf{TIME}$ has an $\forall\exists$ and $\mathsf{coNTIME}$ has a $\forall$, so we're just removing the $\exists$) — this gives

$$\Pi_2\mathsf{TIME}[n] \subseteq \mathsf{coNTIME}[n^c].$$

And then we can do a slowdown again to get

$$\Pi_2\mathsf{TIME}[n] \subseteq \mathsf{coNTIME}[n^c] \subseteq \mathsf{TS}[n^{c^2}, \log n]$$

(this time we're using the assumption directly, so we get the space bound as well). And we have our simulation for TS from Lemma 3.11, so we can again get a square-root speedup and get

$$\Pi_2\mathsf{TIME}[n] \subseteq \mathsf{coNTIME}[n^c] \subseteq \mathsf{TS}[n^{c^2}, \log n] \subseteq \Sigma_2\mathsf{TIME}[n^{c^2/2}].$$

It might not look like we've done very much — this still only gives a contradiction if $c^2 < 2$. But we've actually taken something that's a $\forall\exists$ and made it an $\exists\forall$ (by going from $\Pi_2\mathsf{TIME}$ to $\Sigma_2\mathsf{TIME}$), and this gives us a *new* way to get a slowdown for Σ_3 — Σ_3 has quantifiers $\exists\forall\exists$, and if we can turn the $\forall\exists$ into an $\exists\forall$, then we can replace it with Σ_2. So this gives us

$$\Sigma_3\mathsf{TIME}[n] \subseteq \Sigma_2\mathsf{TIME}[n^{c^2/2}].$$

In other words, we now have a new way of removing a quantifier (when we have at least three) that costs us $\frac{1}{2}c^2$ instead of c — and this is better as long as $c < 2$.

Now we'll remove both quantifiers from the right-hand side (using the original slowdown) to get

$$\Sigma_3\mathsf{TIME}[n] \subseteq \Sigma_2\mathsf{TIME}[n^{c^2/2}] \subseteq \mathsf{TS}[n^{c^4/2}, \mathsf{polylog}(n)].$$

And we saw earlier that we can get a cube-root speedup by going from TS to $\Sigma_3\mathsf{TIME}$, so we get that

$$\mathsf{TS}[n^{c^4/2}, \mathsf{polylog}(n)] \subseteq \Sigma_3\mathsf{TIME}[n^{c^4/6} \cdot \mathsf{polylog}(n)].$$

And this is a contradiction for $c < 6^{1/4} \approx 1.565$ (here we're contradicting a time hierarchy theorem for Σ_3).

So this gets us a better bound; and you can imagine doing hairier and hairier things to improve further.

§4 Oracles and relativization

Today we'll talk about a barrier to proving lower bounds, namely oracles.

We've discussed oracle algorithms earlier (in Subsubsection 2.4.1) — when we have an algorithm with oracle A, we have a read-only input and read-write storage (as usual), as well as an oracle tape. We can write strings y on the oracle tape, and the oracle will tell us whether y is in A (or whether $A(y) = 1$, if we view A as a function rather than a language); we assume oracle calls take a single step.

§4.1 Some examples of relativization

It turns out that lots of theorems in complexity and computability still hold if we add an oracle to our computation model. One canonical theorem is the universal simulation theorem (which is true for any sufficiently expressive model of computation).

Theorem 1.27 (Universal simulation theorem)

There is an algorithm $\mathcal{U}$ such that for all algorithms $\mathcal{M}$, inputs x, and time bounds $t \geq 0$, we have that $\mathcal{U}(\langle \mathcal{M} \rangle, x, 1^t) = 1$ if and only if $\mathcal{M}(x)$ accepts within t steps, and $\mathcal{U}$ runs in polynomial time.

This is a theorem that *relativizes*, meaning that it holds with respect to every oracle.

Theorem 4.1 (Oracle universal simulation theorem)

For all oracles A, there is an algorithm $\mathcal{U}^A$ such that for all oracle algorithms $\mathcal{M}^A$, inputs x, and integers $t \geq 0$, we have that $\mathcal{U}^A(\langle \mathcal{M}^A \rangle, x, 1^t) = 1$ if and only if $\mathcal{M}^A(x)$ accepts within t steps, and $\mathcal{U}^A$ runs in polynomial time.

The main point is that if we think about what's happening in a universal simulation, we're given the input and we're trying to keep track of what $\mathcal{M}$ looks like at every step of the computation. We simulate it step by step — when $\mathcal{M}$ makes a move that changes its storage or state in a certain way, then the simulation makes the same move. And if we add in oracles into the mix — if we give $\mathcal{M}$ some oracle A — then as long as $\mathcal{U}$ has access to that same oracle, it can *still* carry out the simulation step by step.

So the universal simulation theorem relativizes, meaning that it holds for any oracle. This is very powerful, since if you prove a relativizing theorem, then the theorem holds if you change your computation model to have any oracle, which means you get uncountably many corrollaries for free. (This is actually sometimes used in proofs.)

As another example, consider the halting problem, defined as

$$\mathsf{Halt} = \{\langle \mathcal{M}, x \rangle \mid \mathcal{M} \text{ halts on } x\}.$$

We've seen that no Turing machine can decide Halt. And this result also relativizes — if we define

$$\mathsf{Halt}^A = \{\langle \mathcal{M}^A, x \rangle \mid \mathcal{M}^A \text{ halts on } x\},$$

then no Turing machine with oracle A can decide Halt^A.

Remark 4.2. As a subtle point, how do we write down a description of $\mathcal{M}^A$? The point is that we don't actually have to write down what A is in our description of $\mathcal{M}^A$ — we just have to write the instruction that the machine queries A, and what it does upon receiving an answer of 0 and 1. (This is in some sense built into the definition of an oracle Turing machine — the machine transitions into q_{yes} or q_{no} based on what the oracle does, but we don't need to know what exactly the oracle does in order to specify the machine's transitions.)

We've also seen other results that relativize — for example, the time hierarchy theorem. First, we can define time classes for oracle machines in the same way as usual.

Definition 4.3. For an oracle A, we define $\mathsf{TIME}^A[t(n)]$ as the set of decision problems we can solve in time $O(t(n))$ with oracle A.

The time hierarchy theorem (Theorem 1.26) says that there exist functions $f \in \mathsf{TIME}[t^2(n)] \setminus \mathsf{TIME}[t(n)]$ (as long as t is time-constructible). (Here we're just taking $c = 2$ for concreteness.) This also relativizes.

Theorem 4.4 (Oracle time hierarchy theorem)
For all oracles A, there is a function $f \in \mathsf{TIME}^A[t^2(n)] \setminus \mathsf{TIME}^A[t(n)]$.

Similarly, the nondeterministic time hierarchy theorem also relativizes. Simple containments like $\mathsf{P} \subseteq \mathsf{NP}$ also relativize. And so does the fact that $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{P} = \mathsf{PH}$ (i.e., if $\mathsf{P}^A = \mathsf{NP}^A$ then $\mathsf{P}^A = \mathsf{PH}^A$ — this is on the problem set).

§4.2 P vs. NP and the relativization barrier

Lots of the theorems we've seen so far relativize. But it turns out that for the question of P vs. NP, there are oracles that make them equal *and* oracles that make them different! This means P vs. NP has to be approached in a diffefrent way. Many of the methods we've seen so far, like universal simulation and diagonalization and composing algorithms with themselves, don't depend on the model of computation, in the sense that if we throw in an oracle (giving every algorithm the same oracle), everything in the proof still works. So these methods can't be used to resolve P vs. NP.

§4.2.1 An oracle with $\mathsf{P}^A = \mathsf{NP}^A$

First, we'll exhibit an oracle relative to which P and NP become equal.

Theorem 4.5
There exists an oracle A such that $\mathsf{P}^A = \mathsf{NP}^A$.

This already tells us that if we're going to *separate* P from NP (i.e., to prove $\mathsf{P} \neq \mathsf{NP}$), then we'd need to use a method that doesn't relativize (e.g., we need to do something different from all the proofs we saw in Subsection 4.1), since if we proved $\mathsf{P} \neq \mathsf{NP}$ with a relativizing proof, then this would mean $\mathsf{P}^A \neq \mathsf{NP}^A$ for all A, which is the negation of this statement.

Here's the particular oracle we're going to use.

Definition 4.6
We define $\mathsf{ExpAcc} = \{\langle \mathcal{M}, x, 0^t \rangle \mid \mathcal{M} \text{ accepts } x \text{ in at most } 2^t \text{ steps}, |x| \leq t, |\mathcal{M}| \leq t\}$.

Fact 4.7 — We have $\mathsf{ExpAcc} \in \mathsf{TIME}[2^{O(n)}]$.

Proof. We can simply solve ExpAcc with a universal simulator — we're simulating $\mathcal{M}$ on x for $2^t \leq 2^n$ steps, and the simulator has polynomial overhead, so it will take time $2^{O(n)}$. □

Then we'll prove the following more specific statement (in place of Theorem 4.5).

Theorem 4.8

We have $\mathsf{EXP} = \mathsf{P}^{\mathsf{ExpAcc}} = \mathsf{NP}^{\mathsf{ExpAcc}}$.

Recall that $\mathsf{EXP} = \bigcup_k \mathsf{TIME}[2^{n^k}]$.

Proof. We'll show the chain of containments

$$\mathsf{EXP} \subseteq \mathsf{P}^{\mathsf{ExpAcc}} \subseteq \mathsf{NP}^{\mathsf{ExpAcc}} \subseteq \mathsf{EXP}.$$

First, to show $\mathsf{EXP} \subseteq \mathsf{P}^{\mathsf{ExpAcc}}$, suppose that we have an algorithm $\mathcal{A} \in \mathsf{EXP}$ with runtime 2^{n^k}. Then we can define an algorithm $\mathcal{B}^{\mathsf{ExpAcc}}$ which, on input x of length n, forms the query $(\mathcal{A}, x, 0^{n^k})$ and asks whether it's in ExpAcc. This query has polynomial length (as $\mathcal{A}$ is fixed), and writing it down takes polynomial time (and we can ask the query and get an answer in one step). And by definition, the answer will be yes if $\mathcal{A}$ accepts x and no if it doesn't (since $\mathcal{A}$ only runs for 2^{n^k} steps).

Next, of course we have $\mathsf{P}^{\mathsf{ExpAcc}} \subseteq \mathsf{NP}^{\mathsf{ExpAcc}}$ (we have $\mathsf{P}^A \subseteq \mathsf{NP}^A$ for any oracle A, since anything that can be done by a P machine can also be done by an NP machine).

Finally, we need to show $\mathsf{NP}^{\mathsf{ExpAcc}} \subseteq \mathsf{EXP}$. To see this, imagine we're given a nondeterministic machine $\mathcal{N}^{\mathsf{ExpAcc}}$ running in time n^k; we want to simulate it with an exponential time algorithm $\mathcal{M}$ with no oracle.

The idea is to just brute force — our algorithm $\mathcal{M}$, on input x, simply tries all $2^{O(n^k)}$ possible paths of $\mathcal{N}^{\mathsf{ExpAcc}}$ on x, and it'll compute ExpAcc itself to answer each of the queries it sees on the path. The queries can only be of length at most n^k, so by Fact 4.7 we can answer each query in $2^{O(n^k)}$ time. And finally we accept if over the whole brute force, some path of $\mathcal{N}^{\mathsf{ExpAcc}}$ leads to acceptance. □

§4.2.2 An oracle with $\mathsf{P}^B \neq \mathsf{NP}^B$

So far, Theorem 4.5 just shows that relativizing techniques can't prove $\mathsf{P} \neq \mathsf{NP}$, since it's not the case that for every oracle A we have $\mathsf{P}^A \neq \mathsf{NP}^A$. But what if $\mathsf{P} = \mathsf{NP}$? It turns out that relativizing techniques can't prove *that* either, due to the following theorem.

Theorem 4.9

There exists an oracle B such that $\mathsf{P}^B \neq \mathsf{NP}^B$.

(This oracle will be more complicated than the one for Theorem 4.5.)

Together, Theorems 4.5 and 4.9 are key theorems in what's known as the *relativization barrier*. The idea is that there's many separations that we'd *like* to prove, but for most of those separations, there are oracles that make the two classes equal (even if they ordinarily look completely different). So we need techniques that really use the computational model in some way (so that the technique doesn't continue to work if we add in an arbitrary oracle). When we throw in an oracle, we can think of it as black-boxing a portion of the computation (where there's essentially a black box telling us the answers to the oracle queries). So we need proofs that won't work if we black-box part of the computation (e.g., you might have to think step by step about what's happening in the computation).

Proof of Theorem 4.9. We'll phrase things in terms of languages (i.e., we'll describe B as a subset of $\{0,1\}^*$ instead of functions $\{0,1\}^* \to \{0,1\}$).

The basic idea is that for any oracle B, we can define the problem

$$L_B = \{1^n \mid B \cap \{0,1\}^n \neq \emptyset\}.$$

Intuitively, we can think of B as some kind of 'haystack,' and if we restrict to strings of length n, there's 2^n things in this haystack. And we can think of **YES** instances to B as 'needles.' And L_B is essentially the problem where we're given the string of n 1's and we want to figure out whether the nth haystack is empty.

First note that for *any* oracle B, we have $L_B \in \mathsf{NP}^B$ — given 1^n, we can simply guess the string y that's the needle in the haystack. More explicitly, we guess $y \in \{0,1\}^n$ and ask the oracle whether $y \in B$; if the answer is yes then we *accept*, and otherwise we *reject*.

Now we're going to define B to ensure that $L_B \notin \mathsf{P}$. The very high-level intuition is actually quite simple — the idea is that a polynomial time algorithm can only query $\mathsf{poly}(n)$ strings of length n, while the haystack has 2^n strings. So if n is large enough that 2^n exceeds the number of queries, then we'll be able to hide the needle (by placing it somewhere the algorithm doesn't query).

So for each polynomial-time oracle algorithm $\mathcal{M}^B$, we'll find some input length n such that $\mathcal{M}^B(1^n)$ is wrong — if $\mathcal{M}^B(1^n)$ accepts, then we want to set up B such that $B \cap \{0,1\}^n = \emptyset$ (i.e., such that the haystack is really empty), which means 1^n actually isn't in L_B. Meanwhile, if $\mathcal{M}^B(1^n)$ rejects, then we're going to stick in a needle that wasn't queried — i.e., we're going to take some x of length n that $\mathcal{M}^B$ didn't query and stick it into B, which will mean that 1^n actually *is* in L_B.

So that's the high-level idea, but it'll take a bit to actually set the oracle up. First, we need a way of enumerating over polynomial-time oracle algorithms $\mathcal{M}^B$ (or really $\mathcal{M}^\mathcal{O}$ for a general oracle $\mathcal{O}$ — when we describe an oracle Turing machine, the description just involves the appropriate actions the machine takes if the oracle answers yes and no; it doesn't say anything about what the oracle actually does).

To do this, we take a list of *all* oracle Turing machines $\mathcal{M}_1^\mathcal{O}, \mathcal{M}_2^\mathcal{O}, \ldots$. Then for each i, we define $\mathcal{A}_i^\mathcal{O}$ to be $\mathcal{M}_i^\mathcal{O}$ with an 'alarm clock' that rejects after $n^i + i$ steps — so

$$\mathcal{A}_i^\mathcal{O}(x) = \begin{cases} \mathcal{M}_i^\mathcal{O}(x) & \text{if } \mathcal{M}_i^\mathcal{O}(x) \text{ halts within } in^i \text{ steps} \\ \textit{reject} & \text{otherwise.} \end{cases}$$

We'll use $L(\mathcal{A}_i^\mathcal{O})$ to denote the language decided by $\mathcal{A}_i^\mathcal{O}$.

Claim 4.10 — We have $\mathsf{P}^\mathcal{O} = \{L(\mathcal{A}_i^\mathcal{O}) \mid i \in \mathbb{N}\}$.

The point is that this gives us a way of essentially listing all polynomial-time algorithms so that we can go through them one by one.

Proof. First, it's clear that $L(\mathcal{A}_i^\mathcal{O}) \in \mathsf{P}^\mathcal{O}$ for all i, since $\mathcal{A}_i^\mathcal{O}$ runs for at most $n^i + i = \mathsf{poly}(n)$ steps. (Technically, an alarm clock isn't part of our usual computational model, but we can easily implement one with polynomial overhead by keeping a counter for the number of steps of $\mathcal{M}_i^\mathcal{O}$ we've taken.)

For the other direction, consider any $L \in \mathsf{P}^\mathcal{O}$. Then there exists some $\mathcal{M}^\mathcal{O}$ that decides L in at most $n^c + c$ steps for some c. And there are infinitely many machines equivalent to $\mathcal{M}^\mathcal{O}$ (because we can just add in useless states, or dummy code that never gets run), so we can find some such $\mathcal{M}_i^\mathcal{O}$ with $i > c$. Then $\mathcal{A}_i^\mathcal{O}$ is equivalent to $\mathcal{M}_i^\mathcal{O}$ (since $\mathcal{M}_i^\mathcal{O}$ runs within the time bound $n^i + i$, so cutting it off at that time bound has no effect), which means $L = L(\mathcal{A}_i^\mathcal{O})$. □

This gives us a way of listing all the algorithms we need to deal with, and now we're going to construct the oracle B. We'll do this in stages — where we have one stage for each $\mathcal{A}_i^\mathcal{O}$, and the goal of the ith stage is to make sure that $L(\mathcal{A}_i^B) \neq L_B$. (If we can do this, then we'll have guaranteed that $L_B \notin \mathsf{P}^B$.)

The ith stage works as follows.

(1) First, we choose an input length n_i for which we're going to make $L(\mathcal{A}_i^B)$ and L_B disagree on 1^{n_i}. To do so, we set n_i to be greater than the length of all the strings we've already set the values of (where

by the value of a string, we mean whether it's in B or not) — equivalently, larger than all the lengths of the queries asked in previous stages, as well as $n_1, \ldots, n_{i-1}$. This ensures that we haven't yet set the value of any string of length n_i. We also choose n_i large enough that $(n_i)^i + i < 2^{n_i}$ — so that the number of strings we have available in the haystack is more than the number of queries $\mathcal{A}_i^B$ could possibly ask (since it runs for at most $n_i^i + i$ steps).

(2) Now we simulate $\mathcal{A}_i^B$ on 1^{n_i} to figure out what its answer is. We have to be careful here — we haven't defined all of B yet, so how can we simulate $\mathcal{A}_i^B$?

When we're doing this simulation, we first have to be consistent with all the previous stages — so if $\mathcal{A}_i^B$ ever queries a string y whose value we set on a previous stage, then we give the answer consistent with tihs value. (For example, if $\mathcal{A}_i^B$ asks a query of length 2 and we already decided what to do with all 2-bit strings in the first stage, then our answer to this query is forced.)

Meanwhile, whenever $\mathcal{A}_i^B$ queries a string y whose value we *haven't* yet set, we answer no — i.e., we set y to *not* be in B. (So we're answering no to all queries where we have a choice.)

(3) Once we've done this simulation, we know what answer $\mathcal{A}_i^B$ gives on 1^{n_i}. If it accepts, then we want to make 1^{n_i} *not* be in L_B, so we want the haystack to be empty. And we can do this — so far, we haven't put any n_i-bit strings into B (we hadn't set the values of any n_i-bit strings in previous stages, and on this stage, so far we've only placed strings *not* in B). So we can simply set all the remaining n_i-bit strings to not be in B as well. This means the haystack is empty and $1^{n_i} \notin L_B$, so $\mathcal{A}_i^B$ gave the wrong answer (by accepting it).

(4) Meanwhile, if $\mathcal{A}_i^B$ rejects 1^{n_i}, then we want to make 1^{n_i} be in L_B, so we want the haystack to *not* be empty. And this is where we use the fact that there are more n_i-bit strings than queries $\mathcal{A}_i^B$ could have asked — $\mathcal{A}_i^B$ asked at most $(n_i)^i + i < 2^{n_i}$ queries, so there's some n_i-bit string that it *hasn't* queried, and we stick that string into B (we can do this because we haven't already set its value).

So to conclude, $\mathcal{A}_i^B$ accepts 1^{n_i} if and only if $B \cap \{0,1\}^{n_i} = \emptyset$ (by our construction of B), which is true if and only if $1^{n_i} \notin L_B$ (by the definition of L_B). So for each polynomial-time algorithm $\mathcal{A}_i^B$ we've found a bad input 1^{n_i}, which means no such algorithm can decide L_B. □

In fact, we can get something much stronger than $\mathsf{P}^B \neq \mathsf{NP}^B$ out of this proof — the only fact about polynomials we used was that $(n_i)^i + i < 2^{n_i}$. So we could consider algorithms running in time $2^n/\log n$ (for example) instead of polynomial time, and we could run the exact same argument — for sufficiently large n we can still say that $\mathcal{A}_i^B(1^n)$ doesn't query all strings of length n, which is all we need for the argument. (In some sense, the idea behind the needle-in-a-haystack argument is that we construct B so that you're *forced* to look in all places, i.e., to query all strings.) This gives the following statement.

Theorem 4.11

There exists an oracle B such that $\mathsf{NP}^B \not\subseteq \mathsf{TIME}^B[2^n/\log n]$.

The same holds if we replace $\log n$ with any unbounded function of n — the right-hand side just has to be $o(2^n)$. In fact, we just need the algorithm to make $o(2^n)$ *queries* for the argument to work — the oracle doesn't care how long $\mathcal{A}_i^B$ runs for, as long as it's finite.

§4.3 Some more examples of the relativization barrier

In fact, P vs. NP is just the tip of the iceberg — for nearly *every* pair of classes $\mathcal{C}$ and $\mathcal{D}$ for which $\mathcal{C}$ vs. $\mathcal{D}$ is an open problem, there exist oracles making the two classes equal as well as oracles making them different. Here are some particularly weird examples.

Theorem 4.12

There exists an oracle A such that $\mathsf{EXP}^{\mathsf{NP}^A} = \mathsf{BPP}^A$.

In contrast, we *do* know $\mathsf{EXP}^{\mathsf{NP}^B} \neq \mathsf{P}^{\mathsf{NP}^B}$ for all oracles B (because the time hierarchy theorem relativizes).

It's not known whether $\mathsf{EXP}^{\mathsf{NP}} = \mathsf{BPP}$, and perhaps this theorem is the reason why this problem is still open — since it means we can't separate them with e.g. a diagonalization argument. (Note that $\mathsf{EXP}^{\mathsf{NP}^A}$ is more powerful than $\mathsf{EXP}^{\mathsf{NP},A}$, where we have two oracles — e.g., a SAT oracle and an A oracle — and we can call both of them.)

Here's another really weird example.

Theorem 4.13

There exists an oracle A such that $\mathsf{NEXP}^A = \mathsf{P}^{\mathsf{NP}^A}$.

So even though on the left-hand side we get to ask *exponentially* long queries to A, we can still simulate it with the right-hand side (which can only ask polynomial-length queries).

On the other hand, we do know that $\mathsf{NEXP}^B \neq \mathsf{NP}^B$ for all oracles B (since the nondeterministic time hierarchy theorem relativizes).

Remark 4.14. There do exist non-relativizing theorems stating two classes are *equal* — for example, $\mathsf{IP} = \mathsf{PSPACE}$, but there exist oracles B with $\mathsf{IP}^B \neq \mathsf{PSPACE}^B$.

§4.4 Non-relativization of time-space lower bounds

It seems that relativization is in many ways capturing chokepoints in our knowledge of lower bounds — there's not too many lower bounds we know of that don't relativize. But there's at least one that we already know — the lower bound from last lecture.

Question 4.15. Last lecture, we proved $\mathsf{NTIME}[n] \not\subseteq \mathsf{TS}[n^{1.4}, \log n]$. Does this relativize?

This is a subtle question — in general, if you talk about space-bounded classes with respect to oracles, you have to be careful. But for a natural oracle model people have studied, the answer is actually no.

When we're discussing $\mathsf{TS}^A[n^c, \log n]$, we'll imagine that we have a read-only input tape and $O(\log n)$ storage, as usual, and we also have an oracle tape. And we'll say this oracle tape is *write-only* and does *not* count towards the space bound. (It'd be reasonable to use a model that does count the oracle tape towards the space bound, or one that doesn't; here we'll use one that doesn't count it.)

Remark 4.16. For each of these choices of model, on the problem set, we'll prove a statement that doesn't relativize under that model. Specifically, we'll prove on the problem set that $\mathsf{APSPACE} = \mathsf{EXP}$ and $\mathsf{AP} = \mathsf{PSPACE}$. If the oracle tape *does* count towards the space bound, then $\mathsf{APSPACE} = \mathsf{EXP}$ doesn't relativize — this is essentially because with $\mathsf{APSPACE}$ we can only ask polynomial-length queries, while with EXP we can ask exponential-length ones. Meanwhile, if the oracle tape does *not* count, then $\mathsf{AP} = \mathsf{PSPACE}$ doesn't relativize — with PSPACE we can run for exponential time and write down exponentially long queries (keeping a counter to remember where in the query we are).

Under this model, Theorem 3.9 does *not* relativize.

Theorem 4.17

There is an oracle A such that $\mathsf{NTIME}^A[n] \subseteq \mathsf{TS}[n^{1.1}, \log n]$.

Proof. The basic idea behind the oracle A is that with time $n^{1.1}$ and space $\log n$, there's something we can do that we can't do in time n even with nondeterminism — we can ask queries of length $n^{1.1}$. And so we're going to take each oracle machine in $\mathsf{NTIME}^A[n]$ and hide its accept-reject behavior in the oracle using strings slightly longer than that machine could query, exploiting the fact that in $\mathsf{TS}^A[n^{1.1}, \log n]$ we *can* query such strings.

As in the proof of Theorem 4.9, we can enumerate all nondeterministic oracle machines running in time n, which we'll call $\mathcal{M}_1^A$, $\mathcal{M}_2^A$, … (this enumeration doesn't depend on A).

Now we'll construct the oracle A in stages, where on the nth stage, we'll take care of n-bit inputs x. To do so, for each $i = 1, \ldots, n$ and each $x \in \{0,1\}^n$, we simulate $\mathcal{M}_i^A(x)$, where to answer the queries it asks, we're consistent with all previously set values and we answer no to all new queries. (Note that here $\mathcal{M}_i^A$ is nondeterministic, so we do a brute-force simulation — we go through all possible computation paths.)

Then if $\mathcal{M}_i^A$ accepts, we add a long string that the machine couldn't have queried encoding the fact that $\mathcal{M}_i(x)$ accepted x — specifically, we add the string $0^{n^{1.1}}10^i x$ to A. (This string is too long for any of the simulations we've run so far to have queried it.) If $\mathcal{M}_i^A$ rejects, then we set this string to *not* be in A.

This defines our oracle A, and with this oracle we can easily simulate any $\mathcal{M}_i^A$ in $\mathsf{TS}[n^{1.1}, \log n]$ — on an input x, we just write down the query $0^{n^{1.1}}10^i x$ on the oracle tape. (We can do this in space $\log n$ by keeping a counter on our storage tape to keep track of our position in the query on the oracle tape.) Then we ask this query, and the answer tells us whether $\mathcal{M}_i^A$ accepts x or not. $\square$

Question 4.18. Which part of the proof of Theorem 3.9 doesn't relativize?

Recall that in our proof, we showed that if we actually had $\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^{1.4}, \log n]$, then we could contradict the nondeterministic time hierarchy. We started off with $\mathsf{NTIME}[n^2]$; then this assumption means $\mathsf{NTIME}[n^2] \subseteq \mathsf{TS}[n^{2c}, \log n]$ (by padding — this was (1)). Then we performed the *speedup* step (2) where by using $\exists\forall$ quantifiers, we could speed up the $\mathsf{TS}[n^{2c}, \log n]$ computation by a square root to get that $\mathsf{TS}[n^{2c}, \log n] \subseteq \Sigma_2\mathsf{TIME}[n^c \cdot \mathsf{polylog}(n)]$. And finally, we used quantifier removal (again using the assumption $\mathsf{NTIME}[n] \subseteq \mathsf{TS}[n^{1.4}, \log n]$) to get that this is contained in $\mathsf{NTIME}[n^{c^2} \cdot \mathsf{polylog}(n)]$ (this was the *slowdown* step (3)), which is a contradiction to the nondeterministic time hierarchy theorem if $c^2 < 2$.

So which of these parts doesn't relativize? The answer is the speedup step. The speedup step (Lemma 3.11) worked by guessing what the computation looked like at n^c different (equally spaced out) points, and then universally verifying that for each of these points, we really do go from one point to the next in the appropriate number of steps. But when we throw in oracles, to get a faithful simulation we really need to write down the contents of the oracle tape as well — and that could potentially be much longer (i.e., of length $\mathsf{poly}(n)$). (The proof really exploited the fact that we only needed roughly $O(\log n)$ steps to describe a configuration of the machine, and this breaks down with oracles.)

§5 Space complexity

Today we'll talk about space complexity.

Definition 5.1. For $s\colon \mathbb{N} \to \mathbb{N}$, we define $\mathsf{SPACE}[s(n)]$ as the set of decision problems f which can be decided by a Turing machine using $O(s(n))$ space. We define $\mathsf{NSPACE}[s(n)]$ analogously for nondeterministic Turing machines.

(In this definition, we only consider Turing machines that really are deciders, meaning that they always eventually halt.)

Definition 5.2. We define $\mathsf{PSPACE} = \bigcup_k \mathsf{SPACE}[n^k]$.

There are natural complete problems for PSPACE, such as TQBF (the name is short for *true quantified Boolean formula*).

Definition 5.3 (TQBF)

- **Input:** a quantified Boolean formula $\varphi = (\exists\, x_1)(\forall\, x_2)\cdots(Q_n\, x_n)[F(x_1, \ldots, x_n)]$ (where F is some Boolean formula).
- **Decide:** whether φ is true or false.

Today we're mostly going to focus on *nondeterministic* space complexity. This seems like a strange kind of class, but in hindsight it turns out to be very natural in particular ways — it captures some things we like to model in computation very well.

§5.1 Time vs. space

First we'll discuss how to relate nondeterministic space to other things, like space and time. We'll start with relating it to time.

Theorem 5.4

For $s(n) \geq \log n$, we have $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{TIME}[2^{O(s(n))}]$.

(By $\mathsf{TIME}[2^{O(s(n))}]$ we mean $\bigcup_c \mathsf{TIME}[2^{cs(n)}]$.)

The idea behind the proof is to consider all possible *configurations* of $\mathcal{M}$ on x. Intuitively, a configuration is a snapshot of what the machine looks like at a certain point — we want the configuration to store enough information so that if we were to hibernate $\mathcal{M}$ on x at this point, then it'd have all the information we need to start it up again. Here's a more formal definition.

Definition 5.5. A configuration of $\mathcal{M}(x)$ consists of the current state of $\mathcal{M}$, the input pointer (which bit of the input tape we're currently reading), the storage pointer (which cell of the storage we're currently looking at), and the contents of the storage itself.

(Note that a configuration does *not* include the input x itself.)

If $\mathcal{M}$ runs in space $cs(n)$, then we can bound the total *number* of such configurations — there's a constant number of states, n choices for the input pointer, and $cs(n)$ choices for the storage pointer; and if there's d choices for each of the cells in the storage (i.e., the storage alphabet has cardinality d, which is a constant),

then there's $d^{cs(n)}$ possible storage configurations. So

$$\#\text{configurations of } \mathcal{M}(x) \leq O(1) \cdot n \cdot cs(n) \cdot d^{cs(n)} \leq 2^{O(s(n))}$$

(using the fact that $s(n) \geq \log n$ to take care of the factor of n).

We can immediately use this to prove a simpler version of Theorem 5.4 for *deterministic* space.

Lemma 5.6
For $s(n) \geq \log n$, we have $\mathsf{SPACE}[s(n)] \subseteq \mathsf{TIME}[2^{O(s(n))}]$.

Proof. Suppose we have a deterministic machine $\mathcal{M}$ that runs in space $O(s(n))$, so that for any input x of length n, there are $2^{O(s(n))}$ possible configurations for $\mathcal{M}$ on x. And the key point is that if $\mathcal{M}$ on x ever repeats a configuration, then we know it's in an infinite loop — it's deterministic, so if it starts from a configuration c and eventually comes back to c, then it's going to do the same thing again and keep repeating c forever. We defined $\mathsf{SPACE}[s(n)]$ to exclude machines that go into infinite loops, so this means $\mathcal{M}$ can't repeat a configuration, and therefore its runtime is at most the number of possible configurations. So $\mathcal{M}$ *itself* runs in time $2^{O(s(n))}$, and we're done. □

This argument isn't good enough to prove Theorem 5.4 — it *can* be used to say that $\mathcal{M}$ itself runs in time $2^{O(s(n))}$, but now $\mathcal{M}$ is nondeterministic, so this would only give us $\mathsf{NTIME}[2^{O(s(n))}]$ (whereas we really want *deterministic* time). So we have to do something more careful, and the key concept here is something called a *configuration graph.*

§5.1.1 Configuration graphs

Definition 5.7. For a (deterministic or nondeterministic) Turing machine $\mathcal{M}$ and an input x, the configuration graph $\mathcal{G}_{\mathcal{M},x}$ is the directed graph whose nodes are all possible configurations of $\mathcal{M}$ on x, and where we draw an edge $c \to c'$ if and only if $\mathcal{M}(x)$ starting from c can get to c' in one step.

(We write 'can get to' because if $\mathcal{M}$ is nondeterministic, there are multiple transitions you can take from a single configuration c; then we draw an edge for each of them.)

So $\mathcal{G}_{\mathcal{M},x}$ is a graph with a node for the configuration the machine starts in, nodes for all configurations it could potentially go through in between, nodes for all the accepting configurations, and so on. We'll call the starting configuration c_{start} — here both the input and storage pointer are at index 1, and the storage tape is fully blank. In principle, we can also assume there's a unique accepting configuration c_{acc} — the definition of a Turing machine allows for many accepting configurations, but we can modify the machine so that before accepting, it erases all its storage and moves both pointers back to 1. (Then c_{acc} looks just like c_{start}, except that we're in the accepting state rather than the starting state.)

And the edges in this graph just model the transitions that $\mathcal{M}(x)$ could make. If $\mathcal{M}$ is deterministic, then every node has outdegree at most 1. But even if $\mathcal{M}$ is nondeterministic, the outdegree of each node is constant (i.e., independent of $n = |x|$) — the outdegree of c is the number of transitions $\mathcal{M}$ could potentially make from c, and each of these possible transitions has to be represented in the finite control of $\mathcal{M}$.

The point of this definition is the following simple observation.

Fact 5.8 — We have that $\mathcal{M}(x)$ accepts if and only if there is a path from c_{start} to c_{acc} in $\mathcal{G}_{\mathcal{M},x}$.

(This is true essentially by definition.)

Fact 5.9 — If $\mathcal{M}$ runs in space $s(n)$, then for $|x| = n$, given two configurations c and c' of $\mathcal{M}$ on x, we can check whether $c \to c'$ is an edge in $\mathcal{G}_{\mathcal{M},x}$ in $O(s(n))$ space.

(This is true just because we can go through all possible transitions of $\mathcal{M}$ from c, and check whether each of them leads to c'.)

In particular, this means $\mathcal{G}_{\mathcal{M},x}$ can be 'locally constructed' — given any particular node, we can figure out its neighborhood in a very space-efficient way, just by going through all possible configurations and testing them one at a time. (In fact, if we care about time, then we can do this even more efficiently by just going over all possible transitions of $\mathcal{M}$ out of c.)

And now this gives us a way to simulate an $O(s(n))$ space machine — even a nondeterministic one — in $2^{O(s(n))}$ time.

Proof of Theorem 5.4. Let $\mathcal{M}$ be a nondeterministic Turing machine running in space $O(s(n))$. Then to simulate $\mathcal{M}$ on an input x (of length n) in time $2^{O(s(n))}$:

(1) Build the configuration graph $\mathcal{G}_{\mathcal{M},x}$, by going over all possible configurations (we can imagine encoding configurations as bit-strings of length $O(s(n))$), and then going through all pairs of configurations (c, c') and checking whether there should be an edge $c \to c'$.

(2) Check whether there is a path from c_{start} to c_{acc} in $\mathcal{G}_{\mathcal{M},x}$, e.g., by using BFS or DFS.

Since $\mathcal{G}_{\mathcal{M},x}$ has $2^{O(s(n))}$ nodes (and BFS or DFS runs in polynomial time), this takes $2^{O(s(n))}$ time. □

Finally, Theorem 5.4 has some consequences for containments between important classes.

Definition 5.10. We define $\mathsf{L} = \mathsf{SPACE}[\log n]$ and $\mathsf{NL} = \mathsf{NSPACE}[\log n]$.

(We named these classes LOGSPACE and NLOGSPACE much earlier, in Subsubsection 1.2.3; but from now on we're going to use L and NL for convenience.)

Corollary 5.11

We have $\mathsf{NL} \subseteq \mathsf{P}$.

(We don't expect equality to hold, but separating the two might be difficult.)

§5.2 Nondeterministic vs. deterministic space

Now we'll compare nondeterministic to deterministic space. In particular, we'll discuss the following theorem, which is maybe surprising if you mostly think about nondeterministic vs. deterministic *time*.

Theorem 5.12 (Savitch)

For $s(n) \geq \log n$, we have $\mathsf{NSPACE}[s(n)] \subseteq \mathsf{SPACE}[s(n)^2]$.

In particular, we could have imagined defining a complexity class NPSPACE as $\bigcup_k \mathsf{NSPACE}[n^k]$. But Theorem 5.12 means that this class would just be the same as PSPACE (which is the reason we didn't define it).

Proof. Let $\mathcal{M}$ be a nondeterministic machine running in space $s(n)$. Then on inputs x of length n, we want to deterministically figure out whether there's a path from c_{start} to c_{acc} in $\mathcal{G}_{\mathcal{M},x}$. In the proof of Theorem 5.4, we did this using BFS or DFS. Here the idea is that we'll instead use a 'middle-first search.' more

generally, if we want to figure out whether $\mathcal{M}$ can go from c to c' in 2^t steps, we do so by considering the 'middle' configuration c_{mid} — this is true if and only if there exists some c_{mid} such that $\mathcal{M}$ can go from c to c_{mid} in 2^{t-1} steps and from c_{mid} to c' in 2^{t-1} steps. (This is in some sense similar to the speedup step (Lemma 3.11) from our proof of time-space lower bounds, where we guessed a bunch of equally spaced configurations; here we're only guessing one.) So we go through all possible 'middle' configurations c_{mid}, and for each we recursively check whether we can go from c to c_{mid} and from c_{mid} to c' in 2^{t-1} steps.

Finally, in the outermost level of the recursion — to see whether $\mathcal{M}$ can go from c_{start} to c_{acc} at all — we can simply set $t = O(s(n))$ (the configuration graph has $2^{O(s(n))}$ nodes, so if there's a path from c_{start} to c_{acc} at all, then there's one with at most $2^{O(s(n))}$ steps).

To see that this algorithm can be implemented in $O(s(n)^2)$ space, each configuration c takes $O(s(n))$ space to describe. Meanwhile, we've got a recursion of depth $O(s(n))$. At any point in time, we essentially just need to store two or three configurations and the value of t for each level of the recursion (we store the current values of c and c', and which middle configuration c_{mid} we're currently trying — note that after we've computed whether c leads to c_{mid}, we reuse the same space to compute whether c_{mid} leads to c'). So we have recursion depth $O(s(n))$ and we're storing $O(s(n))$ bits at each level, which means our total space usage is $O(s(n)^2)$. □

In particular, Savitch's theorem implies $\mathsf{NL} \subseteq \mathsf{SPACE}[(\log n)^2]$. It's an interesting open problem to improve Savitch's theorem for the case of NL.

Open question 5.13. Is $\mathsf{NL} = \mathsf{L}$?

In fact, it's open even whether we can improve Savitch's theorem by a tiny bit — whether there is *any* unbounded function $\alpha\colon \mathbb{N} \to \mathbb{N}$ for which $\mathsf{NL} \subseteq \mathsf{SPACE}[(\log n)^2/\alpha(n)]$.

§5.3 The class NL

In the remainder of this lecture, we'll mostly talk about NL and the question of L vs. NL. Just as with P vs. NP, one way to think about this question is by looking at complete problems — we want to find problems in NL which are in L if and only if $\mathsf{L} = \mathsf{NL}$. But we have to be careful with how we define NL-completeness — we can't just use polynomial-time reductions, because both L and NL are contained in P (so any problem in one of them is polynomial-time reducible to any other problem). So we need a more fine-grained notion of reductions.

§5.3.1 Log-space reductions

Definition 5.14. A log-space reduction from a language A to a language B (working with a finite alphabet Σ) is a function $f\colon \Sigma^* \to \Sigma^*$ such that there is a Turing machine $\mathcal{M}$ with read-only input, read-write storage with $O(\log n)$ cells, and a *write-only one-way output tape*, such that for all inputs x:

- $\mathcal{M}(x)$ halts with $f(x)$ on its output tape.
- $x \in A$ if and only if $f(x) \in B$.

The point is that we want to reduce from A to B, which means we want to take something that looks like an A-problem and turn it into something that looks like a B-problem. But these problems are potentially long — i.e., of length $\mathsf{poly}(n)$ — which means if we want the reduction to 'run in logarithmic space,' then we can't just have it write down the B-problem on its storage tape. So we instead add in a write-only output tape that *doesn't* count towards the space bound, which is where it writes down the B-problem instead.

Definition 5.15. We say $A \leq_{\mathsf{L}} B$ if there exists a log-space reduction f from A to B.

Before we do things with this notion of reductions, we'll check that it's a 'good' one. There's basically two properties that we want out of reductions (so that we can discuss things like completeness):

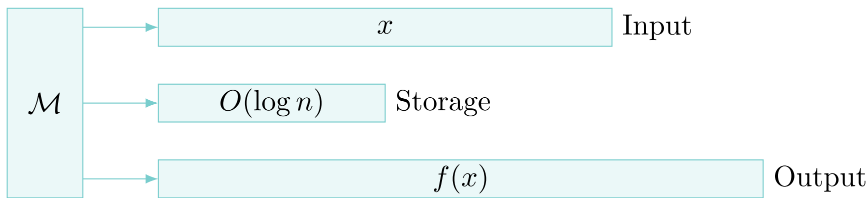
- If A reduces to B and we can solve B efficiently, then we can also solve A efficiently (where 'efficient' depends on what classes or notion of reduction we're working with).
- If A reduces to B and B reduces to C, then A should also reduce to C.

We'll check that both of these properties hold for log-space reductions.

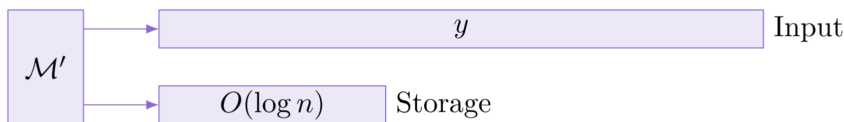
Theorem 5.16

If $A \leq_{\mathsf{L}} B$ and $B \in \mathsf{L}$, then $A \in \mathsf{L}$.

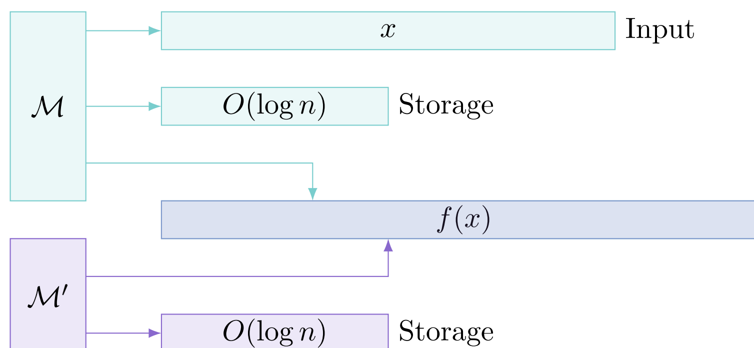
Proof. Suppose we have a reduction f from A to B — this means we've got a machine $\mathcal{M}$ which takes x, has a read-write storage of size $O(\log n)$, and prints $f(x)$ on some write-only output tape.



And since $B \in \mathsf{L}$, we also have a machine $\mathcal{M}'$ which takes in some input y, has a read-write storage of size $O(\log n)$, and eventually accepts or rejects based on whether $y \in B$.



We want to solve A, and we've got a way of turning A into B and a way of solving B, so we'll essentially try to combine them — we'll essentially take the input tape of $\mathcal{M}'$ and replace it with the output tape of $\mathcal{M}$.



So the input tape of our new machine is x (the input tape of $\mathcal{M}$), its storage consists of the storage of both machines, and we want to run $\mathcal{M}'$ on $f(x)$. But we've got a problem — we can't actually write down $f(x)$.

Instead, as we run $\mathcal{M}'$, every time it needs a bit of its input, we hibernate $\mathcal{M}'$, start up $\mathcal{M}$ from scratch and compute that bit — suppose that on a certain step, $\mathcal{M}'$ needs the ith symbol of its input. (In our

simulation of $\mathcal{M}'$, we keep a counter that keeps track of which symbol of its input $\mathcal{M}'$ is currently reading.) Then we start up $\mathcal{M}$ on x from scratch, except that we *don't* store its output tape — instead, as it runs, we store a counter that keeps track of which bit of the output tape it's currently on. We ignore all symbols it writes to the output tape except the ith one. And once this counter reaches i, we get the symbol of $f(x)$ that we needed, give it to $\mathcal{M}'$, and start up $\mathcal{M}'$ again.

This sounds extremely inefficient, and it *is* pretty time-inefficient (if $\mathcal{M}$ takes time n^{100}, then every step of $\mathcal{M}'$ in this simulation could take time n^{100}). But space-wise, it's actually extremely efficient — we just need to store the storage of $\mathcal{M}$ and $\mathcal{M}'$, the $O(\log n)$-bit counter for the input head of $\mathcal{M}'$, and (when we hibernate $\mathcal{M}'$ and run $\mathcal{M}$) the $O(\log n)$-bit counter for the output head of $\mathcal{M}$. So this takes $O(\log n)$ space, showing that $A \in \mathsf{L}$. □

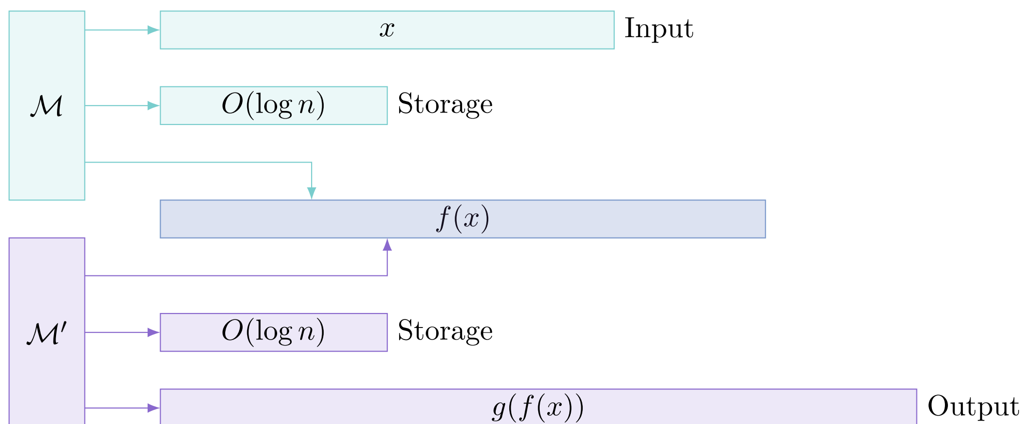
We'll now check that the property of transitivity holds, using essentially the same proof.

Theorem 5.17

If $A \leq_{\mathsf{L}} B$ and $B \leq_{\mathsf{L}} C$, then $A \leq_{\mathsf{L}} C$.

This is important because it means if we want to find NL-complete problems, it's enough to start with some 'helper' problems that we already know are NL-complete, and reduce other problems to those (as we do with NP-completeness).

Proof. Let f be a reduction from A to B computed by $\mathcal{M}$ — so $\mathcal{M}$ reads x, uses some read-write storage, and writes $f(x)$. Similarly, let g be a reduction from B to C computed by $\mathcal{M}'$. Then $g \circ f$ is a reduction from A to C, and we can compute it in the same way as in the proof of Theorem 5.16 — we imagine taking the output $f(x)$ of $\mathcal{M}$ and using it as the input to $\mathcal{M}'$.



And we chain the two machines together in the same way as in the proof of Theorem 5.16 — we start running $\mathcal{M}'$, and every time it needs some bit of $f(x)$, we hibernate it, start up $\mathcal{M}$ on x from scratch, and keep a counter for its location on the output tape and get just the ith bit of its output (ignoring all the other bits). (This is the exact same proof as in Theorem 5.16, except that now $\mathcal{M}'$ computes a reduction rather than a decision problem.) □

We can take the idea of these proofs even further. Last class, we talked about oracles and space complexity — when we're measuring the space complexity of an oracle algorithm, we've got a read-only input tape, a read-write storage, and a write-only one-way oracle tape. The machine writes down its query one symbol at a time on the oracle tape and then calls the oracle; and when it does so, the oracle tape gets erased and reset.

This lets us define L relative to an oracle. A log-space machine can only count up to $O(\log n)$, so it can't make super long queries, but it *can* ask queries of length $\mathsf{poly}(n)$.

And the same proof as that of Theorem 5.16 shows that L is closed under composition — i.e., $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$. (Here we think of $\mathcal{M}$ as the original oracle algorithm and $\mathcal{M}'$ as the algorithm implementing the oracle, and $\mathcal{M}$ has an oracle tape instead of an output tape.)

§5.3.2 A NL-complete problem

Now we'll discuss the notion of NL-completeness.

> **Definition 5.18.** We say B is NL-complete if $B \in \mathsf{NL}$ and for every $A \in \mathsf{NL}$ we have $A \leq_{\mathsf{L}} B$.

(This definition has the same format as the definition of NP-completeness, except that we now have log-space reductions instead of polynomial-time ones.)

The notion of configuration graphs suggests a natural candidate for a NL-complete problem.

Definition 5.19 (DPath)

- **Input:** (G, s, t), where G is a directed graph and s and t are nodes in G.
- **Output:** whether there exists a directed path from s to t.

We can imagine that G is encoded as a $n \times n$ adjacency matrix, and s and t are represented by $O(\log n)$-bit strings telling us the indices of s and t in this adjacency matrix (in binary).

Interestingly, if we consider the *undirected* version of this problem, which we call UPath, it's been known for many decades that UPath can be solved in *randomized* log-space (i.e., log-space with access to a *one-way* random tape — this essentially means you can toss coins, but can't remember their results), and more recently it was shown that $\mathsf{UPath} \in \mathsf{L}$. But the *directed* version is NL-complete (which we will prove shortly) — which means showing it's in L would be equivalent to proving that $\mathsf{L} = \mathsf{NL}$.

Theorem 5.20

The problem DPath is NL-complete.

Proof. First, to see that $\mathsf{DPath} \in \mathsf{NL}$, we can solve DPath nondeterministically by just guessing the next edge to take repeatedly. More precisely, on input (G, s, t), we want to guess a sequence of vertices $v_1, \ldots, v_k$ such that $(s, v_1, \ldots, v_k, t)$ is a path in G. And the idea is that we only need to keep track of the current node (and the next one) at any point. So we start with s, guess an edge to some node v_1, then guess an edge to some node v_2 (and erase v_1), then guess an edge to some node v_3 (and erase v_2), and so on; and if we ever reach t, we *accept*. We also keep a counter storing the number of nodes we've guessed so far — if there's a path from s to t, then there's one whose length is at most the total number of nodes in G (since there's no need to repeat a node), so if our counter exceeds this number, then we *reject*.

Now we'll show DPath is NL-hard. Let $\mathcal{N}$ be a nondeterministic machine running in $O(\log n)$ space. Then to reduce $L(\mathcal{N})$ to DPath, for input x, we simply output $(\mathcal{G}_{\mathcal{N},x}, c_{\text{start}}, c_{\text{acc}})$ (since $\mathcal{N}$ accepts x if and only if there's a path from c_{start} to c_{acc} in its configuration graph).

To do this reduction in logarithmic space, when we're trying to print the adjacency matrix of $\mathcal{G}_{\mathcal{N},x}$, we don't try to write down the entire matrix at once (it'll have size $\mathsf{poly}(n) \times \mathsf{poly}(n)$, which is too big to store). Instead, we print it one entry at a time, using our storage to keep track of which entry (i.e., which pair of configurations c and c') we're on — and for each such pair, we can check whether there's an edge from c to c' in $O(\log n)$ space. □

§5.3.3 NL = coNL

Trying to find a path in a graph is a prety basic problem, so the fact that it's NL-complete might suggest that NL is a basic class. But it turns out that NL actually captures lots of things. For example, you can imagine defining a log-space hierarchy (similar to the polynomial hierarchy) by considering $\mathsf{NL}^{\mathsf{NL}}$, $\mathsf{NL}^{\mathsf{NL}^{\mathsf{NL}}}$, and so on. (And people *did* define this hierarchy at some point.) But it turns out that this hierarchy collapses all the way to NL! (So in some sense, not only is L closed under composition, but so is NL.) This follows from the following theorem.

Theorem 5.21
We have NL = coNL.

(We define coNL as the set of complements to problems in NL, as usual.)

The proof is really beautiful, so we'll go over it. It's one of the reasons it's difficult to understand NL, because it says NL is unexpectedly powerful in some way — we can prove that there *is* a path in log-space by just guessing it, but it's much more surprising that we can also prove there's *no* path.

Proof. We'll show that $\overline{\mathsf{DPath}}$ (i.e., the complement of DPath) is in NL. At a very high level, the proof will involve *lots* of guessing. On an input (G, s, t), we want to be able to check (in nondeterministic log-space) that there is *no* path from s to t. We'll first imagine that a little birdy has told us a secret — the *number* of nodes in G which are reachable from s. We'll call this number r (note that r can be written with $O(\log n)$ bits in binary). We'll first show how to check $(G, s, t) \notin \mathsf{DPath}$ (with a NL algorithm) if we're given r; and then we'll later show how to obtain r. '

The idea (behind how we prove that there's no path from s to t, given r) is that we go through all nodes $v \neq t$ and check if there's at least r reachable nodes among them — if we know there's at least r reachable nodes which are not t, then since there's only r reachable nodes in *total*, we know t is not reachable.

More precisely, we go through all nodes $v \neq t$ one at a time, keeping a counter c for the number of reachable nodes we've found so far. For each, we guess whether v is reachable from s (this is a single bit); and if our guess is yes, then we guess a path from s to v (one vertex at a time, as in the proof that $\mathsf{DPath} \in \mathsf{NL}$). If we succeed at finding such a path, then we increment c by 1. Meanwhile, if we fail (i.e., we guessed that v was reachable but the path we were guessing broke), then we *reject.*

And once we've gone through all nodes (except t), if $c = r$ then we *accept*, and otherwise we *reject.* The point is that if $c = r$ then we've found r different reachable nodes from s (none of which is t), so if there's only r reachable nodes, then t can't be reachable.

(This is an example of how just knowing the *number* of good objects can let you verify that an object is bad. This is a paradigm that comes up in other parts of complexity theory as well — sometimes if you want to check that something is bad, and someone tells you the *number* of things which are good, then you can find that many *other* things and verify that they're all good, so this one has to be bad.)

Now the big question is how we compute r. We'll do this inductively — we'll let r_k be the number of nodes reachable from s by a path of at most k edges (where k is between 0 and $|V| - 1$). For example, $r_0 = 1$ (the only node reachable from s by a path of 0 edges is itself), $r_1 = 1 + \operatorname{outdeg}(v)$ (the extra 1 is from s itself), and $r_{|V|-1} = r$ (since if a node is reachable, then it's reachable in at most $|V| - 1$ steps).

Suppose we're given r_i, and we want to compute r_{i+1} with a NL algorithm — so we've got a little birdy secret and we want a bigger birdy secret. To do so, we initialize a counter c at 0. We'll then go through all nodes v one at a time, and we'll use c to count up the nodes we've found that are reachable by a path of at most $i + 1$ edges (so that in the end c will be r_{i+1}).

For each node v, in order to figure out whether v is reachable by a path of at most $i+1$ edges (i.e., whether we should increment c), we'll try to go through all nodes u reachable by a path of at most i edges and see

if there is an edge $u \to v$ for any such u. To do so, we'll define a new counter $d = 0$ (which will store the number of such nodes u we've found). Then we go through all nodes u one at a time. For each u, we guess whether u is reachable from s in at most i edges or not. If we've guessed yes, then we guess a path of at most i edges from s to u (as before, guessing one vertex at a time); and if we successfully find such a path, then we increment d by one (otherwise — if we guessed that u was reachable but failed to find a path — we *reject*). If we did successfully find a path and $u \to v$ is an edge, then we immediately know v is reachable by a path of at most $i + 1$ edges, so we can jump out of the loop over u and increment c by 1 (for v).

Otherwise, once we've gone through all possible nodes u, we need to verify that we really found all nodes u reachable by a path of at most i edges (i.e., we guessed all of them were reachable and found the corresponding paths). To do so, we check that $d = r_i$; if $d \neq r_i$ then this means we undercounted, so we *reject*. If $d = r_i$, then this means we really did consider all nodes u reachable by a path of at most i edges, and we found that none of them had an edge to v; so v is *not* reachable by a path of at most $i + 1$ edges, so we don't increment c and we move to the next vertex v.

Finally, once we've gone through all v, we set $r_{i+1} = c$.

The point is that in this proof we do tons of guessing, and every time we're trying to do something with guessing, we have a way of knowing that we've messed up — in which case we just reject. Meanwhile, some thread will make all the right guesses and get the right answer. □

Remark 5.22. Given that $\mathsf{NL} = \mathsf{coNL}$, how do you show that $\mathsf{NL}^{\mathsf{NL}} = \mathsf{NL}$? The analogous statement for the polynomial hierarchy — that if $\mathsf{NP} = \mathsf{coNP}$ then $\mathsf{PH} = \mathsf{NP}$ — can be proven by using the quantifier characterization of PH. Specifically, the assumption $\mathsf{NP} = \mathsf{coNP}$ allows us to turn an $\exists$ into a $\forall$ and vice versa, so we can take any chain of $\exists$'s and $\forall$'s and reducing it to just one.

And there's a similar quantifier characterization for NL and coNL, where instead of storing the whole witness, we imagine that the witness is passing through in a streaming sort of way. This lets us think about $\mathsf{NL}^{\mathsf{NL}}$ and so on in terms of $\exists$'s and $\forall$'s, and the fact that $\mathsf{NL} = \mathsf{coNL}$ again lets us flip between $\exists$'s and $\forall$'s to reduce to just having one quantifier.

The fact that $\mathsf{NL} = \mathsf{coNL}$ has other implications as well.

Definition 5.23 (2SAT)

- **Input:** a 2CNF φ (i.e., an **AND** of clauses, where each clause is an **OR** of two literals).
- **Decide:** whether φ is satisfiable.

Theorem 5.24

The problem 2SAT is NL-complete.

The interesting part of this theorem isn't the fact that 2SAT is NL-hard, but that it's even in NL. We don't have space to *store* a satisfying assignment, but it turns out that we can get away with not doing that using nondeterminism. And this proof really uses the fact that NL is closed under complement.

Proof sketch. Suppose we're given a 2CNF $\varphi = (\ell_1 \vee \ell_2) \wedge (\ell_3 \vee \ell_4) \wedge \cdots$ (where each ℓ_i is a literal). We then imagine making a graph G with a node for each literal — so the nodes of G are $x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}$. And for each clause $\ell_1 \vee \ell_2$, we draw directed edges corresponding to implications from that clause — this clause gives the implications $\overline{\ell_1} \implies \ell_2$ and $\overline{\ell_2} \implies \ell_1$, and so we draw the corresponding edges in the graph. Then we can think of finding paths in this graph as uncovering consequences in the formula.

Fact 5.25 — The formula φ is *unsatisfiable* if and only if there exists some $i \in [n]$ for which G has a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i.

In words, such a path means that from φ we can infer that x_i is true if and only if $\overline{x_i}$ is, which is impossible. We're not going to prove this fact. One direction of this fact is clear — if such paths exist, then φ can't have a satisfying assignment for this reason. The converse is harder, but is true as well.

Then in order to solve the *complement* of $\mathsf{2SAT}$, we just need to call DPath $2n$ times, and we know $\mathsf{DPath} \in \mathsf{NL}$. So $\overline{\mathsf{2SAT}} \in \mathsf{NL}$, and since NL is closed under complement, this means $\mathsf{2SAT} \in \mathsf{NL}$ as well. □

Remark 5.26. Most people believe that $\mathsf{P} \neq \mathsf{NL}$. But we do know that *alternating logspace* (or AL) — an extremely souped-up version of $\mathsf{NL}^{\mathsf{NL}^{\cdots}}$ where you sort of get to switch between NL and coNL wherever you want — is actually P (this is on the problem set), and that together with the fact that $\mathsf{NL} = \mathsf{coNL}$ gives some pause.

§6 Circuit complexity

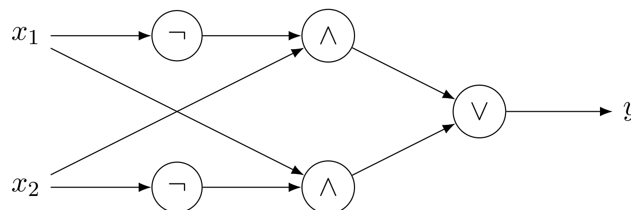
Today's lecture is on circuit complexity.

§6.1 Boolean circuits

Definition 6.1. A Boolean circuit with n inputs and m outputs is a directed acyclic graph with n sources $x_1, \ldots, x_n$ and m sinks (corresponding to the different outputs). All other nodes are labelled by one of $\wedge$, $\vee$, and $\neg$; each $\neg$ has one input (and one output), and each $\wedge$ and $\vee$ has two inputs (and one output). We define the size of such a circuit as the total number of $\wedge$'s and $\vee$'s.

Note that $\neg$ is 'free' — it doesn't cost us anything in terms of the size of the circuit.

Any Boolean circuit computes a function $C\colon \{0,1\}^n \to \{0,1\}$. For example, the following circuit computes the **XOR** function.



Remark 6.2. We don't bound the out-degrees of nodes in a circuit — x_1 or x_2 or any intermediate node can be used as many times as we want. (In contrast, in a *formula* everything can only be used once.)

This model is great for understanding *finite* functions (i.e., functions on a fixed number of inputs). But in this class, we're really interested in decision problems, which are functions $f\colon \{0,1\}^* \to \{0,1\}$. So we'll extend this notion to such functions as follows — instead of a single circuit, we'll give a separate circuit for each input length n, and when we get an input of length n, we run the nth circuit on it.

Definition 6.3. A circuit family is a collection of circuits $\{C_n\} = \{C_1, C_2, \ldots\}$ where for each n, we have $\mathsf{inputs}(C_n) = n$ and $\mathsf{outputs}(C_n) = 1$.

Definition 6.4. We say a circuit family $\{C_n\}$ computes a function $f\colon \{0,1\}^* \to \{0,1\}$ if for all $x \in \{0,1\}^*$ with $x \neq \varepsilon$, we have $f(x) = C_{|x|}(x)$.

So the way we compute using a circuit family is that we take our input, check its length, find the appropriate circuit in our infinite family, and run it. (We don't worry about the empty string ε because a circuit with 0 sources doesn't really make sense.) This is a *non-uniform* model of computation, where we don't just have a single algorithm we're running for all inputs — the algorithm itself varies depending on input length (so there's an infinite list of algorithms, in some sense).

When we think about the circuit complexity of a function f, we consider f restricted to n-bit inputs (for each n), and we want to understand how small the corresponding circuit can be.

Definition 6.5. For a function $s\colon \mathbb{N} \to \mathbb{N}$, we define $\mathsf{SIZE}[s(n)]$ as the set of functions $f\colon \{0,1\}^* \to \{0,1\}$ such that there exists a circuit family $\{C_n\}$ computing f with $\mathsf{size}(C_n) \leq s(n)$ for all n.

Note that here we have $s(n)$ instead of $O(s(n))$ — this is because sometimes we're interested in even the leading constants. This means these complexity classes will depend on which gates we use in the circuit. For instance, our example circuit computing $x_1 \oplus x_2$ has size 3, and it's possible to show that with only **AND** and **OR** gates we can't do better (more generally, we need $3(n-1)$ gates to compute the **XOR** of n inputs). But if we allowed **XOR** gates too, then of course we could get away with a circuit of size 1.

In the remainder of this lecture, we'll see several facts about circuit complexity and what's going on with these size classes — we have a non-uniform model where our circuit families have infinite descriptions, so things can get pretty weird.

§6.2 Bounds on maximum possible circuit complexity

First, we'll see some bounds on how large the circuit complexity of a function can possibly be; we'll see that there are hard functions, but they can't get *too* hard.

Proposition 6.6

There is some $\varepsilon > 0$ and some $f\colon \{0,1\}^* \to \{0,1\}$ with $f \notin \mathsf{SIZE}[\varepsilon \cdot 2^n/n]$.

This means there exist decision problems that need exponential-size circuits — even though we're allowed to construct a separate circuit for each input length.

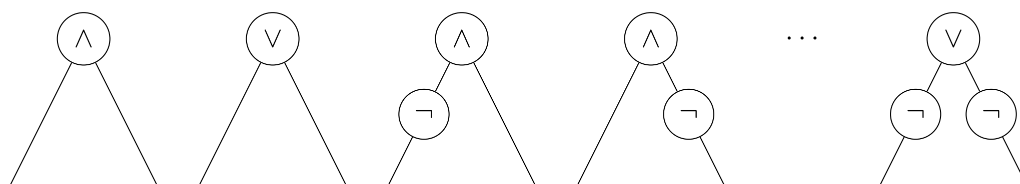
Proof. We'll show there is some $\varepsilon > 0$ such that for each sufficiently large n, there is a function $f_n\colon \{0,1\}^n \to \{0,1\}$ which doesn't have a circuit of size $\varepsilon \cdot 2^n/n$; then we can form $f\colon \{0,1\}^* \to \{0,1\}$ by putting together such functions. The idea is a counting argument — the number of *total* functions $f_n\colon \{0,1\}^n \to \{0,1\}$ is 2^{2^n} (since there's 2^n possible inputs, and two choices for our output on each of them). Meanwhile, we'll show that the number of functions with circuits of size $s = \varepsilon \cdot 2^n/n$ is much smaller than this (for appropriately chosen ε). In fact, we'll upper-bound just the number of *circuits* of size s — this suffices because each circuit computes only one function. (Some circuits could compute the same function, so the actual number of functions with small circuits could be even less, but at least this gives an upper bound.)

Claim 6.7 — There is an absolute constant c such that for all n and s, the number of circuits with n inputs and size s is at most $2^{cs\log(n+s)}$.

Proof. Ryan likes to think of this as an encoding argument — we'll show that we can encode every circuit of size s in at most $cs\log(n+s)$ bits. (If we have a unique encoding of each circuit using this many bits, then the number of possible encodings, and therefore the number of circuits, is at most $2^{cs\log(n+s)}$.)

The idea is that our circuit has input nodes $x_1, \ldots, x_n$ and intermediate nodes $1, \ldots, s$ (which we refer to as *gates*). At each gate $i \in [s]$, we need to specify where its inputs are coming from — out of $x_1, \ldots, x_n$ and the at most s preceding gates — and what type of gate it is (e.g., $\wedge$ or $\vee$). Each of its two inputs takes $O(\log(n+s))$ bits to write down; and the gate type takes a constant numbr of bits to write down.

There's a subtlety — $\neg$ gates don't count towards the size of the circuit. But we can take these gates and fold them into our types of gates — so instead of just having $\wedge$ and $\vee$ gates, we have $\wedge$ gates, $\vee$ gates, $\wedge$ gates with a $\neg$ on the left, and so on. But there's still only a finite number of gate types, so the gate type still takes $O(1)$ bits to write down.



So then we've got s different gates, and we're using $O(\log(n+s))$ bits to describe each one, so the entire encoding takes $O(s\log(n+s))$ bits. $\square$

Then for any s with $2^{2^n} > 2^{cs\log(n+s)}$, there is a function $f_n\colon \{0,1\}^n \to \{0,1\}$ that can't be computed by a size-s circuit. And this inequality will be true if $s = \varepsilon \cdot 2^n/n$ (where ε is a small constant to cancel out c), as then we have $cs\log(n+s) \approx \varepsilon c \cdot 2^n/n \cdot \log(2^n/n) \approx \varepsilon c \cdot 2^n$. $\square$

In fact, this argument shows that if ε is a sufficiently small constant, then even a *random* function on n bits is likely to need a circuit of size $\varepsilon \cdot 2^n/n$.

> **Remark 6.8.** By a much more complicated argument, it's possible to show that for all ε, there exist functions $f \notin \mathsf{SIZE}[(1-\varepsilon)2^n/n]$.

On the other hand, we can't get a lower bound much larger than this.

Proposition 6.9

For all functions $f\colon \{0,1\}^* \to \{0,1\}$, we have $f \in \mathsf{SIZE}[O(2^n/n)]$.

Together, Propositions 6.6 and 6.9 mean that the circuit complexity of a function can be as large as $2^n/n$, but it can't get any larger — every function has a circuit with only $2^n/n$ gates (ignoring constant factors).

> **Remark 6.10.** In fact, it's true even that for all $\varepsilon > 0$, we have $f \in \mathsf{SIZE}[(1+\varepsilon)2^n/n]$ for all functions f — so $2^n/n$ is genuinely the correct bound, not just up to constant factors.

We're not going to prove Proposition 6.9; instead, we'll prove a simpler bound.

Fact 6.11 — For all functions f, we have $f \in \mathsf{SIZE}[O(2^n)]$.

Proof. Suppose we restrict our function to inputs of length n; then we want to find a circuit for $f_n\colon \{0,1\}^n \to \{0,1\}$. To do so, we can write $f_n(x_1, \ldots, x_n)$ in the form

$$f_n = (x_n \wedge f^1_{n-1}(x_1, \ldots, x_{n-1})) \vee (\overline{x_n} \wedge f^0_{n-1}(x_1, \ldots, x_{n-1})) \tag{1}$$

for some $f^0_{n-1}, f^1_{n-1}\colon \{0,1\}^{n-1} \to \{0,1\}$. What's going on in this formula is that we can imagine writing out the whole truth table of f such that everything with $x_n = 0$ is in the first half, and everything with $x_n = 1$ is in the second; then f^0_{n-1} is the first half of the truth table, and f^1_{n-1} is the second. And if we want to figure out the output of f, we first look at x_n; if x_n is 0 then we want to look at the first half of the truth table, and if x_n is 1 then we want to look at the second half. So what we've done is essentially that we took the 2^n-bit truth table of f (saying the value of f on all possible inputs) and split it into two 2^{n-1}-bit truth tables (corresponding to what happens when we fix x_n).

000	001	010	011	100	101	110	111
1	0	1	1	0	0	1	0

f^0_{n-1} (first four columns) $\quad f^1_{n-1}$ (last four columns)

And (1) involves 3 gates (since we're not counting $\neg$ in our number of gates), so we've paid 3 gates and reduced the question of computing one function on n bits to computing 2 functions on $n-1$ bits. This gives a recurrence $s(n) = 3 + 2s(n-1)$, and $s(1) \leq 3$, so solving the recurrence gives $s(n) \leq 3(2^n - 1)$. $\square$

We won't prove Proposition 6.9 (which improves this bound by a factor of n), but here's the main idea.

Proof sketch of Proposition 6.9. We again use (1) to repeatedly split the problem of computing f into a problem of computing two functions on one fewer bit, but instead of doing this all the way until we're left with one input, we only do this until we're left with t inputs — so now we've got 2^{n-t} different functions of $x_1, \ldots, x_t$ that we need to compute, each of which corresponds to a different setting of $x_{t+1}, \ldots, x_n$.

Then we can construct a circuit D with t inputs, 2^{2^t} outputs, and $O(2^t \cdot 2^{2^t})$ gates such that $D(y)$ outputs the concatenation of $g(y)$ over all 2^{2^t} functions $g\colon \{0,1\}^t \to \{0,1\}$ (in some fixed order) — we can do this because we can compute *each* function g with a circuit of size $O(2^t)$ by Fact 6.11, and then we just put all these circuits together. Finally, for each of the 2^{n-t} functions we need to compute, we simply wire in the appropriate bit of $D(y)$. (The point is that if t is small, then the number of distinct functions $\{0,1\}^t \to \{0,1\}$ is much smaller than the number of functions we're trying to compute (2^{2^t} compared to 2^{n-t}), so it's more efficient to compute *all* functions at once and then reference this computation as needed than to compute each function separately every time we need it.)

This gives a circuit of size $O(2^{n-t} + 2^t \cdot 2^{2^t})$, and setting $t = \frac{1}{2}\log n$ makes this $O(2^n/n^{1/2})$ (this isn't exactly what we wanted, but it shows how to shave off a polynomial factor from Fact 6.11). $\square$

§6.3 An undecidable problem

The next weird fact we'll mention is that we can solve undecidable problems using circuits. More precisely, Proposition 6.9 already means that any function f (even an undecidable one) can be solved by *some* circuit family, but even more strikingly, there are undecidable functions that can be solved by extremely efficient circuits.

Proposition 6.12

There exist undecidable functions $f \in \mathsf{SIZE}[O(1)]$.

Proof. Let L be any undecidable problem over $\{0,1\}^*$ (e.g., the halting problem or A_{TM}), and let $\text{bin}\colon \mathbb{N} \to \{0,1\}^*$ be any bijection such that both it and its inverse are computable (the name is short for *binary*, and we'll think of this as the binary string representing n — for example, one possible bin is the function that takes the binary encoding of n and removes the leading 1).

Then as a first attempt, as we saw in Subsection 1.5, we can define the language $L_0 = \{1^n \mid \text{bin}(n) \in L\}$. So in words, we're taking our string of n 1's, converting it to a binary string, and asking whether that binary string is in L. Then $L_0 \in \mathsf{SIZE}[O(n)]$ — for each input length n, either no string is accepted, in which case we're trying to compute the all-0's function (which has a circuit of constant size), or just 1^n is accepted, in which case we're trying to compute the **AND** function (which has a circuit of size $O(n)$). And L_0 is undecidable because there's a mapping reduction from L to L_0 (where we just convert a given binary string into the corresponding unary string; then the unary string is in L_0 if and only if the binary one was in L).

We really wanted $O(1)$ circuit complexity, not just $O(n)$. But we can get this by refining the above idea — instead of L_0, we take

$$L' = \{x \mid \text{bin}(|x|) \in L\}.$$

(So for each input length n we're still checking whether $\text{bin}(n) \in L$, but now instead of adding just 1^n to our language when this is the case, we add *all* strings of length n.) This problem is still undecidable for the same reason. But now for each input length n, either L restricted to inputs of length n is the all-0's function, or it's the all-1's function (since whether x is in L only depends on $|x|$). And we can compute either with a circuit of constant size (e.g., $x_1 \wedge \overline{x_1}$ or $x_1 \vee \overline{x_1}$). $\square$

The main point of this fact is to illustrate how different circuit complexity is from things like computability theory — the fact that we have separate algorithms for each input length means that things can get crazy.

§6.4 From algorithms to circuits

The next fact we'll see is that any algorithm can be made into a circuit family.

Proposition 6.13

There is a universal constant c such that $\mathsf{TIME}[t(n)] \subseteq \mathsf{SIZE}[ct(n)^c]$ for all $t\colon \mathbb{N} \to \mathbb{N}$.

The value of c can depend on what computation model you start with (e.g., one-tape or multi-tape Turing machines or random access machines, or so on). We'll prove this for one-tape Turing machines (this implies the same statement for other reasonable models as well, since machines in those models can be converted to one-tape Turing machines with a polynomial increase in the runtime).

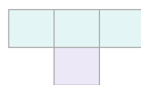
Proof. Suppose we have a one-tape Turing machine $\mathcal{M}$ that computes a function f in time $t(n)$ (on inputs of length n). We'll show that from $\mathcal{M}$ we can obtain a circuit C of size $O(t(n)^2)$ that also computes f. The idea is to work with the *computation tableau* of $\mathcal{M}$ on the input x, i.e., the table whose rows are the configurations $\mathcal{M}$ goes through on x.

We'll let Q be the set of states of $\mathcal{M}$, and for simplicity we'll assume that the tape alphabet of $\mathcal{M}$ is just $\{0, 1, \sqcup\}$ (we can do so without loss of generality). We'll write our configurations as rows of length $t(n)$ where each entry has two slots — the second slot stores the tape symbol at the corresponding cell, and the first slot stores the current state if that's where the head of $\mathcal{M}$ is, and $\sqcup$ otherwise. So for example, the initial configuration (the first row of the tableau) looks like the following.

$q_0\, x_1$	$\sqcup\, x_2$	$\sqcup\, x_3$	$\cdots$	$\sqcup\, x_n$	$\sqcup\sqcup$	$\sqcup\sqcup$	$\cdots$	$\sqcup\sqcup$

Then as we go down the table, we step through configurations one by one. Since $\mathcal{M}$ runs for time $t(n)$, we can assume the table has height and width $t(n)$. And we want to figure out whether in the end it reaches an accepting state. Without loss of generality we can assume $\mathcal{M}$ has a unique accepting state and it moves its head all the way to the left before accepting; then we want to figure out whether the tableau has q_{acc} in the bottom-left corner or not.

The idea is that the computation tableau can be constructed very locally — in particular, we can determine an entry in one configuration by the three entries above it in the previous configuration.



In fact, the function determining the purple cell from the three blue cells is the same at all locations in the tableau — so if we let $T(i, j)$ be the (i, j)th entry of the tableau, then there's some function ψ, depending only on $\mathcal{M}$, such that

$$T(i, j) = \psi(T(i-1, j-1) + T(i-1, j) + T(i-1, j+1))$$

for all i and j. Suppose that each entry consists of k bits (where k is a constant depending on $\mathcal{M}$), so that ψ is a fixed function $\psi\colon \{0,1\}^{3k} \to \{0,1\}^k$. Then by Fact 6.11, ψ has a circuit of size $O(k \cdot 2^{3k})$ (since we can make an $O(2^{3k})$-size circuit for each of its output bits), and computing the computation tableau of $\mathcal{M}$ on x (after the initial configuration) amounts to computing ψ at each cell. Meanwhile, for the initial configuration, we feed in $x_1, \ldots, x_n$ from the input wires (and we can imagine q_0 and all the $\sqcup$'s as being

hardcoded). And finally, in the end we check that the state in the bottom-left corner is q_{acc} (and *accept* if and only if this is true).

This gives us a circuit of size $O(t(n)^2)$ — there are $t(n)^2$ cells in the tableau, and each involves a constant amount of circuitry (we're computing ψ in each cell). □

Corollary 6.14

We have $\mathsf{P} \subseteq \mathsf{SIZE}[\mathsf{poly}]$.

(We use $\mathsf{SIZE}[\mathsf{poly}]$ to denote $\bigcup_c \mathsf{SIZE}[cn^c]$.)

§6.4.1 A P-complete problem

This construction — where we convert from an algorithm to a circuit — is extremely useful. In particular, it lets us talk about P-complete problems in a natural way. Here we're using log-space reductions — so if a problem is P-complete, then it's in NL if and only if $\mathsf{NL} = \mathsf{P}$.

Definition 6.15 (CircEval)

- **Input:** (C, a), where C is a Boolean circuit with 1-bit output and a is an input to C.
- **Decide:** whether $C(a) = 1$.

Theorem 6.16

The problem CircEval is P-complete under log-space reductions — i.e., $\mathsf{CircEval} \in \mathsf{P}$, and for every $f \in \mathsf{P}$ we have $f \leq_{\mathsf{L}} \mathsf{CircEval}$.

Proof. To see that $\mathsf{CircEval} \in \mathsf{P}$, we can simply do dynamic programming to figure out the value at each gate when we run C on a. (We just follow our nose — there's always some gate in the circuit we haven't evaluated whose two inputs we *have* evaluated, and we can keep evaluating such gates until we're done.)

Now we'll show that CircEval is P-hard. The idea is to stare at the proof of Proposition 6.13 and realize that it's actually a log-space reduction. More precisely, let $\mathcal{M}$ be a polynomial-time one-tape Turing machine. Then we need to show $L(\mathcal{M}) \leq_{\mathsf{L}} \mathsf{CircEval}$ (where $L(\mathcal{M})$ is the language it decides). So given x, we want to find a pair (C, a) such that $C(a) = 1$ if and only if $\mathcal{M}$ accepts x. So we simply output the pair $(C_{|x|}, x)$, where $C_{|x|}$ is the polynomial-size circuit for $\mathcal{M}$ on $|x|$-bit inputs as given by Proposition 6.13. (Here $\mathcal{M}$ is hardcoded into our reduction.)

To see that this is a log-space reduction, the point is that the construction in Proposition 6.13 is very local, and we're doing the same thing at each cell. We can hardcode into our reduction what the function ψ is (since it only depends on $\mathcal{M}$). Then in order to print the circuit, we first print all the hardcoded stuff in the first row; and then we iterate over all i and j and repeatedly print the circuit computing ψ on the appropriate inputs. (We essentially only need to keep track of the current values of i and j, so we can do this in logarithmic space.) □

Corollary 6.17

We have $\mathsf{NL} = \mathsf{P}$ if and only if $\mathsf{CircEval} \in \mathsf{NL}$.

One interesting fact is that if our circuit only had **OR** gates (and no **AND** or **NOT** gates), then we could evaluate it in NL — this is really a graph connectivity problem, where we want to know whether there's a path from a 1 to the output node. And if we only had **AND** gates, then we could evaluate it in NL as well (we could naturally evaluate it in coNL, and coNL = NL). If we had a layer of **AND**s followed by a layer of **OR**s, this is *also* in NL (e.g., because it's in $\mathsf{NL}^{\mathsf{NL}}$). But if we allow *arbitrary* alternation, then we run into problems (and evaluation becomes P-complete instead).

§6.5 Algorithms with advice

Today we'll look at a generalization of the circuit model called *algorithms with advice* — this is another way of thinking about what's going on when we have a separate circuit for each input length. This will lead to the class P/poly. We'll then prove the Karp–Lipton theorem, which relates the question of NP vs. P/poly to *uniform* classes. And finally, we'll see how it can be used to prove circuit lower bounds.

In the standard computation models we've seen, we have the following setup: we have a function f that we want to compute, and we say we've computed it if there's an algorithm $\mathcal{A}$ such that for all inputs x we have $\mathcal{A}(x) = f(x)$. We call this a *uniform* computation model because $\mathcal{A}$ is fixed once and for all — it's supposed to work regardless of the length of the input.

But this isn't the only way we might be able to solve hard problems. Modern software somehow keeps increasing in size over time — as we handle more and more data, there's always some new update that makes it bigger and bigger. So we don't have a fixed algorithm that works forever regardless of how big the input gets — instead, we have a program that somehow scales with the input size.

So we imagine that we've got polynomial time to compute f, but on inputs of length n, we also get some program with $p(n)$ lines of code — as our input length gets larger and larger, we get programs of size $p(n)$ rather than a fixed program. (Here $p(n)$ could be polynomial in n, or it could be much smaller than n. It could even be constant — this is still different from the uniform model, because we're still allowed a separate program for each input length, just of constant size.)

This is pretty different from the uniform model we're familiar with. The circuit model does something like this — we have a different circuit for each input length. But this model is even more general, since $p(n)$ could potentially be $\log n$ or $\mathsf{polylog}(n)$ or so on.

> **Question 6.18.** What problems can we solve in this model — where we allow the size of our program to grow with the input length?

To formalize this, we use the following definition.

> **Definition 6.19.** Given $s\colon \mathbb{N} \to \mathbb{N}$ and a decision problem f, we say an algorithm $\mathcal{A}$ decides f with $s(n)$ advice if for all n, there exists $a_n \in \{0,1\}^*$ of length $|a_n| = s(n)$ such that for all $x \in \{0,1\}^n$, we have $f(x) = \mathcal{A}(x, a_n)$.

So we've got a fixed algorithm $\mathcal{A}$, and for each input length n, it gets a special advice string — this advice strin can be different for each value of n, and when we're computing, we sort of stick it in along with $\mathcal{A}$. One way to view this is to think of $\mathcal{A}$ as some sort of 'program simulator' and a_n as the 'program' you run on n-bit inputs.

> **Remark 6.20.** This definition looks kind of similar to that of NP in some sense, but here the existential quantifier on a_n is *prior* to seeing the input — it's a sort of witness that's supposed to give you the right answer on *all* inputs x of length n (whether that right answer is 0 or 1).

§6.5.1 The class P/poly

We'll now define complexity classes based on the model of algorithms with advice.

Definition 6.21. For $s\colon \mathbb{N} \to \mathbb{N}$, we define $\mathsf{P}/s(n)$ as the set of decision problems f such that f can be decided by a polynomial-time algorithm $\mathcal{A}$ with $s(n)$ advice.

For example, last class (in Proposition 6.12) we showed that there is an undecidable problem $f \in \mathsf{SIZE}[O(1)]$ — we defined f by taking an undecidable problem and encoding it in 'unary' so that for every input length, f is either the all-0's or all-1's function. Then we also have $f \in \mathsf{P}/1$ (we can encode whether f on inputs of length n is the all-0's or all-1's function with a single bit, and take that to be our advice).

The class that people often look at, which is very natural, is the class of polynomial-time algorithms with polynomial-sized advice — so we let the program size grow *polynomially* with the length of the input.

Definition 6.22. We define $\mathsf{P}/\mathsf{poly} = \bigcup_k \mathsf{P}/(kn^k)$.

It turns out that this class corresponds exactly to the problems solvable with polynomial-sized circuits.

Theorem 6.23

We have $\mathsf{P}/\mathsf{poly} = \mathsf{SIZE}[\mathsf{poly}]$.

The proof isn't too hard — we just use the facts we've seen so far regarding converting between algorithms and circuits (from Subsection 6.4).

Proof. For one direction, we need to show that any problem with a polynomial-sized circuit family can be solved in P/poly; it's enough to show that for all k we have $\mathsf{SIZE}[kn^k] \subseteq \mathsf{P}/\mathsf{poly}$ (then we can take a union over k). Consider any function in $\mathsf{SIZE}[kn^k]$, and let $\{C_n\}$ be a circuit family computing it with $\mathsf{size}(C_n) = kn^k$. Then for each n, we simply take a_n to be a description of C_n (as seen in the proof of Claim 6.7, we can encode a circuit of size kn^k with $O(n^k \log n)$ bits). And we define $\mathcal{A}$ to be an algorithm computing $\mathsf{CircEval}$ — i.e., so that $\mathcal{A}(x, a_n) = \mathsf{CircEval}(a_n, x)$. (We saw last class (in Theorem 6.16) that $\mathsf{CircEval} \in \mathsf{P}$, so $\mathcal{A}$ is a polynomial-time algorithm.)

Now we'll show the reverse direction — that for all c and d, we have $\mathsf{TIME}[n^c]/(dn^d) \subseteq \mathsf{SIZE}[\mathsf{poly}]$. Suppose we start with some algorithm $\mathcal{A}$, which for simplicity we'll assume is a 1-tape Turing machine running in time $O(n^c)$ on inputs of length n, and we've got a sequence of advice strings $\{a_n\}$ with $|a_n| \leq dn^d$ for all n. Then we'll use the same tableau-to-circuit reduction from Proposition 6.13 to build from $\mathcal{A}$ a circuit C_n for each input length n — as in the proof of Proposition 6.13, we've got a big table where we feed in x and a_n at the top, and then we compute a fixed function ψ in every cell, and in the end we check whether the bottom-left corner has q_{acc}. And the key point is that we can just hardcode the advice a_n into our circuit; then this gives a circuit simulating $\mathcal{A}$ on any given x with the appropriate advice.

And the size of this circuit is the square of the runtime of $\mathcal{A}(x, a_n)$. And $\mathcal{A}$ runs in time $O(n^c)$ on inputs of length n; here our input is (x, a_n), where x has length n and a_n has length dn^d, so we get that this circuit has size $O((n + dn^d)^{2c}) = \mathsf{poly}(n)$. □

Remark 6.24. Usually in the literature $\mathsf{SIZE}[\mathsf{poly}]$ is called P/poly everywhere, because it doesn't make a difference. But if we care in a more fine-grained way about the runtime vs. advice length, then the definition of algorithms with advice captures more than what we could capture with just circuit classes.

§6.5.2 NP vs. P/poly

Even if $\mathsf{P} \neq \mathsf{NP}$, it could still be true that $\mathsf{NP} \subseteq \mathsf{P/poly}$ — it could be that you could cook up nice, efficient circuits to solve SAT on each input length. Still, we think this is *not* true. (If it were true, it would still be quite a revolution in SAT-solving — for example, you could design a polynomial-sized piece of hardware that solves SAT with a million variables.)

> **Conjecture 6.25** — We have $\mathsf{NP} \not\subseteq \mathsf{P/poly}$.

Proving this conjecture seems much harder than proving that $\mathsf{P} \neq \mathsf{NP}$, because P/poly can even solve some undecidable problems — so it's not even clear where to start. Nevertheless, lots of approaches to trying to prove $\mathsf{P} \neq \mathsf{NP}$ tried to prove this instead. The intuition was that Turing machines are complicated, but we might be able to understand circuits by looking at them one gate at a time. Unfortunately, this doesn't work — people found that we don't understand circuits either. This makes sense becaues circuits can simulate Turing machines (as seen in Proposition 6.13), so going from Turing machines to circuits really just makes the problem harder.

Another thing to note is that all the notions of completeness work for P/poly as well — for example, if we show a single NP-complete problem is in P/poly, it follows that $\mathsf{NP} \subseteq \mathsf{P/poly}$. The point is that any polynomial-time reduction can be converted to a polynomial-sized circuit — so if we have a problem $g \in \mathsf{NP}$ and a reduction from g to f, and we manage to get a polynomial-sized circuit for f, then we can make a polynomial-sized circuit for our reduction as well, and putting them together gives a circuit for g. So all the things we've seen about NP-completeness with regards to NP vs. P also hold for NP vs. P/poly.

§6.6 The Karp–Lipton theorem

The problem of NP vs. P/poly is probably very difficult, but we'd still like to get a handle on it in some way. One way to get a handle on P vs. NP is to assume $\mathsf{P} = \mathsf{NP}$ and see what consequences we get — for example, we saw in Theorem 2.44 that $\mathsf{P} = \mathsf{NP}$ implies $\mathsf{P} = \mathsf{PH}$. By an analogous argument, we can show that if $\mathsf{NP} \subseteq \mathsf{P/poly}$ then $\mathsf{PH} \subseteq \mathsf{P/poly}$ (now we've got algorithms with advice instead of just algorithms, but the same proof still goes through).

But this just says that if we can solve uniform stuff with non-uniform algorithms, then more uniform stuff can also be solved with non-uniform algorithms. What would be really interesting is if we could show something like if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\mathsf{P} = \mathsf{NP}$ – this would mean that if we could cook up non-uniform algorithms for NP-complete problems, then we could somehow get rid of the non-uniformity and actually get a nice polynomial-time algorithm.

We don't know how to show this statement, but we *do* have results along these lines — that if a uniform class can be simulated in a non-uniform way, then we get interesting consequences in the uniform world.

> **Theorem 6.26** (Karp–Lipton 1980)
> If $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\mathsf{PH} = \Sigma_2\mathsf{P}$.

So we can't necessarily say that $\mathsf{NP} \subseteq \mathsf{P/poly}$ means that the whole polynomial hierarchy collapses to P (which is what we'd really like), but we *can* say it collapses to $\Sigma_2\mathsf{P}$. This theorem has been improved — $\Sigma_2\mathsf{P}$ has been replaced with some more esoteric characters in the complexity zoo (which we are not going to define in this course). But we don't know whether we can replace it with P^{NP} — this is a huge open problem, and it turns out that if you could prove this, you'd prove new circuit lower bounds (there's some kind of deep connection between such theorems and constructing functions with high circuit complexity, which we'll see later in this lecture).

We'll now prove the Karp–Lipton theorem.

§6.6.1 The solution printer lemma

A major component of the proof is a concept called *self-reducibility* — that if we have some kind of algorithm that can decide SAT (or other NP problems), then we can use it to construct one that actually prints *solutions* to NP problems. For example, we saw the following fact on the first problem set.

> **Fact 6.27** — If SAT $\in$ P, then there is a polynomial-time algorithm that, given a formula φ, prints a satisfying assignment (if one exists).

This is often called *search to decision*. And a major component of the proof of Theorem 6.26 is some kind of search to decision, but for P/poly.

Lemma 6.28 (Solution printer lemma)
Assume that $\mathsf{NP} \subseteq \mathsf{P/poly}$. Then for every polynomial-time verifier $\mathcal{V}$ and every $k \geq 1$, there exists a polynomial-sized circuit family $\{C_n\}$ where C_n has kn^k outputs such that for every input x of length n, if there exists y with $|y| = kn^k$ and $\mathcal{V}(x, y) = 1$, then $\mathcal{V}(x, C_n(x)) = 1$.

In words, what this says is that if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then given a verifier $\mathcal{V}$ and some polynomial bound kn^k on the witness length, we can come up with a polynomial-time circuit family that, just given x, will actually print a witness that makes the verifier accept. (Here $C_n(x)$ is the witness that the circuit prints.) Note that this is an if and only if — i.e., there exists y with $\mathcal{V}(x, y) = 1$ if and only if $\mathcal{V}(x, C_n(x)) = 1$ — since if the circuit can print something, then it certainly exists.

Proof. The idea is that given a verifier $\mathcal{V}$ (and k), we define a function $g(x, y)$ that checks whether y is a *prefix* for a good witness — i.e., $g(x, y)$ is 1 if and only if there exists z of length $kn^k - |y|$ such that $\mathcal{V}(x, yz) = 1$. So g essentially answers the question, given x and a prefix y, is there a way to complete y to a witness for x?

And $g \in \mathsf{NP}$, so the hypothesis means $g \in \mathsf{P/poly}$ — so we've got a polynomial-sized circuit that tells us, given a prefix, whether there's a way to complete it.

And then we first stick in $y = 0$ and $y = 1$ to see what this circuit says. If it rejects both, then there's no way to complete either to a good witness, so there *is* no good witness, and we can just reject. Meanwhile, if one accepts, then we take it as our prefix. And then we stick in another bit — for example, if we took 0 as our prefix, then we run the circuit on 00 and 01. And we keep on doing this — repeatedly extending our prefix by one bit and using the circuit for g to check that it can still be completed to a good witness — until we get the full witness. $\square$

§6.6.2 Proof of the Karp–Lipton theorem

Now we'll use the solution printer lemma to prove the Karp–Lipton theorem (Theorem 6.26). We'll actually prove the following statement.

Lemma 6.29
If $\mathsf{NP} \subseteq \mathsf{P/poly}$, then $\Sigma_2\mathsf{P} = \Pi_2\mathsf{P}$.

This suffices to imply Theorem 6.26, because it means that if we have an $\exists\forall[\cdots]$ (where $[\cdots]$ is some polynomial-time predicate), then we can convert it into $\forall\exists[\cdots]$ (and vice versa). And so if we take anything in $\Sigma_3\mathsf{P}$, which can be written in the form $\exists\forall\exists[\cdots]$, then we can swap the final $\forall\exists$ to an $\exists\forall$ to get something of the form $\exists\exists\forall[\cdots] = \exists\forall[\cdots]$. And we can show that $\Sigma_4\mathsf{P} = \Sigma_3\mathsf{P}$ similarly, and so on — we just keep

taking the final two quantifiers and flipping them, which lets us remove one quantifier at a time. So once we've got $\Sigma_2\mathsf{P} = \Pi_2\mathsf{P}$, any chain $\exists\forall\exists\forall\cdots$ comes crashing down to just $\exists\forall$.

Proof. We'll show $\Pi_2\mathsf{P} \subseteq \Sigma_2\mathsf{P}$ (the reverse inclusion then follows by taking complements). Consider some function $f \in \Pi_2\mathsf{P}$, so there's some polynomial-time algorithm $\mathcal{A}$ such that

$$f(x) = 1 \iff (\forall\, y_1)(\exists\, y_2)[\mathcal{A}(x, y_1, y_2) = 1]$$

(where y_1 and y_2 both have polynomial length). Now we're going to look at the part $(\exists\, y_2)[\mathcal{A}(x, y_1, y_2) = 1]$ and collapse it into a NP-verifier — so we define a verifier $\mathcal{V}$ which takes (x, y_1) as its input and y_2 as its witness, and then runs $\mathcal{A}$ and outputs the answer, i.e.,

$$\mathcal{V}((x, y_1), y_2) = \mathcal{A}(x, y_1, y_2).$$

Now the solution printer lemma (Lemma 6.28) tells us that because $\mathsf{NP} \subseteq \mathsf{P/poly}$, there's a polynomial-sized circuit family such that there *exists* y_2 with $\mathcal{V}((x, y_1), y_2) = 1$ if and only if $C_m(x, y_1)$ *outputs* some y_2 such that $\mathcal{A}(x, y_1, y_2)$ accepts (where $m = |x| + |y_1| = \mathsf{poly}(n)$). In other words, we have a circuit that takes in x and y_1, and *prints* a good y_2 (i.e., a good solution to the last existential quantifier).

And then we can say that

$$f(x) = 1 \iff (\exists\, C_m)(\forall\, y_1)[\mathcal{A}(x, y_1, C_m(x, y_1)) = 1].$$

What we've done here is that we guess the solution printer at the start (before seeing y_1 or y_2), and then we see y_1 and use the solution printer to get y_2. And we've now got an $\exists\forall$ statement, showing $f \in \Sigma_2\mathsf{P}$. □

Remark 6.30. There are generalizations of Karp–Lipton for larger complexity classes, but they're trickier to prove — for example, if $\mathsf{PSPACE} \subseteq \mathsf{P/poly}$, then we could get $\mathsf{PSPACE} = \Sigma_2\mathsf{P}$. There's a similar statement for EXP — in fact, there's even such a connection for NEXP, and this is a crucial component in certain connections between CircuitSAT and circuit lower bounds.

§6.7 Some circuit lower bounds

We'll now see that using the Karp–Lipton theorem, we can prove circuit lower bounds for $\Sigma_2\mathsf{P}$.

We'll start with a fact about constructing hard functions for circuits.

Question 6.31. Are there 'easy' functions f such that $f \notin \mathsf{SIZE}[n^{10}]$ — for example, is $\mathsf{P} \not\subseteq \mathsf{SIZE}[n^{10}]$?

Most people's intuition is that there should be functions in e.g. $\mathsf{TIME}[n^{100}]$ that can't be solved with circuits of size n^{10} — we can't simulate all n^{100}-time algorithms even in time n^{99} by the time hierarchy theorem, so it'd be surprising if non-uniformity allowed us to go from n^{100} to n^{10}. But non-uniformity is weird, so this is hard to prove. However, we *can* make the following weaker statement.

Lemma 6.32

For all k, there is some $f_k \in \mathsf{PH}$ with $f_k \notin \mathsf{SIZE}[kn^k]$.

So there's no universal k such that every function in the polynomial *hierarchy* has circuits of size kn^k.

Remark 6.33. Note that this does *not* mean $\mathsf{PH} \not\subseteq \mathsf{P/poly}$ — that would mean there *exists* a function $f \in \mathsf{PH}$ such that for *all* k, we have $f \notin \mathsf{SIZE}[kn^k]$. So the quantifiers are permuted, which makes an enormous difference — Lemma 6.32 is not too hard, and $\mathsf{PH} \not\subseteq \mathsf{P/poly}$ is a major open problem.

Given this and the Karp–Lipton theorem, we can already prove a circuit lower bound for $\Sigma_2\mathsf{P}$.

Theorem 6.34 (Kannan 1981)
For all k, there exists $f_k \in \Sigma_2\mathsf{P}$ with $f_k \notin \mathsf{SIZE}[kn^k]$.

Proof. First, if $\mathsf{NP} \not\subseteq \mathsf{P/poly}$, then $\mathsf{SAT} \notin \mathsf{P/poly}$, so we can just take $f_k = \mathsf{SAT}$ for all k.

Meanwhile, if $\mathsf{NP} \subseteq \mathsf{P/poly}$, then by the Karp–Lipton theorem PH collapses to $\Sigma_2\mathsf{P}$, which means the hard function f_k from Lemma 6.32 is actually in $\Sigma_2\mathsf{P}$. □

This proof is sort of annoying — it's super non-constructive and conditioned on something we're presumably not going to resolve in our lifetimes. Nevertheless, people have made this argument sort of constructive (using weird functions that interpolate between SAT and the hard function from Lemma 6.32, which we *will* describe constructively).

Now we'll prove Lemma 6.32 — in fact, our construction will give functions $f_k \in \Sigma_3\mathsf{P}$.

Proof of Lemma 6.32. Fix k. We want to describe a PH algorithm and argue that the function it compute doesn't have small circuits. And the main idea is to try to *guess* a hard function (and then use our quantifiers to check that it really is hard).

As a starting point, by Proposition 6.6, for all large ℓ there exists a function $g_\ell \colon \{0,1\}^\ell \to \{0,1\}$ without circuits of size $2^\ell/\ell^2$. (We could replace this with $\varepsilon \cdot 2^\ell/\ell$, but this weaker bound is sufficient.) Our goal is to somehow isolate such a function using an algorithm in PH. We're eventually going to take ℓ logarithmic in n — specifically, $\ell \approx (k+1)\log n$ — so that g_ℓ doesn't have circuits of size $n^{k+1}/(\log n)^2 > kn^k$. If we can get our hands on such a function g_ℓ, then we can simply have f_k mimic it (we have to be a bit careful because g_ℓ only takes $(\log n)$-bit inputs, while f_k is supposed to take n-bit inputs — but we can handle this just by padding).

First, we can simply guess the truth table of g_ℓ (i.e., the string describing what g_ℓ does on every possible input) — tihs truth table has roughly n^{k+1} bits, so this is a polynomial-length guess.

Now we want to check that our guessed function g_ℓ really is hard, meaning that it doesn't have circuits of size kn^k. We can do this with more quantifiers — we check that *for all* circuits C of size $2^\ell/\ell^2$ with ℓ inputs, C fails to compute g_ℓ. And we can check that C fails to compute g_ℓ in polynomial time — this means there exists $z \in \{0,1\}^\ell$ with $C(z) \neq g_\ell(z)$, and since there's only $2^\ell = \mathsf{poly}(n)$ possible values of z, we can just enumerate over all of them (i.e., we don't need an $\exists$ quantifier for this).

So far, we've guessed a function g_ℓ and verified that it really is hard. But we're not done — the problem is that what we really want is to ensure that $f_k(0^{n-\ell}y) = g_\ell(y)$ for all inputs y (i.e., f_k computes exactly a padded version of g_ℓ), so that if g_ℓ doesn't have small circuits, then neither does f_k (since if f_k did have small circuits, then we could plug in a bunch of 0's to get small circuits for g_ℓ). But right now, the issue is that we haven't guaranteed we're getting the same function g_ℓ on all inputs — we might be guessing a different hard function g_ℓ for different inputs x of length n, and then this argument won't work (since we haven't isolated a particular function independent of the input).

So to fix this issue, we're not just going to ask that g_ℓ is hard (i.e., it doesn't have $2^\ell/\ell^2$-sized circuits) — we're going to ask that it's the *lexicographically first* hard function (so that we really do get the same g_ℓ on every input).

To ensure this, we need to check that all lexicographically smaller functions actually have smaller circuits. So we universally quantify over all functions $h \colon \{0,1\}^\ell \to \{0,1\}$ with $h \prec g_\ell$ (in the lexicographic order on 2^ℓ-bit strings). And then we existentially guess a circuit D of size at most $2^\ell/\ell^2$ and verify that D really does compute h — i.e., that $D(z) = h(z)$ for all z (again, we don't need a quantifier for this — there are $2^\ell = \mathsf{poly}(n)$ possible values of z, so we can go through all of them one at a time). This ensures that all $h \prec g_\ell$ do have small circuits, so our function g_ℓ really is the first hard function (in lexicographical order).

So now we've got a function g_ℓ which is hard and doesn't depend on x (it only depends on $n = |x|$). And then we're done — if x is of the form $0^{n-\ell}y$ then we output $g_\ell(y)$, and otherwise we output 0. This means for all y of length ℓ, we have $f_k(0^{n-\ell}y) = g_\ell(y)$ (which is what we wanted).

And our quantifiers here are $(\exists\, g_\ell)(\forall\, C, h)(\exists\, D)$, so we've got an $\exists\forall\exists$ procedure to compute f_k, showing that $f_k \in \Sigma_3\mathsf{P}$.

Finally, we'll check more carefully that $f_k \notin \mathsf{SIZE}[kn^k]$. Suppose that it is, and let $\{C_n\}$ be a circuit family for f_k with $\mathsf{size}(C_n) \leq kn^k$. Then we can choose n large enough such that $kn^k < n^{k+1}/((k+1)\log n)^2 = 2^\ell/\ell^2$; then we've got $C_n(0^{n-\ell}y) = g_\ell(y)$ for all $y \in \{0,1\}^\ell$. But $C_n(0^{n-\ell}y)$ has size $kn^k < 2^\ell/\ell^2$, contradicting the fact that g_ℓ is supposed to require circuits of size greater than $2^\ell/\ell^2$. □

§6.7.1 Some extensions

This idea is really useful for many other lower bounds. For example, we can use the same idea to show exponential-sized circuit lower bounds, such as the following.

Theorem 6.35
There exists $\varepsilon > 0$ and $f \in \mathsf{EXPSPACE}$ such that $f \in \mathsf{SIZE}[\varepsilon \cdot 2^n/n]$.

Note that in the proof of Lemma 6.32 we could have used PSPACE instead of $\Sigma_3\mathsf{P}$ — we could have enumerated over truth tables and circuits instead of using quantifiers (but the statement with quantifiers is stronger); this is how we get $\mathsf{EXPSPACE}$ here. In fact, there's even a notion of $\Sigma_3\mathsf{EXP}$, and we can get that as well (and we'll slightly improve this on our problem set).

There was a major advance last year due to Chen–Hirahara–Ren and Z. Li, who gave a Σ_2-procedure for such things. In fact, they showed how to solve a problem called RangeAvoidance, which is a very interesting problem — very roughly speaking, imagine we've got a box B that takes n inputs and $n+1$ outputs. Our task is to find some y outside the range of the box — i.e., such that for all x, we have $B(x) \neq y$. The box has more outputs than inputs, so there has to *exist* such a y — there's 2^{n+1} possible strings y and at most 2^n are in the range of B — and in fact, with randomness this problem is not too hard (if we sample a random string y, we'll get something outside the range of the box at least half the time). But the question of whether there's an efficient *deterministic* algorithm for this is very hard. And the authors gave a Σ_2-procedure, which has lots of implications for circuit lower bounds and cryptography.

Another big open problem is to replace $\Sigma_2\mathsf{P}$ in Theorem 6.34 with something smaller.

Open question 6.36. Can we show $\mathsf{P}^{\mathsf{NP}} \not\subseteq \mathsf{SIZE}[kn^k]$? Even more ambitiously, can we show this of P?

We'd expect this to be true because of the time hierarchy theorem — we don't think the translation from algorithms to circuits is so inefficient that you can always compress circuits (for problems with arbitrary polynomial-time algorithms) to have a *fixed* polynomial size. But on the other hand, we've shown that this kind of question is equivalent to the question of improving the Karp–Lipton theorem — for example, showing that if $\mathsf{NP} \subseteq \mathsf{P/poly}$ then PH collapses to another class $\mathcal{C}$ (which is very close to P^{NP}). So in some sense, the weird roundabout way of proving lower bounds we saw in Theorem 6.34 seems 'equivalent' to proving a lower bound in the first place — if we prove a lower bound, then we can't help but get a collapse theorem.

In contrast, what's the best we know regarding the question for P? If we allow $\oplus$, $\vee$, and $\wedge$ gates, then the best-known statement is that there exists $f \in \mathsf{P}$ with $f \notin \mathsf{SIZE}[3.1n]$. (This was a paper in STOC two years ago, and it took a lot of work; before this the constant used to be 3.01, and before that the constant had been 3 for 30 years.) Meanwhile, if we ban $\oplus$, then we can improve the bound to $5n$ (and this hasn't improved since 2005). So the current approaches are very far from being able to prove something like $\mathsf{P} \not\subseteq \mathsf{SIZE}[kn^k]$.

§7 Randomized complexity

Today our main topic is randomized complexity; we'll begin by formalizing the computation model.

§7.1 Randomized computation

We can think of randomized computation as a nondeterministic model, but with different conditions on acceptance. A *nondeterministic* algorithm has multiple computation paths and accepts if *at least one* computation path goes to an accept state, and a *conondeterministic* algorithm accepts if *all* paths go to an accept state. Meanwhile, what we want from a *randomized* algorithm is that it should accept if *most* paths go to an accept state and reject if most paths go to a reject state, and moreover, one of these two cases should always happen (i.e., on all inputs, either a large fraction of the threads accept or a large fraction reject, with nothing in between).

We'll actually define our randomized computation model as a simple extension of a deterministic model, where we allow some number of random coin tosses — so on an input x of length n, we flip $r(n)$ coins to get bits $c_1, \ldots, c_{r(n)} \in \{0, 1\}$ (which are uniform and mutually independent). And then we run a *deterministic* algorithm $\mathcal{A}(x; c_1 \ldots c_{r(n)})$ on our input and the outcomes of all these coin tosses. We define

$$\mathbb{P}[\mathcal{A} \text{ accepts } x] = \mathbb{P}_{c_1,\ldots,c_r}[\mathcal{A}(x; c_1 \ldots c_r) = 1]$$

(i.e., the number of sequences of coin tosses that $\mathcal{A}$ accepts, divided by 2^r).

Definition 7.1. We say a randomized algorithm $\mathcal{A}$ computes f with $r(n)$ randomness and probability p if for all $x \in \{0, 1\}^*$ (letting $r = r(|x|)$), we have

$$\mathbb{P}_{c_1,\ldots,c_r}[\mathcal{A}(x; c_1 \ldots c_r) = f(x)] \geq p.$$

There's a certain sense in which we have to be careful about this kind of definition — for example, if we set $p = \frac{1}{2}$, then the definition is meaningless.

Fact 7.2 — For every decision problem f, there is a constant-time randomized algorithm that computes f with probability $\frac{1}{2}$.

Proof. There's only two possible inputs — 0 and 1 — so we can simply ignore the input x and just flip a coin to give a random output, and this will agree with f with probability $\frac{1}{2}$. □

So this means that to some extent, we have to be careful with what probabilities we stick into the definition. But it turns out that any probability sufficiently far away from $\frac{1}{2}$ is sufficient (i.e., they're all equivalent for the purposes of defining polynomial-time randomized computation, as we'll see soon).

Definition 7.3. We say a randomized algorithm $\mathcal{A}$ computes f in time $O(t(n))$ if:

- $\mathcal{A}$ computes f with $O(t(n))$ randomness and with probability at least $\frac{2}{3}$.
- For all inputs x of length n and coin tosses c of length $O(t(n))$, $\mathcal{A}(x; c)$ halts in $O(t(n))$ steps.

(Reading one random bit should cost you a step, so we don't allow more than $O(t(n))$ bits of randomness.)

This definition guarantees that on every single input x, there's two possibilities for what $\mathcal{A}$ looks like (if we enumerate over all 2^r possible sequences of coin tosses in a tree) — either at least $\frac{2}{3}$ of these coin tosses make $\mathcal{A}$ print 1 (in the case $f(x) = 1$), or at most $\frac{1}{3}$ do (in the case $f(x) = 0$). The choice of $\frac{2}{3}$ is mostly arbitrary — it's just a constant greater than $\frac{1}{2}$ that we use for concreteness, but any such constant would work thanks to *error reduction*.

Lemma 7.4 (Error reduction)

Suppose a randomized algorithm $\mathcal{A}$ computes f in time $t(n)$ with probability $\frac{1}{2} + \delta$ (for some $\delta > 0$). Then for every (efficiently computable) $e\colon \mathbb{N} \to \mathbb{N}$, there exists an algorithm $\mathcal{B}$ that computes f in time $O(t(n)e(n)\delta^{-2})$ and with probability at least $1 - 2^{-e(n)}$.

(In fact, δ is also allowed to depend on n — it doesn't have to be a constant.)

In words, we imagine that we've got an algorithm that computes f with probability only slightly better than the trivial algorithm from Fact 7.2 (of just flipping a coin and outputting its result). Then for any error parameter $e(n)$, we can mutiply our runtime by $e(n)\delta^{-2}$ and get our probability of being correct *exponentially* close to 1 (in $e(n)$).

Proof. We simply run a bunch of independent trials of $\mathcal{A}$ and output their majority — more explicitly, the new algorithm $\mathcal{B}$ works by choosing $k = O(e(n)\delta^{-2})$ independent random strings $r_1, \ldots, r_k$, and then outputting $\mathsf{Majority}(\mathcal{A}(x, r_1), \ldots, \mathcal{A}(x, r_k))$. (So we compute $\mathcal{A}$ with each of these choices of randomness, and then output the more common answer.) The reason this works is Chernoff bounds. □

Remark 7.5. In the model, we always think of the coins as being on a separate tape as the input (otherwise if we just had a 1-tape Turing machine, we'd get a linear overhead from having to jump between the input and coins). So we have one tape with x, one tape with the coins, and one tape for our storage.

Finally, we'll define the corresponding complexity classes.

Definition 7.6. We define $\mathsf{BPTIME}[t(n)]$ as the set of functions f which can be computed by a randomized algorithm in time $O(t(n))$.

Here BP stands for *bounded-error probabilistic*; this corresponds to a 2-sided error model. (There's also R and coR, which only allow error on one side; we'll see these later.)

Definition 7.7. We define $\mathsf{BPP} = \bigcup_k \mathsf{BPTIME}[n^k]$.

§7.2 Polynomial identity testing

There are many problems which were first shown to be in BPP and were later shown to be in P. One of the most famous examples is primality testing (which was shown to be in BPP in the 1970s, and in P in the early 2000s). But there are some problems for which this is still open. One of the most famous examples of a problem in BPP which is still not known to be in P is the problem PIT (which stands for *polynomial identity testing*). It's a problem related to a topic we haven't discussed much called *algebraic complexity theory* — the idea is similar to circuit complexity, except that instead of computing circuits in $x_1, \ldots, x_n$ with $\vee$ and $\wedge$, we're trying to compute polynomials in $x_1, \ldots, x_n$ (and we want to know how many times we need to compute a $+$ or $\times$ to do so).

Definition 7.8 (PIT)

- **Input:** two formulas F and G over $x_1, \ldots, x_n$, consisting of constants in $\mathbb{Z}$ as well as the operations $+$ and $\times$.
- **Decide:** is $F \equiv G$ — i.e., do the two formulas compute the same polynomial?

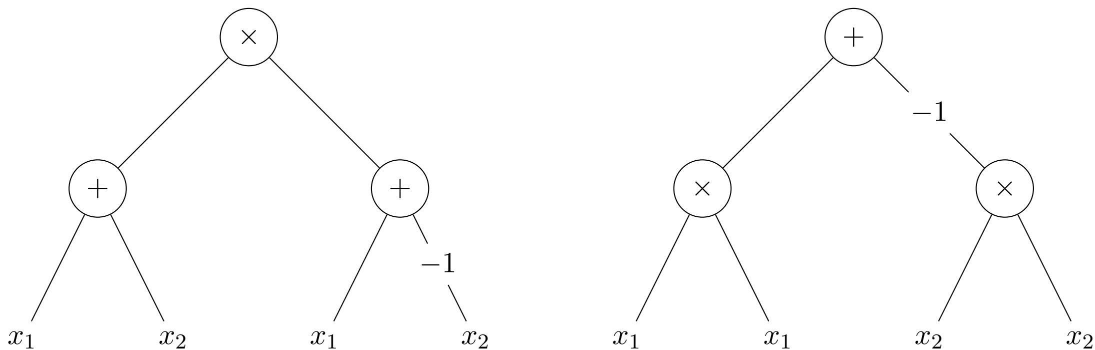
This question is pretty subtle — if instead of being over $+$ and $\times$ and computing polynomials, our formulas were over $\wedge$ and $\vee$ and $\neg$ and computed Boolean functions (and we wanted to know whether they computed the same Boolean function or not), then this problem would be coNP-complete (i.e., very difficult). In contrast, we'll see that PIT itself is in BPP (which means it's probably much easier).

Example 7.9

As an example of two equivalent (but not identical) formulas, we have

$$(x_1 + x_2)(x_1 + (-1)x_2) = x_1 \cdot x_1 + (-1)x_2 \cdot x_2.$$

We could imagine drawing out both sides as formulas (putting scalars on the wires) as follows.



We could imagine solving PIT deterministically by taking any formula and repeatedly expanding to write it in the form $\sum a \prod x_i^{e_i}$. But this could blow up the formula really quickly — if we started out with d binomials, then we could end up with 2^d terms — so it could take exponential time. So there isn't an obvious efficient *deterministic* algorithm for PIT. But it turns out that there is a *randomized* one.

Theorem 7.10

We have PIT $\in$ BPP.

The idea behind the proof is pretty simple — we'll just evaluate F and G on *random* inputs, and see if they give us the same answer. The big warhorse behind why this works is the following theorem (which is extremely important in randomized algorithms).

Definition 7.11. We say a polynomial has degree at most d if in its expansion as a sum of monomials, for each monomial $\prod x_i^{e_i}$ we have $\sum e_i \le d$.

Theorem 7.12 (Schwarz–Zippel)

Let $\mathbb{F}$ be a field (or $\mathbb{Z}$), and let S be any subset of $\mathbb{F}$. Suppose that $p(x_1, \ldots, x_n) \in \mathbb{F}[x_1, \ldots, x_n]$ is a polynomial which is not identically zero, and $\deg(p) \le d$. Then if we choose uniform random values $r_1, \ldots, r_n \in S$ to plug into p, we have

$$\mathbb{P}_{r_1,\ldots,r_n}[p(r_1, \ldots, r_n) = 0] \le \frac{d}{|S|}.$$

This theorem says that if we have a nonzero polynomial of bounded degree, then if we pick a sufficiently big set S from which to draw inputs, there's a large fraction of points from that set which *witness* that the polynomial is nonzero (meaning that when we plug them in, we get something nonzero). This is what makes

PIT very different from the setting of Boolean formulas. And this theorem is extremely important — there are tons of randomized algorithms that work by cooking up a cool polynomial and evaluating it on a cool set, and showing that these evaluations can be done quickly and the polynomial being nonzero translates to something interesting.

We won't prove Theorem 7.12; in the case $n = 1$ it's immediate from the fact that a polynomial of degree at most d (in one variable) has at most d roots. There are proofs of the general case in the books of Sipser and Arora–Barak, as well as on Wikipedia; there's several very lovely proofs.

Given this, we can get a simple randomized algorithm for PIT.

Proof of Theorem 7.10. Our algorithm works as follows: suppose F and G have size s (where we measure the size of a formula as the number of operations plus the number of occurrences of variables — the precise definition doesn't matter, as long as it's a reasonable proxy for input length). We then set $S = \{1, \ldots, 3s\}$, choose n values $r_1, \ldots, r_n \in S$ completely at random (which we represent by a single vector $r \in S^n$), and accept if and only if $F(r) = G(r)$.

We need to verify two things about this algorithm — that it runs in polynomial time, and that it gives the right answer with high probability. We'll start with the runtime.

First, the key observation is that a formula of size s corresponds to a polynomial of degree at most s (as each $+$ corresponds to taking the maximum of the two input degrees, and each $\times$ corresponds to adding the two degrees).

Then $F(r)$ and $G(r)$ will both be some integer at most $(3s \cdot an)^s$, where a is the maximum magnitude of their constants — this is because if we imagine expanding them out as a sum of monomials, since they have degree at most s, there's at most n^s monomials. For each of these monomials, we're plugging in numbers between 1 and $3s$, and the total exponent is at most s; so the greatest the monomial could possibly be is $(3s)^s$, and its coefficient is at most a^s.

This in particular means $F(r)$ and $G(r)$ take $O(s(\log n + \log s + \log a))$ bits to describe; and this bound is polynomial in the input length (we have $n \leq s$ if our size bound includes the number of variables, and the input length is at least s; it's also at least $\log a$, since the input includes a written in binary). The same is true for all the intermediate values we'll have while computing them; so we can compute $F(r)$ and $G(r)$ in polynomial time.

Now we'll prove that the algorithm is correct with high probability. First, if $F \equiv G$ then of course $F(r) = G(r)$ for all r, so the algorithm accepts with probability 1. Meanwhile, if $F \not\equiv G$, then $F - G$ is a nonzero polynomial of degree at most s. So by Schwarz–Zippel (Theorem 7.12), we have

$$\mathbb{P}[\text{accept}] = \mathbb{P}_r[(F - G)(r) = 0] \leq \frac{s}{3s} = \frac{1}{3}$$

(since our polynomial has degree at most s, and our domain S has size $3s$). In fact, this algorithm actually has a *one-sided* guarantee — we only have error in the case $F \not\equiv G$. □

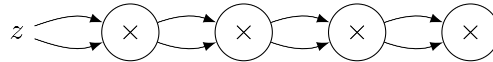
§7.2.1 An extension to arithmetic circuits

We can think of PIT as a problem of testing whether two arithmetic *formulas* are equivalent (the output of a computation can't be used in multiple places).

Question 7.13. What if we used arithmetic *circuits* instead of formulas?

If we use circuits instead of formulas, then we still have $+$ and $\times$ gates, but now we can reuse nodes as many times as we like.

First, what goes wrong in our algorithm for PIT when we replace formulas with circuits? The issue is that it's no longer true that the degrees are at most s, since we can reuse nodes — for example, if we start with a single variable z and keep squaring it by repeatedly sending two copies to a $\times$ gate, then we end up with a polynomial of degree 2^s.



In particular, this means our numbers get way too big, so we can't do computations with them efficiently (e.g., we can't necessarily evaluate $F(r)$ and $G(r)$ in polynomial time).

At a high level, here's a sketch of how to patch this (to get an algorithm that works for PIT with circuits). Now F and G have degree at most roughly 2^s, so the upper bound we get on $F(r)$ and $G(r)$ is roughly 2^{2^s}. Numbers this big are way too big to compute with, but the point is that they don't have too many prime factors — so we're going to pick a random prime p and compute everything mod that prime (and there'll be a good chance that this prime doesn't divide $F(r) - G(r)$).

First, as before, we take $S = \{1, \dots 2^{3s}\}$ (chosen to have much larger size than the bound on degrees, which is 2^s) and sample $r_1, \dots, r_n \in S$ uniformly and independently (and let $r \in S^n$ be the vector encoding all of them). Then we'll still have $F(r) \neq G(r)$ with decent probability.

But instead of computing $F(r)$ and $G(r)$ directly (which we can't do, because they're too big), we choose a prime $p \in [2, 2^{s^2}]$ (we can do this by randomly sampling numbers and testing whether they're prime; it should take $O(s^2)$ trials to find a prime by the prime number theorem). And then we compute $F(r)$ and $G(r)$ mod p and test whether $p \mid F(r) - G(r)$. All the values in our computation are at most $p \leq 2^{s^2}$, so they take $O(s^2)$ bits to write down, and we really can compute $F(r)$ and $G(r)$ mod p in polynomial time.

And the reason this works is that if $F(r) \neq G(r)$ and $|F(r) - G(r)| \leq 2^{2^s}$ (this bound isn't actually true, but it's qualitatively similar to the true bound), then $F(r) - G(r)$ has at most 2^s prime factors. So since we're choosing p from a set of primes much larger than this, then no matter what r is (as long as $F(r) \neq G(r)$, which is true with high probability), we have $F(r) \not\equiv G(r) \pmod{p}$ with high probability.

§7.3 BPP vs. some other complexity classes

Now we'll discuss where BPP fits within all the other complexity classes. We'll start with a simple fact.

Fact 7.14 — We have BPP $\subseteq$ PSPACE.

Proof. We can just iterate over all poly(n) strings of coin tosses, stick them into our deterministic algorithm $\mathcal{A}(x; c)$ one at a time, and count the number that accept and reject (and then choose whether to accept or reject based on how big these numbers are), □

Here's a more interesting theorem.

Theorem 7.15
We have BPP $\subseteq$ P/poly.

Proof. Consider some $f \in$ BPP. Then by error reduction, there exists an algorithm $\mathcal{A}$ and a polynomial p such that $\mathcal{A}$ computes f in $p(n)$ time using $p(n)$ randomness, and with probability $1 - 2^{-2n}$. (Error reduction, as in Lemma 7.4, means we can make the probability of failure exponentially small by repeating the algorithm enough times.)

We want to construct an algorithm with advice, so let's fix an input length n. For $x \in \{0, 1\}^n$, we say a sequence of coin tosses c is *bad for x* if it makes $\mathcal{A}$ give the wrong answer — i.e., if $\mathcal{A}(x; c) \neq f(x)$.

Claim 7.16 — For each n, there exists a sequence of coin tosses c_n which is good for *every* $x \in \{0,1\}^n$.

Proof. Imagine we choose c_n completely at random and ask, what's the probability that there exists an input x which c_n is bad for? We can bound this probability by applying the union bound; this gives

$$\mathbb{P}_{c_n}[\text{exists } x \in \{0,1\}^n \text{ s.t. } c_n \text{ bad for } x] \leq \sum_x \mathbb{P}[c_n \text{ bad for } x] \leq \frac{2^n}{2^{2n}} \leq \frac{1}{2}.$$

This means a randomly chosen c_n works with probability at least $\frac{1}{2}$, so some c_n certainly works. □

Then $\mathcal{A}(x; c_n) = f(x)$ for all $x \in \{0,1\}^n$, so we can simply take such a sequence c_n to be our advice (for inputs of length n) and $\mathcal{A}$ to be our deterministic algorithm. □

Remark 7.17. Another way to think of this proof is that if we have an algorithm $\mathcal{A}$ that decides f with probability $\frac{2}{3}$, then we know we can decide f by going over all $2^{\mathsf{poly}(n)}$ possible strings of coin tosses and taking their majority. But we can actualy pick a set of just $\mathsf{poly}(n)$ strings that substitute for the whole set of coin tosses — if we choose a uniform random set of $\mathsf{poly}(n)$ size, then with high probability their majority vote still gives the right answer on all inputs (because of the Chernoff bounds).

So we take our advice to be this set of $\mathsf{poly}(n)$ strings that approximates what the full set would do, and our advice algorithm simulates $\mathcal{A}$ on them and takes their majority.

(This is equivalent to the above proof, since the proof of error reduction (Lemma 7.4) involves just running $\mathcal{A}$ several times and taking a majority vote.)

§7.4 BPP $\subseteq \Sigma_2$P

Open question 7.18. Is BPP $\subseteq$ NP? Or is BPP $\subseteq \mathsf{P}^{\mathsf{NP}}$?

It's not at all obvious how you'd show BPP $\subseteq$ NP — given a randomized algorithm $\mathcal{A}$, you'd somehow have to prove that there's at least $\frac{2}{3}$ choices of randomness (as opposed to at most $\frac{1}{3}$) that make $\mathcal{A}$ accept, and it's not at all obvious how to do so.

And in fact, there exist oracles relative to which these containments are not true — so we'd have to do something non-relativizing (e.g., we actually have to open up the algorithm in some way). For example, there's an oracle A such that $\mathsf{EXP}^{\mathsf{NP}^A} = \mathsf{BPP}^A$, which means BPP is *much* harder than P^{NP} relative to this oracle A.

Remark 7.19. There do also exist oracles relative to which the containments are true. For example, we've seen (Theorem 4.5) that there exists an oracle relative to which P = NP. And we're going to show today that BPP is in the polynomial hierarchy, so P = BPP relative to that oracle as well.

So we don't know how to show a containment for NP or P^{NP}. However, we *can* show a containment for Σ_2P!

Theorem 7.20 (Sipser–Gacs)
We have BPP $\subseteq \Sigma_2$P.

This is a really cool theorem, and the ideas in this proof have been used all over the place. (It's not just that BPP happens to have a simple $\exists\forall$ description — what's going on is deeper, and the ideas involved are really useful.)

§7.4.1 Overview of the proof

First, how do we even start? With BPP, we have an algorithm where on each input, either $\frac{2}{3}$ of the coin tosses accept or $\frac{2}{3}$ reject; and somehow we want to simulate this with an $\exists\forall$.

The first idea you might have, inspired by the proof of $\mathsf{BPP} \subseteq \mathsf{P/poly}$, is that maybe we want to guess a sequence of coin tosses c that makes the algorithm $\mathcal{A}$ give the right answer — i.e., such that $\mathcal{A}(x, c) = f(x)$. But if we did this, how would we verify that c actually gives the right answer? It's unclear how to do this without knowing how to simulate $f(x)$ in the first place.

But we're going to be doing an $\exists\forall$, so we are going to be guessing *something*. And our goal is to somehow figure out whether or not a *lot* of coin tosses make $\mathcal{A}$ accept.

One of the key ideas in the proof is to sort of shift perspective on what we're trying to do — what's happening when we're trying to understand if a BPP algorithm accepts or rejects? Once we fix an input x, we can look at the whole space of coin tosses and consider the ones that make $\mathcal{A}$ accept; and either this is a large set (at least a $\frac{2}{3}$-fraction of the space) or a small set (at most a $\frac{1}{3}$-fraction). So when we're trying to simulate a BPP algorithm, we're really trying to distinguish a 'small set' from a 'large set.'

To be more precise, consider some $f \in \mathsf{BPP}$, and suppose that $\mathcal{A}$ computes f using $p(n)$ randomness; using error reduction (Lemma 7.4), we can assume $\mathcal{A}$ is correct with probability at least $1 - 2^{-n}$. (This bound will be a bit of an overkill for our argument, but it'll certainly suffice.) So now we've got an algorithm such that for each input x, either a vast majority of coin tosses accept, or a vast majority reject — and we want to use an $\exists\forall$ to figure out which.

To formalize this idea, for each $x \in \{0, 1\}^n$, we define

$$S_x = \{c \in \{0, 1\}^{p(n)} \mid \mathcal{A}(x; c) = 1\}$$

as the set of coin tosses that make $\mathcal{A}$ accept x. Then if $f(x) = 1$ we have $|S_x| \geq 2^{p(n)}(1 - 2^{-n})$, while if $f(x) = 0$ then $|S_x| \leq 2^{p(n)-n}$ — so either there's an enormous number of accepting strings or there's a very small number, and we somehow want to tell the difference between these two cases in $\Sigma_2\mathsf{P}$.

To be able to distinguish a very big set from a very small set, we're going to use *translates*.

> **Definition 7.21.** For $z \in \{0, 1\}^m$ and $S \subseteq \{0, 1\}^m$, we define $z \oplus S = \{z \oplus y \mid y \in S\}$.

By $\oplus$ we mean the bitwise **XOR** — so we think of the strings z and y as m-bit vectors, and we take the **XOR** their first components, then the **XOR** of their second components, and so on. For example, $001 \oplus 101 = 100$.

So given a bit-string z and a set of strings S, we can produce a new set by **XOR**ing every string in S with z; we can think of this new set $z \oplus S$ as a 'translate' of S. And we're going to try to get a handle on whether S_x is big or small by using translates. First, we'll make the following simple observation.

> **Fact 7.22** — We have $|S| = |z \oplus S|$ for all z and S.

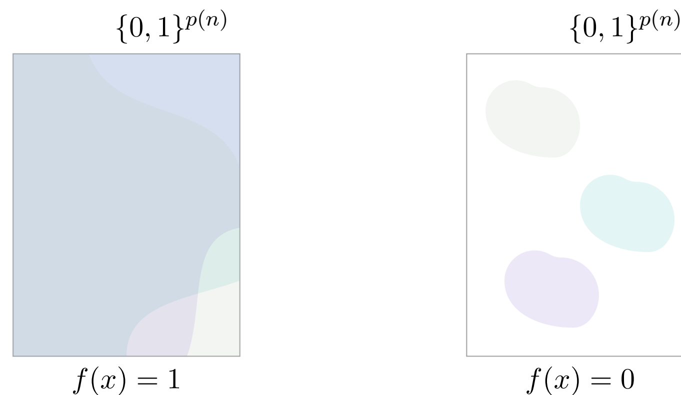
Proof. This is simply because the map $y \mapsto z \oplus y$ is a bijection $\{0, 1\}^m \to \{0, 1\}^m$ (for any fixed z) — in fact, it's an involution (i.e., it is its own inverse), since $z \oplus (z \oplus y) = y$ for any y. ☐

So when we take $z \oplus S$ we're really just shifting around S in some way, without affecting its cardinality.

Now the key idea is to guess a sequence of $p(n)$ translates $z_1, \dots, z_{p(n)} \in \{0, 1\}^{p(n)}$ such that $\bigcup_i (z_i \oplus S) = \{0, 1\}^{p(n)}$ — we'll show that this is possible if and only if $f(x) = 1$. The idea is that if $f(x) = 1$, then S_x is super big — only a 2^{-n}-fraction of strings are *not* in S_x. So with one translate, we'll already have covered nearly all of the $2^{p(n)}$ strings in our space. And then if we choose another translate, we'll cover a bit more of

the remaining strings; and so on. And we're going to show that if we choose $p(n)$ translates, we can actually cover *all* strings.

Meanwhile, if $f(x) = 0$, then S_x is tiny — it's at most a 2^{-n}-fraction of all possible strings. So each of our translates also only covers a 2^{-n}-fraction of the space, which means that we can't cover the entire space with a polynomial number of translates.



That's the main idea; we'll now go through the details. There's two pieces to the argument. The first is that this actually works — i.e., that when S_x is big we really can find $p(n)$ translates that cover the entire space, and when S_x is small we can't. And the second is that we can describe this condition with an $\exists\forall$ statement.

§7.4.2 Big set vs. small set

We'll first show that our method for distinguishing the cases where the set S_x is big vs. small — by checking whether there exist $p(n)$ translates of S_x that cover the entire space — really works.

We'll first handle the small-set case (which is easier, and can essentially be seen directly from the picture).

Claim 7.23 — Suppose that $S \subseteq \{0,1\}^{p(n)}$ and $|S| \leq 2^{p(n)-n}$ (for some polynomial p). Then if n is sufficiently large, there do not exist $z_1, \dots, z_{p(n)} \in \{0,1\}^{p(n)}$ such that $\bigcup_i (z_i \oplus S) = \{0,1\}^{p(n)}$.

Proof. For any choice of $z_1, \dots, z_{p(n)}$, each translate $z_i \oplus S$ has the same cardinality as S, so the size of their union is at most

$$\sum_i |z_i \oplus S| \leq p(n) \cdot 2^{p(n)-n} < 2^{p(n)}$$

(since $p(n) < 2^n$ for all large n), which means their union is far too small to be all of $\{0,1\}^{p(n)}$. □

(We'll assume throughout the argument that n is sufficiently large — in our $\Sigma_2\mathsf{P}$ algorithm, we can simply handle small n by brute force.)

We'll now handle the big-set case (which is the harder case) — we need to show that if S_x is big, then there do exist $p(n)$ translates that cover the entire space. The idea is that we'll imagine choosing these translates *randomly*, and we'll show that this works with positive probability; this means there must be *some* choice of translates that works.

Claim 7.24 — Suppose that $S \subseteq \{0,1\}^{p(n)}$ and $|S| \geq (1 - 2^{-n})2^{p(n)}$ (for some polynomial p). Then for uniform independent random $z_1, \dots, z_{p(n)} \in \{0,1\}^{p(n)}$, we have $\mathbb{P}[\bigcup_i (z_i \oplus S) = \{0,1\}^{p(n)}] > 0$.

Proof. We'll show that for each $y \in \{0,1\}^{p(n)}$, the probability that $\bigcup_i (z_i \oplus S)$ fails to cover y is tiny; then by union-bounding over all y, it'll follow that the probability there *exists* such a y (i.e., that $\bigcup_i (z_i \oplus S) \neq \{0,1\}^{p(n)}$) is tiny as well.

Fix y. First, for each $i \in [p(n)]$, we have $y \in z_i \oplus S$ if and only if $z_i \in y \oplus S$, and since we're choosing z_i uniformly at random, this means

$$\mathbb{P}_{z_i}[y \notin (z_i \oplus S)] = \mathbb{P}_{z_i}[z_i \notin (y \oplus S)] = 1 - \frac{|y \oplus S|}{2^{p(n)}} = 1 - \frac{|S|}{2^{p(n)}} \leq \frac{1}{2^n}.$$

Since $z_1, \ldots, z_{p(n)}$ are independent, this means $\mathbb{P}[y \notin \bigcup_i (z_i \oplus S)] = \mathbb{P}[y \notin (z_i \oplus S) \text{ for all } i] \leq 2^{-n \cdot p(n)}$. Finally, union-bounding over all $2^{p(n)}$ strings y, we get that the probability $\bigcup_i (z_i \oplus S)$ is *not* the entire space $\{0,1\}^{p(n)}$ — equivalently, that there exists $y \in \{0,1\}^{p(n)}$ with $y \notin \bigcup_i (z_i \oplus S)$ — is at most $2^n \cdot 2^{-n \cdot p(n)} < 1$, which means $\bigcup_i (z_i \oplus S) = \{0,1\}^{p(n)}$ with positive probability. □

§7.4.3 An $\exists\forall$ characterization

Together, Claims 7.23 and 7.24 show that our method of distinguishing a big set from a small set — by checking whether $p(n)$ translates cover the entire space — really does work. To prove Theorem 7.20, it remains to turn this condition into an $\exists\forall$ statement.

> **Claim 7.25 —** For $x \in \{0,1\}^n$, we have $f(x) = 1$ if and only if
>
> $$(\exists\, z_1, \ldots, z_{p(n)})(\forall\, y \in \{0,1\}^{p(n)})[\mathcal{A}(x; y \oplus z_i) = 1 \text{ for some } i \in [p(n)]].$$

This immediately implies Theorem 7.20, as it gives an $\exists\forall[\cdots]$ characterization of f. Note that the predicate that $\mathcal{A}(x, y \oplus z_i) = 1$ for some i is checkable in polynomial time (we can brute force over all $p(n)$ values of i and run $\mathcal{A}(x; y \oplus z_i)$ on each).

Proof. By Claims 7.23 and 7.24, it suffices to show that this statement is equivalent to the statement that there exist $z_1, \ldots, z_{p(n)}$ such that $\bigcup_i (z_i \oplus S_x) = \{0,1\}^{p(n)}$. But this is pretty immediate — we can rewrite the statement $\bigcup_i (z_i \oplus S_x) = \{0,1\}^{p(n)}$ as the statement that for all $y \in \{0,1\}^{p(n)}$, we have $y \in z_i \oplus S_x$ for some i; and $y \in z_i \oplus S_x$ if and only if $y \oplus z_i \in S_x$, which by the definition of S_x means $\mathcal{A}(x; y \oplus z_i) = 1$. □

§7.5 Some open problems about BPP

We'll now discuss some more open questions about BPP and some reasons why they're difficult to resolve.

> **Open question 7.26.** Is there a BPP-complete problem?

To find such a problem, we might *try* taking something like

$$\mathsf{BPSimulation} = \{\langle \mathcal{B}, x, 1^t \rangle \mid \mathcal{B} \text{ accepts } x \text{ in at most } t \text{ steps}\}.$$

Why doesn't this work? The problem is that only some algorithms $\mathcal{B}$ will satisfy the BP promise — that on each input x, either $\frac{2}{3}$ of the threads accept or $\frac{2}{3}$ reject. And for an arbitrary algorithm $\mathcal{B}$, we don't have a way of verifying whether it satisfies this promise. This means BPSimulation isn't in BPP (or at least, it isn't obviously in BPP) — if we tried to decide it by simulating $\mathcal{B}$ step-by-step (using our own coins to simulate its coins), then if even a single $\mathcal{B}$ we simulate doesn't satisfy the BP promise, our simulation won't satisfy it either.

Remark 7.27. Later, we will see something that's 'akin' to a BPP-complete problem, but it won't be a normal decision problem.

Here's another open question, where we run into similar issues.

Open question 7.28. Is there a time hierarchy theorem for BPTIME — e.g., is $\mathsf{BPTIME}[n] \neq \mathsf{BPP}$?

One main difficulty with answering this (similar to above) is that we don't have a universal simulator for randomized algorithms — because if we're given code that doesn't satisfy the BP promise, then if we simulate it step-by-step, we ourselves won't satisfy the BP promise either.

In fact, we don't even know whether this is true relative to any oracle.

Open question 7.29. Is there an oracle A such that $\mathsf{BPTIME}^A[n] = \mathsf{BPP}^A$?

So as far as we know, there might even be a relativizing proof of a BPTIME hierarchy.

However, it *is* known that if we cheat a little bit and allow ourselves 1 bit of advice — defining $\mathsf{BPTIME}[n^k]/1$ in the way we'd expect (where the randomized algorithm gets 1 bit of advice) — then we *can* prove e.g.

$$\mathsf{BPTIME}[n^k]/1 \subsetneq \mathsf{BPTIME}[n^{k+1}]/1.$$

(The main idea is that we use the 1 bit of advice to say whether the BP promise holds.)

Finally, here's a somewhat embarassing open question.

Open question 7.30. Is $\mathsf{BPP} \neq \mathsf{EXP}^{\mathsf{NP}}$?

We believe these classes are *very* far from being equal — in fact, we believe that $\mathsf{P} = \mathsf{BPP}$ (we'll discuss the reasons why in subsequent lectures), and if we could prove this, then of course we'd get $\mathsf{BPP} \neq \mathsf{EXP}^{\mathsf{NP}}$ (in fact, we'd even get $\mathsf{BPP} \neq \mathsf{EXP}$). Even if we could show $\mathsf{BPP} \subseteq \mathsf{P}^{\mathsf{NP}}$, this would be enough to resolve the question (by the (deterministic) time hierarchy theorem, or rather the fact that it relativizes).

Remark 7.31. We do know $\mathsf{BPP} \neq \mathsf{NEXP}^{\mathsf{NP}}$, at least.

§7.6 Other versions of randomized computation

We'll now discuss some variants on BPP.

§7.6.1 The classes RP and coRP

The first class we'll consider is RP (which, for some reason, stands for *randomized polynomial time*); here we have one-sided error in the **YES** case (i.e., we can only make a mistake in the case $f(x) = 1$). Similarly, there's also coRP, where we have one-sided error in the **NO** case. Here are the more formal definitions.

Definition 7.32. We say $f \in \mathsf{RP}$ if there is a polynomial-time algorithm $\mathcal{A}$ such that for all x:

- If $f(x) = 1$, then $\mathbb{P}_c[\mathcal{A}(x;c) = 1] \geq \frac{2}{3}$.
- If $f(x) = 0$, then $\mathbb{P}_c[\mathcal{A}(x;c) = 1] = 0$.

So if $f(x) = 0$ then our algorithm *never* accepts; and if $f(x) = 1$, then it *usually* accepts (over the random coin tosses).

Definition 7.33. We say $f \in \mathsf{coRP}$ if there is a polynomial-time algorithm $\mathcal{A}$ such that for all x:

- If $f(x) = 0$, then $\mathbb{P}_c[\mathcal{A}(x; c) = 0] \geq \frac{2}{3}$.
- If $f(x) = 1$, then $\mathbb{P}_c[\mathcal{A}(x; c) = 0] = 0$.

Example 7.34

We have $\mathsf{PIT} \in \mathsf{coRP}$ — our algorithm (in the proof of Theorem 7.10) always accepts in the **YES** case where $F \equiv G$ (as then $F(r) = G(r)$ for all r), which means it has one-sided error only in the **NO** case.

Note that $\mathsf{RP} \subseteq \mathsf{NP}$ and $\mathsf{coRP} \subseteq \mathsf{coNP}$ — for example, the condition of RP is that when x is a **YES** instance, at least $\frac{2}{3}$ of all witnesses should be accepted; while when x is a **NO** instance, no witnesses should be accepted. This is stronger than the condition of NP (when x is a **YES** instance, the condition of NP only requires that *some* witness is accepted; with RP we're asking that not only does there *exist* a witness that satisfies the verifier, but even a *random* witness does).

§7.6.2 The class ZPP

The next class we'll consider is ZPP, which is the class of problems we can solve in 'probabilistic polynomial time with zero error.' There are several ways of thinking about this. One is by considering an algorithm model where we have more than two possible outputs — we have a 'yes for sure' answer (which we denote by 1), a 'no for sure' answer (which we denote by 0), and a 'I don't know, try again' answer (which we denote by ?).

Definition 7.35. We say $f \in \mathsf{ZPP}$ if there is a polynomial-time algorithm $\mathcal{A}$ with outputs in $\{0, 1, ?\}$ such that for all x, the following two statements hold:

- $\mathbb{P}_c[\mathcal{A}(x; c) = f(x)] \geq \frac{2}{3}$.
- $\mathcal{A}(x; c) \in \{f(x), ?\}$ for all c.

In other words, $\mathcal{A}$ never outputs the *wrong* answer (e.g., it never outputs 0 when $f(x) = 1$), and it outputs an answer (i.e., not ?) with probability at least $\frac{2}{3}$.

Note that if we have such an algorithm $\mathcal{A}$, then we could try to compute $f(x)$ with it by repeatedly running it until it gives an answer (every time we get a ?, we try again). So another way to think about ZPP is that these are the functions computable in *worst-case expected polynomial time* — i.e., if we think of the runtime of the algorithm on x, which we denote by $t(x)$, as a random variable (depending on the coin tosses), then $\mathbb{E}[t(x)] = \mathsf{poly}(n)$ for all x. (The point is that if we have an algorithm with outputs in $\{0, 1, ?\}$, then we can just keep running it until it gives an answer; and if it gives an answer with probability $\frac{2}{3}$, then in expectation it should just take $\frac{3}{2}$ tries to get an answer.)

Open question 7.36. Does ZPP equal RP or BPP?

This isn't known, but we *can* relate ZPP to the classes we've defined so far (giving another way of thinking about it).

Fact 7.37 — We have $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$.

Proof. To show that $\mathsf{ZPP} \subseteq \mathsf{RP}$, suppose we've got a function $f \in \mathsf{ZPP}$ and an algorithm for f as in the definition of ZPP (so the algorithm always outputs the correct answer with probability at least $\frac{2}{3}$, and ? otherwise). Then we can get an algorithm as in the definition of RP by simply replacing ? with 0; then we'll

always reject when $f(x) = 0$, while when $f(x) = 1$ we'll still be outputting 1 with probability at least $\frac{2}{3}$. Similarly, to get a coRP algorithm, we can instead replace ? with 1. This shows $\mathsf{ZPP} \subseteq \mathsf{RP} \cap \mathsf{coRP}$.

To prove the other inclusion, the idea is that given a RP and a coRP algorithm for f, we run both algorithms at the same time. No matter what $f(x)$ is, we know one of these algorithms isn't allowed to make an error (if $f(x) = 0$ then the RP one can't make an error, and if $f(x) = 1$ then the coRP one can't). So if the two algorithms agree, then we output whatever they give (e.g., if both answer 1, then we output 1 as well). Meanwhile, if they *don't* agree, then we don't know which one's wrong, so we just output ?. This gives a ZPP algorithm for f, showing $\mathsf{RP} \cap \mathsf{coRP} \subseteq \mathsf{ZPP}$. □

§7.6.3 Some relationships and open questions

We've seen that $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$, and of course RP and coRP are both contained in BPP (the conditions in the definition of RP and coRP are only stronger than the ones for BPP).

Fact 7.38 — We have $\mathsf{BPP} \subseteq \mathsf{RP}^{\mathsf{NP}}$.

The point is that in our proof of $\mathsf{BPP} \subseteq \Sigma_2\mathsf{P} = \mathsf{NP}^{\mathsf{NP}}$ (Theorem 7.20), we actually showed that it's enough to choose $z_1, \ldots, z_{p(n)}$ at *random* and check whether the corresponding translates of S_x cover the entire space (choosing the z_i's can be done in RP, and checking that the corresponding translates cover the space can be done in coNP).

Finally, here are some open questions.

Open question 7.39. Is $\mathsf{P} = \mathsf{ZPP}$?

Open question 7.40. Is $\mathsf{RP} = \mathsf{BPP}$?

We believe the answers to both questions are yes — in fact, we believe *all* these classes are equal to P. However, we're pretty far from being able to show this — even the following questions are still open.

Open question 7.41. Is $\mathsf{EXP} = \mathsf{ZPP}$? Is $\mathsf{NEXP} = \mathsf{BPP}$?

We believe the answer is no, but it's striking that we don't know how to prove it — no one believes that these classes are equal, but somehow separating them is highly nontrivial (and proving they're different would somehow give interesting derandomization results).

Remark 7.42. We do know that $\mathsf{NEXP} \neq \mathsf{ZPP}$, because $\mathsf{ZPP} \subseteq \mathsf{NP}$ (and $\mathsf{NP} \subsetneq \mathsf{NEXP}$ by the nondeterministic time hierarchy theorem).

§8 Derandomization

As mentioned earlier, we believe that all the randomized classes we've defined — BPP, RP, and ZPP — are actually equal to P. Today and in the next couple of lectures, we'll discuss why we believe this.

§8.1 The problem CAPP

As a starting point for our discussion of derandomization, we'll define a problem that we might think of as 'complete' for derandomization — a problem that we'd want to solve in an efficient deterministic way if we want to prove that all these randomized classes collapse to P. We often think of the P vs. NP problem in terms of a single NP-complete problem (e.g., 3SAT or CircuitSAT). In a similar fashion, we can view P vs. BPP in terms of a specific problem. The problem we'll consider will be a sort of circuit analysis problem (like CircuitSAT, where we're given a circuit and want to know whether there exists an input making it print a particular value).

Definition 8.1 (CAPP)

- **Input:** a circuit C with n inputs and one output.
- **Promise:** one of the following two cases holds:
  (1) For uniform random $x \in \{0,1\}^n$, we have $\mathbb{P}_x[C(x) = 1] \geq \frac{2}{3}$.
  (2) For uniform random $x \in \{0,1\}^n$, we have $\mathbb{P}_x[C(x) = 0] \geq \frac{2}{3}$.
- **Decide:** which of (1) or (2) is true.

(CAPP stands for *circuit approximation probability problem.*)

This is a different kind of problem than the ones we've seen before, in that it comes with a *promise* — we're promised that our circuit C satisfies (1) or (2) (i.e., either it accepts most inputs, or it doesn't). Of course some circuits won't satisfy this, but when we talk about solving CAPP, we won't care about what our algorithm does on those circuits.

Definition 8.2. We say an algorithm $\mathcal{A}$ solves CAPP if given a circuit C satisfying (1) or (2), $\mathcal{A}$ correctly outputs which of (1) or (2) it satisfies.

Importantly, we don't care what $\mathcal{A}$ outputs on circuits not satisfying (1) or (2) — if $\mathbb{P}_x[C(x) = 1] = \frac{1}{2}$ (for example), then $\mathcal{A}$ can output whatever it likes.

Remark 8.3. Usually, CAPP refers to a stronger version of this problem, where we're given a circuit C and we want to *estimate* its acceptance probability — more precisely, to output $v \in [0,1]$ such that

$$|v - \mathbb{P}_x[C(x) = 1]| \leq \frac{1}{10}.$$

(This problem doesn't have a promise — it's defined on all circuits — but there's several possible v's we could output for any circuit.)

This is a more difficult problem, in that if we can estimate $\mathbb{P}_x[C(x) = 1]$ with an additive error of $\frac{1}{10}$, then we can definitely distinguish between the cases where it's $\frac{2}{3}$ vs. $\frac{1}{3}$. But the version in Definition 8.1 is enough for our purposes.

First, it's easy to distinguish between the cases (1) and (2) if we have randomness — we can just sample 20 values of x at random, compute the fraction for which $C(x) = 1$, and check whether it's above or below $\frac{1}{2}$.

(Given that one of (1) and (2) holds, this will give the right answer with high probability.) So the question we're interested in (for derandomization) is whether we can do this *deterministically*.

We think of CAPP as being BPP-hard in the following sense.

Theorem 8.4

If there is a polynomial-time algorithm that solves CAPP, then P = BPP.

In other words, if we get a polynomial-time algorithm for CAPP, then we get one for *every* problem in BPP.

Proof. Let $\mathcal{A}$ be a polynomial-time algorithm that solves CAPP. Consider some $f \in \mathsf{BPP}$, and let $\mathcal{B}$ be a randomized polynomial-time algorithm for f (this means $\mathcal{B}$ takes two inputs — the actual input x, and the randomness c). Our goal is to get a *deterministic* algorithm for f.

The main idea is that given an input x, we first construct a circuit whose input is the randomness, and which simulates $\mathcal{B}$ — i.e., we construct a circuit C_x (of polynomial size) such that $C_x(y) = \mathcal{B}(x; y)$. We can do this by the same 'tableau-to-circuit' construction from our proof that $\mathsf{P} \subseteq \mathsf{SIZE}[\mathsf{poly}]$ (Proposition 6.13) — we convert $\mathcal{B}$ into a one-tape Turing machine and then construct the circuit for building its computation tableau. (There's two inputs to $\mathcal{B}$ — the input x and the randomness y — and in our circuit, we hardcode x and take y to be the input to the circuit.)

And then we just call $\mathcal{A}$ on C_x and output its answer. Why does this work? Since $\mathcal{B}$ is a BPP algorithm for f, we know $\mathbb{P}_y[\mathcal{B}(x; y) = f(x)] \geq \frac{2}{3}$ (by the definition of BPP). So $\mathbb{P}_y[C_x(y) = f(x)] \geq \frac{2}{3}$ for each fixed x, which means C_x really does satisfy (1) or (2) in the promise for CAPP. And since $\mathcal{A}$ solves CAPP, it'll correctly figure out which one C_x satisfies, which gives us the value of $f(x)$. □

Remark 8.5. The setup of this proof is a bit similar to how you'd prove CircuitSAT is NP-complete — we start out with a polynomial-time algorithm $\mathcal{V}$ that takes in some input x and some witness y, and code up a circuit (with input y, and with x hardcoded) that simulates $\mathcal{V}(x, \bullet)$. There we're trying to check whether there *exists* y that makes the algorithm accept. Here we're trying to figure out whether *most* y make the algorithm accept, and this naturally gives CAPP.

§8.1.1 PromiseBPP

The converse of Theorem 8.4 isn't necessarily true — it's possible that P = BPP but there is no fast deterministic algorithm for CAPP. This is because CAPP isn't in BPP — this wouldn't be a statement that makes sense, because CAPP is a promise problem and isn't really even *defined* on all inputs. However, we can define a 'promise' version of BPP, and CAPP *will* be complete for this class.

Definition 8.6. For disjoint subsets $Y, N \subseteq \{0,1\}^*$, we say $(Y, N) \in$ PromiseBPP if there is a polynomial-time algorithm $\mathcal{A}$ such that for all x:

- If $x \in Y$, then $\mathbb{P}_y[\mathcal{A}(x; y) = 1] \geq \frac{2}{3}$.
- If $x \in N$, then $\mathbb{P}_y[\mathcal{A}(x; y) = 0] \geq \frac{2}{3}$.

We think of (Y, N) as a *promise problem* — there's a 'promise' that our input x is in either Y (which we think of as the **YES** case) or N (which we think of as the **NO** case), and we don't care what the algorithm does if it isn't (so we get the BPP promise, but only on $Y \cup N$).

We could also phrase this definition in terms of functions rather than subsets of $\{0,1\}^*$. Here we'll have functions $f \colon \{0,1\}^* \to \{0,1,?\}$, where we think of ? as meaning 'I don't care' — we call these *partial functions*, since they're not really defined (i.e., they take values of 'I don't care') on some inputs.

Definition 8.7. We say $f\colon \{0,1\}^* \to \{0,1,?\}$ is in PromiseBPP if there is a polynomial-time algorithm $\mathcal{A}$ such that for all x:

- If $f(x) = 1$, we have $\mathbb{P}_y[\mathcal{A}(x,y) = 1] \geq \frac{2}{3}$.
- If $f(x) = 0$, we have $\mathbb{P}_y[\mathcal{A}(x,y) = 0] \geq \frac{2}{3}$.

For example, we can think of BPP as the set of functions $f \in$ PromiseBPP which never output ?.

Theorem 8.8

The problem CAPP is PromiseBPP-complete — in other words, CAPP $\in$ PromiseBPP, and for every $f \in$ PromiseBPP we have $f \leq_{\mathsf{p}}$ CAPP.

We've already seen all the ideas involved in the proof — CAPP is already set up as a promise problem, and it's easy to see it's in PromiseBPP (we can just sample a bunch of values of x and compute the fraction of our sample for which $C(x) = 1$); and the proof that it's PromiseBPP-hard is essentially the same as that of Theorem 8.4. So we won't go through the details here.

Remark 8.9. We could also define the class PromiseP in the same way — where $(Y, N) \in$ PromiseP if there's a polynomial-time algorithm $\mathcal{A}$ such that for all x, if $x \in Y$ then $\mathcal{A}(x) = 1$, and if $x \in N$ then $\mathcal{A}(x) = 0$. Then Theorem 8.8 implies that CAPP $\in$ PromiseP (i.e., there's a polynomial-time algorithm solving CAPP) if and only if PromiseBPP $=$ PromiseP.

§8.2 Pseudorandom generators

People often think about derandomization in terms of CAPP — we're given a circuit, and we want to approximate its acceptance probability (or at least be able to distinguish between tiny and large probabilities).

Question 8.10. How could we (deterministically) solve CAPP?

In principle, we could imagine trying to solve CAPP by taking the description of the circuit and somehow trying to understand what it's doing — maybe we can figure out whether it accepts lots of inputs by staring at the **AND**s and **OR**s really hard and doing some sort of weird analysis.

But it turns out the way people think about solving CAPP is very different. They believe you barely have to look at the circuit at all — instead, you just come up with an appropriate set of strings that effectively simulates *all* the strings. If we're trying to deterministicaly solve CAPP, the obvious deterministic algorithm tries all possible inputs $x \in \{0,1\}^n$ and sees what C does on each (which takes $2^n \cdot \mathsf{poly}(\mathsf{size}(C))$ time). And the way people try to solve CAPP is by trying to replace $\{0,1\}^n$ — which has size 2^n — with a much smaler space. More specifically, we want a set S_n consisting of $\mathsf{poly}(n)$ n-bit strings, such that randomly sampling from S_n does roughly the same thing as randomly sampling from the set of *all* strings — i.e., such that

$$\mathbb{P}_{x \in S_n}[C(x) = 1] \approx \mathbb{P}_{x \in \{0,1\}^n}[C(x) = 1]$$

for every circuit C (of the appropriate size). If we could come up with such a set S_n, then we could distinguish between (1) and (2) (to solve CAPP) by simply going over all the strings in S_n and running the given circuit C on each (which we can do efficiently, since there's only $\mathsf{poly}(n)$ such strings).

This is a bit counterintuitive — you might think that you could get a lot of information (to distinguish (1) and (2)) by staring at the circuit, but here we're barely looking at the circuit at all. Still, people strongly believe that this can be done — that not only can we solve CAPP deterministically, but we can even solve it in this way (which is a much stronger statement).

§8.2.1 Definition of PRGs

This motivates the definition of a pseudorandom generator — a pseudorandom generator is essentially a function that's supposed to produce a set S_n such that $\mathbb{P}_{x \in S_n}[C(x) = 1] \approx \mathbb{P}_{x \in \{0,1\}^n}[C(x) = 1]$.

Definition 8.11. We say a function $g \colon \{0,1\}^* \to \{0,1\}^*$ is an $r(n)$-seed ε-error pseudorandom generator (abbreviated PRG) if:

- g maps $\{0,1\}^{r(n)} \to \{0,1\}^n$ for all n.
- For all circuits C of size n and with $k \leq n$ inputs, we have

$$\left|\mathbb{P}_{x \in \{0,1\}^n}[C(x) = 1] - \mathbb{P}_{y \in \{0,1\}^{r(n)}}[C(g(y)) = 1]\right| \leq \varepsilon. \tag{2}$$

We think of $r(n)$ as being less than n — so g takes short strings and *stretches* them to longer strings. And the idea is that if we take the image of g — i.e., $g(y)$ over all possible strings y — and compute the acceptance probability of C on this image, it should be close to the acceptance probability of C on *all* possible inputs.

We call the property (2) *indistinguishability* (it's a really common inequality in PRGs or cryptography), and when it holds we say that C is *fooled* by the outputs of g. The point is that it's behaving basically as if the outputs of g are uniform random strings — it can't distinguish the distribution $g(\{0,1\}^{r(n)})$ from $\{0,1\}^n$.

Remark 8.12. In this definition, x and $g(y)$ have n bits, but the definition allows C to have $k \leq n$ inputs. If C has fewer than n inputs, then we define $C(x)$ as $C(x_1 \ldots x_k)$ — i.e., we just take the first k bits of x and chop off the extra bits.

Remark 8.13. It's also possible to define PRGs for other models of computation (where instead of asking the PRG to fool all circuits, we stick in a different model of computation). We're working with circuits here because we want to use PRGs to solve CAPP.

We're interested in using PRGs for derandomization, so we typically think of ε as being a sufficiently small constant (e.g., $\frac{1}{10}$) to let us distinguish between $\frac{2}{3}$ and $\frac{1}{3}$ (or it could even shrink with n — for example, it could be $\frac{1}{n}$). We also ideally want seed length $r(n) = O(\log n)$ — then $\{0,1\}^{r(n)}$ has polynomial size, so we're working with a space of $\mathsf{poly}(n)$ strings $g(y)$.

§8.2.2 Existence of PRGs

So far, we haven't said anything about how difficult it is to construct such a function g (the definition doesn't have any requirement on being able to compute g efficiently — though we'll certainly need this to use it for derandomization). But the first thing we'll say is that PRGs with these parameters do *exist* (though we're still not saying anything about whether we can efficiently *construct* them).

Theorem 8.14

For all $\varepsilon > 0$, there exists an $O(\log(n/\varepsilon))$-seed ε-error PRG.

Proof. The idea is that we want to construct a set (the image of g) which is 'pseudorandom,' and we'll do so by fighting pseudorandomness with true randomness — we'll pick g uniformly at random, and we'll show that it'll satisfy (2) with high probability (by a Chernoff bound–union bound argument — this is a very common type of argument in randomized algorithms and the probabilistic method).

We first fix n, and let $t = 2n^2\varepsilon^{-2}$ and $r = \log_2 t$ (we can round t up to be a power of 2, so that r is an integer). We then define $g_n\colon \{0,1\}^r \to \{0,1\}^n$ by choosing each $g_n(y)$ uniformly and independently at random — so we're essentially picking $t = 2^r$ random strings to be the image of g.

To show that this works, for each circuit C, we define $\mathsf{Diff}(C, g_n) = |\mathbb{P}_x[C(x) = 1] - \mathbb{P}_y[C(g_n(y)) = 1]|$ (where $x \in \{0,1\}^n$ and $y \in \{0,1\}^r$ are chosen uniformly at random) — this essentially measures how well g_n fools C. Then using a Chernoff bound, we can show that for all C we have

$$\mathbb{P}_{g_n}[\mathsf{Diff}(C, g_n) \geq \varepsilon] \leq \frac{1}{2^{n^2/3}}$$

(the reason we get n^2 in the exponent is because we've got roughly n^2 strings). So the probability that g_n fails to fool any fixed circuit C is *really* small. And once we've got a bound like this, we can just union-bound over all circuits C — the probability there exists *any* circuit C (with $\mathsf{size}(C) \leq n$) which fails to be fooled by g_n is

$$\mathbb{P}_{g_n}[\text{exists } C \text{ with } \mathsf{Diff}(C, g_n) \geq \varepsilon] \leq \sum_C \mathbb{P}_{g_n}[\mathsf{Diff}(C, g_n) \geq \varepsilon].$$

And there's only $2^{O(n \log n)}$ circuits of size n (by Claim 6.7), while our failure probability for each is $2^{-n^2/3}$ (which is tiny compared to this number) — so we get

$$\mathbb{P}_{g_n}[\text{exists } C \text{ with } \mathsf{Diff}(C, g_n) \geq \varepsilon] \leq \frac{2^{O(n \log n)}}{2^{n^2/3}} \ll 1.$$

This tells us that with high probability, a *random* function g (with these parameters) fools all circuits of the appropriate size. □

Remark 8.15. In fact, the seed length $O(\log n)$ is optimal — there does not exist an $o(\log n)$-seed $\frac{1}{n}$-error PRG. Very roughly, the reason is that we're dealing with circuits of size n, and so if our seed length goes below $\log n$, then we can store all the strings in the image of g in a circuit of size n. So we can construct a circuit that goes through all $n^{o(1)}$ strings in the image of g and checks whether its given input x is one of them. This circuit will have size at most n, but g will certainly not fool it.

Of course, Theorem 8.14 doesn't help us at all with proving $\mathsf{P} = \mathsf{BPP}$, because we just proved the *existence* of PRGs (and our method of constructing g used randomness, which is exactly what we wanted to avoid in the first place).

Question 8.16. Are there deterministic and efficiently computable PRGs?

§8.2.3 Derandomization from PRGs

First, to see what parameters we want for our PRGs (in particular, what we want 'efficient' to mean), we'll see how to get derandomization from PRGs.

Theorem 8.17

Suppose that there exists an $r(n)$-seed $\frac{1}{10}$-error PRG g which we can compute in $2^{O(m)}$ time on m-bit seeds. Then we can solve CAPP (deterministically) in $2^{O(r(n))} \cdot \mathsf{poly}(n)$ time.

So it's okay even if our PRG takes *exponential* time (in the seed length) to compute. In particular, if we can get seed length $r(n) = O(\log n)$ (so that we're spending $\mathsf{poly}(n)$ time to compute the PRG on $O(\log n)$-bit seeds), then we could solve CAPP in *polynomial* time.

Remark 8.18. The parameters for complexity-theoretic PRGs are fairly different from the ones used in cryptography — here we want seed lengths of $O(\log n)$, and our PRG is allowed to run in exponential time (in the seed length). In contrast, in cryptography we want our PRGs to run in polynomial time (in the seed length), but we also don't need seeds this small — we want seed length n^ε (for some $\varepsilon > 0$) rather than $O(\log n)$.

Proof. We want to use our PRG to solve CAPP, so suppose we're given a circuit C with $\mathsf{size}(C) = n$. Then the idea is that we just compute C on all $2^{r(n)}$ outputs of g, and use this to estimate the acceptance probability of C.

More precisely, we go through all $r(n)$ seeds y one at a time and compute $v = \mathbb{P}_{y \in \{0,1\}^{r(n)}}[C(g(y)) = 1]$ by simply evaluating C on each string $g(y)$; this takes $2^{r(n)} \cdot \mathsf{poly}(n)$ time. And then we output (1) (i.e., that we're in the case $\mathbb{P}_x[C(x) = 1] \geq \frac{2}{3}$) if $v \geq \frac{1}{2}$, and (2) (i.e., that $\mathbb{P}_x[C(x) = 0] \geq \frac{2}{3}$) if $v < \frac{1}{2}$.

The reason this works is just that because g is a PRG, we have

$$\left| v - \mathbb{P}_{x \in \{0,1\}^n}[C(x) = 1] \right| \leq \frac{1}{10}.$$

This means if C satisfies (1) then we must have $v \geq \frac{2}{3} - \frac{1}{10} > \frac{1}{2}$, while if it satisfies (2) then we must have $v \leq \frac{1}{3} + \frac{1}{10} < \frac{1}{2}$. So in either case, this gives us the right answer. ☐

We can get similar consequences for derandomizing BPP.

Theorem 8.19

Suppose that there exists an $r(n)$-seed $\frac{1}{10}$-error PRG g which we can compute in $2^{O(m)}$ time on m-bit seeds. Then for time-constructible $t\colon \mathbb{N} \to \mathbb{N}$, we have

$$\mathsf{BPTIME}[t(n)] \subseteq \mathsf{TIME}[2^{O(r(t(n)^2))} \cdot \mathsf{poly}(t(n))].$$

Proof sketch. The idea is that we just take a randomized $t(n)$-time algorithm, convert it to a $t(n)^2$-sized circuit, and then apply Theorem 8.17. Note that in the definition of a PRG, n corresponds to the circuit size; and here our circuit size is $t(n)^2$, which is why we have $r(t(n)^2)$. ☐

Corollary 8.20

Suppose that there exists an $O(\log n)$-seed PRG with $\frac{1}{10}$ error that can be computed in $\mathsf{poly}(n)$ time (on $O(\log n)$-bit seeds). Then $\mathsf{P} = \mathsf{BPP}$.

The point is that $2^{O(\log t)} = \mathsf{poly}(t)$; so if we have PRGs with these parameters, then Theorem 8.19 says that $\mathsf{BPTIME}[t(n)] \subseteq \mathsf{TIME}[\mathsf{poly}(t(n))]$.

For this reason, we'll refer to such PRGs as *good*.

Definition 8.21. We say a good PRG is an $O(\log n)$-seed $\frac{1}{10}$-error PRG that can be computed in $\mathsf{poly}(n)$ time (on $O(\log n)$-bit seeds).

Question 8.22. Do there exist good PRGs?

Next class, we're going to see that under hypotheses that most people believe, the answer is yes! In particular, we're going to see that certain circuit lower bounds imply the existence of good PRGs (so if you believe in circuit lower bounds, then you should also believe in derandomization).

§8.3 PRGs from circuit lower bounds

Today we'll discuss how to get PRGs from circuit lower bounds. In particular, we're going to talk about the Nisan–Wigderson PRG; this is one of the most important constructions in complexity theory, and its analysis involves lots of nice things (e.g., a hybrid argument from cryptography, designs from combinatorics, and so on).

Definition 8.23. We define $\mathsf{E} = \mathsf{TIME}[2^{O(n)}]$.

Theorem 8.24 (IW 1997)

Suppose there exists $f \in \mathsf{E}$ such that for all circuit families $\{C_n\}$ computing f, for all large n we have $\mathsf{size}(C_n) > 2^{n/100}$. Then there exist good PRGs (and therefore $\mathsf{P} = \mathsf{BPP}$).

So the assumption here is roughly that there exists a problem in E that doesn't have subexponential-sized circuits. People generally believe that such circuit lower bounds should be true, and this is why they believe in good PRGs. (There are connections in the other direction as well — there are certain problems for which if we could derandomize them, then we'd get circuit lower bounds.)

The constant $\frac{1}{100}$ here is not important, and can be substituted with any $\alpha > 0$; we'll prove it with a very concrete value of α so that the proof is simpler.

In fact, this theorem gives a very explicit, very concretely described g constructed out of f (we'll define g by looking at both our $r(n)$-bit seed y and the truth table of f on a specific input length, and we'll use the fact that f is computable in $2^{O(n)}$ time to get that g is efficiently computable).

We'll now go through the argument (which involves several steps).

§8.3.1 Step 1 — worst-case to average-case hardness

The first step of the proof, which we unfortunately have to black-box, is going from worst-case to average-case hardness. In the hypothesis of Theorem 8.24, we're given that every circuit family $\{C_n\}$ that computes f has $\mathsf{size}(C_n) > 2^{n/100}$. This is a *worst-case* hardness statement — for every circuit C_n with $\mathsf{size}(C_n) \leq 2^{n/100}$, there just has to be *one* input x for which C_n fails to compute $f(x)$. The first step is going from turning this into an *average-case* hardness statement — i.e., showing that if we've got one function f cooked up such that every small circuit fails to compute f on *one* input, then we can cook up another function f' (which is also in E) such that every small circuit fails to compute f' on a *very large fraction* of inputs.

Definition 8.25. Given $f\colon \{0,1\}^n \to \{0,1\}$, we say f has average-case hardness at least $s(n)$ if for all circuits C with $\mathsf{size}(C) \leq s(n)$, we have

$$\mathbb{P}_x[C(x) = f(x)] < \frac{1}{2} + \frac{1}{s(n)}.$$

First, why does this definition make sense? Note that for *any* function f, there's a circuit that gets it right on at least half of all inputs — we can take C to be either the all-0's circuit or the all-1's circuit (there's *some* value that f takes at least half of the time, and we can hardcode our circuit to just always output that value). So for a function to have average-case hardness at least $s(n)$ means that $s(n)$-sized circuits can't do much better than the trivial bound.

The theorem that encapsulates this step — which we'll unfortunately have to state as a black box — is that given a function f which doesn't have small circuits (i.e., is worst-case hard), we can get another function which is *average-case* hard. First, we'll use the following definition to make things more convenient to state.

Definition 8.26. We say $f \in \mathsf{ioSIZE}[s(n)]$ if there exist infinitely many n for which f on n-bit inputs can be computed by a circuit C_n with $\mathsf{size}(C_n) \leq s(n)$.

Here io stands for *infinitely often*. The point of this definition is that saying $f \notin \mathsf{ioSIZE}[s(n)]$ means that for *all* sufficiently large n we have that f on n-bit inputs can't be computed by a circuit of size $s(n)$ (which is the kind of worst-case hardness statement we have in the hypothesis of Theorem 8.24); in contrast, saying $f \notin \mathsf{SIZE}[s(n)]$ would only mean this is true for infinitely many n.

Theorem 8.27

There is a universal constant c such that if there exists a function $f \in \mathsf{E}$ with $f \notin \mathsf{ioSIZE}[s(n)]$, then there is a function $f' \in \mathsf{E}$ such that for all sufficiently large n, we have that f' on n-bit inputs has average-case hardness at least $s(n/c)^{1/c}$.

In general, this type of statement is called *hardness amplification* — we take a function which is worst-case hard, and show there's another function building on it which is actually *average-case* hard (with a little bit of loss).

This is Theorem 19.27 in Arora and Barak; the fact that there are 26 results before it is suggestive of why we don't have time to prove it in class. But here's a very high-level overview of what the proof looks like.

The proof uses *error-correcting codes* — an error-correcting code E maps messages x to codewords $E(x)$ such that even if a bunch of bits of the codeword are corrupted, you can still recover the original message — i.e., given a string that differs from $E(x)$ in a bunch of indices, we can recover x.

Then the idea behind the proof of Theorem 8.27 is that we take the truth table of f (which has size 2^n) and apply some really funky error-correcting code to it (or really, various error-correcting codes applied in a nice way) to get a $2^{O(n)}$-bit string, which we take as the truth table of f'. So to compute f' on a specific input, we first compute f on all 2^n inputs (which takes $2^n \cdot 2^{O(n)}$ time) to get its truth table, then apply the polynomial-time error-correcting code (which also takes $2^{O(n)}$ time) to get the truth table of f', and then look up the appropriate bit of this truth table; this shows $f' \in \mathsf{E}$.

And the point is that if we have a circuit that even *approximately* computes f', then by using the recovery process for the error-correcting code, we can get a circuit that *exactly* computes f.

§8.3.2 An overview of Step 2 — average-case hardness to PRGs

The next step of the argument is to show that from average-case hardness we can get good PRGs. More precisely, we start with a function which is average-case hard for *exponential*-sized circuits — for all circuits of size $2^{\alpha n}$, the probability that the circuit computes our function correctly on a randomly chosen (n-bit) input is at most $\frac{1}{2} + \frac{1}{2^{\alpha n}}$. And we're going to show that from such a function, we can get a PRG — this will be our main theorem for today.

Theorem 8.28 (Nisan–Wigderson 1994)

There exists a polynomial-time computable function $g\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ such that for all large n, if $f\colon \{0,1\}^{5\log n} \to \{0,1\}$ has average-case hardness at least $10n^2$, then the function $g_f = g(f, \bullet)$ is a $200 \log n$-seed $\frac{1}{10}$-error PRG (for circuits of size n).

Here we think of g as taking in two bit-strings — the first is the truth table of an (average-case) hard function, and the other is a seed. The truth table of f will have n^5 bits (so the total input length to g is $n^5 + 200 \log n$, which means g runs in $\mathsf{poly}(n)$ time). Note that f is a function on $m = 5 \log n$ bits, so its

average case hardness is $10n^2 = 10 \cdot 2^{2m/5}$ (which is exponential in m). (The constants here (e.g., 5, 10, and 200) are not super important, and there are ways to tune them to other values.)

Together, Theorems 8.27 and 8.28 give a construction of good PRGs from circuit lower bounds as in Theorem 8.24 (though we need to tweak the parameters) — we start with some $f \in \mathsf{E}$ satisfying the worst-case hardness hypothesis in Theorem 8.24, so that $f \notin \mathsf{ioSIZE}[2^{n/100}]$. Then we can use Theorem 8.27 to get $f' \in \mathsf{E}$ which is *average-case* hard. And the fact that $f' \in \mathsf{E}$ means that we can compute its entire truth table on $O(\log n)$-bit inputs in $\mathsf{poly}(n)$ time (there's $2^{O(\log n)} = \mathsf{poly}(n)$ inputs to go over, and it takes $2^{O(\log n)} = \mathsf{poly}(n)$ time to compute f' on each). And then we can plug this truth table into the function g from Theorem 8.28, and we'll get a good PRG $g_{f'}$. (This isn't exactly true, because Theorem 8.28 asks for average-case hardness roughly $2^{2m/5}$ on m-bit inputs (as opposed to $2^{n/100c}$, which is what we'd get from this argument), but the parameters can be tuned so that this does work.)

Remark 8.29. In some sense, Theorem 8.28 is a statement that a lower bound implies an algorithm — if we have a sufficiently strong lower bound on f (saying that it's average-case hard for circuits), we can turn it into an actual construction for producing PRGs. This is fascinating on a philosophical level, in addition to the proof being mathematically cool.

In the rest of this class, we're going to prove Theorem 8.28.

§8.3.3 A baby PRG

As a first step towards proving Theorem 8.28, we're going to prove a much easier statement that'll already illustrate some of the key ideas. We *want* to start with an average-case hard function f and use it to get a $O(\log n)$-seed PRG. We're first going to show how to get a $(n-1)$-seed PRG instead. This doesn't help us at all in terms of derandomization — if you want to simulate using the PRG's output instead of truly random strings, you'll now have to go over 2^{n-1} seeds instead of 2^n strings (which isn't much of an improvement). But being able to stretch $n-1$ bits to n bits is already a nontrivial and interesting statement, and proving that the construction works will give us some good insights.

We're given a really hard function f, and we want to stretch $(n-1)$-bit strings x to n-bit strings. The natural thing to do is that we know f is really hard, so we'll just stick $f(x)$ at the end of x — i.e., we define $g(f, x) = xf(x)$. (So our PRG takes the 2^{n-1}-bit truth table of f, looks up $f(x)$, and then concatenates that to x.)

Theorem 8.30 (Baby PRG theorem)

Suppose that for all circuits C' of size s, we have $\mathbb{P}_x[C'(x) = f(x)] \leq \frac{1}{2} + \varepsilon$. Then for all circuits C of size s, we have

$$|\mathbb{P}_x[C(xf(x)) = 1] - \mathbb{P}_{x,b}[C(xb) = 1]| \leq \varepsilon.$$

(Here if we think of ε as $\frac{1}{s}$, then the hypothesis says that f has average-case hardness at least s.)

This says that if f is average-case hard, then no small circuit can tell the difference between taking a $(n-1)$-bit x and tacking on $f(x)$ compared to taking x and tacking on a random bit b. Note that if x and b are uniformly distributed, then so is xb; so this means the distribution $xf(x)$ really is pseudorandom.

Proof. We'll prove the contrapositive — we'll show that if a circuit C isn't fooled by the mapping $x \mapsto xf(x)$, then there exists a circuit C' that can predict f with probability better than $\frac{1}{2} + \varepsilon$. (So we're going from a circuit that can *distinguish* between our generator and the uniform distribution to one that can *predict* f.)

First, without loss of generality we can remove the absolute value signs and assume that

$$\mathbb{P}_x[C(xf(x)) = 1] - \mathbb{P}_{x,b}[C(xb) = 1] \geq \varepsilon. \tag{3}$$

(This is because if the left-hand side were instead at most $-\varepsilon$, then we could replace C with $\neg C$ — the circuit that flips the output of C. We have $\mathbb{P}_y[C(y) = 1] = 1 - \mathbb{P}_y[\neg C(y) = 1]$, so replacing C with $\neg C$ toggles the sign of the left-hand side.)

Intuitively, (3) says that C is ε more likely to output 1 when the last bit of its input is $f(x)$ compared to when the last bit is completely random, and we want to use this to compute $f(x)$. The idea is that we can imagine picking a bit $b \in \{0, 1\}$ at random (which is essentially our 'initial guess' for what $f(x)$ is), and then running $C(xb)$ and checking whether or not it's 1. The point is that it's more likely to be 1 when $b = f(x)$ (i.e., when our guess was correct) than when $b \neq f(x)$ (i.e., when our guess was incorrect). So if C outputs 1 then we stick with our original guess and output b, while if C outputs 0 then we flip it and output $1 - b$.

To make this precise, we imagine choosing $b \in \{0, 1\}$ at random and defining the circuit $C'_b(x) = C(xb) \oplus (1 - b)$. (Here b is hardcoded into the circuit — so $C'_0(x) = \neg C(x0)$ and $C'_1(x) = C(x1)$. In particular, this means both have size at most $\mathsf{size}(C)$ — we're essentially just plugging in one bit into C, which can only simplify it.) We can show that

$$\mathbb{P}_{x,b}[C'_b(x) = f(x)] \geq \frac{1}{2} + \varepsilon$$

by a direct computation from (3) (this statement essentially just formalizes the above intuition). And this means there *exists* some $b \in \{0, 1\}$ for which $\mathbb{P}_x[C'_b(x) = f(x)] \geq \frac{1}{2} + \varepsilon$ (the *average* over b is at least $\frac{1}{2} + \varepsilon$, so we can take whichever b maximizes the probability), and we can take C' to be C'_b for this choice of b. □

> **Remark 8.31.** In this proof, the circuit isn't choosing b; we ourselves are choosing b (in the construction of C') and then hardcoding it into the circuit.

So we've got a baby PRG where we take x and tack on $f(x)$, and that extends it by one bit. But we really want something that takes $200 \log n$ bits and expands them to n bits, so we need to do something way more sophisticated than just tacking on f once. Somehow we want to tack on f *many* times — i.e., we want to evaluate f at many dfferent points (each coming from a subset of our $200 \log n$ bits). And we need to do this in a clever way, so that the different subsets of bits we're computing f on are 'uncorrelated' in some way — enough that we still get to make the statement that these different values of f can't be predicted from each other.

§8.3.4 NW-designs and the PRG construction

First, here's a high-level picture of what we want our generator $g(f, \bullet)$ to look like — suppose we start with a seed x of $200 \log n$ bits. We've got a hard function on inputs of length $5 \log n$, so we're going to take n (not necessarily contiguous) substrings $y_1, \ldots, y_n$ of x, each of length $5 \log n$ (by fixing n subsets of $[200 \log n]$ — the universe of all indices — each of size $5 \log n$), and then we're going to plug them all into f and set

$$g(f, x) = f(y_1) \ldots f(y_n).$$

We want to choose the index subsets for $y_1, \ldots, y_n$ in such a way that this looks pseudorandom, and intuitively this means we want the y_i's to be 'uncorrelated.' More specifically, we want these index subsets to have small pairwise intersections.

> **Definition 8.32.** A NW-design is a collection of sets $\mathcal{S} = \{S_1, \ldots, S_n\}$ computable in $\mathsf{poly}(n)$ time such that each S_i is a subset of $[200 \log n]$ with size $|S_i| = 5 \log n$, and such that for all $i \neq j$ we have $|S_i \cap S_j| < \log n$.

We're not going to discuss how to construct such a set system, but there's many such designs, with different parameters. Often they can be constructed by greedy algorithms — we've got a universe of size $200 \log n$, which means there's only n^{200} subsets to consider.

We'll also use the following piece of notation for convenience.

Notation 8.33. For a set S, we let x_S denote the substring of x with bits indexed by S.

For example, if $x = 101011$ and $S = \{1, 3, 5\}$, then $x_S = 111$.

We're now ready to define the final PRG — we take

$$g(f, x) = f(x_{S_1}) \cdots f(x_{S_n}),$$

where $\{S_1, \ldots, S_n\}$ is a NW-design. Note that g can be evaluated in polynomial time (we can build the NW-design in $\mathsf{poly}(n)$ time, and then given the truth table of f and our $200 \log n$-bit seed x, we can evaluate f on the appropriate substrings to get a n-bit string).

We'll now state the final theorem we need to prove, which essentially says that this construction really does give us a PRG.

Theorem 8.34

Suppose that for some f, there exists a circuit C of size n such that

$$|\mathbb{P}_x[C(x) = 1] - \mathbb{P}_z[C(g(f, z)) = 1]| \geq \frac{1}{10}.$$

Then there exists a circuit C' of size at most $10n^2$ such that

$$\mathbb{P}_x[C(x) = f(x)] \geq \frac{1}{2} + \frac{1}{10n}.$$

(Here x is chosen from $\{0, 1\}^n$, and z from $\{0, 1\}^{200 \log n}$.)

This is the contrapositive of what we need to show — it states that if there is a circuit that distinguishes between our generator $g(f, \bullet)$ and the uniform distribution, then f is not average-case hard (i.e., there's a circuit that predicts $f(x)$ substantially better than a coin flip).

We're now going to prove this (which will finish the proof of Theorem 8.28).

§8.3.5 A hybrid argument

To prove Theorem 8.34, the idea is that we want to show that if there exists C satisfying the given hypothesis (i.e., distinguishing $g(f, z)$ from random), then there should be some i for which C distinguishes $f(x|_{S_i})$ from random. In the baby case in Theorem 8.30, we only had a single copy of $f(x)$, so we automatically got that C distinguished $f(x)$ from random; but here we've got *lots* of copies, and we want to be able to isolate one of them.

To do so, we use a *hybrid argument*. We define $n + 1$ distributions $\mathcal{H}_0$, …, $\mathcal{H}_n$ where $\mathcal{H}_i$ is defined in the following way:

- Sample a PRG seed $z \in \{0, 1\}^{200 \log n}$ and let $y = g(f, z)$.
- Choose $n - i$ uniform random bits $r_{i+1}, \ldots, r_n \in \{0, 1\}$.
- Output $(y_1, \ldots, y_i, r_{i+1}, \ldots, r_n)$.

So we're taking the first i bits from our PRG's distribution and sticking in completely random bits for the remainder.

First, $\mathcal{H}_0$ is just the uniform distribution on $\{0, 1\}^n$ (because we're just taking n random bits), and $\mathcal{H}_n$ is the PRG's distribution (i.e., the distribution of $g(f, z)$ for a uniform seed z), so the given hypothesis states

$$|\mathbb{P}_{x \sim \mathcal{H}_0}[C(x) = 1] - \mathbb{P}_{x \sim \mathcal{H}_n}[C(x) = 1]| \geq \frac{1}{10}.$$

And the point is that all the intermediate hybrid distributions interpolate between these two extremes — we've got some bits from the PRG's output and some uniform random bits — such that when we go from $\mathcal{H}_{i-1}$ to $\mathcal{H}_i$, we're only changing one bit (namely, we're changing the ith bit from being chosen randomly to being taken from the PRG).

First, for each $i \in \{0, \ldots, n\}$, we define

$$p_i = \mathbb{P}_{x \sim \mathcal{H}_i}[C(x) = 1],$$

so that the hypothesis states that $|p_0 - p_n| \geq \frac{1}{10}$. Just as in the baby case (Theorem 8.30), we can remove the absolute value signs — so we'll assume without loss of generality that $p_n - p_0 \geq \frac{1}{10}$. And we can write this as a telescoping sum — we have

$$p_n - p_0 = \sum_{i=1}^{n} (p_i - p_{i-1}) \geq \frac{1}{10},$$

so since there are only n terms in this sum and they sum to at least $\frac{1}{10}$, at least one of them must be at least $\frac{1}{10n}$ — this means there exists some i for which $p_i - p_{i-1} \geq \frac{1}{10n}$.

Now we're going to fix this index i; our circuit C distinguishes $\mathcal{H}_{i-1}$ from $\mathcal{H}_i$, and we're going to use this to get a circuit that predicts f.

§8.3.6 A circuit predicting f

We've now got a small circuit C such that

$$\mathbb{P}_{x \sim \mathcal{H}_i}[C(x) = 1] - \mathbb{P}_{x \sim \mathcal{H}_{i-1}}[C(x) = 1] \geq \frac{1}{10n}, \tag{4}$$

and we want to use it to obtain a circuit C' that predicts f. The key is that $\mathcal{H}_i$ and $\mathcal{H}_{i-1}$ only differ in the ith bit — the ith bit of $\mathcal{H}_i$ is $f(x_{S_i})$, while the ith bit of $\mathcal{H}_{i-1}$ is completely random. Meanwhile, both have $f(x_{S_j})$ in the jth bit for all $j < i$, and completely random bits for all $j > i$. For notational convenience, we'll assume $S_i = \{1, \ldots, 5 \log n\}$.

So we'll define C' as follows (for now, the construction of C' is going to involve randomness; in the end, we'll fix choices for all these random things and hardcode them into the circuit). The input to C' is a $5 \log n$-bit string x (since we're trying to compute f, which takes $5 \log n$-bit inputs), so we choose a random $195 \log n$-bit string x' uniformly at random to fill out the rest of the seed (we treat xx' as our seed). And we also choose uniform random bits $r_i, r_{i+1}, \ldots, r_n \in \{0, 1\}$.

Then for each $j = 1, \ldots, i-1$, we define $y_j = f((xx')_{S_j})$. We then compute $C(y_1, \ldots, y_{i-1}, r_i, \ldots, r_n)$; if it's 1 then we output r_i, and otherwise we output $1 - r_i$.

There's two things we need to show — that this really does predict f (for appropriate choices of all the randomness), and that it can be implemented with a circuit of size at most $10n^2$. The main point behind why it predicts f (on average) is that for uniform random $x \in \{0, 1\}^{5 \log n}$ (so that xx' is a uniform random $200 \log n$-bit seed), the distribution of $(y_1, \ldots, y_{i-1}, \bullet, r_{i+1}, \ldots, r_n)$ is exactly the distribution given by both $\mathcal{H}_i$ and $\mathcal{H}_{i-1}$ (except for the ith coordinate), and in the ith coordinate $\mathcal{H}_i$ has $f(x)$ while $\mathcal{H}_{i-1}$ has a truly random bit. And (4) means that C is more likely to output 1 when we plug in $f(x)$ in the ith bit than when we plug in a random bit — so if we think of r_i as our 'guess' for $f(x)$, it's more likely to output 1 when our guess is correct. Then by a very similar argument to the baby case (Theorem 8.30), we get

$$\mathbb{P}_{x, x', r_i, \ldots, r_n}[C'(x) = f(x)] \geq \frac{1}{2} + \frac{1}{10n}$$

(where C' is the circuit defined in this way for the given choice of x', r_i, $\ldots$, r_n), which means there is some choice of x', r_i, $\ldots$, r_n such that $\mathbb{P}_x[C'(x) = f(x)] \geq \frac{1}{2} + \frac{1}{10n}$.

The more interesting part of the argument is that for fixed x', r_i, $\dots$, r_n (which we hardcode), we can implement C' with a circuit of size at most $10n^2$. This is *a priori* surprising — we're trying to run C on the input $(y_1, \dots, y_{i-1}, r_i, \dots, r_n)$ where $y_j = f((xx')_{S_j})$ for each j, and it looks like this involves computing f, which has high circuit complexity (we can't compute f in any naive way, because f is supposed to be hard). So how on earth can we do this with a circuit of size only $10n^2$?

This is the magical part, and where the assumption $|S_i \cap S_j| < \log n$ comes in. The idea is that for each j, since $S_i = \{1, \dots, 5 \log n\}$ (so that $x = (xx')_{S_i}$) and $|S_i \cap S_j| < \log n$, we get that $(xx')_{S_j}$ only involves less than $\log n$ bits of x (and all the other bits are constants — they come from x', which is fixed). And so $f((xx')_{S_j})$ is really a function of only $\log n$ bits, which means by Proposition 6.9 that it has a circuit of size at most $2^{\log n}/(\log n) = O(n/(\log n))$.

And we need to do this at most $i - 1 \leq n$ times (we need a separate circuit computing $y_j = f((xx')_{S_j})$ for each $j < i$), so we get a total circuit size of $o(n^2)$, which is good enough.

This completes the proof of Theorem 8.34.

§9 Counting complexity

In the last few classes, we talked about randomized computation and derandomization; the problem of derandomization boils down to essentially trying to estimate $\mathbb{P}_x[C(x) = 1]$ for a given circuit C. Today we're going to discuss counting complexity, which can in some sense be viewed as a strengthening of this — now roughly speaking, we want to compute $\mathbb{P}_x[C(x) = 1]$ *exactly* (for a given circuit C). As you might imagine, this seems like a much harder problem.

More generally, in counting complexity we want to count the *number of solutions* to a NP problem — i.e., the number of witnesses that make its verifier accept. For example, the canonical NP-complete problem is CircuitSAT, where the verifier takes an input C and witness x and checks whether $C(x) = 1$. If we're computing $\mathbb{P}_x[C(x) = 1]$ exactly (under the uniform distribution over x), then we're exactly counting the number of solutions x to the verifier.

§9.1 The class #P

When we think about counting complexity, we're trying to *count* the number of solutions to a NP problem, which means our problems are no longer decision problems, but *function* problems — they're represented by functions $f\colon \{0,1\}^* \to \{0,1\}^*$ rather than $\{0,1\}^* \to \{0,1\}$, where we interpret the output as a nonnegative integer using binary. (By convention, we ignore leading 0's — we say 0^ℓ means 0 and $0^\ell x$ means x.)

Definition 9.1. We say $f \in$ #P if there exists k and a polynomial-time verifier $\mathcal{V}$ such that for all x,

$$f(x) = \#\{y \in \{0,1\}^{k|x|^k} \mid \mathcal{V}(x,y) = 1\}.$$

So we're fixing a length kn^k for our witnesses (on inputs x of length n), and then f prints the exact number of witnesses y of that length which make the verifier output 1.

Another way to think about this definition is that $f \in$ #P if there is a nondeterministic polynomial-time $\mathcal{N}$ such that for all x, the configuration graph $\mathcal{G}_{\mathcal{N},x}$ has exactly $f(x)$ paths from c_{start} to c_{acc}. (This is because we can view the witness y as telling us which sequence of choices to make, so each witness that makes $\mathcal{V}$ accept corresponds to such a path.) But we'll use the verifier definition of #P because it's cleaner.

Remark 9.2. We pronounce #P as 'sharp P' and not 'hashtag P.' Why this class is called #P instead of #NP isn't clear, but we'll see another class with an even more confusing name later today.

§9.1.1 Two classes of problems

There's two classes of #P problems we'll look at. The first consists of problems where testing whether there *exists* a solution is easy, but counting all of them may be more tricky; so it's not obvious whether the problem is easy or hard. One example is counting the number of cycles in a graph.

Definition 9.3 (#Cycle)

- **Input:** a directed graph G.
- **Output:** the number of simple cycles in G.

(A cycle is *simple* if it doesn't visit any vertex more than once — for example, if G has n vertices, then a simple cycle of length n is a Hamiltonian cycle.)

To see that $\#\mathsf{Cycle} \in \#\mathsf{P}$, we can simply make a verifier $\mathcal{V}$ which accepts (G, C) if and only if C is a simple cycle in G (using some canonical encoding of cycles so that each cycle has a unique encoding). Then counting the number of witnesses that make $\mathcal{V}$ accept means we're counting the number of cycles.

Note that checking whether $\#\mathsf{Cycle}(G) = 0$ is in P — trying to figure out whether a graph has a cycle at all can be done in polynomial time by a simple graph search algorithm (e.g., BFS or DFS). So testing whether there's a nonzero number of cycles is easy; but here we want to actually know the *exact number* of cycles.

That's the first type of problem, and it's quite subtle — it's often not clear which problems in this class are easy or hard.

The first class is sort of the more interesting one. Meanwhile, the second class consists of problems obviously defined from NP-hard or NP-complete problems; so you'd naturally expect these to be the hardest problems in #P. For example, we can define counting versions of CircuitSAT or 3SAT in the obvious way.

Definition 9.4

We define $\#\mathsf{CircuitSAT}(C) = \#\{x \mid C(x) = 1\}$ (for a circuit C).

Definition 9.5

We define $\#\mathsf{3SAT}(F) = \#\{x \mid F(x) = 1\}$ (for a 3CNF formula F).

For both of these problems, we know it's already NP-hard to check whether the function is even *nonzero*. As we'll see soon, lots of things we see with NP-completeness extend here.

§9.2 #P vs. FP

There's two natural questions to ask about #P.

Question 9.6. Can #P be solved efficiently?

Question 9.7. Can we identify the 'hardest' problems in #P?

To answer Question 9.6, we need a notion of *feasibility*, or of what it means for a function problem to be efficiently solvable. (To answer Question 9.7, we need a notion of *reductions*; we'll return to this later.)

We define the class FP (which stands for 'function P') to capture what we mean by function problems which can be solved efficiently.

Definition 9.8. We say a function $f\colon \{0,1\}^* \to \{0,1\}^*$ is in FP if there exists a polynomial-time algorithm $\mathcal{A}$ that computes $f(x)$ (on all inputs x).

So here our algorithms $\mathcal{A}$ have an output tape; and we want that on all inputs x, it should print $f(x)$ on its output tape. (This is the most natural way to define what it means for a function problem to be efficiently solvable.)

The first thing we'll say about #P is that it's at least as interesting as FP.

Theorem 9.9

We have $\mathsf{FP} \subseteq \#\mathsf{P}$.

In other words, this says that any polynomial-time computable function corresponds to counting the number of solutions to some verifier.

Proof. Suppose that $\mathcal{A}$ computes some function $f \in \mathsf{FP}$. For simplicity, we'll assume that $\mathcal{A}$ always has outputs of the same length kn^k on all inputs of length n (so there exists k such that $|f(x)| = k\,|x|^k$ for all x) — for most reasonable functions, you can imagine padding them so that this is true.

We then define a verifier $\mathcal{V}$ in the following way.

Algorithm 9.10 ($\mathcal{V}$) **—** On input (x, y):

- Compute $f(x)$ using $\mathcal{A}$.
- If $|y| = k\,|x|^k$ and $y < f(x)$, then *accept*; otherwise *reject*.

(When we write $y < f(x)$, we view both as numbers in $\{0, 1, \ldots, 2^{k|x|^k} - 1\}$.)

Then counting the number of solutions y to $\mathcal{V}$ corresponds exactly to computing $f(x)$. For example, letting $t = k\,|x|^k$, we have $f(x) > 0$ if and only if $\mathcal{V}(x, 0^t)$ accepts, $f(x) > 1$ if and only if $\mathcal{V}(x, 0^{t-1}1)$ accepts, and so on — in general, $f(x)$ is exactly the number of strings of length t which are less than $f(x)$ when interpreted as numbers, which is exactly the number of strings y accepted by $\mathcal{V}$. □

Then we can formalize Question 9.6 in the following way.

Question 9.11. Is $\mathsf{FP} = \#\mathsf{P}$?

The first thing to observe about this question is that it's easier (or at least as easy) to separate #P from FP than to separate NP from P, in the following sense.

Fact 9.12 — If $\mathsf{FP} = \#\mathsf{P}$, then $\mathsf{P} = \mathsf{NP}$.

The point is that if $\mathsf{FP} = \#\mathsf{P}$, then we can even count the *number* of solutions to 3SAT in polynomial time; and if we can do this, we can certainly figure out whether this number is nonzero (which is what we need to do to solve 3SAT).

§9.3 #P-completeness

We'll now address Question 9.7. We want to be able to understand the 'hardest' problems in #P — what can we expect to count, and what can we expect to *not* be able to count (unless $\mathsf{P} = \mathsf{NP}$)? For this, we first need a notion of reductions which will be useful for counting. We're already working with function problems, so we're going to work with Turing reductions here (rather than many-to-one reductions).

Definition 9.13. For $f, g \colon \{0,1\}^* \to \{0,1\}^*$, we say f has a polynomial-time Turing reduction to g if $f \in \mathsf{FP}^g$ — i.e., f can be computed in polynomial time with an oracle for g.

This is a bit different from the reductions we've seen earlier in this class. In particular, here we're allowed to solve f by making a *bunch* of calls to g (rather than just turning a f-problem into a g-problem and calling g only once).

First, here's an example.

Definition 9.14 (MLE)

- **Input:** n Boolean formulas $F_1, \ldots, F_n$ (each taking some vector $x \in \{0,1\}^n$ as input), and n bits $b_1, \ldots, b_n \in \{0,1\}$.
- **Compute:** $\mathbb{P}[x_i = 1 \mid F_j(x) = b_j \text{ for all } j]$ for each $i \in [n]$ (where the prior distribution of x is the uniform distribution on $\{0,1\}^n$).

The idea is that there's some hidden variables $x_1, \ldots, x_n \in \{0, 1\}$ chosen uniformly at random (which we think of as a vector $x = x_1 \ldots x_n$). And we're given Boolean formulas $F_1, \ldots, F_n$, and we get to see their outputs on x — i.e., we're given $b_j = F_j(x)$ for each j. And what we want to figure out is, conditioned on seeing these outputs, what's the likelihood that each x_i was set to 1?

(The name MLE stands for *maximum likelihood estimate* — it's an inference problem where we see some visible output and want to get an estimate on the invisible inputs. This is actually sort of a baby version of the real MLE problem, which is more complicated.)

Theorem 9.15

We have $\mathsf{MLE} \in \mathsf{FP}^{\#\mathsf{SAT}}$.

This means if we can solve #SAT, then we can also solve MLE — i.e., we can compute all these conditional probabilities.

Proof. To solve MLE, for each $i \in [n]$ we want to compute

$$p_i = \frac{\mathbb{P}[x_i = 1 \wedge (F_j(x) = b_j \text{ for all } j)]}{\mathbb{P}[F_j(x) = b_j \text{ for all } j]}.$$

(We've just expanded out the conditional probability using Bayes's rule.) And we can compute both the numerator and denominator by counting the number of solutions to a Boolean formula — define the formula $G(x) = \bigwedge_j (F_j(x) = b_j)$. Then the numerator and denominator of p_i are the number of satisfying assignments to G and to $G \wedge x_i$, respectively, divided by 2^n, so

$$p_i = \frac{\#\mathsf{SAT}(G \wedge x_i)}{\#\mathsf{SAT}(G)}.$$

This means we can solve MLE by just making $n + 1$ calls to #SAT — on inputs G and $G \wedge x_i$ for each $i \in [n]$ — and computing all the desired probabilities. □

We'll now define #P-completeness, in the way we'd expect.

Definition 9.16. We say g is #P-complete if $g \in \#\mathsf{P}$, and for all $f \in \#\mathsf{P}$ we have $f \in \mathsf{FP}^g$.

(The latter condition could equivalently be written as $\#\mathsf{P} \subseteq \mathsf{FP}^g$.)

Fact 9.17 — If some #P-complete problem g is in FP, then $\#\mathsf{P} = \mathsf{FP}$.

The point is that FP is closed under composition — i.e., $\mathsf{FP}^{\mathsf{FP}} = \mathsf{FP}$. So if there were some $g \in \mathsf{FP}$ which were #P-complete, then we'd get that every $f \in \#\mathsf{P}$ is in $\mathsf{FP}^g \subseteq \mathsf{FP}^{\mathsf{FP}} = \mathsf{FP}$.

§9.3.1 #P-completeness for NP-complete problems

First we'll consider the more straightforward class of problems in #P — problems which are built from NP-complete problems. It turns out that lots of these problems are #P-complete; this is pretty natural, but the proofs are a bit subtle.

Theorem 9.18

The problem #CircuitSAT is #P-complete.

Proof. We've already seen #CircuitSAT is in #P. To see that it's #P-hard, the idea is to take the reduction used to show CircuitSAT is NP-complete, stare at it really hard, and realize that it preserves the number of solutions.

More precisely, consider any $f \in \#\mathsf{P}$, and let $\mathcal{V}$ be its verifier — so $f(x) = \#\{y \mid \mathcal{V}(x, y) = 1\}$ for each x. We want to show $f \in \mathsf{FP}^{\#\mathsf{CircuitSAT}}$. The idea is that in order to compute $f(x)$ (with a #CircuitSAT oracle), we take x and build the circuit C_x such that $C_x(y) = \mathcal{V}(x, y)$ for all y of the appropriate length. (We're just performing the transformation we've seen many times in this course where we take an algorithm and turn it into a circuit — $\mathcal{V}$ takes two inputs x and y, and we've fixed x and hardcoded it into the circuit, which takes just y as input.)

Then $f(x) = \#\{y \mid \mathcal{V}(x, y) = 1\} = \#\{y \mid C_x(y) = 1\}$ (since C_x is the circuit corresponding to $\mathcal{V}(x, \bullet)$). But this count is exactly $\#\mathsf{CircuitSAT}(C_x)$, so we can compute $f(x)$ by calling the #CircuitSAT oracle on C_x. ☐

The moral is that this proof worked because the standard reduction from a general NP problem to CircuitSAT is *parsimonious*, meaning that it preserves the number of solutions (i.e., the number of solutions to the original NP problem is the same as the number of solutions to the CircuitSAT problem that the reduction produces), so it can be turned into a reduction from the corresponding #P problem to #CircuitSAT.

You might think that this is because circuits are so close to algorithms. But in fact, if you stare at all the standard NP-completeness reductions you've seen before, you'll (probably) find that they're *all* parsimonious — they preserve the number of solutions. For example, we can prove 3SAT is NP-hard by reducing from CircuitSAT. In this reduction, we start off with a circuit C taking in an input x, and we map it to a 3CNF formula $F(x, y)$ whose input is the same x together with a bunch of extra variables y_g that we throw in to simulate the circuitry (we create a variable y_g for every gate g, which is meant to encode the value we get at that gate in the circuit, and add in clauses enforcing that the y_g's satisfy the logical relations they're supposed to). And the point is that once we fix x, all the y_g's are completely determined — they're forced to be the values at the gates when we run C on x. This means

$$\#\{x \mid C(x) = 1\} = \#\{(x, y) \mid F(x, y) = 1\}$$

(because for each x, if $C(x) = 1$ then there's exactly one y with $F(x, y) = 1$, and if $C(x) = 0$ then there's no such y). So $\#\mathsf{CircuitSAT}(C) = \#\mathsf{3SAT}(F)$, which means $\#\mathsf{CircuitSAT} \in \mathsf{FP}^{\#\mathsf{3SAT}}$. And since #CircuitSAT is #P-complete, this means #3SAT is as well.

And this is true for tons of NP-completeness reductions — the point is that these reductions all involve a nice mapping from solutions to one problem to solutions to the other. So we get a bunch of #P-completeness results for free, just by staring at the corresponding NP-completeness proofs and realizing they're parsimonious in this same way.

Example 9.19

The problems #3SAT, #VertexCover, #IndependentSet, #HamCycle, and #SubsetSum are all #P-complete.

§9.3.2 #P-completeness of #Cycle

So far, we've just seen that for all the standard NP-complete problems, their counting versions are #P-complete; this is to be expected. But what's more interesting is that #P-completeness extends ti way more problems — there are problems where it's easy to decide whether a solution *exists*, but it's nevertheless hard to *count* (or even approximate) the number of solutions. So #P-completeness is quite subtle and complicated compared to NP-completeness, and trying to understand whether a counting problem is easy or hard can be very tricky. The next theorem is an example of this phenomenon.

Theorem 9.20

The problem #Cycle is #P-complete.

So even though we can easily determine whether a cycle in a directed graph *exists*, counting the *number* of cycles is as hard as e.g. counting the number of *Hamiltonian* cycles (where it's much harder to determine whether one exists).

Proof. We've already seen that #Cycle $\in$ P. So now we have to show that it's #P-hard, and we'll do so by reducing #HamCycle (the problem of counting the number of *Hamiltonian* cycles) to #Cycle. (This suffices because #HamCycle is #P-complete, as mentioned in Example 9.19.)

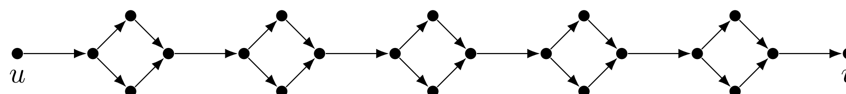
Suppose we're given a directed graph G, and we want to compute #HamCycle(G) (with an oracle for #Cycle). The idea is that we'll construct a bigger graph G' such that every Hamiltonian cycle in G maps to an *enormous* number of cycles in G', while every non-Hamiltonian cycle in G corresponds to a much smaller number of cycles in G'. Then if we can get the number of cycles in G', this will tell us the number of Hamiltonian cycles in G. (In fact, this same argument shows that even *approximately* counting the number of cycles is #P-hard.)

More precisely, we'll construct a polynomial-time reduction that takes a directed graph G on n nodes and maps it to a bigger directed graph G' on $O(n^4)$ nodes, such that

$$\#\mathsf{HamCycle}(G) = k \iff \#\mathsf{Cycle}(G') \in [k \cdot 2^{n^3}, (k+1) \cdot 2^{n^3}). \tag{5}$$

(These numbers are exponentially large, but that's fine — they only take $O(n^3)$ bits to write down, so a polynomial-time algorithm *could* in principle write them down.) Then knowing the number of cycles in G' tells us the number of Hamiltonian cycles in G.

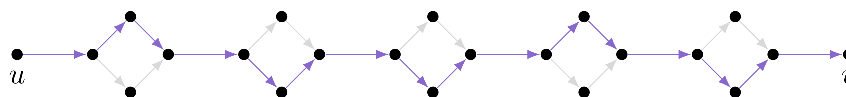
The idea behind the reduction is that we first take all the vertices in G and place them into G'. Then we take every *edge* in G and blow it up in a certain way — for each edge $u \to v$ in G, in G' we draw n^2 diamond gadgets chained together in sequence, as shown below.



(All of the diamonds consist of completely new nodes specific to this one edge, and we have n^2 of these diamonds in sequence.)

Then the number of new nodes in G' is $O(|E| \cdot n^2) = O(n^4)$ (where E is the edge set of G).

And the point of this construction is that if we want to get from u to v in G', we've got n^2 diamonds, and for each we can choose to walk along either the top or the bottom, which gives us 2^{n^2} ways.



This means every cycle of length ℓ in G maps to exactly $2^{n^2\ell}$ cycles in G' — for each edge of the cycle in G, we have 2^{n^2} choices for how to take that edge in G'. (Meanwhile, we can check that every cycle in G' comes from some cycle in G.)

This immediately means that if G has k Hamiltonian cycles, these contribute exactly $k \cdot 2^{n^3}$ cycles to G' (since each Hamiltonian cycle has n edges, and therefore corresponds to exactly 2^{n^3} cycles in G'). So now it remains to show that the non-Hamiltonian cycles in G have much smaller contribution — that in total, they contribute fewer than 2^{n^3} cycles to G'.

First, the *number* of non-Hamiltonian cycles in G is at most n^n — there's fewer than n choices for the length ℓ of the cycle (which must satisfy $3 \le \ell \le n-1$), and then there's at most n choices for each of its $\ell \le n-1$ nodes, giving a total of at most $n \cdot n^{n-1} = n^n$ cycles. (This is a crude bound, but it's more than enough for our purposes.)

And each such cycle in G corresponds to at most $2^{n^2\ell} \le 2^{n^2(n-1)}$ cycles in G', so in total, these cycles in G contribute at most

$$2^{n^2(n-1)} \cdot n^n = 2^{n^3 - n^2 + n\log n} < 2^{n^3}$$

cycles to G' (since $n \log n$ is much less than n^2).

So the k Hamiltonian cycles in G contribute $k \cdot 2^{n^3}$ cycles to G', and all the non-Hamiltonian cycles in G contribute a total of fewer than 2^{n^3}; this gives (5), and we're done. □

There are several other examples of this phenomenon as well (problems whose decision versions are easy, but whose counting versions are #P-complete).

Example 9.21

The problems #2SAT and #PerfectMatching are #P-complete.

§9.4 The class PP — a decision version of counting

It's not obvious that we have to move to function problems to talk about counting. And it turns out that in principle we *don't*, if we allow ourselves polynomial-time Turing reductions — we can get away with using certain decision problems instead. So we're back to working with decision problems — i.e., functions $f : \{0,1\}^* \to \{0,1\}$.

Definition 9.22. For a decision problem $f : \{0,1\}^* \to \{0,1\}$, we say $f \in$ PP if there is some $k \in \mathbb{N}$ and a polynomial-time $\mathcal{V}$ such that for all x, we have

$$f(x) = 1 \iff \mathbb{P}_{y \in \{0,1\}^{k|x|^k}}[\mathcal{V}(x,y) = 1] \ge \frac{1}{2}.$$

So we're supposed to accept if the verifier $\mathcal{V}$ accepts a *majority* of all possible witnesses (of the appropriate length), and reject otherwise.

The name PP is supposed to stand for 'probabilistic P.' This name is kind of unfortunate, since we don't really think of it in terms of anything probabilistic — it's really an exact counting version of P (it's telling us the first bit of the number of solutions to the verifier). Still, the reason for the name is that we can think of this as an extremely precise version of randomized complexity — with BPP we're supposed to accept when $\mathbb{P}_y[\mathcal{V}(x,y) = 1] \ge \frac{2}{3}$ and reject when $\mathbb{P}_y[\mathcal{V}(x,y) = 1] \le \frac{1}{3}$ (with the guarantee that one always holds), which gives us room for error. In contrast, here we don't have room for error — we're supposed to be able to distinguish between $\mathbb{P}_y[\mathcal{V}(x,y) = 1]$ being e.g. $\frac{1}{2} + 2^{-k|x|^k}$ (in which case we're supposed to accept) vs. $\frac{1}{2} - 2^{-k|x|^k}$ (in which case we're supposed to reject).

This class turns out to be very overpowered (Ryan thinks it should be called OP for 'overkill P').

Fact 9.23 — We have NP $\subseteq$ PP.

Proof. Suppose we've got a NP verifier $\mathcal{V}$ for a problem $f \in$ NP — so $f(x) = 1$ if and only if there is *any* y such that $\mathcal{V}(x,y) = 1$. We want to turn it into a PP verifier $\mathcal{V}'$ — so we want $\mathcal{V}'$ to accept at least *half* of all witnesses if and only if $\mathcal{V}$ accepts *any* witness.

The way we do this is by adding an extra bit to the witness y for $\mathcal{V}$; and if that bit is 0 we run $\mathcal{V}$, while if the bit is 1 then we automatically accept all witnesses except one. Explicitly, for y of the witness length m taken by $\mathcal{V}$ and $b \in \{0, 1\}$ a single bit, we define

$$\mathcal{V}'(x, yb) = \begin{cases} \mathcal{V}(x, y) & \text{if } b = 0 \\ 1 & \text{if } b = 1 \text{ and } y \neq 0^m \\ 0 & \text{if } b = 1 \text{ and } y = 0^m. \end{cases}$$

This ensures that we accept at least half of all witnesses if and only if $\mathcal{V}$ accepted some witness. (We're essentially just padding witnesses to shift from 0 to $\frac{1}{2}$.) □

Fact 9.24 — We have $\mathsf{BPP} \subseteq \mathsf{PP}$.

Proof. The point is that if we've got a BPP algorithm $\mathcal{A}(x; y)$ for some f (where x represents the input and y represents the randomness), then we want to figure out whether $\mathcal{A}(x; y)$ accepts or rejects on a $\frac{2}{3}$-fraction of all y's; and we can certainly do this if we can compare the fraction of y's it accepts with $\frac{1}{2}$. Explicitly, we just take $\mathcal{V}$ in the definition of PP to be $\mathcal{A}$ — the fact that it satisfies the BPP requirement (that $f(x) = 1$ means $\mathbb{P}_y[\mathcal{A}(x; y) = 1] \geq \frac{2}{3}$ and $f(x) = 0$ means $\mathbb{P}_y[\mathcal{A}(x; y) = 1] \leq \frac{1}{3}$) means that it also satisfies the PP one (that $f(x) = 1$ if and only if $\mathbb{P}_y[\mathcal{A}(x; y) = 1] \geq \frac{1}{2}$). □

Fact 9.25 — We have $\mathsf{P} = \mathsf{PP}$ if and only if $\mathsf{FP} = \#\mathsf{P}$.

The backwards direction is easy — to solve a problem in PP with verifier $\mathcal{V}$ (on input x), we're just trying to find the most significant bit of $\#\{y \mid \mathcal{V}(x, y) = 1\}$ (since this bit determines whether it's at least half of the total number of y's or not). And the assumption $\mathsf{FP} = \#\mathsf{P}$ means that we can even *count* this number, so we can certainly find its most significant bit.

Meanwhile, the forwards direction follows from the following stronger statement.

Theorem 9.26

We have $\#\mathsf{P} \subseteq \mathsf{FP}^{\mathsf{PP}}$.

This means PP is as powerful as #P, in the sense of having polynomial-time access to an oracle.

Proof sketch. The idea is binary search — suppose we have a verifier $\mathcal{V}$, and we want to count $\#\{y \mid \mathcal{V}(x, y) = 1\}$ in polynomial time with a PP oracle (where $|y| = k\,|x|^k$). Roughly speaking, we can use the same idea as in Fact 9.23 to use our PP oracle to compare the count with *any* number (instead of just half the total number), and this lets us do binary search to find the exact count.

Explicitly, we define the PP oracle as follows: we define a new verifier $\mathcal{V}'$ which takes input (x, c) and a witness yb of length $k\,|x|^k + 1$ (where y has length $k\,|x|^k$ — as with the witnesses to $\mathcal{V}$ for x — and b is a single bit), and such that:

- If $b = 0$, then $\mathcal{V}'((x, c), y0) = \mathcal{V}(x, y)$.
- If $b = 1$, then $\mathcal{V}'((x, c), y1)$ accepts for all but c witnesses y (e.g., it accepts if and only if y is not one of the c lexicographically first strings of length $k\,|x|^k$).

Then $\mathcal{V}'((x, c), yb)$ accepts at least half of all witnesses yb if and only if $\mathcal{V}(x, y)$ accepts at least c witnesses y. So we can define our PP oracle to be the decision problem corresponding to $\mathcal{V}'$; and then we can call this oracle with various values of c to find $\#\{y \mid \mathcal{V}(x, y) = 1\}$ by binary search. □

As we can see, PP is a very powerful class; and getting complete problems for it is pretty tricky. Here's one natural candidate.

Definition 9.27 (MajCircuitSAT)

- **Input:** a circuit C.
- **Decide:** whether $\mathbb{P}_x[C(x) = 1] \geq \frac{1}{2}$ (i.e., whether C accepts at least half of all inputs).

This problem is indeed PP-complete, by the same proof as Cook–Levin. And MajSAT (where we're given a *formula* and want to know if at least half of all assignments are satisfying) is also PP-complete. It was believed for a while that Maj-kSAT is also PP-complete, but it's not — in fact, it's actually in P! This is really surprising — for most classes we've dealt with (such as NP or Σ_2P), restricting clause width (so that there's only e.g. three literals in each clause) never created issues. But it turns out that here it makes the problem go from being PP-complete to actually being solvable in polynomial time. So this is one example of how counting complexity is weird — there's lots of cool stuff going on.

§9.5 The Valiant–Vazirani theorem

Today we'll investigate the following question.

Question 9.28. Does the number of solutions to a problem affect whether it's NP-hard?

For a NP problem, the number of solutions could be anywhere from 0 to 2^w (where w is the length of the witness). And we want to know, does the size of this number affect how hard the problem is? It's plausible that the difficulty of NP-hard problems comes about because of the number of solutions or the way that they're distributed.

In fact, in statistical physics, people study the random 3SAT problem — we fix some number n of variables (which we think of as large, e.g., 1000) and some clause-to-variable ratio α (so the number of clauses is $m = \alpha n$), and we build a formula by sampling each clause uniformly at random out of the space of all possible clauses $\ell_i \vee \ell_j \vee \ell_k$. When the ratio α is 1, with overwhelming probability the formula is satisfiable; and as we increase the ratio, somewhere around $\alpha \approx 4.2$ maybe the probability the formula is satisfiable becomes $\frac{1}{2}$; and beyond that all kinds of weird things happen to the solution space. You can imagine drawing solutions on the 2^n-vertex hypercube (with 2^n nodes and edges between nodes that differ in just one bit), and people study how they're distributed — they're in clusters and then they break apart and then there's no solutions. (The places where these changes happen are called *phase transitions* in satisfiability.)

The intuition is that somehow the solution spaces are 'complicated' and this makes the problem difficult. For example, there's two paradigms of how you might try solving 3SAT. One is backtracking — we start with nothing, pick a variable and assign it a certain value (0 or 1), pick another variable and assign it, and keep doing this until we've broken something; and then we backtrack and change the last assignment, and so on. The other way you could try solving 3SAT is by starting with a full assignment (which is presumably broken) and trying to fix it, by finding a bad clause and flipping some of its variables. And indeed, if the solution spaces are complicated, then these algorithms are going to break; so you might imagine that the difficulty does come from the solution spaces being complicated.

But the main point of today's lecture is that in *general*, this intuition is wrong — the NP-hardness of a problem doesn't necessarily depend on the solution space. In particular, we're going to see that SAT is hard even if we only consider formulas with just *one* solution! Here the solution space is very simple (in some sense, at least), but the problem is still hard.

Theorem 9.29 (Valiant–Vazirani 1986)

Suppose there exists a polynomial-time algorithm $\mathcal{M}$ such that for all formulas F, if $\#\mathsf{SAT}(F) = 1$ then $\mathcal{M}(F)$ outputs a satisfying assignment to F. Then $\mathsf{NP} = \mathsf{RP}$.

In the hypothesis, the algorithm could have arbitrary behavior on all other formulas — for example, if the number of satisfying assignments is 2, it could just output that the formula is unsatisfiable. We only care about its behavior on formulas with exactly one satisfying assignment. And this theorem says that even finding a satisfying assignment for such formulas is hard (assuming $\mathsf{NP} \neq \mathsf{RP}$ — which we think is true, since if we believe in derandomization and circuit lower bounds (as in Theorem 8.24) we even have $\mathsf{RP} = \mathsf{P}$).

§9.5.1 The proof idea

We'll show that under the given hypothesis, we have $\mathsf{SAT} \in \mathsf{RP}$ — given a formula F (with any number of satisfying assignments), we'll produce a randomized algorithm that uses $\mathcal{M}$ to get a satisfying assigment.

For now, imagine that we've got a formula F (which takes inputs $x \in \{0,1\}^n$) with $\#\mathsf{SAT}(F) = 2^k$ for some k. The high-level idea is that we'd like to modify this formula to have only one satisfying assignment, and we'll do so by adding some constraints to the formula that should reduce the number of satisfying assignments (but hopefully not kill all of them). And the way we'll do this is by hashing.

For now, imagine that we take a completely random 'hash function' $h\colon \{0,1\}^n \to \{0,1\}^k$ (which takes n bits and outputs k bits), and construct the new formula

$$G(x) = F(x) \wedge (h(x) = 0^k).$$

So we're taking our original formula F, and imposing the additional constraint that the assignment should hash to the all-0's string. This should reduce the number of satisfying assignments — we have

$$\mathbb{P}_h[h(x) = 0^k] = \frac{1}{2^k}$$

for each $x \in \{0,1\}^n$ (since we're choosing h to be a completely random function, so $h(x)$ is equally likely to be any of the 2^k possible values), so by linearity of expectation

$$\mathbb{E}_h[\#\mathsf{SAT}(G)] = \frac{\#\mathsf{SAT}(F)}{2^k} = 1.$$

This in some sense gives us a 'reduction' from SAT to the problem of SAT where there's just one satisfying assignment.

But there's at least two problems with this idea (or ways in which it doesn't make sense).

The first problem is that if we're picking h to be a completely random function, we'd need $\Omega(2^n \cdot k)$ bits to describe it. So this reduction is certainly not polynomial-time. But this can be fixed — we don't need h to be completely random, just pseudorandom (in ways we'll make precise later).

The second problem is that when we chose our hash function h, we assumed that $\#\mathsf{SAT}(F)$ was 2^k. But this doesn't make sense — when we're trying to solve SAT, we're trying to figure out whether this number of solutions is 0 or not, so we certainly don't know this number.

But the way we'll get around this is that for the pseudorandom hashing argument to work, we won't actually need to know the *exact* value of $\#\mathsf{SAT}(F)$ — we'll only need to know it within a factor of 2. And then there's only $n+1$ possible values we need to try (i.e., 2^0, 2^1, ..., 2^n), and we can just try them all one at a time — as long as $\#\mathsf{SAT}(F)$ is nonzero, this is going to work for some value of k.

§9.5.2 Pairwise independent hash families

Instead of choosing h to be a completely random function, we're going to choose h from a *pairwise independent hash family*; we'll first define these and set up their relevant properties.

Definition 9.30. We say a family of functions $\mathcal{H} = \{h\colon \{0,1\}^n \to \{0,1\}^k\}$ is pairwise independent if for all $x, y \in \{0,1\}^n$ with $x \neq y$ and all $a, b \in \{0,1\}^k$, we have

$$\mathbb{P}_{h\in\mathcal{H}}[(h(x) = a) \wedge (h(y) = b)] = \frac{1}{2^{2k}}.$$

This definition states that for $h \in \mathcal{H}$ chosen uniformly at random, the distribution of $(h(x), h(y))$ is uniform in $\{0,1\}^k \times \{0,1\}^k$ — so if we just look at two inputs at a time, h looks as if it's a completely random function. (In particular, this definition means $\mathbb{P}_h[h(x) = a] = \frac{1}{2^k}$ for all a, so each $h(x)$ is uniformly distributed.)

On the problem set, we proved the following fact.

Fact 9.31 — The family of functions

$$\mathcal{H}^*_{n,k} = \{h_{M,r}\colon x \mapsto Mx \oplus r \mid M \in \{0,1\}^{k\times n},\ r \in \{0,1\}^k\}$$

is a pairwise independent family.

Here our family of hash functions is indexed by a $k \times n$ matrix M and a k-bit vector r, and the function corresponding to M and r is defined as $h_{M,r}(x) = Mx \oplus r$ (where all computations are performed mod 2). It's much easier to write down such a function — we just need kn bits to specify M and k to specify r, which is much fewer than the $2^n k$ bits we'd need to describe a completely random function $\{0,1\}^n \to \{0,1\}^k$.

The main lemma we'll need is the following (we'll state it in a quite general way, and it's super useful — not just for this, but for several other arguments as well).

Lemma 9.32

Suppose that $S \subseteq \{0,1\}^n$ and k are such that $|S| \in [2^{k-2}, 2^{k-1}]$. Let $\mathcal{H}_{n,k}$ be any pairwise independent hash family, and suppose we choose $h \in \mathcal{H}_{n,k}$ uniformly at random. Then

$$\mathbb{P}_h[\text{there is a unique } x \in S \text{ with } h(x) = 0^k] \geq \frac{1}{8}.$$

So we've got an arbitrary set S of n-bit strings, and we're hashing these strings onto some range of size 2^k (where $|S|$ is a constant factor smaller than this range). And we want to know, what's the probability that there's *exactly one* string in S that hashes to 0^k? (Here 0^k is a placeholder for an arbitrary string in $\{0,1\}^k$ — the same holds for any string.) We can imagine there's 2^k buckets, one for each possible output, and we're hashing the strings in S into these buckets; and we want to know the probability that the 0^k-bucket has exactly one string put into it. And Lemma 9.32 states that this happens with decent probability.

The idea is that once we have this lemma, we can use it to isolate exactly one solution by imposing the condition $h(x) = 0^k$ on our satisfying assignments.

We'll now prove this. The proof is a bit technical, but this sort of thing is super useful in complexity theory (and algorithms), and we'll use various versions of it in later lectures.

Proof of Lemma 9.32. First, for notational convenience, let $p = \frac{1}{2^k}$ be the probability that any fixed string $x \in \{0,1\}^n$ is hashed to 0^k. Then by pairwise independence, for all $x \neq y$ we have $\mathbb{P}_h[h(x) = h(y) = 0^k] = p^2$. Meanwhile, the fact that $2^{k-2} \leq |S| \leq 2^{k-1}$ means that $\frac{1}{4} \leq p\,|S| \leq \frac{1}{2}$.

We'll first *fix* some $x \in S$ and try to calculate the probability that it's the *only* element of S that hashes to 0^k, which we can write as

$$\mathbb{P}_h[h(x) = 0^k] - \mathbb{P}_h[h(x) = 0^k \text{ and } h(y) = 0^k \text{ for some } y \in S \setminus \{x\}]. \qquad (*)$$

We know $\mathbb{P}_h[h(x) = 0^k] = p$; now let's try to upper-bound the second term. First, by a union-bound over all possible y's, it's at most

$$\sum_{y \in S \setminus \{x\}} \mathbb{P}_h[h(x) = h(y) = 0^k] = (|S| - 1)p^2$$

(by pairwise independence). And $p\,|S| \leq \frac{1}{2}$, so this is at most $\frac{1}{2}p$; plugging this in gives

$$(*) \geq p - (|S| - 1)p^2 \geq p - \frac{1}{2}p = \frac{1}{2}p.$$

So now we've analyzed the probability that a *fixed* $x \in S$ is hashed to 0^k. This probability is pretty low — it's just $\frac{1}{2^{k+1}}$ — but now we're going to take this calculation and use it to get the probability that there is *some* such $x \in S$, which is

$$\sum_{x \in S} \mathbb{P}[x \text{ is the only element in } S \text{ with } h(x) = 0^k] \geq \frac{1}{2}p\,|S|$$

(crucially, this is true because these events over different choices of x are disjoint — there can't be two distinct $x, x' \in S$ which are *both* the only element of S hashing to 0^k). And finally, we had $p\,|S| \geq \frac{1}{4}$, so this probability is at least $\frac{1}{8}$. □

So the only property we need of the function h to get the desired 'isolation' behavior is pairwise independence — we don't actually need h to be a completely random function.

§9.5.3 The isolation lemma

We're now going to use pairwise independent hashing and Lemma 9.32 to reduce from SAT to the special case of SAT where there's exactly one satisfying assignment — we're essentially going to implement the construction from Lemma 9.32 on the solutions to the SAT problem in an algorithmic way.

Lemma 9.33 (Isolation lemma)

There is a randomized polynomial-time algorithm $\mathcal{A}$ such that for all n and k, $\mathcal{A}(1^n, 1^k)$ outputs a formula $G_{n,k}(x, y)$ where $|x| = n$ and $|y| = \mathsf{poly}(n)$ and such that for any function $F\colon \{0,1\}^n \to \{0,1\}$, if $\#\mathsf{SAT}(F) \in [2^{k-2}, 2^{k-1}]$, then

$$\mathbb{P}_{\mathcal{A}}[\#\mathsf{SAT}(F \wedge G_{n,k}) = 1] \geq \frac{1}{8}.$$

So $\mathcal{A}$ is given n and k in unary (which will correspond to the domain and range of the hash function from Lemma 9.32); and it's going to toss some coins and output some formula $G_{n,k}$ (the formula it outputs will be different over different choices of its random coins — so it's really producing a *distribution* over formulas). And for *any* function $F\colon \{0,1\}^n \to \{0,1\}$, with decent probability our algorithm $\mathcal{A}$ — which doesn't look at F at all — produces a formula $G_{n,k}$ that isolates one solution to F. (We think of a solution to F as an input x with $F(x) = 1$ — in particular, $\#\mathsf{SAT}(F)$ counts the number of such inputs.) (Note that here F can be arbitrarily complicated — this isn't important for our purposes, where F will just be the given input formula, but it's useful in other places.)

Proof. The idea is that we'll implement Lemma 9.32 in an algorithmic way, taking $S = \{x \mid F(x) = 1\}$.

So given n and k, we first choose a hash function $h \in \mathcal{H}^*_{n,k}$ at random (where $\mathcal{H}^*_{n,k}$ is the hash family from Fact 9.31). We then construct a circuit C that takes in an input x and checks whether $h(x) = 0^k$ — we can do this because the hash functions $h_{M,r}$ in the family are efficiently computable in the sense that there's a polynomial-time algorithm to compute $h_{M,r}(x)$ given (M, r, x), and so we can have $\mathcal{A}$ run the usual algorithm-to-circuit reduction and hardcode in the values of M and r it chose.

And then we can convert C to a 3CNF formula G (taking in x as well as extra variables y) using the Cook–Levin reduction. And the Cook–Levin reduction has the nice property that it's *parsimonious* (as discussed last lecture, in Subsubsection 9.3.1) — so for each $x \in \{0,1\}^n$, if $h(x) = 0^k$ (i.e., if $C(x) = 1$) then there is a *unique* y such that $G(x, y) = 1$ (and if $h(x) \neq 0^k$, then there is no such y).

This means $\#\mathsf{SAT}(F(x) \wedge G(x, y))$ is precisely $\#\{x \mid F(x) = 1 \text{ and } h(x) = 0^k\}$. Then Lemma 9.32 with $S = \{x \mid F(x) = 1\}$ gives precisely that the probability we want is at least $\frac{1}{8}$. □

§9.5.4 Proof of the Valiant–Vazirani theorem

Now we're finally ready to prove Theorem 9.29.

Suppose that there's a polynomial-time algorithm $\mathcal{M}$ such that for all formulas F, when there's a unique satisfying assignment to F, it'll find that assignment. We want to use $\mathcal{M}$ to get a RP algorithm for SAT in general (this is enough to imply $\mathsf{RP} = \mathsf{NP}$, since SAT is RP-complete).

The idea is that given a formula F, we'll essentially use Lemma 9.33 to try to get a formula with a unique satisfying assigment (by calling $\mathcal{A}$ — we'll have to call $\mathcal{A}$ some constant number of times so that it actually succeeds with high probability). And we don't actually know the right value of k, so we're going to brute-force over all possibilities.

Algorithm 9.34 — On input F (in n variables), for each $k \in \{2, \ldots, n+1\}$, perform 20 trials as follows:

- Run $\mathcal{A}(1^n, 1^k)$ to produce some $G_{n,k}$, and call $\mathcal{M}$ on $F \wedge G_{n,k}$.
- Check whether the output of $\mathcal{M}$ is a satisfying assignment to F. If yes, then *accept*.

In the end, if we've gone through all values of k and all trials without ever having found a satisfying assignment, then *reject*.

(Note that this is an oracle reduction, in the sense that we're calling $\mathcal{M}$ several times.)

To see that this algorithm works, first suppose F is unsatisfiable. Then no matter how magical $\mathcal{M}$ is, it can't produce a satisfying assignment to F when such a thing doesn't exist. So in this case, our algorithm will *always* reach the end and reject.

On the other hand, now suppose F is satisfiable. Then there must exist some $k \in \{2, \ldots, n+1\}$ such that $\#\mathsf{SAT}(F) \in [2^{k-2}, 2^{k-1}]$. And for this value of k, by Lemma 9.33 we have

$$\mathbb{P}_{\mathcal{A}}[\#\mathsf{SAT}(F \wedge G_{n,k}) = 1] \geq \frac{1}{8}$$

(where the probability is over the randomness of $\mathcal{A}$) for each of the 20 trials, so the probability there is *some* trial for which it occurs is at least $1 - (\frac{7}{8})^{20} \geq \frac{2}{3}$. And if this ever occurs, then $\mathcal{M}$ has to give us a satisfying assignment (as $F \wedge G_{n,k}$ has a unique satisfying assignment), which means we'll accept (and we even get a satisfying assignment to F).

Remark 9.35. As we can see, this algorithm is randomized, and derandomizing it (i.e., changing the conclusion of Theorem 9.29 from NP = RP to P = NP) is a big open problem. There's a sense in which if we're going to do our isolation at an extremely high level — where it's supposed to work regardless of the function F (as in Lemma 9.33) — then we need randomness. But in principle, if we're given F and we're trying to isolate a solution to it, it might be possible to stare at F and get enough information from it to do this isolation in a deterministic way. And in fact, there are circuit lower bounds that imply this algorithm can be derandomized.

§9.6 Toda's theorem — $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$

Today we'll prove a very powerful theorem about counting — we've previously talked about the polynomial hierarchy, and then counting complexity. And it turns out that counting can simulate *everything* in the polynomial hierarchy.

Theorem 9.36 (Toda's theorem)
We have $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

(Note that $\mathsf{P}^{\#\mathsf{P}} = \mathsf{P}^{\mathsf{PP}}$, so we could make the same statement about P^{PP}.)

For example, this means #SAT is Σ_kP-hard for all k (using Turing reductions) — so things like logic minimization problems (e.g., the problem where we're given a formula and want to find the smallest formula equivalent to it) can be solved by counting.

And we've previously seen that $\mathsf{PH} \subseteq \mathsf{PSPACE}$ (for example, the problem TQBF is complete for PSPACE, and TQBF with at most k alternating quantifiers is Σ_kP-complete). Toda's theorem means that counting lies somewhere in between — i.e., $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}} \subseteq \mathsf{PSPACE}$. (The fact that $\mathsf{P}^{\#\mathsf{P}} \subseteq \mathsf{PSPACE}$ is just because we can count with a polynomial amount of space by just trying all possible solutions.)

§9.6.1 Relativized SAT

The proof of Toda's theorem shows the power of relativization. We've talked about relativization as a *barrier*, but it's also very powerful — when a theorem relativizes, you get uncountably many corollaries for free, and today we'll exploit this a lot.

In earlier classes, we've defined relativized complexity classes such as P^A and NP^A (where we have a machine model that can make A-oracle queries). Now we're going to define certain *problems* (in particular, SAT) with respect to oracles.

Definition 9.37. An A-oracle formula in variables $x_1, \ldots, x_n$ is a Booean formula over $\wedge$, $\vee$, and $\neg$ whose atoms are constants, variables and their negations, and expressions of the form $A(x_{i_1} \ldots x_{i_k})$ (for all $k \in \mathbb{N}$ and indices $i_1, \ldots, i_k \in [n]$).

In a typical formula, the atoms are just variables and their negations; but here we allow A-oracle queries as atoms as well.

Example 9.38
One example of an A-oracle formula is $(x_1 \vee A(x_2x_3) \vee \overline{x_3}) \wedge \overline{x_1} \wedge A(x_2x_3x_4)$.

For example, if $A(000) = 1$, then $x_1 = x_2 = x_3 = x_4 = 0$ is a satisfying assignment to this formula.

Definition 9.39 (SAT^A)

- **Input:** an A-oracle formula φ^A.
- **Decide:** whether φ^A has a satisfying assignment.

This gives us a version of SAT relative to any function A (which we call SAT^A). Note that A is *not* part of the input to SAT^A — the input is just a formula that has some A-predicates in it, as in Example 9.38.

Remark 9.40. In this definition, we don't allow nested A's — for example, we're not allowed to have something like $A(A(x_1, x_2), \dots)$. But this doesn't really matter — we could imagine defining a new variable y, creating a different clause that asserts $y = A(x_1, x_2)$, and then replacing $A(x_1, x_2)$ with y so that we have $A(y, \dots)$ here instead.

And the fact that SAT is NP-complete relativizes in the following sense.

Theorem 9.41 (Relativized Cook–Levin)

For any oracle A, SAT^A is NP^A-complete.

(Even $\mathsf{3SAT}^A$ is NP^A-complete.)

The idea of the proof is that we can still use the translation from computation tableaus to circuits or formulas that we've used many times — imagine we have some computation tableau of an A-oracle Turing machine, with a part of its tape devoted to A-oracle calls. If we think about how to translate this tableau into a circuit (and then into a formula), each of these oracle calls ends up corresponding to one of these A-predicates.

Remark 9.42. When we define NP^A-completeness, we're still using ordinary polynomial-time reductions (the reductions don't need to use an A-oracle).

Example 9.43

The problem $\mathsf{SAT}^{\mathsf{SAT}}$ is complete for $\mathsf{NP}^{\mathsf{NP}} = \Sigma_2\mathsf{P}$.

§9.6.2 The class $\oplus\mathsf{P}$

We'll also need to define one new complexity class $\oplus\mathsf{P}$ (which is another class of decision problems), since the shortest path to Toda's theorem goes through it.

Definition 9.44. We say a decision problem $f \colon \{0,1\}^* \to \{0,1\}$ is in $\oplus\mathsf{P}$ if there exists $g \in \#\mathsf{P}$ such that for all x, we have $f(x) \equiv g(x) \pmod 2$.

So $\oplus\mathsf{P}$ is essentially #P mod 2 — we can think of PP as grabbing the most significant bit of a #P function $g(x) = \#\{y \mid \mathcal{V}(x, y) = 1\}$, and $\oplus\mathsf{P}$ as grabbing the *least* significant bit. Equivalently, $f \in \oplus\mathsf{P}$ if there is a nondeterministic polynomial-time $\mathcal{N}$ such that $f(x)$ is the number of accepting paths of $\mathcal{N}(x)$ mod 2.

Definition 9.45 ($\oplus\mathsf{SAT}$)

- **Input:** a Boolean formula φ.
- **Decide:** whether the number of satisfying assignments to φ is odd or even.

(In other words, $\oplus$SAT is the problem of finding #SAT mod 2.)

Lemma 9.46

The problem $\oplus$SAT is $\oplus$P-complete.

Proof sketch. This again follows from the fact that the usual NP-completeness reduction to SAT is parsimonious (i.e., it preserves the number of witnesses) — if we have an arbitrary problem in $\oplus$P with verifier $\mathcal{V}$ such that $\mathcal{V}$ on x accepts k witnesses, then when we put it through the Cook–Levin reduction, the formula φ we get will also have k satisfying assignments, so it certainly has the same *parity* of the number of satisfying assignments. □

§9.6.3 A sequence of lemmas

Toda's theorem will follow from a sequence of lemmas, which we'll prove one by one in the remainder of the lecture. The first is very similar to the Valiant–Vazirani theorem (Theorem 9.29) from last class.

Lemma 9.47

We have $\mathsf{NP} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}}$. Furthermore, this relativizes, i.e., $\mathsf{NP}^A \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}^A}$ for all oracles A.

Last class (in Theorem 9.29), we essentially saw that if we have an oracle that distinguishes between whether a formula has 0 or 1 solutions, then we can solve SAT. (This isn't exactly what we proved, but as we'll see later today, the same proof gives this statement.) And a $\oplus$P oracle can definitely do this (since if we've got the parity of the number of solutions, then we can certainly distinguish between 0 and 1).

The next lemma is a remarkable property of $\oplus$P that explains why we're going through it in this proof.

Lemma 9.48

We have $\oplus\mathsf{P}^{\oplus\mathsf{SAT}} \subseteq \oplus\mathsf{P}$. Furthermore, this relativizes, i.e., $\oplus\mathsf{P}^{\oplus\mathsf{SAT}^A} \subseteq \oplus\mathsf{P}^A$ for all oracles A.

This essentially means $\oplus$P is closed under composition (as $\oplus$SAT is complete for $\oplus$P). This is quite surprising — we don't believe this is true of e.g. NP or Σ_2P or PP, but it happens to be true for $\oplus$P. (It's also true for P and BPP, but those are much less surprising.)

(Of course, in both lemmas we can replace $\subseteq$ with $=$; but we only need the direction of containment stated in the lemmas.)

The next lemma is sort of a generalization of the fact that if NP = P then PH = P (Theorem 2.44); and we showed on a problem set that this proof relativizes.

Lemma 9.49

For all oracles A, if $\mathsf{NP}^A \subseteq \mathsf{BPP}^A$, then $\mathsf{PH}^A \subseteq \mathsf{BPP}^A$.

We proved this with P in place of BPP (in both locations). But it's a lot more subtle with BPP — the best containment we have regarding BPP and the polynomial hierarchy is $\mathsf{BPP} \subseteq \Sigma_2\mathsf{P}$, so this isn't at all obvious, and there's a trick involved.

And the final lemma, which we will not prove in this lecture, is the following.

Lemma 9.50
We have $\mathsf{BPP}^{\oplus\mathsf{P}} \subseteq \mathsf{P}^{\#\mathsf{P}}$.

This is some kind of derandomization result — on the left-hand side we've got a randomized algorithm that gets to make calls grabbing the least significant bit of a #P function. And somehow we can derandomize it if we're allowed to instead get *all* the bits of a #P function.

First we'll see why Toda's theorem follows from these lemmas; then we'll prove them one by one.

§9.6.4 Proof of Toda's theorem from the lemmas

First we'll see how we prove Toda's theorem (Theorem 9.36) assuming all these lemmas. In this proof, we'll see why it's useful to have relativizing statements — we're going to plug in several oracles A into the lemmas to get the proofs to go through.

We first start with $\mathsf{NP}^{\oplus\mathsf{SAT}}$ — if we stick in $A = \oplus\mathsf{SAT}$ into Lemma 9.47, it tells us that

$$\mathsf{NP}^{\oplus\mathsf{SAT}} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}^{\oplus\mathsf{SAT}}}.$$

And then Lemma 9.48 tells us $\oplus\mathsf{SAT}^{\oplus\mathsf{SAT}} \in \oplus\mathsf{P}$ (because $\oplus\mathsf{SAT}^A$ is in $\oplus\mathsf{P}^A$ for any A, and $\oplus\mathsf{P}^{\oplus\mathsf{SAT}} = \oplus\mathsf{P}$), so we can replace the $\oplus\mathsf{SAT}^{\oplus\mathsf{SAT}}$ oracle with an $\oplus\mathsf{P}$-oracle. And $\oplus\mathsf{SAT}$ is $\oplus\mathsf{P}$-complete (Lemma 9.46), so we can replace this $\oplus\mathsf{P}$-oracle with an $\oplus\mathsf{SAT}$-oracle; this gives

$$\mathsf{BPP}^{\oplus\mathsf{SAT}^{\oplus\mathsf{SAT}}} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}}.$$

Combining these, we've now got the statement $\mathsf{NP}^{\oplus\mathsf{SAT}} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}}$, so by Lemma 9.49 with $A = \oplus\mathsf{SAT}$ we get that $\mathsf{PH}^{\oplus\mathsf{SAT}} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}}$. And finally, by Lemma 9.50 we get $\mathsf{BPP}^{\oplus\mathsf{SAT}} \subseteq \mathsf{P}^{\#\mathsf{P}}$. So in the end, we get $\mathsf{PH}^{\oplus\mathsf{SAT}} \subseteq \mathsf{P}^{\#\mathsf{P}}$ (which of course means $\mathsf{PH} \subseteq \mathsf{P}^{\#\mathsf{P}}$ — the statement we get is slightly stronger, but the difference isn't super important because the main reason we defined $\oplus\mathsf{SAT}$ was to prove this theorem).

Remark 9.51. Even without Lemma 9.50 (which we're not going to prove), this is quite remarkable — it shows that $\mathsf{PH} \subseteq \mathsf{BPP}^{\oplus\mathsf{P}}$, which means for any problem in the polynomial hierarchy, we can get a *randomized* simulation with an oracle that lets us count mod 2.

§9.6.5 Proof of Lemma 9.47 — $\mathsf{NP} \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}}$

We'll start with Lemma 9.47, which should be the easiest — it's basically a recap of some proofs we saw last lecture (in proving Theorem 9.29).

Last lecture we proved the isolation lemma (Lemma 9.33); here's a slight restatement of it.

Lemma 9.52 (Isolation lemma)
There is a randomized polynomial-time algorithm $\mathcal{M}$ such that for all n, $\mathcal{M}(1^n)$ outputs a formula $G(x, y)$ with the property that for every $F\colon \{0,1\}^n \to \{0,1\}$ (with input $x \in \{0,1\}^n$), we have

$$\mathbb{P}_{\mathcal{M}}[\#\mathsf{SAT}(F \wedge G) = 1] \geq \frac{1}{8n}.$$

Last class, we said $\mathcal{M}$ takes in both 1^k and 1^n, and if $k \approx \log_2 \#\mathsf{SAT}(F)$ then we get a probability of at least $\frac{1}{8}$. But we can remove the parameter k by just having $\mathcal{M}$ randomly guess it — this just loses a factor of $\frac{1}{n}$ (since there are at most n possibilities for k, one of which works).

Proof sketch. We randomly pick $k \in \{2, \dots, n+1\}$ and a pairwise independent hash function $h\colon \{0,1\}^n \to \{0,1\}^k$ (from some efficiently computable hash family), and then define $G(x, y)$ as the formula asserting $h(x) = 0^k$ (given by the Cook–Levin reduction). □

Note that $\mathcal{M}$ doesn't take in the function F — this function can be arbitrarily complicated, and in particular it could even be an A-oracle formula — which is why we'll be able to stick oracles into Lemma 9.47.

Now we can use this to prove Lemma 9.47 — i.e., to give a randomized algorithm for SAT^A with an $\oplus\mathsf{SAT}^A$ oracle. (This suffices because SAT^A is NP^A-complete.)

Algorithm 9.53 — On input F^A (in n variables), repeat the following $20n$ times:

- Run $\mathcal{M}(1^n)$ to get a formula G.
- Call the $\oplus\mathsf{SAT}^A$ oracle on $F^A \wedge G$; if it outputs 1, then *accept*.

If all $20n$ iterations fail, then *reject*.

The reason this works is that if F^A is satisfiable, then the isolation lemma (Lemma 9.52) means that with high probability, there is some trial on which $\#\mathsf{SAT}^A(F^A \wedge G) = 1$ (i.e., $F^A \wedge G$ has exactly one satisfying assignment), which of course means $\oplus\mathsf{SAT}^A(F^A \wedge G) = 1$ as well. (We're performing $20n$ trials so that our failure probability $(1 - \frac{1}{8n})^{20n}$ is tiny.)

On the other hand, if F^A is *not* satisfiable, then no matter what A is we'll have $\#\mathsf{SAT}^A(F^A \wedge G) = 0$ (i.e., there are no satisfying assignments), which of course means $\oplus\mathsf{SAT}^A(F^A \wedge G) = 0$ as well.

So this gives us $\mathsf{SAT}^A \in \mathsf{BPP}^{\oplus\mathsf{SAT}^A}$, and since SAT^A is NP^A-complete, we get $\mathsf{NP}^A \subseteq \mathsf{BPP}^{\oplus\mathsf{SAT}^A}$.

Remark 9.54. In fact, this algorithm only has one-sided error (if F^A is unsatisfiable, we'll never say it's satisfiable), so we can even replace BPP with RP.

§9.6.6 Proof of Lemma 9.49 — from NP ⊆ BPP to PH ⊆ BPP

Lemma 9.48 is by far the most involved and mind-bending (how on earth can we simulate $\oplus\mathsf{P}^{\oplus\mathsf{P}}$ with $\oplus\mathsf{P}$?), so we'll jump to Lemma 9.49 first (it's got a trick too, but the trick is less tricky).

To prove Lemma 9.49, we need to show that if $\mathsf{NP}^A \subseteq \mathsf{BPP}^A$, then $\Sigma_k\mathsf{P}^A \subseteq \mathsf{BPP}^A$ for all k (this suffices, because $\mathsf{PH}^A = \bigcup_k \Sigma_k\mathsf{P}^A$). This is done by induction on k. The case $k = 1$ is vacuous ($\Sigma_1\mathsf{P}$ is just NP). The case $k = 2$ is already interesting, and shows all the ideas needed for the inductive step, so we'll only do this case — so we're given that $\mathsf{NP}^A \subseteq \mathsf{BPP}^A$, and we want to conclude that $\Sigma_2\mathsf{P}^A \subseteq \mathsf{BPP}^A$.

Suppose we start with some $f \in \Sigma_2\mathsf{P}^A$ — this means there's a polynomial-time verifier $\mathcal{V}^A$ such that

$$f(x) = 1 \iff (\exists^{\mathsf{P}}\, y)(\forall^{\mathsf{P}}\, z)[\mathcal{V}^A(x, y, z) = 1] \tag{6}$$

(where $\exists^{\mathsf{P}}\, y$ means 'there exists y of polynomial length').

The first observation is that if $\mathsf{NP}^A \subseteq \mathsf{BPP}^A$, then $\mathsf{coNP}^A \subseteq \mathsf{BPP}^A$ (because BPP^A is closed under complement — we can just flip the output of our BPP algorithm — and the complements of NP problems are coNP problems).

And now we can stare at (6) and note that if we're given x and y as input, then the remainder of (6) — the part $(\exists^{\mathsf{P}}\, z)[\mathcal{V}^A(x, y, z) = 1]$ — is a coNP^A problem. And so the assumption $\mathsf{coNP}^A \subseteq \mathsf{BPP}^A$ lets us replace it with a BPP^A algorithm $\mathcal{W}^A$ — more precisely, a deterministic algorithm $\mathcal{W}^A$ that takes x, y, and some randomness r. So we've got a universal quantifier over z in (6), and we're replacing it with randomness r.

More precisely, let $|x| = n$ and $|y| = p(n)$ (for some polynomial p). Then using error reduction (Lemma 7.4), we can assume that the error probability of $\mathcal{W}^A$ is at most $2^{-2p(n)}$ — so it's much smaller than the total number of possible witnesses y. Once we do this, if $f(x) = 1$, then there exists y such that

$$\mathbb{P}_r[\mathcal{W}^A(x, y; r) = 1] \geq 1 - \frac{1}{2^{2p(n)}},$$

while if $f(x) = 0$, then for *all* y we have

$$\mathbb{P}_r[\mathcal{W}^A(x, y; r) = 1] \leq \frac{1}{2^{2p(n)}}.$$

So far, what we've done is we started with a function $f \in \Sigma_2\mathsf{P}^A = \mathsf{NP}^{\mathsf{NP}^A}$ and showed it's in $\mathsf{NP}^{\mathsf{BPP}^A}$ (since we're existentially guessing a witness y, and then $\mathcal{W}^A$ is a BPP^A algorithm). What we want to do next is to flip the NP and BPP — we want to show that $f \in \mathsf{BPP}^{\mathsf{NP}^A}$. (Right now we've got a NP algorithm that calls a BPP^A oracle, and we want to instead get a BPP algorithm that calls a NP^A oracle.) Why do we want this? If we can show $f \in \mathsf{BPP}^{\mathsf{NP}^A}$, then since we're assuming $\mathsf{NP}^A \subseteq \mathsf{BPP}^A$, we can replace the NP^A oracle with a BPP^A one to get $f \in \mathsf{BPP}^{\mathsf{BPP}^A}$. But BPP is closed under composition, so $\mathsf{BPP}^{\mathsf{BPP}^A} = \mathsf{BPP}^A$; so this will mean $f \in \mathsf{BPP}^A$, which is precisely what we wanted to show.

And the idea behind how we swap the NP and the BPP is that we literally just swap the y and r — right now we're first quantifying over y and then r, and we're going to swap them to quantify over r and then y. The point is that we've made the error probability of $\mathcal{W}^A$ so small that this will work (by a union bound).

Let's see why this is true. First let's consider the case $f(x) = 1$ — so there exists some y such that

$$\mathbb{P}_r[\mathcal{W}^A(x, y; r) = 1] \geq 1 - \frac{1}{2^{2p(n)}}.$$

Let y^* be such a choice of y. Then we certainly have

$$\mathbb{P}_r[(\exists\, y)[\mathcal{W}^A(x, y; r) = 1]] \geq \mathbb{P}_r[\mathcal{W}^A(x, y^*; r) = 1] \geq 1 - \frac{1}{2^{2p(n)}}.$$

Meanwhile, for the case $f(x) = 0$, we can just union-bound over all possible y's to get

$$\mathbb{P}_r[(\exists\, y)[\mathcal{W}^A(x, y; r) = 1]] \leq \sum_y \mathbb{P}_r[\mathcal{W}^A(x, y; r) = 1] \leq \sum_y \frac{1}{2^{2p(n)}} = \frac{1}{2^{p(n)}}.$$

(The point is that each y has an error probability of at most $2^{-2p(n)}$, and there's only $2^{p(n)}$ possible y's.)

And now this is a $\mathsf{BPP}^{\mathsf{NP}^A}$ computation — we're first picking r randomly, and then calling the NP^A oracle to see whether there exists y such that $\mathcal{W}^A(x, y; r) = 1$ (and with high probability over r, the answer will be yes if $f(x) = 1$ and no if $f(x) = 0$).

> **Remark 9.55.** This argument also unconditionally shows that $\mathsf{NP}^{\mathsf{BPP}} \subseteq \mathsf{BPP}^{\mathsf{NP}}$ — the trick is that we make the error probability in the BPP part so small that we can union-bound over all possible witnesses, and this allows us to choose the randomness first and then check whether there's a witness.
>
> (The reverse inclusion is probably not true — if we think that $\mathsf{BPP} = \mathsf{P}$, then $\mathsf{NP}^{\mathsf{BPP}}$ would just be NP and $\mathsf{BPP}^{\mathsf{NP}}$ would contain coNP.)
>
> This sort of quantifier-swapping trick between NP and BPP is very important — it also comes up with Arthur–Merlin games, which we'll talk about next week.

And for larger values of k, the idea is that we just keep on doing this — we keep flipping the NP and BPP parts to eventually get $\Sigma_k\mathsf{P}^A \subseteq \mathsf{BPP}^A$.

§9.6.7 Proof of Lemma 9.48 — $\oplus$P is closed under composition

Finally we'll prove Lemma 9.48, which essentially states that $\oplus$P is closed under composition — this is a really, really strange phenomenon that turns out to be true because we can do ingenious cancellations with mod 2 counting.

Suppose we start with some $f \in \oplus\mathsf{P}^{\oplus\mathsf{SAT}^A}$; our goal is to somehow get rid of the $\oplus$SAT and show that $f \in \oplus\mathsf{P}^A$. First, the definition of $\oplus$P in terms of accepting paths of a nondeterministic machine means that there's some nondeterministic algorithm $\mathcal{N}$ with an $\oplus\mathsf{SAT}^A$ oracle such that

$$f(x) \equiv \#\text{accepting paths of } \mathcal{N}(x) \pmod 2$$

for all x. And we want to construct a nondeterministic $\mathcal{N}'$ that only has an A oracle — we want to bypass $\oplus$SAT entirely — and such that the same equation holds, i.e.,

$$f(x) \equiv \#\text{accepting paths of } \mathcal{N}'(x) \pmod 2.$$

First, without loss of generality we can assume $\mathcal{N}(x)$ (on inputs of length $|x| = n$) always makes exactly $t = t(n)$ queries (on every path).

The idea is that $\mathcal{N}'$ is going to be guessing the answers $(a_1, \ldots, a_t)$ to the queries $(F_1, \ldots, F_t)$ that $\mathcal{N}$ makes; so the number of accepting paths of $\mathcal{N}'$ is going to involve a sum over queries (F_i) and answers (a_i). And we want to somehow rig the sum so that when we guess the wrong answers, things cancel out. The big equation we'll need is the following (which gives us such a sum).

Claim 9.56 — We can write #accepting paths of $\mathcal{N}(x) \pmod 2$ as

$$\sum_{F_i, a_i} (\#\text{accepting paths of } \mathcal{N}(x) \text{ with queries } (F_i) \text{ and answers } (a_i)) \prod_i (1 - a_i + \#\mathsf{SAT}^A(F_i)). \tag{7}$$

Proof. The main idea is that if there's some a_i which is the *wrong* answer for F_i — i.e., $a_i \neq \oplus\mathsf{SAT}^A(F_i)$ — then we'll have $1 - a_i + \#\mathsf{SAT}^A(F_i) \equiv 0 \pmod 2$, which means $\prod_i(1 - a_i + \#\mathsf{SAT}^A(F_i)) \equiv 0 \pmod 2$. So all terms in the sum where we guess some wrong answer will cancel out.

Conversely, if we guess all the correct answers — i.e., we have $a_i = \oplus\mathsf{SAT}^A(F_i)$ for all i — then $\prod_i(1 - a_i + \#\mathsf{SAT}^A(F_i)) \equiv 1 \pmod 2$ (as each of the terms in the product is 1 mod 2). So terms in the sum where all our guessed answers are correct contribute 1 mod 2.

And this means we're counting each valid path of $\mathcal{N}(x)$ exactly once (mod 2), corresponding to the term where we correctly guess all the oracle's answers on that path (and we're not counting anything we shouldn't be — all the paths where our answers are wrong cancel out). □

Now our goal is to try to design $\mathcal{N}'$ so that the number of accepting paths of $\mathcal{N}'(x)$ matches (7). We'll do this in the following way.

Algorithm 9.57 ($\mathcal{N}'$) — On input x:

- Simulate $\mathcal{N}$ on x, step by step (whenever $\mathcal{N}$ makes a guess, we do as well). In the end, we *accept* if $\mathcal{N}$ accepts and *reject* if $\mathcal{N}$ rejects.
- Each time $\mathcal{N}$ tries to make an oracle query F_i, we guess an answer $a_i \in \{0, 1\}$.
  - If $a_i = 0$, then guess $b \in \{0, 1\}$. If $b = 0$, then continue the simulation of $\mathcal{N}$ with answer a_i.
  - If $a_i = 1$ or $a_i = 0$ and $b = 1$, guess an assignment y to F_i. If $F_i(y) = 1$, then continue the simulation; otherwise *reject*.

The idea is that we're trying to mimic (7); so every time there's a query F_i, we're guessing an answer a_i and then trying to multiply our number of paths by $1 - a_i + \#\mathsf{SAT}^A(F_i)$. So once we've guessed a_i, we run a subroutine that spawns this many parallel branches in our computation. To do this, if $a_i = 1$ (so that we want to spawn $\#\mathsf{SAT}^A(F_i)$ branches), we simply guess an assignment to F_i, use the A-oracle to check whether it's satisfying, and continue only if it is — so we'll get one new branch for each of the $\#\mathsf{SAT}^A(F_i)$ satisfying assignments. And if $a_i = 0$, we do the extra guessing with b to get the $+1$.

So then #accepting paths of $\mathcal{N}'(x)$ is precisely the quantity in (7), which means

$$\#\text{accepting paths of } \mathcal{N}'(x) \equiv \#\text{accepting paths of } \mathcal{N}(x) \pmod 2$$

by Claim 9.56. This means we've taken our $\oplus\mathsf{P}$ machine $\mathcal{N}$ with a $\oplus\mathsf{SAT}^A$ oracle and converted it into an equivalent $\oplus\mathsf{P}$ machine $\mathcal{N}'$ with just an A-oracle, which completes the proof of Lemma 9.48.

§10 Interactive proofs

Today we'll move to a very different topic, interactive proofs — these are extremely important both on a theoretical level and in practice.

§10.1 Setup of interactive proofs

The notion of NP (with a witness and verifier) is in some sense based on formalizing what it means to convince someone of something. We can imagine viewing NP in the following way: for all claimed theorems, if the theorem is true, then there's a prover $\mathcal{P}$ who constructs a proof of it and sends it over to the verifier $\mathcal{V}$ (so we've got a one-way communication $\mathcal{P} \to \mathcal{V}$ of a proof), and the verifier $\mathcal{V}$ checks that this proof is correct. (In the verifier characterization of NP in Fact 1.17, the input x is the 'theorem' and the witness y is the 'proof.') And this proof system should have three properties:

- All true theorems have a proof — we call this *completeness*.
- No false theorems have a proof — we call this *soundness*.
- The verifier is efficient. (The prover doesn't have to be efficient, but the verifier does — if the proof wasn't checkable efficiently, then what would be the point?)

Today we'll consider a generalization of the idea of a proof system where these properties get relaxed in certain ways. For example, we'll still require the verifier to be efficient, but we might allow *randomness* — so the verifier can toss coins while it's performing the check. Then we need to allow it to make an error — maybe it's still true that all true theorems have a proof, but the verifier might reject the proof with some tiny probability. Or maybe every false theorem doesn't have a proof, but the verifier might make a mistake and accept a bad proof with tiny probability. So we allow a tiny chance of soundness and completeness going wrong.

We'll also think about what happens if $\mathcal{P}$ and $\mathcal{V}$ get to *interact* — so after $\mathcal{P}$ sends over a proof, $\mathcal{V}$ can send something back (e.g., a message 'can you elaborate on this part of the proof?') For example, imagine $\mathcal{V}$ sends a summary of a much larger proof to $\mathcal{V}$; and $\mathcal{V}$ says they're okay with most of the lemmas but skeptical of one, so they ask $\mathcal{P}$ to elaborate on that one; and so on. And we want to know, what theorems can an efficient verifier $\mathcal{V}$ be convinced of through a short *interaction*?

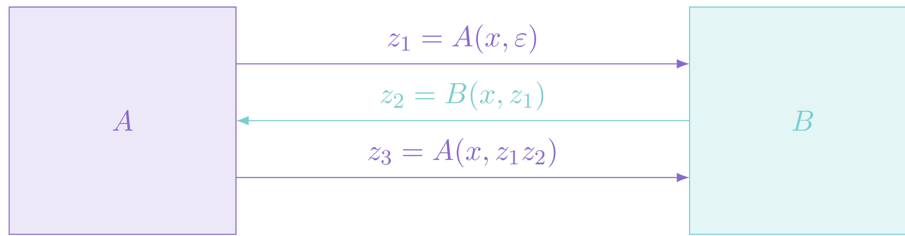
§10.1.1 Modelling interaction

The first thing we'll do is formalize what we mean by $\mathcal{P}$ and $\mathcal{V}$ interacting. Intuitively, we imagine there's a common input x that both $\mathcal{P}$ and $\mathcal{V}$ see. Then $\mathcal{P}$ looks at this input and sends over a message z_1; and then $\mathcal{V}$ looks at x and z_1 and sends over a message z_2; and then $\mathcal{P}$ looks at x, z_1, and z_2 and sends over a message z_3; and so on.

To formalize this, we'll model $\mathcal{P}$ and $\mathcal{V}$ as *communication protocols*. In a general communication protocol, we imagine having two functions $A, B\colon \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ which take in two inputs — the first input is their shared input $x \in \{0,1\}^*$, and the second input is the transcript of all the communication so far. So in the first round A takes in x and the empty transcript and produces z_1; then B takes in x and z_1 and produces z_2; and so on.

Definition 10.1. The k-round interaction of $\langle A, B\rangle$ on x is the sequence of strings $z_1, \ldots, z_k$ defined as $z_1 = A(x, \varepsilon)$, $z_2 = B(x, z_1)$, $z_3 = A(x, z_1 z_2)$, $z_4 = B(x, z_1 z_2 z_3)$, and so on. We use $\langle A, B\rangle_k(x)$ to denote z_k (the output of the kth round of the protocol).

So A speaks on odd-numbered rounds and B speaks on even-numbered rounds, and each chooses a message to send as a function of all the communication that's occurred so far (and their common input).



§10.1.2 The deterministic verifier setting

As a warmup, we'll start by thinking of the verifier as deterministic; this will give us some definitions and a framework that'll be useful for the randomized case as well.

We want our prover and verifier to look at x and have some conversation, and this conversation shouldn't last too long — it should have length $\mathsf{poly}(|x|)$. So we'll start by defining the space of 'feasible communication functions.'

Definition 10.2. We define $\mathcal{F}[n, t(n)]$ as the set of all $A\colon \{0,1\}^n \times \{0,1\}^{\leq t(n)} \to \{0,1\}^{\leq t(n)}$.

We think of A as taking in an n-bit input and a transcript of length at most $t(n)$ (which we think of as all the previous messages z_i put together), and outputting the next message; and this transcript should have length at most $t(n)$, which we think of as $\mathsf{poly}(n)$.

Next, we'll define what it means for such a proof system to solve a decision problem $f\colon \{0,1\}^* \to \{0,1\}$. We'll assume that k is even, so that B (which will be the verifier $\mathcal{V}$) has the last word (i.e., z_k is something that the verifier outputs).

Definition 10.3. We say f has a k-round deterministic interactive proof system (abbreviated DIP) if there exists $c \in \mathbb{N}$ and a verifier $\mathcal{V} \in \mathsf{FP}$ such that for all inputs x:

- If $f(x) = 1$, then there exists $\mathcal{P} \in \mathcal{F}[|x|, |x|^c]$ such that $\langle \mathcal{P}, \mathcal{V}\rangle_k(x) = 1$.
- If $f(x) = 0$, then for *all* $\mathcal{P} \in \mathcal{F}[|x|, |x|^c]$, we have $\langle \mathcal{P}, \mathcal{V}\rangle_k(x) = 0$.

Note that the fact we're requiring $\mathcal{P}$ to be in $\mathcal{F}[|x|, |x|^c]$ automatically limits the length of the transcript — the transcript has to have length at most $|x|^c$ at all points in time, or else the prover's output wouldn't even be defined. Other than this, $\mathcal{P}$ can be arbitrary — there's no constraints on $\mathcal{P}$ other than on its input and output length. On the other hand, we're requiring $\mathcal{V}$ to be efficient (with the condition $\mathcal{V} \in \mathsf{FP}$).

And if $f(x) = 1$, then there is some 'good prover' $\mathcal{P}$ which can convince $\mathcal{V}$ in k rounds; while if $f(x) = 0$, then there is no such prover — i.e., no matter what the prover does, the verifier always rejects.

First, here's a simple example to demonstrate how these interactive proofs work.

Proposition 10.4
If $f \in \mathsf{NP}$, then f has a 2-round DIP.

Proof. The idea is that the prover $\mathcal{P}$ will just send over the NP witness, and the verifier $\mathcal{V}$ will check that it's valid. Explicitly, given $f \in \mathsf{NP}$, we'll take $\mathcal{V}$ to be the polynomial-time verifier for f from the verifier definition of NP. Then for x with $f(x) = 1$, we take the good prover $\mathcal{P}$ to just send over a witness y, i.e., any y such that $\mathcal{V}(x, y) = 1$. Meanwhile, for x with $f(x) = 0$, no matter what string y the prover sends, we'll have $\mathcal{V}(x, y) = 0$. So this protocol satisfies the definition. $\square$

So 2-round DIPs correspond exactly to NP (the converse is true as well — any verifier for a 2-round DIP is also a verifier in the sense of NP).

> **Question 10.5.** If we allow extra interaction (e.g., polynomial-round DIPs), can we solve more problems?

It turns out the answer is no!

> **Theorem 10.6**
> For all polynomials p, a decision problem f has a $2p(n)$-round DIP if and only if $f \in \mathsf{NP}$.

(The reason we've got $2p(n)$ instead of $p(n)$ is just to ensure that there's an even number of rounds, so that $\mathcal{V}$ speaks last.)

Proof. We've already proven the backwards direction — if $f \in \mathsf{NP}$ then we even have a 2-round DIP.

For the forwards direction, suppose we start with a polynomial-round interaction; we want to make it a two-round one (so that $\mathcal{P}$ is just sending over a single witness and $\mathcal{V}$ is verifying it, as in NP). The idea is to have our new prover send over the *whole* transcript of the original protocol, all at once.

To formalize this, let our original prover and verifier be $\mathcal{P}$ and $\mathcal{V}$. This means when $f(x) = 1$, there *exists* a sequence $z_1, \ldots, z_{2p(n)}$ with $|z_i| = \mathsf{poly}(n)$ such that $\mathcal{P}(x, \varepsilon) = z_1$, $\mathcal{V}(x, z_1) = z_2$, and so on, until finally the verifier outputs $\mathcal{V}(x, z_1 \ldots z_{2p(n)}) = 1$. So we'll create a new verifier $\mathcal{V}'$ that takes in an entire transcript $z = z_1 \ldots z_{2p(n)}$ and check that all the verifier's computations have been done correctly — i.e., that $\mathcal{V}(x, z_1) = z_2$, $\mathcal{V}(x, z_1 z_2 z_3) = z_4$, and so on, and in the end $\mathcal{V}$ accepts.

And if $f(x) = 1$, then there exists z such that $\mathcal{V}'(x, z) = 1$ (namely, the correct transcript for $\langle \mathcal{P}, \mathcal{V} \rangle$ on x). Meanwhile, if $f(x) = 0$, then no matter what transcript we give $\mathcal{V}'$, it'll always reject (because there's no prover $\mathcal{P}$ that makes $\mathcal{V}$ accept, so there can't be a transcript that's correct on the verifier end and results in acceptance). This is exactly the guarantee we need for $\mathcal{V}'$ to be a NP verifier, showing $f \in \mathsf{NP}$. □

The point is that if the protocol is deterministic, then we don't really gain any benefit from having multiple rounds — the prover can just send the entire transcript at once (since they know exactly what $\mathcal{V}$ would say at each step), and the verifier can just check it.

§10.1.3 The randomized verifier setting

We've now seen that *deterministic* interactive proof systems are just NP, which is why we have not heard of the complexity class DIP before. But things become more interesting when the verifier is randomized — this means $\mathcal{V}$ gets to toss some coins that $\mathcal{P}$ can't see, and we allow some small probability of error.

First, here's an 'example' of a protocol with a randomized verifier that does something interesting. (This is kind of silly and informal, but as we'll see later today, the ideas are actually useful.)

> **Example 10.7**
> Suppose we've got a verifier $\mathcal{V}$ who's red-green colorblind and a prover $\mathcal{P}$ who's not colorblind. Then there's a protocol for $\mathcal{P}$ to prove to $\mathcal{V}$ that he (i.e., $\mathcal{P}$) really can tell red from green.

Proof. Imagine $\mathcal{P}$ has a red ball and a green ball. He puts the red ball in the left hand of $\mathcal{V}$, and the green ball in the right hand of $\mathcal{V}$. Then $\mathcal{V}$ turns their back and tosses a coin (which $\mathcal{P}$ can't see); if this coin comes up heads then they swaps the two balls, while if it comes up tails then they don't swap the balls.

Then $\mathcal{V}$ turns around and shows $\mathcal{P}$ the red and green balls in their new positions, and asks $\mathcal{P}$ to guess what their coin toss was.

If $\mathcal{P}$ can tell red from green, then they can *always* correctly say what the coin toss was (by seeing whether the red ball is now in the left or right hand of $\mathcal{V}$).

On the other hand, if $\mathcal{P}$ cannot tell red from green, then he has to somehow predict an unbiased coin from no information, so he's got only a $\frac{1}{2}$ chance of correctly guessing it.

Now suppose we repeat this protocol k times (either in parallel or sequence). If $\mathcal{P}$ can tell red from green, then he should get all trials correct. Meanwhile, if $\mathcal{P}$ *can't* tell red from green, then the probability he guesses everything correctly is just $\frac{1}{2^k}$. So if $\mathcal{P}$ gets everything correct, then $\mathcal{V}$ can be convinced with high confidence that $\mathcal{P}$ really can tell red from green. $\square$

Remark 10.8. Interestingly, $\mathcal{V}$ gains no information out of this protocol — now $\mathcal{V}$ is convinced that $\mathcal{P}$ can distinguish red from green, but they've learned nothing about *how* $\mathcal{P}$ does so. And in fact, $\mathcal{V}$ could have predicted everything that $\mathcal{P}$ would say; but the difference is that $\mathcal{V}$ has the coin and $\mathcal{P}$ doesn't. This property is called *zero-knowledge* — meaning that there's a way to simulate everything that $\mathcal{P}$ is telling $\mathcal{V}$ if you also get to see (or simulate) the coin tosses.

Remark 10.9. In this protocol, the private randomness looks crucial. But it turns out that if you allow more rounds of interaction, then private coins can actually be simulated with public ones! So the difference isn't as big as we might think from this example.

Now we'll formally define private coin protocols.

Definition 10.10. The k-round private coin interaction of $\langle \mathcal{P}, \mathcal{V} \rangle$ on an input x and randomness r is the sequence of strings $z_1, \ldots, z_k \in \{0,1\}^*$ such that $z_1 = \mathcal{P}(x, \varepsilon)$, $z_2 = \mathcal{V}(x, z_1; r)$, $z_3 = \mathcal{P}(x, z_1 z_2)$, $z_4 = \mathcal{V}(x, z_1 z_2 z_3; r)$, and so on; we write $\langle \mathcal{P}, \mathcal{V} \rangle_k (x; r) = z_k$.

The reason this is called *private coin* is that only $\mathcal{V}$ has access to the randomness r (it's in the input to $\mathcal{V}$, but not $\mathcal{P}$). (We'll discuss public coin protocols in a future lecture.)

Now we'll define interactive proofs in the private coin setting.

Definition 10.11. We say a decision problem f has a k-round interactive proof system, which we write as $f \in \mathsf{IP}[k]$, if there is some $c \in \mathbb{N}$ and a verifier $\mathcal{V} \in \mathsf{FP}$ such that for all x:

- If $f(x) = 1$, then there exists $\mathcal{P} \in \mathcal{F}[|x|, |x|^c]$ such that $\mathbb{P}_r[\langle \mathcal{P}, \mathcal{V} \rangle_k (x; r) = 1] \geq \frac{2}{3}$.
- If $f(x) = 0$, then for all $\mathcal{P} \in \mathcal{F}[|x|, |x|^c]$, we have $\mathbb{P}_r[\langle \mathcal{P}, \mathcal{V} \rangle_k (x; r) = 1] \leq \frac{1}{3}$.

Here we call $\frac{2}{3}$ (the probability of accepting a true theorem) the *completeness parameter*; ideally we'd want it to be 1 (in which case every true theorem has a prover that makes the verifier always accept). And we call $\frac{1}{3}$ (the probability of accepting a *false* theorem) te *soundness parameter*; ideally we'd want it to be 0 (ideally, you'd want the verifier to always reject all proofs of a false theorem). But if we set the completeness and soundness parameters to 1 and 0, then we'd be back to deterministic proof systems, which we saw can't do very much. So we really do need to allow some error in order to get something new.

Definition 10.12. We define $\mathsf{IP} = \bigcup_c \mathsf{IP}[cn^c]$.

So IP is the class of problems which can be solved by a *polynomial*-round interactive proof system with a randomized verifier.

§10.1.4 Some facts about IP

We'll now mention (but not prove) a few facts about interactive proofs.

First, just as with BPP, the parameters $\frac{2}{3}$ and $\frac{1}{3}$ are arbitrary (they just have to be bounded away from $\frac{1}{2}$ in the appropriate directions) — we can get the completeness and soundness arbitrarily close to 1 and 0.

> **Fact 10.13** — If there is an interactive proof system for f, then there is one with completeness parameter $1 - 2^{-p(n)}$ and soundness parameter $2^{-p(n)}$ for any polynomial p.

When discussing BPP, the way we did error reduction (as in Lemma 7.4) was by performing a bunch of independent trials and taking the majority. And we can d othe same thing here — $\mathcal{V}$ tosses $\mathsf{poly}(n)$ separate random coins to get independent randomnesses $r_1, \ldots, r_{\mathsf{poly}(n)}$, and we then run $\langle \mathcal{P}, \mathcal{V} \rangle (x; r_i)$ for each i and take the majority vote of their final decisions. (The reason this works is very similar to the one for BPP.)

More surprisingly, we can get the completeness parameter to actually *equal* 1!

> **Fact 10.14** — If there is an interactive proof system for f, then there is one with *perfect completeness* (i.e., completeness parameter 1).

So in other words, we can set things up so that when the theorem is true, there's a proof that the verifier is *always* convinced by (no matter what its randomness is).

On the other hand, this is *not* true of the soundness parameter.

> **Fact 10.15** — It is not necessarily true that if there is an interactive proof system for f, then there is one with perfect soundness (i.e., soundness parameter 0), unless $\mathsf{IP} = \mathsf{DIP} = \mathsf{NP}$.

The point is that if we could get a protocol $\langle \mathcal{P}, \mathcal{V} \rangle$ with perfect soundness (but imperfect completeness), then we could create a new protocol where the prover sends over the randomness r that would make $\mathcal{V}$ accept, since such randomness exists when $f(x) = 1$ and not when $f(x) = 0$. So then we'd get a protocol with no randomness, i.e., one in DIP.

Finally, we've seen that making $\mathcal{V}$ randomized makes the proof system more powerful (we don't think $\mathsf{IP} = \mathsf{DIP}$); so does letting $\mathcal{P}$ be randomized make it even more powerful? The answer is no.

> **Fact 10.16** — Letting $\mathcal{P}$ be randomized doesn't change IP at all.

The point is that there's no constraint on the computability of $\mathcal{P}$, so if we had a randomized prover $\mathcal{P}$, then we could obtain a deterministic $\mathcal{P}'$ where we just take $\mathcal{P}$ and hardcode the best outcomes of the coin tosses (i.e., the ones maximizing the probability that the verifier accepts).

§10.2 An example — graph non-isomorphism

We'll now see an example of an interesting problem that can be solved with an interactive protocol.

First, we'll consider the graph isomorphism problem — where we're given two graphs, and we want to figure out whether they're isomorphic.

> **Definition 10.17** (GI)
>
> - **Input:** a pair of graphs (G_1, G_2) (with the same number of vertices).
> - **Decide:** whether G_1 and G_2 are isomorphic — i.e., whether there is a permutation $\pi : V(G_1) \to V(G_2)$ such that edges in G_1 are mapped to edges in G_2 and non-edges in G_1 are mapped to non-edges in G_2.

We have $\mathsf{GI} \in \mathsf{NP}$ — we can just guess the isomorphism (i.e., the permutation of vertices) and check that it works. So coming up with an interactive proof system for GI is not particularly interesting (every problem in NP has a deterministic interactive proof system, as see in Proposition 10.4). Instead, we'll consider its *complement*.

Definition 10.18 (GNI)

- **Input:** a pair of graphs (G_1, G_2) (wit the same number of vertices).
- **Decide:** whether G_1 and G_2 are *non*-isomorphic.

We know $\mathsf{GNI} \in \mathsf{coNP}$; whether it's in NP is actually a major open question. What we know about the deterministic time complexity of GNI (or equivalently GI) is quite striking — we don't know whether it's in P, but Babai in 2015 showed that $\mathsf{GI} \in \mathsf{TIME}[2^{\mathsf{polylog}(n)}]$ via a deep and fascinating algorithm that uses lots of symmetries of graphs (this is way faster than trying all $n! = 2^{O(n \log n)}$ vertex permutations). For this reason, we don't think GI is very hard — for example, if it were NP-complete, then this would mean *every* problem in NP would be in $\mathsf{TIME}[2^{\mathsf{polylog}(n)}]$ (i.e., deterministic quasipolynomial time), which would mean exhaustive search (over all $2^{\mathsf{poly}(n)}$ possible witnesses) would be very far from necessary.

We'll now see the following interactive proof system for GNI.

Theorem 10.19

There is a constant r such that $\mathsf{GNI} \in \mathsf{IP}[r]$.

In fact, it's possible to show $\mathsf{GNI} \in \mathsf{IP}[3]$ if $\mathcal{V}$ speaks first.

Proof. The main idea is to use some version of the red-green protocol from Example 10.7. If G_1 and G_2 are really non-isomorphic, then a prover can tell them apart in the sense that if we randomly permute the vertices, they'll look different. Meanwhile, if they're isomorphic, then if we permute their vertices they'll look the same. So we'll essentially have the verifier randomly permute the vertices of the graphs and see if the prover can distinguish between them — this should be possible if they're not isomorphic but not if they are isomorphic.

More formally, we repeat the following for t trials (where t is some parameter we'll set later):

- $\mathcal{V}$ picks $c \in \{1, 2\}$ uniformly at random (privately), randomly permutes the vertices of G_c to get some graph H, and sends H over to $\mathcal{P}$.
- $\mathcal{P}$ sens back some $c' \in \{1, 2\}$, and $\mathcal{V}$ checks that $c' = c$.

In the end, $\mathcal{V}$ accepts if and only if $c' = c$ on every single trial.

To see that this works, first suppose G_1 and G_2 are not isomorphic (so $(G_1, G_2) \in \mathsf{GNI}$). Then when the prover sees H, it's isomorphic to either G_1 or G_2, but not both — if it were isomorphic to both, then G_1 and G_2 would themselves be isomorphic. So the prover can figure out which one it's isomorphic to, which tells it what c was.

On the other hand, now suppose G_1 and G_2 are isomorphic (so $(G_1, G_2) \notin \mathsf{GNI}$). Then the distribution of H is the same in the case $c = 1$ as in the case $c = 2$ (i.e., randomly permuting the vertices of G_1 and of G_2 both produce the same distribution — a random member of their isomorphism class). This means the graph H that the prover sees is completely independent of c, so no matter what the prover does, we have

$$\mathbb{P}_H[\mathcal{P} \text{ sends } c' \text{ such that } c' = c] \leq \frac{1}{2}.$$

(The point is that H doesn't tell the prover any information about c whatsoever, so the prover can't do any better than chance. The only reason this is an inequality and not an equality is because $\mathcal{P}$ could theoretically do something silly like not answering at all.)

This means after t trials, the probability that $\mathcal{P}$ answers correctly on *every* trial is at most 2^{-t}, which we can make as small as we want (by choosing t to be a large constant).

Finally, we can reduce the number of rounds by doing the t trials in parallel rather than in sequence — so $\mathcal{V}$ chooses $c_1, \dots, c_t \in \{1, 2\}$ independently, obtains graphs $H_1, \dots, H_t$ by permuting one of G_1 and G_2 randomly (where H_i is a permutation of G_{c_i} for each i), and sends them to $\mathcal{P}$. Then $\mathcal{P}$ sends back $c'_1, \dots, c'_t \in \{1, 2\}$, and $\mathcal{V}$ checks that all these guesses are correct (i.e., $c'_i = c_i$ for all i). □

Remark 10.20. As with the red-green protocol from Example 10.7, in this proof $\mathcal{V}$ doesn't really learn *anything* about how to tell G_1 and G_2 apart — the only information $\mathcal{P}$ is giving $\mathcal{V}$ is the value of a private coin that $\mathcal{V}$ already knew. So $\mathcal{V}$ could easily simulate everything $\mathcal{P}$ is sending; this is the idea behind zero-knowledge proofs (which are an important concept in cryptography, but we won't discuss them more here).

§10.3 Arthur–Merlin and Merlin–Arthur protocols

We'll now discuss a slightly different quantification of interactive proofs, namely *Arthur–Merlin protocols*.

Definition 10.21. We say $f \in \mathsf{AM}[k]$ if there exists a k-round interactive proof system for f such that $\mathcal{V}$ speaks first and only sends uniform randomness.

This means the only thing the verifier ever does is send uniform randomness to the prover (and eventually decide whether to accept or reject). This motivates the name Arthur–Merlin — we think of the prover as all-powerful Merlin, and the verifier is supposed to be King Arthur uttering random garbage.

Definition 10.22. We define $\mathsf{AM} = \bigcup_k \mathsf{AM}[k]$.

Note that in the definition of IP, we allow polynomially many rounds of interaction; but with AM we're only allowing a constant number of rounds.

One way to think about Arthur–Merlin protocols is as 'public-coin interactive proof systems' — because $\mathcal{V}$ is sending over their uniform randomness, which means $\mathcal{P}$ gets to see this randomness when choosing their response.

We'll also define *Merlin–Arthur protocols*, which in some sense correspond to a randomized version of NP.

Definition 10.23. We define $\mathsf{MA} = \mathsf{IP}[2]$.

This means that the prover speaks first, and the verifier then decides whether to accept or reject. In particular, even though the definition has IP in it, there's no interaction involved — here Merlin sends over a proof, and Arthur tosses a bunch of coins and decides whether to accept or reject. This is very similar to NP, except that in NP we've got a deterministic verifier, and with MA we've got a randomized one.

Conjecture 10.24 — We have $\mathsf{MA} = \mathsf{AM} = \mathsf{NP}$.

The reason people believe these equalities is that they're implied by certain circuit lower bounds (somewhat similarly to Theorem 8.24, where we saw that certain circuit lower bounds imply $\mathsf{BPP} = \mathsf{P}$).

We'll now see some facts about AM.

Fact 10.25 — We have $\mathsf{AM}[k] = \mathsf{AM}[2]$ for any constant k.

This means any k-round Arthur–Merlin protocol can be converted to one that just takes 2 rounds (where $\mathcal{V}$ sends uniform randomness, $\mathcal{P}$ sends back some proof, and $\mathcal{V}$ decides *deterministically* whether to accept or reject); we'll show this on the problem set.

Fact 10.26 — We have $\mathsf{AM} \subseteq \mathsf{BPP}^{\mathsf{NP}} \subseteq \Sigma_3\mathsf{P}$.

The fact that $\mathsf{AM} \subseteq \mathsf{BPP}^{\mathsf{NP}}$ is because we can (very roughly) think of the verifier as the BPP part, and the prover as the NP oracle. More precisely, by Fact 10.25, for any problem in AM we can get a protocol where the verifier $\mathcal{V}$ sends uniform randomness, the prover sends a proof, and $\mathcal{V}$ accepts or rejects; then we can make a $\mathsf{BPP}^{\mathsf{NP}}$ algorithm where the NP oracle takes in the randomness sent by $\mathcal{V}$ (as its query) and answers the question 'does there exist a proof that the verifier would accept?'

§10.3.1 Private-coin to public-coin protocols

Next, we'll see some relations between AM and IP.

Fact 10.27 — For all $k = \mathsf{poly}(n)$, we have $\mathsf{AM}[k] \subseteq \mathsf{IP}[k+1]$.

This is unsurprising (and has basically no content) — any AM protocol is also an IP one. (The reason we have a $+1$ is just that in IP the prover speaks first, while in AM the verifier speaks first; so to convert from a AM protocol to an IP one, we need to add a useless round at the start where the prover speaks.)

What *is* surprising is that there's an inclusion in the reverse direction!

Theorem 10.28 (Goldwasser–Sipser)
For all $k = \mathsf{poly}(n)$, we have $\mathsf{IP}[k] \subseteq \mathsf{AM}[k+2]$.

This is very surprising because it essentially means that for any private-coin interactive protocol, we can replace the private coins with *public* ones. This is pretty unexpected — for example, our protocol for graph non-isomorphism (in Theorem 10.19) involved $\mathcal{V}$ choosing private randomness and asking $\mathcal{P}$ to predict what randomness it used (since $\mathcal{P}$ can do this if the two graphs are not isomorphic, but not if they are). But if we used public coins for this instead (i.e., if $\mathcal{V}$ sent over their randomness), then $\mathcal{P}$ would already know what $\mathcal{V}$'s randomness was, so this protocol would make no sense. Theorem 10.28 in particular means that there's a different way of solving the graph non-isomorphism problem that *doesn't* require the coins to be private.

Remark 10.29. Together, Fact 10.27 and Theorem 10.28 mean $\mathsf{IP}[O(1)] = \mathsf{AM}$ — i.e., IP with a *constant* number of rounds is just AM. So the main difference between the two classes is the fact that IP allows a *polynomial* number of rounds, while AM doesn't (we might have expected the private vs. public coin distinction to also be a key difference, but thanks to Theorem 10.28, it actually isn't).

We're not going to go through all the details of Theorem 10.28, but we'll discuss the high-level idea. For simplicity, let's imagine we have a 3-round private-coin protocol where $\mathcal{V}$ has private randomness r — so $\mathcal{V}$ sends a message m (computed as a function of the input x and randomness r), then $\mathcal{P}$ sends back some message m' (computed as a function of x and m), and then $\mathcal{V}$ decides whether to accept or reject (based on x, m, m', and their randomness r). And we know that in the **YES** case there exists a good prover that makes $\mathcal{V}$ accept with very high probability, while in the **NO** case, no matter what the prover does, $\mathcal{V}$ will accept with very low probability (over r).

The main idea to convert this to a protocol with public randomness is that the new prover $\mathcal{P}'$ is going to try to convince the verifier $\mathcal{V}'$ that $\mathcal{V}$ *would* have accepted on most choices of r. And the main tool that enables this is pairwise independent hash functions.

We're not going to say too much about the details, because they're closely related to a problem on the problem set (about approximate counting). But here's the outline — suppose that the randomness r in the original protocol had length ℓ. Then the new public coin protocol works as follows:

- $\mathcal{V}'$ chooses a hash function $h: \{0,1\}^\ell \to \{0,1\}^k$ from a pairwise independent hash family (for some appropriately chosen k), and sends it over to $\mathcal{P}$ (by sending over its seed).
- Then $\mathcal{P}'$ sends back a 3-tuple (r, m, m').
- Finally, $\mathcal{V}'$ accepts if and only if the original $\mathcal{V}(x; r)$ would have sent m and would have accepted if $\mathcal{P}$ had sent back m', and $h(r) = 0^k$.

The point is that if there's a lot of good randomness r that would make $\mathcal{V}$ accept, then with high probability there should be one that hashes to 0^k. Meanwhile, if there's very little such randomness r (i.e., the set of randomness r on which $\mathcal{V}$ accepts is tiny), then the probability some such r hashes to 0^k is very low.

§10.4 An interactive proof for Count-SAT

We've mentioned that it's believed $\mathsf{MA} = \mathsf{AM} = \mathsf{NP}$. But it's *not* believed that $\mathsf{IP} = \mathsf{NP}$ (though this is still open). In fact, it turns out that IP is extremely powerful, thanks to the following theorem.

Theorem 10.30 (Shamir 1990)
We have $\mathsf{IP} = \mathsf{PSPACE}$.

We're going to prove this next class. Today, as a warmup, we're going to cover an easier protocol that highlights many of the important ideas. Specifically, we'll consider the problem Count-SAT.

Definition 10.31 (Count-SAT)

- **Input:** (φ, k), where φ is a Boolean formula and $k \in \mathbb{N}$.
- **Output:** whether $\#\mathsf{SAT}(\varphi) \geq k$.

It's easy to see that Count-SAT is NP-hard, and that $\#\mathsf{SAT} \in \mathsf{FP}^{\mathsf{Count\text{-}SAT}}$ (since given a Count-SAT oracle, we can just do binary search on k to find $\#\mathsf{SAT}(\varphi)$ for a given φ).

(The only reason we're discussing Count-SAT rather than #SAT is because we only defined IP with respect to decision problems, and #SAT is not a decision problem.)

Today, we'll provide an interactive proof for Count-SAT. More specifically, we'll prove the following theorem.

Theorem 10.32
There is a polynomial-round interactive proof system such that for all formulas φ:

- There exists a good prover $\mathcal{P}$ whose first message is $k = \#\mathsf{SAT}(\varphi)$ and for which $\mathcal{V}$ accepts with probability 1.
- For any prover $\mathcal{P}$ whose first message is *not* $\#\mathsf{SAT}(\varphi)$, $\mathcal{V}$ rejects with probability at least $1 - \frac{1}{n}$.

(This immediately implies $\mathsf{Count\text{-}SAT} \in \mathsf{IP}$ — given input (φ, k) to Count-SAT, we can use this protocol to convince $\mathcal{V}$ of the exact value of $\#\mathsf{SAT}(\varphi)$, and then $\mathcal{V}$ can check whether or not this value is at least k.)

So in words, in our protocol $\mathcal{P}$ first sends over an integer k which it claims is the number of satisfying assignments to φ. And if $\mathcal{P}$ is not lying, there's some way for them to continue such that $\mathcal{V}$ always accepts. Meanwhile, if $\mathcal{P}$ does lie on the first message (i.e., it gives the wrong number of satisfying assignments), then with high probability $\mathcal{V}$ will eventually catch them lying and reject.

§10.4.1 Arithmetization

The first key idea in the proof of Theorem 10.32 is *arithmetization* — this essentially means that we'll take Boolean formulas and turn them into *polynomials* over some field.

Definition 10.33. A field is a set $\mathbb{F}$ that supports the usual arithmetic operations $+$, $-$, $\times$, and division by nonzero elements.

We'll work specifically with the finite field consisting of the integers mod p (where p is a prime).

Definition 10.34. For p prime, we define $\mathbb{F}_p$ as the field with elements $\{0, 1, \ldots, p-1\}$, where all operations are done mod p.

To encode a Boolean formula as a polynomial, we'll essentially interpret `True` as $1 \in \mathbb{F}_p$ and `False` as $0 \in \mathbb{F}_p$, using the following correspondences to represent the Boolean operations.

Formula	Polynomial
$x \wedge y$	$x \cdot y$
$\neg x$	$1 - x$
$x \vee y$	$1 - (1-x)(1-y)$

Now we're ready to arithmetize SAT over $\mathbb{F}_p$ — suppose we're given a Boolean formula φ in the variables $x_1, \ldots, x_n$. We'll then turn it into a polynomial P_φ in variables $y_1, \ldots, y_n$, by just replacing each x_i with y_i and replacing the Boolean operations with arithmetic ones using the above formulas.

Example 10.35

If $\varphi = (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_4})$, then

$$P_\varphi(y_1, y_2, y_3, y_4) = (1 - y_1(1-y_2)y_3)(1 - y_1 y_4).$$

Note that $\deg P_\varphi \leq |\varphi|$ (where $|\varphi|$ is the size of φ) — each Boolean operation at worst adds together the degrees of its two inputs.

Why are we converting Boolean formulas to polynomials in $\mathbb{F}_p$? One reason is that it allows us to plug in more inputs — instead of plugging in values from $\{0, 1\}$ (i.e., Boolean values), we can plug in any values in $\mathbb{F}_p$. And this in some sense expands the set of questions $\mathcal{V}$ can ask $\mathcal{P}$ (since we can evaluate φ on a bigger set), which makes the verifier more powerful.

§10.4.2 The sum-check protocol

We'll now go into the proof of Theorem 10.32. The protocol we'll use is essentially a sum-check protocol (this is an idea that's so important that it has a name).

First, we'll describe the high-level intuition behind this protocol. The idea is that given φ, we can write

$$\#\mathsf{SAT}(\varphi) = \sum_{a_1, \ldots, a_n \in \{0,1\}} P_\varphi(a_1, \ldots, a_n).$$

Here we're plugging in all possible assignments $(a_1, \ldots, a_n)$ to our variables and summing the results — on Boolean inputs, P_φ evaluates to 1 whenever the input is a satisfying assignment and 0 whenever it isn't, so when we sum P_φ over all possible assignments, we get the number of satisfying ones.

And $\mathcal{P}$ wants to prove that $\#\mathsf{SAT}(\varphi) = k$, so they want to convince the verifier of the equation

$$k = \sum_{a_1,\ldots,a_n \in \{0,1\}} P_\varphi(a_1, \ldots, a_n). \tag{8}$$

Throughout this process, we're going to be working in $\mathbb{F}_p$ for a big prime p — so the prover really wants to convince the verifier of (8) mod p. As long as $p > 2^n$, this is enough — we don't get weird cancellations because $\#\mathsf{SAT}(\varphi)$ is at most $2^n < p$, so if we can get $\#\mathsf{SAT}(\varphi) \equiv k \pmod{p}$, this means it's actually k.

How can $\mathcal{P}$ try to convince $\mathcal{V}$ of (8)? The main idea is to send a *polynomial* — specifically, we consider the polynomial $Q(z_1)$ where we relax the assignment a_1 into a variable z_1 (but still sum over all possible assignments for the other variables), so that

$$Q(z_1) = \sum_{a_2,\ldots,a_n \in \{0,1\}} P_\varphi(z_1, a_2, \ldots, a_n). \tag{9}$$

Note that Q is a polynomial in a single variable with $\deg Q \leq |\varphi|$ (since $\deg P_\varphi \leq |\varphi|$). In particular, even though this definition of Q has an exponential-sized sum, it's possible to write Q down as a polynomial-sized message — if we write it in the form $\sum_{i=0}^m c_i z_1^i$ (where $m = \deg Q$), then there's only polynomially many terms, and each coefficient has polynomial size (the coefficients are elements of $\mathbb{F}_p$, so they take $\log p \approx n$ bits to write down).

So we imagine that $\mathcal{P}$ sends over a polynomial R, which is supposed to be Q (i.e., the good prover sends Q, but a cheating prover might send some other polynomial). Then $\mathcal{V}$ wants to do two things. First, they check that if $\mathcal{P}$ really sent over the correct polynomial (i.e., $R = Q$), then (8) is true — they can do this by plugging in 0 and 1 into R and checking that $R(0) + R(1) = k$ (the point is that the right-hand side of (8) can be written as $Q(0) + Q(1)$.)

Next, $\mathcal{V}$ wants to check that $\mathcal{P}$ really sent over the correct polynomial — i.e., that R really does equal Q. And the idea behind how we'll do this is the same as the one behind polynomial identity testing (as in Theorem 7.10) — $\mathcal{V}$ picks a random element $r \in \mathbb{F}_p$, and tries to check whether $Q(r) = R(r)$. The point is that Q and R are both polynomials of bounded degree, so if they're not the same polynomial, then with high probability — specifically, probability at least $1 - \frac{|\varphi|}{p}$ — they're not going to agree at r. (We're guaranteed that $\deg Q \leq |\varphi|$; we're not guaranteed this about R, but if $\deg R$ is too big then $\mathcal{V}$ can just automatically reject, since it immediately knows R is not the correct polynomial. So then $\deg(Q - R) \leq |\varphi|$, which means $Q - R$ has at most $|\varphi|$ roots; and if we're selecting a random element from $\mathbb{F}_p$, the probability we hit one of these roots is at most $\frac{|\varphi|}{p}$.)

What have we accomplished here? We've reduced the problem to the problem of checking whether $Q(r) = R(r)$. And $\mathcal{V}$ can definitely evaluate $R(r)$, since they've been given R; but they have no way of evaluating $Q(r)$ (as Q is a sum of exponentially many terms). So instead, we ask the *prover* to evaluate $Q(r)$ — more specifically, we send over the value of r and ask the prover to prove the statement

$$R(r) = \sum_{a_2,\ldots,a_n \in \{0,1\}} P_\varphi(r, a_2, \ldots, a_n). \tag{10}$$

(The right-hand side is precisely the definition of $Q(r)$.)

And (10) is an equation that looks a lot like (8) (we're summing some quantity over a bunch of $a_i \in \{0,1\}$), but now we're summing over $n - 1$ variables instead of n (we've fixed a value $r \in \mathbb{F}_p$ for the first one). And so we can repeat this over and over again to fix the values of more variables — so in the end, we've plugged in random values $r_1, \ldots, r_n \in \mathbb{F}_p$ for *all* the variables, and the statement we're asking to be convinced of is that $P_\varphi(r_1, \ldots, r_n)$ is a certain specific value. And at this point, the verifier can check this themselves, because we don't have a sum anymore — they can just plug in $r_1, \ldots, r_n$ to the arithmetized version of φ and check that this gives the claimed value.

We'll now formalize how this algorithm works.

Algorithm 10.36 ($\langle \mathcal{P}, \mathcal{V} \rangle$) — On input φ, where φ is a Boolean formula in variables $x_1, \ldots x_n$:

- First, $\mathcal{P}$ sends k and a prime $p \in [2^n |\varphi|, 2^{n+1} |\varphi|]$ (such a prime exists by Bertrand's postulate). Then $\mathcal{V}$ checks that p is really a prime, and *rejects* if not.
- Let $k_0 = k$. Then for all $i = 1, \ldots, n$:
  - $\mathcal{P}$ sends a polynomial $R_i \in \mathbb{F}_p[z_i]$ of degree at most $|\varphi|$.
  - $\mathcal{V}$ checks that $R_i(0) + R_i(1) = k_{i-1}$, and *rejects* if not.
  - $\mathcal{V}$ picks a random $r_i \in \mathbb{F}_p$. We then set $k_i = R_i(r_i)$ and send r_i to $\mathcal{P}$.

  (We call this entire procedure the ith round.)
- Finally, $\mathcal{V}$ evaluates $P_\varphi(r_1, \ldots, r_n)$; they *accept* if it's equal to k_n and *reject* otherwise.

Note that in fact, this protocol only involves $\mathcal{V}$ sending over uniform randomness.

§10.4.3 Proof of completeness

We now need to show the completeness and soundness of this protocol. We'll first prove completeness — we'll show that if $\#\mathsf{SAT}(\varphi)$ really is k, then there is a strategy for $\mathcal{P}$ such that $\mathcal{V}$ always accepts.

The idea is that the good prover should just send the polynomials Q as in (9) — more generally, for each i, the good prover sends the polynomial

$$Q_i(z_i) = \sum_{a_{i+1}, \ldots, a_n \in \{0,1\}} P_\varphi(r_1, \ldots, r_{i-1}, z_i, a_{i+1}, \ldots, a_n)$$

(where we're plugging in all the previously chosen randomness for the first $i - 1$ variables, leaving the ith variable as a variable, and plugging in all values in $\{0, 1\}$ for the remaining variables).

Then on the first round, $\mathcal{V}$ is checking that $Q_1(0) + Q_1(1) = k$, which is true (as $k = \#\mathsf{SAT}(F)$ — here Q_1 is the polynomial Q from the intuition discussed in Subsubsection 10.4.2). In the remaining rounds, we've defined $k_{i-1} = Q_{i-1}(r_{i-1})$, so $\mathcal{V}$ will be checking whether $Q_i(0) + Q_i(1) = Q_{i-1}(r_{i-1})$; and this is true by definition (since in $Q_{i-1}(r_{i-1})$ we're plugging in the ith variable to be 0 and 1, and in $Q_i(z_i)$ we're leaving it as a variable z_i).

And finally, in the end $\mathcal{V}$ will be checking that $P_\varphi(r_1, \ldots, r_n) = Q_n(r_n)$, which is also true by definition; so $\mathcal{V}$ is going to accept.

§10.4.4 Proof of soundness

Now we'll prove the soundness of this protocol — that if $\#\mathsf{SAT}(\varphi) \neq k$, then no matter what the prover $\mathcal{P}$ tries to do, $\mathcal{V}$ is going to reject with high probability.

Suppose that $\#\mathsf{SAT}(\varphi) \neq k$ — intuitively, this means the prover is lying on their first message (where they claim $\#\mathsf{SAT}(\varphi)$ is k). We're essentially going to show that with high probability they'll have to *keep* lying, and then $\mathcal{V}$ will catch them in the end.

First, $\mathcal{P}$ can't send $\mathcal{V}$ the correct polynomial Q_1 on the first round — we have $Q_1(0) + Q_1(1) = \#\mathsf{SAT}(\varphi)$ (essentially by definition), and since this is not k, if $\mathcal{P}$ sent over Q_1 then $\mathcal{V}$ would immediately reject. So to get past this check, $\mathcal{P}$ has to send some incorrect polynomial R_1 (i.e., a polynomial with $R_1 \neq Q_1$).

And then since $R_1 \neq Q_1$ and both have degree at most $|\varphi|$, we must have

$$\mathbb{P}_{r_1}[R_1(r_1) \neq Q_1(r_1)] \geq 1 - \frac{|\varphi|}{p} \geq 1 - \frac{1}{2^n}$$

(because $R_1 - Q_1$ has at most $\deg(R_1 - Q_1) \leq |\varphi|$ roots, and we only get $R_1(r_1) = Q_1(r_1)$ when r_1 is one of its roots).

And the point is that now $\mathcal{P}$ is trying to prove to $\mathcal{V}$ that $Q_1(r_1) = R_1(r_1)$, and this statement is still false — so if its original claim was a lie, then with high probability this new claim is a lie as well. Explicitly, on the next step it'll need to provide a polynomial R_2 with $R_2(0) + R_2(1) = R_1(r_1)$; and since $Q_2(0) + Q_2(1) = Q_1(r_1) \neq R_1(r_1)$, it must give an incorrect polynomial $R_2 \neq Q_2$ at this step as well. And so on — the point is that if the prover is trying to prove a false claim at the start of one round, then with probability at least $1 - 2^{-n}$ the claim they'll be trying to prove at the start of the next round will *also* be false.

So either $\mathcal{V}$ catches $\mathcal{P}$ at some point (i.e., the check $R_i(0) + R_i(1) = k_{i-1}$ fails), or with probability at least $(1 - 2^{-n})^n$ we'll end up with $R_n(r_n) \neq Q_n(r_n)$. (More formally, we can show by induction on i that if the intermediate checks all pass, then for each i, we have $R_i(r_i) \neq Q_i(r_i)$ with probability at least $(1 - 2^{-n})^i$.)

But $P_\varphi(r_1, \ldots, r_n)$ is $Q_n(r_n)$ (by definition), so in this case it's *not* $k_n = R_n(r_n)$. And this means the final check $\mathcal{V}$ performs will fail.

So we've shown that if $\#\mathsf{SAT}(\varphi) \neq k$, then $\mathcal{V}$ rejects with probability at least $(1 - 2^{-n})^n \geq 1 - \frac{1}{n}$.

§10.5 IP = PSPACE

Last lecture, we mentioned the following theorem.

Theorem 10.30 (Shamir 1990)
We have $\mathsf{IP} = \mathsf{PSPACE}$.

Today we're going to sketch a proof of this; it'll use several of the same ideas from the proof of Theorem 10.32 (that $\mathsf{Count\text{-}SAT} \in \mathsf{IP}$).

§10.5.1 IP ⊆ PSPACE

We'll first handle the easy direction — that if $f \in \mathsf{IP}$, then $f \in \mathsf{PSPACE}$ as well. The idea is that if we've got an interactive protocol for f with verifier $\mathcal{V}$, then the communication history of this protocol is polynomially bounded, so by carefully iterating over all possible transcripts, we can just compute $\max_{\mathcal{P}} \mathbb{P}[\langle \mathcal{P}, \mathcal{V}\rangle \text{ accepts}]$ (i.e., the probability that the verifier would accept if we used the 'optimal' prover) in polynomial space — and this probability is large if $f(x) = 1$ (in which case there is a good prover $\mathcal{P}$ that makes $\mathcal{V}$ likely to accept) and small if $f(x) = 0$ (in which case there is no such prover), so if we can compute it, then we can decide f.

To formalize this, for each i (representing the round number) and partial transcript z (representing all the messages sent before the ith round), we define $\mathsf{Prob}(x, i, z)$ as the maximum (over all possible provers $\mathcal{P}$) probability (over the remaining randomness of the verifier $\mathcal{V}$) that the protocol accepts when started from this partial transcript. In particular, the probability we're interested in is $\mathsf{Prob}(x, 1, \varepsilon)$ — since here we're imagining that we start at the beginning of the protocol with an empty transcript, and we're asking for the maximum probability over all provers $\mathcal{P}$ that $\mathcal{V}$ accepts.

And we can compute the values of $\mathsf{Prob}(x, i, z)$ recursively — if i is odd (meaning that the prover speaks in the ith round) then we have

$$\mathsf{Prob}(x, i, z) = \max_m \mathsf{Prob}(x, i + 1, zm)$$

(where m represents the message the prover sends on the ith round, and zm denotes the original transcript z with m tacked on — the prover has to choose their message on this round as a function of just x and z, so we're trying to find the best possible message $\mathcal{P}$ could send — the one that maximizes the probability $\mathcal{V}$ accepts — and plug it in).

Meanwhile, if i is even (meaning that the verifier speaks), then we have

$$\mathsf{Prob}(x, i, z) = \sum_m \mathbb{P}[\mathcal{V}(x, z) = m]\mathsf{Prob}(x, i+1, zm)$$

(we're going through all possible messages m the verifier could send, finding the probability (over its internal randomness) that it sends each m, and multiplying by the probability (with the optimal prover) that if it does, the protocol ends in acceptance).

So this gives us a recursion for $\mathsf{Prob}(x, 1, \varepsilon)$ that we can compute in polynomial space (all the messages have polynomial length, and there's polynomially many rounds), and therefore a PSPACE algorithm for f.

§10.5.2 Arithmetization of paths

We'll now handle the more interesting direction, that $\mathsf{PSPACE} \subseteq \mathsf{IP}$. This proof will use configuration graphs and ideas from Savitch's theorem (Theorem 5.12) together with arithmetization.

First, we'll start by recalling the definition of a configuration graph.

Definition 5.7. For a Turing machine $\mathcal{M}$ and an input x, the configuration graph $\mathcal{G}_{\mathcal{M},x}$ is the directed graph whose nodes are all possible configurations of $\mathcal{M}$ on x, and where we draw an edge $c \to c'$ if and only if $\mathcal{M}(x)$ starting from c can get to c' in one step.

Note that $\mathcal{M}$ accepts x if and only if $\mathcal{G}_{\mathcal{M},x}$ has a path from the start state to an accepting state. At a very high level, we're going to start with some problem in PSPACE and take a polynomial-space machine $\mathcal{M}$ that decides it, and we're going to construct an interactive proof system in which $\mathcal{P}$ tries to convince $\mathcal{V}$ that $\mathcal{M}$ accepts x by proving to $\mathcal{V}$ that there exists such a path.

To formalize this, we can assume $\mathcal{M}$ is a 1-tape Turing machine (since we're only dealing with polynomial space bounds) with a unique accepting state c_{acc}. We'll write configurations of $\mathcal{M}$ as bit-strings storing the current tape contents, head location, and state of $\mathcal{M}$ in some nice way; and we'll let $s(n) = \mathsf{poly}(n)$ be such that the configurations of $\mathcal{M}$ on inputs of length n can be described with $s(n)$ bits (we can take $s(n)$ to be some large constant multiple of the space bound on $\mathcal{M}$). For notational convenience, we'll write s in place of $s(n)$.

Then $\mathcal{G}_{\mathcal{M},x}$ has at most 2^s nodes, so if there's a path in $\mathcal{G}_{\mathcal{M},x}$ from c_{start} to c_{acc}, then there's one of length at most 2^s. For convenience, we'll assume $\mathcal{G}_{\mathcal{M},x}$ has an edge from c_{acc} to itself, so that there's a path of length *exactly* 2^s.

We'll then define a useful function

$$\mathsf{PATH}_k(c, c') = \begin{cases} 1 & \text{if there is a path from } c \text{ to } c' \text{ of length } 2^k \\ 0 & \text{otherwise,} \end{cases}$$

so that the goal of our interactive proof is for $\mathcal{P}$ to convince $\mathcal{V}$ that $\mathsf{PATH}_{s(n)}(c_{\text{start}}, c_{\text{acc}}) = 1$. (We'll also define $\mathsf{PATH}_k(c, c')$ to be 0 if c is a valid configuration and c' isn't; we won't yet define it when c isn't a valid configuration.)

The first main idea is to arithmetize PATH. We'll first have $\mathcal{P}$ pick a prime $p \in [2^{2s}, 2^{4s}]$ (we'll then work over $\mathbb{F}_p$ for the rest of the protocol), and we'll try to define a version of PATH_k (for each k) that's a *polynomial* and takes inputs which are arbitrary vectors $x \in \mathbb{F}_p^s$ rather than configurations $c \in \{0, 1\}^s$.

We'll do this recursively. The base case $k = 0$ is quite complicated to describe, so we won't give the details, but it's possible to define a polynomial $\mathsf{PATH}_0(x, y)$ such that:

- It's possible to evaluate the polynomial in $\mathsf{poly}(s(n))$ time (on given inputs $x, y \in \mathbb{F}_p^s$). (In particular, it has a description of length $\mathsf{poly}(s)$.)

- Every variable x_i and y_i has degree at most d for some constant d.
- If $x, y \in \{0,1\}^s$ and x is a valid configuration, then $\mathsf{PATH}_0(x, y)$ is 1 if and only if y is also a valid configuration and $\mathcal{G}_{\mathcal{M},x}$ has an edge from x to y.

(We don't have any requirements on $\mathsf{PATH}_0(x, y)$ when x is not a valid configuration or when one of x and y has non-Boolean values.)

The idea behind proving this is that if we write configurations in a way similar to the one in the proof of Proposition 6.13, then checking whether a configuration x leads to y can be done by taking the **AND** of a bunch of very local checks (each involving a constant number of variables); and we can write each of these local checks as a constant-degree polynomial (outputting 0 if the check passes and 1 if it fails) and take their product. Each variable will be involved in a constant number of these local checks, and therefore will have constant degree.

Meanwhile, for the recursive step (where we want to define PATH_k from PATH_{k-1}), the main idea is that for configurations c and c', we have

$$\mathsf{PATH}_k(c, c') = \sum_{c'' \in \{0,1\}^s} \mathsf{PATH}_{k-1}(c, c'')\mathsf{PATH}_{k-1}(c'', c').$$

(Intuitively, we're guessing the 'middle configuration' c'' — if $\mathcal{M}$ doesn't go from c to c' in 2^k moves, then there's no middle configuration c'' such that it goes from c to c'' and from c'' to c' in 2^{k-1} moves, so each term in this sum is 0. Meanwhile, if it does, then since it's deterministic, there's a unique such c''.) And we can just arithmetize this recursion directly — so we define

$$\mathsf{PATH}_k(x, y) = \sum_{c'' \in \{0,1\}^s} \mathsf{PATH}_{k-1}(x, c'')\mathsf{PATH}_{k-1}(c'', y). \tag{11}$$

Note that in each term of this sum we're plugging in some *constant* c'', and the variables in the two factors $\mathsf{PATH}_{k-1}(x, c'')$ and $\mathsf{PATH}_{k-1}(c'', y)$ are disjoint; this means if all variables in PATH_{k-1} had degree at most d, the same is true of all variables in PATH_k.

So we've now defined a sequence of polynomials $\mathsf{PATH}_0, \ldots, \mathsf{PATH}_s$ (over $\mathbb{F}_p$, and in $2s$ variables $x_1, \ldots, x_s$ and $y_1, \ldots, y_s$) which are essentially arithmetized versions of path-checking functions, and with the following properties:

- PATH_0 is efficiently computable (on any given inputs). (The remaining polynomials may not be.)
- All variables in all the polynomials PATH_k have degree at most d, where d is some constant.
- $\mathcal{M}$ accepts x if and only if $\mathsf{PATH}_s(c_{\text{start}}, c_{\text{acc}}) = 1$.

We're now going to use these polynomials to get an interactive protocol checking whether $\mathcal{M}$ accepts x.

§10.5.3 The interactive protocol

We want an interactive protocol in which $\mathcal{P}$ convinces $\mathcal{V}$ that $\mathsf{PATH}_s(c_{\text{start}}, c_{\text{acc}}) = 1$. More generally, we'll consider the problem where $\mathcal{P}$ wants to convince $\mathcal{V}$ that $\mathsf{PATH}_k(x, y) = a_k$ for some fixed $x, y \in \mathbb{F}_p^s$ and $1 \leq k \leq s$. (This is what we'll need to do at intermediate steps.)

The first idea is that (11) gives an expression for PATH_k as a sum over $c''_1, \ldots, c''_n \in \{0,1\}$. And we can use the same sum-check protocol from Subsubsection 10.4.2 to replace each c''_i with a single random value (peeling off one variable at a time); so after we run the sum-check protocol, $\mathcal{P}$ now wants to convince $\mathcal{V}$ that

$$\mathsf{PATH}_{k-1}(x, r) \cdot \mathsf{PATH}_{k-1}(r, y) = b_k,$$

where $r \in \mathbb{F}_p^s$ is some random vector (consisting of all the random values $r_1, \ldots, r_s \in \mathbb{F}_p$ that $\mathcal{V}$ chose in the sum-check protocol).

Remark 10.37. In more detail, every term in the sum is of the form $\mathsf{PATH}_{k-1}(x, c'') \cdot \mathsf{PATH}_{k-1}(c'', y)$, and the sum-check protocol reduces the problem of proving such a sum has a certain value a_k to the problem of proving that a single term of this form with c'' replaced by some random vector has a certain value b_k. Note that all the polynomials involved will be of the form

$$\sum_{c''_{i+1},\ldots,c''_s\in\{0,1\}} \mathsf{PATH}_{k-1}(x, r_1 \ldots r_{i-1} z c''_{i+1} \ldots c''_s)\mathsf{PATH}_{k-1}(r_1 \ldots r_{i-1} z c''_{i+1} \ldots c''_s, y)$$

(where the polynomial is in the variable z), so they'll have degree at most $2d$; and this means if the prover is lying at the start, then with probability at least $(1 - \frac{2d}{p})^s$ they'll be lying in the end as well.

Now here's where the next trick comes in — we define $q\colon \mathbb{F}_p \to \mathbb{F}_p^s \times \mathbb{F}_p^s$ as

$$q(t) = (1 - t)(x, r) + t(r, y)$$

(we're essentially 'linearly interpolating' between the vectors (x, r) and (r, y); explicitly, this means $q(t) = ((1 - t)x_1 + tr_1, \ldots, (1 - t)r_1 + ty_1, \ldots))$. Then $q(0) = (x, r)$ and $q(1) = (r, y)$. (This is essentially a nice trick to collapse two points down into one.)

And now $\mathcal{P}$ sends a polynomial $Q_{k-1}(t)$ which they claim is $\mathsf{PATH}_{k-1}(q(t))$. Since PATH_{k-1} is a polynomial in $2s$ variables, each of which has degree at most d, we have $\deg Q_{k-1} \leq 2sd$.

Then $\mathcal{V}$ first checks that if Q_{k-1} is correct, then so is b_k — i.e., they check that $Q_{k-1}(0) \cdot Q_{k-1}(1) = b_k$. And now $\mathcal{V}$ wants to check that Q_{k-1} is actually correct; to do so, they pick a random input $r \in \mathbb{F}_p$ and ask $\mathcal{P}$ to prove that $\mathsf{PATH}_{k-1}(q(r)) = Q_{k-1}(r)$.

So now we've reduced k by 1 (replacing (x, y) with $q(r)$ and setting $a_{k-1} = Q_{k-1}(r)$). And we can keep doing this until we get down to the case $k = 0$ — i.e., when we want to prove the statement $\mathsf{PATH}_0(x, y) = a_0$. Finally, when we do get here, $\mathcal{V}$ can simply verify this statement themselves (as PATH_0 is efficiently computable).

Finally, we'll briefly discuss the correctness and soundness for this protocol. If $\mathsf{PATH}_s(c_{\text{start}}, c_{\text{acc}})$ is really 1, then $\mathcal{P}$ can convince $\mathcal{V}$ with probability 1 by simply sending all the polynomials they're supposed to. Meanwhile, if it's *not* 1, then the original claim of $\mathcal{P}$ is a lie. And for each k, if the claim $\mathsf{PATH}_k(x, y) = a_k$ at the start of the kth round is a lie, then as discussed in Remark 10.37, the claim $\mathsf{PATH}_{k-1}(x, r) \cdot \mathsf{PATH}_{k-1}(r, y) = b_k$ we'll have at the end of the sum-check protocol is also a lie with probability at least $(1 - \frac{2d}{p})^s$. And $Q_{k-1}(t)$ and $\mathsf{PATH}_{k-1}(q(t))$ are polynomials of degree at most $2sd$, so the resulting claim $\mathsf{PATH}_{k-1}(q(r)) = Q_{k-1}(r)$ will also be a lie with probability at least $1 - \frac{2ds}{p}$.

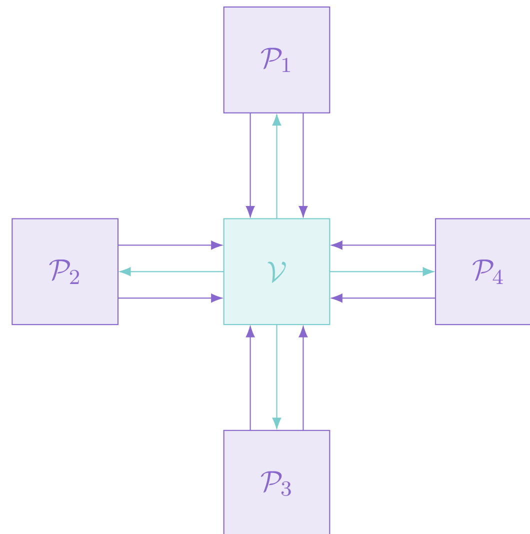
And if the lie gets maintained all the way down to $k = 0$, then $\mathcal{V}$ will catch the prover in the end. So this means $\mathcal{V}$ rejects with probability at least

$$\left(1 - \frac{2d}{p}\right)^{s^2} \left(1 - \frac{2ds}{p}\right)^{s} \geq 1 - \frac{4ds^2}{p},$$

which is very close to 1 (for appropriately chosen p).

§10.6 Multiparty interactive proofs

In all the interactive proofs we've looked at so far, there's one prover and one verifier, and they send messages back and forth. Now we'll consider a slightly different model where there's still one verifier $\mathcal{V}$, but *many* provers $\mathcal{P}_1, \ldots, \mathcal{P}_k$ (where $k = \mathsf{poly}(n)$). The provers want to collaborate to convince $\mathcal{V}$ that $f(x) = 1$; but they can't talk to each other (specifically, $\mathcal{P}_i$ can only see its own transcript history — it can't see the transcript history of the interaction between $\mathcal{V}$ and $\mathcal{P}_j$ for any $j \neq i$).



We define MIP as the class of problems which can be solved with such a system (with a polynomial number of provers and messages).

Definition 10.38. We say $f \in \mathsf{MIP}$ if there is a polynomial-time verifier $\mathcal{V}$ and some $k \leq \mathsf{poly}(n)$ such that for all x:

- If $f(x) = 1$, there exist provers $\mathcal{P}_1, \ldots, \mathcal{P}_k$ such that $\mathcal{V}$ accepts with probability at least $\frac{2}{3}$.
- If $f(x) = 0$, then for all provers $\mathcal{P}_1, \ldots, \mathcal{P}_k$, the probability $\mathcal{V}$ accepts is at most $\frac{1}{3}$.

We've allowed an arbitrary polynomial number of provers in this definition, but it turns out that we only need two.

Theorem 10.39

For all $f \in \mathsf{MIP}$, there is a verifier $\mathcal{V}$ that only needs two provers.

Proof. The main idea is that we can use one prover to check another. Imagine we start with a proof system with verifier $\mathcal{V}$ and provers $\mathcal{P}_1, \ldots, \mathcal{P}_k$. We now create a new proof system with verifier $\mathcal{V}'$ and just two provers $\mathcal{P}_1'$ and $\mathcal{P}_2'$ in the following way.

First, we use $\mathcal{V}'$ and $\mathcal{P}_1'$ to simulate the interaction of $\mathcal{V}$ with *all* the provers $\mathcal{P}_1, \ldots, \mathcal{P}_k$ — so every time $\mathcal{V}$ would ask $\mathcal{P}_i$ a message m, we instead have $\mathcal{V}'$ ask $\mathcal{P}_1'$ the message (i, m) (essentially, we're asking our first prover the question 'if $\mathcal{V}$ sent $\mathcal{P}_i$ the message m, how would it respond?') This gives a simulation of the original protocol, so eventually we've queried a bunch of pairs $(i_1, m_1), \ldots, (i_\ell, m_\ell)$ and our simulation of $\mathcal{V}$ has come to a decision.

So far, we have an issue — $\mathcal{P}_1'$ gets to see *all* the transcripts, so it could be lying (i.e., it could be changing its answers from the perspective of $\mathcal{P}_i$ based on the queries we're trying to ask $\mathcal{P}_j$). The idea is that we're now going to use $\mathcal{P}_2'$ to check that $\mathcal{P}_1'$ is *not* lying in this way. So $\mathcal{V}'$ chooses some random $i \in [k]$ and tries to use $\mathcal{P}_2'$ to check that $\mathcal{P}_1'$ faithfully simulated what $\mathcal{P}_i$ was supposed to do — this means $\mathcal{V}'$ first sends i to $\mathcal{P}_2'$ (essentially telling it which prover it's supposed to be simulating), and then sends the messages m_j with $i_j = i$ one at a time and considers the messages that $\mathcal{P}_2'$ sends back. If the behavior of $\mathcal{P}_2'$ doesn't match that of $\mathcal{P}_1'$ (on these queries), this tells us $\mathcal{P}_1'$ was cheating when simulating $\mathcal{P}_i$, so we *reject*. (The point is that $\mathcal{P}_2'$ only gets to see the messages meant for $\mathcal{P}_i$, so the fact that $\mathcal{P}_1'$ responds in the same way as $\mathcal{P}_2'$ means that the way it's really choosing the messages for its simulation of $\mathcal{P}_i$ really doesn't depend on the transcripts with other provers $\mathcal{P}_j$.)

Finally, we repeat this with polynomially many independent trials. □

§10.6.1 Probabilistic oracle machines

Now we'll consider a slightly different-looking model that actually turns out to be equivalent.

Here we've got a verifier $\mathcal{V}$ which has access to a very long proof $\mathcal{P}$, of length $2^{\mathsf{poly}(n)}$, instead of a prover. Of course $\mathcal{V}$ can't read the whole proof, but it's allowed to ask for specific bits of it — specifically, it gets to toss some coins and choose $\mathsf{poly}(n)$ strings $y_1, \ldots, y_k$ (which we think of as indices into $\mathcal{P}$), and then ask for $\mathcal{P}(y_1), \ldots, \mathcal{P}(y_k)$. (This means it's choosing the strings nonadaptively — before it gets to read the proof.) And once it reads these k bits of the proof, it decides to accept or reject.

Definition 10.40. We say f has a probabilistic polynomial time oracle machine (abbreviated PPTOM) if there exists a polynomial-time verifier $\mathcal{V}$ such that for all x:

- If $f(x) = 1$, then there exists an oracle $\mathcal{P}$ such that $\mathcal{V}^{\mathcal{P}}$ accepts with probability at least $\frac{2}{3}$.
- If $f(x) = 0$, then for all $\mathcal{P}$, the probability $\mathcal{V}^{\mathcal{P}}$ accepts is at most $\frac{1}{3}$.

The difference between this model and an interactive proof (with one prover) is that here the prover can't adaptively lie — the bits of $\mathcal{P}$ (i.e., the answers it'll give) are all fixed beforehand, so it can't change them based on the queries $\mathcal{V}$ makes (while the prover in an interactive proof could).

However, it turns out that this model is the same as *multiparty* interactive proofs!

Theorem 10.41

We have $f \in \mathsf{MIP}$ if and only if f has a probabilistic polynomial time oracle machine.

Proof sketch. The idea for the forwards direction is that if f has a MIP, then we can make an oracle $\mathcal{P}$ that encodes the provers' responses to all possible queries. Meanwhile, for the backwards direction, suppose f has an oracle machine. Then to get a multiparty interactive proof (with two provers), the idea is that we can treat $\mathcal{P}_1$ as the oracle and use $\mathcal{P}_2$ to check that $\mathcal{P}_1$ isn't adaptively changing its answers — so once we've made all our queries $y_1, \ldots, y_k$ to $\mathcal{P}_1$, we choose some random $r \in [k]$, query y_r to $\mathcal{P}_2$, and check that it gives the same answer. $\square$

The final theorem we'll see is that multiparty interactive proofs are actually *extremely* powerful.

Theorem 10.42 (BFL 1991)

We have $\mathsf{MIP} = \mathsf{NEXP}$.

The idea for $\mathsf{MIP} \subseteq \mathsf{NEXP}$ is that if $f \in \mathsf{MIP}$ then it has a probabilistic polynomial time oracle machine $\mathcal{V}$; and then we can just nondeterministically guess the good proof $\mathcal{P}$ of $2^{\mathsf{poly}(n)}$ length and compute the probability $\mathcal{V}^{\mathcal{P}}$ accepts. (If $f(x) = 1$ then there should be some $\mathcal{P}$ such that this probability is large; and if $f(x) = 0$ then for all $\mathcal{P}$ this probability should be small.) We won't discuss the reverse direction (which is the harder one).

§11 Probabilistically checkable proofs

Our next topic is *probabilistically checkable proofs* (abbreviated PCPs — but this has nothing to do with the Post correspondence problem). These are extremely tricky objects — they essentially show that every problem in NP can be verified in a way that just sounds impossible, using the power of randomness in a crucial way. And there's a deep connection between PCPs and showing that problems in NP are hard to even *approximately* solve — you might think some optimization problems are hard to solve *exactly*, but it turns out that for many problems, even getting something remotely *close* to the answer is hard.

§11.1 The definition of PCPs

The basic setup of a PCP is that we have a polynomial-time verifier $\mathcal{V}$, which takes an input x. It tosses coins to get some randomness r, and then it makes a bunch of queries to some proof (which we can think of as an oracle — we query a certain index, and it'll give us a 0 or 1). (These queries are nonadaptive, meaning that $\mathcal{V}$ chooses all of them before getting to see the answers.) We're going to measure both the number of coin tosses, which we denote by $r(n)$, and the number of queries, which we denote by $q(n)$. (For comparison, in the normal NP setting we have no randomness and we read the entire proof; here we've got randomness and we're only reading part of the proof.)

One way to phrase this definition is in terms of probabilistic polynomial time oracle machines (as defined last class).

Definition 11.1. For $r, q\colon \mathbb{N} \to \mathbb{N}$ (measuring the randomness and the number of queries), $s \in [0,1]$ (called the *soundness parameter*), and a decision problem f, we say $f \in \mathsf{PCP}_s[r(n), q(n)]$ if there is a probabilistic polynomial time oracle machine $\mathcal{V}^{\mathcal{P}}$ such that for every x of length n, we have that $\mathcal{V}$ tosses at most $r(n)$ coins and makes at most $q(n)$ queries, and:

- If $f(x) = 1$, then there exists a proof $\mathcal{P}$ such that $\mathbb{P}_r[\mathcal{V}^{\mathcal{P}}(x; r) \text{ accepts}] = 1$.
- If $f(x) = 0$, then for all proofs $\mathcal{P}$, we have $\mathbb{P}_r[\mathcal{V}^{\mathcal{P}}(x; r) \text{ accepts}] < s$.

Note that even though we don't have any constraints on how long the proof $\mathcal{P}$ is in this definition, we can always assume $|\mathcal{P}| \leq q(n) \cdot 2^{r(n)}$ — there's $2^{r(n)}$ possible coin tosses that $\mathcal{V}$ could make, and for each choice of coin tosses, it'll make $q(n)$ queries. So we can imagine $\mathcal{P}$ as being indexed by $\{0,1\}^{r(n)} \times [q(n)]$ (corresponding to the choice of randomness and the query number) — any other bits in the original proof don't matter, as they'll never get read.

First, here are a few more or less self-explanatory facts about PCPs.

Fact 11.2 — We have $\mathsf{NP} \subseteq \mathsf{PCP}_0[0, \mathsf{poly}(n)]$.

The idea is that for any problem in NP, there's a verifier with no randomness that reads the whole witness and decides whether to accept or reject; and we can think of the witness as the proof, and reading it as making $\mathsf{poly}(n)$ queries.

Fact 11.3 — For all $\delta \in [0,1]$, we have $\mathsf{PCP}_\delta[O(\log n), \mathsf{poly}(n)] \subseteq \mathsf{NP}$.

The idea is that if we've got a PCP verifier for f that tosses only $O(\log n)$ coins, then to get a NP verifier, we take the proof $\mathcal{P}$ to be our witness. And we can just enumerate over all $O(\log n)$-bit strings corresponding to the outcomes of the coin tosses (of which there are at most $\mathsf{poly}(n)$), run the verifier on every single one, and check whether it accepts on every single run (in which case $f(x) = 1$) or not (in which case $f(x) = 0$).

Fact 11.4 — We have $\mathsf{NP} \subseteq \mathsf{PCP}_1[O(\log n), 3]$.

Here we're only tossing $O(\log n)$ coins and only making 3 queries. But we've set the soundness parameter to 1, so either the verifier accepts on every random string (in the **YES** case) or there's *some* random string on which it doesn't accept (in the **NO** case) — this is the tiniest possible gap we could have. This statement looks really weird, but it turns out to be highly motivating for the connection between PCPs and approximation (which we'll see later).

Proof. First, every NP problem can be reduced in polynomial time to 3SAT, so it suffices to give a PCP with these parameters for 3SAT. And we'll give a very direct one (it's not a coincidence that the two 3's — in 3SAT and our number of queries — are equal.)

Given a formula $\varphi = C_1 \wedge \cdots \wedge C_m$ (where each clause C is an **OR** of at most three literals), we take the proof $\mathcal{P}$ to be a satisfying assignment to φ. And then $\mathcal{V}^{\mathcal{P}}(\varphi)$ works as follows: it chooses a random clause C of φ, queries $\mathcal{P}$ on the three variables in that clause C, and accepts if and only if C is satisfied. So in other words, we're picking a random clause and just checking that the assignment satisfies this clause.

Why does this work? First, if φ is satisfiable, then when $\mathcal{P}$ is an actual satisfying assignment, *every* clause will be satisfied; so the verifier always accepts.

Meanwhile, if φ is unsatisfiable, then no matter what assignment $\mathcal{P}$ we stick in, there'll be *some* clause that's not satisfied. And we pick that clause with probability $\frac{1}{m}$, which means the probability we accept is at most $1 - \frac{1}{m} < 1$. □

§11.2 The PCP theorem

In Fact 11.4, we saw that there's a PCP for 3SAT with just $O(\log n)$ randomness and a constant number of queries, but where the probability of accepting a bad proof is $1 - \frac{1}{m}$. This is certainly less than 1, but it's *extremely* close to 1, so it sort of feels like we got off on a technicality.

> **Question 11.5.** Can we get a statement like Fact 11.4 with soundness parameter $s \leq 0.9$?

(Really, we just want s to be a constant in $(0, 1)$, so that it's bounded away from 1.)

It turns out that the answer is yes!

> **Theorem 11.6** (PCP theorem, ALMSS 1992)
> There exists $\delta < 1$ such that $\mathsf{NP} = \mathsf{PCP}_\delta[O(\log n), O(1)]$.

What this says is that every NP proof system whatsoever can be transformed into some sort of spot-checking — we start with an arbitrary NP verifier $\mathcal{V}$ that reads x and the *entire* proof, and we can somehow transform it into one where $\mathcal{V}$ gets to toss a few coins but only reads a *constant* number of bits of the proof on any run. (Note that $r(n) = O(\log n)$, so the length of the proof is $\mathsf{poly}(n)$.)

And moreover, once we've got such a statement for a constant $\delta < 1$, then we can make δ arbitrarily close to 0 by 'repeating' the PCP over and over (we can imagine running the verifier for some constant number of independent trials, and accepting if and only if all runs accept).

This is extremely remarkable — it means that on a **YES** instance we always accept, while on a **NO** instance we're only querying a *constant* number of bits, and we still accept with probability only at most 10^{-9}.

> **Remark 11.7.** Is there work on what the $O(1)$ term (for the number of queries) is? The answer is yes — there's lots of work on the optimal tradeoff between the number of queries and the soundness. In fact, it can be made 3 (with $\delta = \frac{7}{8} + \varepsilon$ for any $\varepsilon > 0$) — there's a famous 3-bit PCP by Håstad (which we'll talk about later), while it can't be made 2 unless $\mathsf{P} = \mathsf{NP}$ (we'll see this on our problem set).

§11.3 PCPs and hardness of approximation

PCPs are already very cool on a mathematical level (and the proof of the PCP theorem involves lots of nice stuff such as error-correcting codes). But one of their main applications to theoretical computer science, and the reason they're so important, is that they imply that many NP-hard problems are even hard to *approximately* solve; and this is what we'll discuss now.

§11.3.1 Constraint satisfaction problems

Previously, we've seen the problem k-SAT — where we're given a Boolean formula with clauses of length up to k (and want to figure out whether it's satisfiable). We can define an optimization version of k-SAT, called Max-k-SAT, where we're given a collection of clauses (which may or may not be simultaneously satisfiable) and want to find an assignment satisfying the *maximum* number of these clauses.

Definition 11.8 (Max-k-SAT)

- **Input:** $\ell \in \mathbb{N}$ and a collection of clauses $C_1, \ldots, C_n$, each of which is an **OR** of at most k literals from $x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}$.
- **Decide:** whether there is an assignment to the variables x_i satisfying at least ℓ clauses.

(We've written this as a decision problem, where we want to figure out whether there's an assignment satisfying a certain number of clauses.)

In fact, we're going to generalize this even further — in Max-k-SAT we've got a collection of clauses where each clause involves k variables and is an **OR** of variables and their negations. We'll generalize this to the problem Max-k-CSP, where each constraint still involves at most k variables, but the constraints are allowed to be *arbitrary* Boolean functions on these k variables. (We now call them *constraints* instead of clauses.)

Definition 11.9 (Max-k-CSP)

- **Input:** $\ell \in \mathbb{N}$ and a collection $\mathcal{C}$ of functions $f_1, \ldots, f_n$, such that each f_i is a function on a k-element subset of $\{x_1, \ldots, x_n\}$.
- **Decide:** whether $\mathsf{OPT}(\mathcal{C}) \geq \ell$, where $\mathsf{OPT}(\mathcal{C})$ is the maximum number of functions $f_i \in \mathcal{C}$ that can be simultaneously satisfied by an assignment to the variables x_i.

Here we think of k as a constant, so we can imagine specifying each function f_i by giving its entire truth table (which has constant size).

Example 11.10

The problem Max-k-SAT is the special case of Max-k-CSP where each f_i is an **OR** of literals.

So Max-k-CSP is a generalization of Max-k-SAT where we allow our functions to be arbitrary, rather than just **OR**s of literals. It turns out that the k-CSP problem captures lots of things beyond k-SAT — there are lots of situations where you'll have local constraints each involving a small number of variables, and you want to know if they can be simultaneously satisfied.

We're going to discuss the problem of *approximating* Max-k-CSP. First, we can formally define what it means to approximate Max-k-CSP in the following way.

Definition 11.11. For $s \in (0, 1)$, we say an s-approximation to Max-k-CSP is an algorithm that, given a k-CSP $\mathcal{C}$, outputs an assignment that satisfies at least $s \cdot \mathsf{OPT}(\mathcal{C})$ of its constraints.

So the algorithm might not be finding an assignment that satisfies the maximum possible number of constraints, but it finds one that gets reasonably close — specifically, within a s-fraction of the maximum.

(This is not a bad choice of notation — we're actually going to see that the parameter s here is related to the soundness parameter s from PCPs.)

First, we'll see that nontrivial approximations do exist.

Proposition 11.12
For all k, there is a polynomial-time $\frac{1}{2}$-approximation to Max-k-SAT.

So we probably can't find an assignment that actually achieves the maximum (that's the problem Max-k-SAT itself, which is NP-hard), but we can easily get within a factor of $\frac{1}{2}$.

Proof. Every clause is satisfied by either the all-0's assignment or the all-1's assignment (if it has some negative literal then it's satisfied by all-0's, and otherwise it's satisfied by all-1's). This means for any collection $\mathcal{C}$ of clauses, by the pigeonhole principle either the all-0's assignment or the all-1's assignment satisfies at least half the clauses, and we're done (we can plug in both and see which one works). □

(Unsurprisingly, most approximation algorithms are more complicated than this.)

Remark 11.13. This can be improved — if all variables are distinct, then we can get a $(1 - \frac{1}{2^k})$-approximation. (A *random* assignment is expected to satisfy a $(1 - \frac{1}{2^k})$-fraction of the clauses, and this can be derandomized.) And amazingly, this is tight — there are PCPs showing you can't do any better than this.

Remark 11.14. For the more general problem Max-k-CSP, the best approximation algorithm we know of is a $\frac{ck}{2^k}$-approximation (for some constant c). It's still an open problem whether this is the right answer, but people believe it is (people believe there should exist PCPs that show this is tight, but we don't know for sure).

§11.3.2 PCPs and inapproximability

We want to discuss the hardness of approximating Max-k-CSP, and for this it's better to have a decision problem — so we'll now define a decision version of what it means to approximate Max-k-CSP (rather than the version we've stated where we're trying to *output* an assignment), and then we'll show that the NP-hardness of this decision problem is *equivalent* to PCPs in a very strong sense.

Definition 11.15 (s-Gap-k-CSP)

- **Input:** an instance of Max-k-CSP (i.e., a set of constraints $\mathcal{C}$ forming a k-CSP).
- **Promise:** either $\mathsf{OPT}(\mathcal{C}) = |\mathcal{C}|$ (which we call the **HIGH** case), or $\mathsf{OPT}(\mathcal{C}) < s\,|\mathcal{C}|$ (which we call the **LOW** case).
- **Decide:** whether we're in the **HIGH** case or **LOW** case.

This is a promise problem, similarly to the problem CAPP we saw when discussing derandomization — we've got a promise, and we only care about what the algorithm does on instances satisfying that promise (we don't care what the algorithm outputs if $\mathcal{C}$ isn't in either the **HIGH** or **LOW** case). Here our promise says that either *all* the constraints can be simultaneously satisfied, or we can't even find an assignment satisfying a s-fraction of the constraints.

And the big theorem is that there's an equivalence between the NP-hardness of this decision problem and the existence of PCPs; this is one of the major motivations for why we think about PCPs.

Theorem 11.16 (Equivalence theorem)
We have $\mathsf{NP} = \mathsf{PCP}_s[O(\log n), k]$ if and only if there is a polynomial-time reduction from 3SAT to s-Gap-k-CSP.

We always have $\mathsf{NP} \supseteq \mathsf{PCP}_s[O(\log n), k]$ by Fact 11.3, so this theorem really says that the reverse inclusion is true — i.e., *every* problem in NP has a PCP with these parameters — if and only if s-Gap-k-CSP is NP-hard. (Here 3SAT is just a proxy for any NP-complete problem, which we use for concreteness.)

To be very precise, the statement that there exists a reduction R from 3SAT to s-Gap-k-CSP means that for all formulas φ, if φ is a **YES** instance of 3SAT (i.e., it's satisfiable), then the k-CSP $R(\varphi)$ should be in the **HIGH** case, meaning that all its constraints can be simultaneously satisfied. And conversely, if φ is a **NO** instance of 3SAT (i.e., it's unsatisfiable), then $R(\varphi)$ should be in the **LOW** case, meaning that every assignment satisfies at most a s-fraction of the constraints.

What does this have to do with the hardness of approximating Max-k-CSP? The point is that approximating Max-k-CSP is related to this decision problem in the following way.

Proposition 11.17
If there is a polynomial-time algorithm which is a s-approximation to Max-k-CSP, then there is a polynomial-time algorithm for s-Gap-k-CSP.

This is how PCPs are used to show the harndess of approximation — if there's a PCP with the appropriate parameters, then by Theorem 11.16 there's a reduction from 3SAT to s-Gap-k-CSP; and then if $\mathsf{P} \neq \mathsf{NP}$ this means there can't be a polynomial-time algorithm for s-Gap-k-CSP, so there can't be one for s-approximating Max-k-CSP either (by the contrapositive of Proposition 11.17).

Proof. Suppose we've got an algorithm that, given $\mathcal{C}$, always outputs an assignment satisfying at least a s-fraction of the optimum number of constraints; and we want to turn it into an algorithm for solving s-Gap-k-CSP. The idea is that if $\mathcal{C}$ is in the **HIGH** case, then the s-approximation outputs an assignment satisfying at least $s \cdot \mathsf{OPT}(\mathcal{C}) = s\,|\mathcal{C}|$ of the constraints. Meanwhile, if $\mathcal{C}$ is in the **LOW** case, then no matter what the algorithm does, the assignment it outputs is going to have to satisfy fewer than $s\,|\mathcal{C}|$ constraints (since this is true of every assignment whatsoever). So we can just run our s-approximation and compare the number of satisfied constraints to $s\,|\mathcal{C}|$, and this lets us distinguish between the **HIGH** and **LOW** cases. □

§11.3.3 Proof of the equivalence theorem

Now we're going to prove the equivalence theorem (Theorem 11.16) — i.e., we're going to see how PCPs are basically equivalent to reductions to these gap versions of k-CSP.

Proof of the forwards direction. First we'll show the forwards direction — that if every problem in NP has a PCP with soundness s only making k queries, then we can get a reduction from 3SAT to s-Gap-k-CSP.

We'll start with a PCP for 3SAT — let $\mathcal{V}$ be a PCP verifier that makes k queries to some proof $\mathcal{P}$ and tosses $O(\log n)$ coins (so $\mathcal{P}$ has length $\mathsf{poly}(n)$), and let φ be the 3CNF we're given as input.

Now we'll make a CSP in variables $x_1, \ldots, x_{|\mathcal{P}|}$ — so we've got one variable for each bit of the proof, and coming up with an assignment to these variables corresponds to filling in the proof.

There's $O(\log n)$ random bits, so there's $\mathsf{poly}(n)$ choices for the randomness r. For each of these, we make a constraint f_r which outputs 1 if and only if $\mathcal{V}^{\mathcal{P}}(\varphi, r)$ accepts — so f_r is a function of the k variables x_i that $\mathcal{V}$ queries on the randomness r, and for each of the 2^k possible assignments to those variables, it records whether the verifier would accept or reject if $\mathcal{P}$ answered with those bits. (We can imagine obtaining the truth table of f_r by going through all 2^k possible answers of $\mathcal{P}$ one at a time and simulating $\mathcal{V}$ to see whether it'd accept or reject on those answers.)

If φ is satisfiable, then there's an assignment(corresponding to the good proof $\mathcal{P}$) such that the verifier accepts on *all* choices of the randomness, which means f_r is satisfied for all r. So then we're in the **HIGH** case — there's an assignment such that all the constraints are satisfied.

On the other hand, if φ is unsatisfiable, this means no matter what assignment A we take (equivalently, no matter what proof we put in), the probability that the verifier accepts is less than s; and since we defined a separate constraint for each of the possible strings of randomness, this means $\mathbb{P}_r[f_r(A) = 1] < s$. So every assignment satisfies less than a s-fraction of the constraints, which means we're in the **LOW** case.

So the collection of constriants $\{f_r\}$ is an instance of s-Gap-k-CSP with the properties we want of a reduction — if φ is satisfiable then it's in the **HIGH** case, and if φ is unsatisfiable then it's in the **LOW** case. ☐

Proof of the backwards direction. Now we'll prove the backwards direction, that if there's a polynomial-time reduction R from 3SAT to s-Gap-k-CSP, then we can get a PCP out of this reduction. We'll make a PCP just for 3SAT; we can then get a PCP for any problem in NP by first reducing it to 3SAT.

Suppose the input to our 3SAT problem is a 3CNF φ. The idea is that we're going to take the proof $\mathcal{P}$ to be an assignment to the variables of the k-CSP $R(\varphi)$ satisfying the maximum number of constraints.

Then to get our PCP verifier, we first run the reduction to get the s-Gap-k-CSP instance $R(\varphi)$, which means we've got a set of constraints $\mathcal{C} = \{f_i\}$ over some variables $y_1, \ldots, y_{\mathsf{poly}(n)}$. And we then choose a random constraint $f_i \in \mathcal{C}$, and then query $\mathcal{P}$ (which is supposed to be an assignment of the variables y_j) to see whether f_i is satisfied — the point is that f_i only depends on k variables, so we just query those k entries of $\mathcal{P}$. And finally, we accept if and only if f_i is satisfied.

The point is that if φ is satisfiable, then $R(\varphi)$ (the output of the reduction) is in the **HIGH** case, so there's some assignment satisfying all the constraints — i.e., $\mathsf{OPT}(\mathcal{C}) = |\mathcal{C}|$ — and this means there's some proof on which $\mathcal{V}$ accepts with probability 1 (namely, the assignment that satisfies all the constraints).

Meanwhile, if φ is unsatisfiable, then $R(\varphi)$ is in the **LOW** case, which means $\mathsf{OPT}(\mathcal{C}) < s\,|\mathcal{C}|$. So no matter what assignment we stick in $\mathcal{P}$, it'll satisfy less than a s-fraction of the constraints; and our verifier is picking a random constraint and checking whether it's satisfied, so the probability it accepts is less than s. ☐

§11.3.4 Hardness of approximation for Max-Clique

There's lots of problems you might want to solve that aren't about arbitrary functions, but concrete objects such as graphs. One example of an interesting problem is Max-Clique.

Definition 11.18 (Max-Clique)
- **Input:** a graph $G = (V, E)$.
- **Output:** the maximum $S \subseteq V$ such that for all distinct $u, v \in S$, we have $\{u, v\} \in E$.

In other words, we're given a graph G, and we're trying to find the largest clique (a *clique* in a graph is a collection of vertices such that every pair has an edge).

Notation 11.19. We use $\omega(G)$ to denote the size of the largest clique in a graph G.

We saw in 18.404 that finding the maximum clique in a graph is NP-hard. So we'll consider the problem of *approximately* solving Max-Clique.

Definition 11.20. A s-approximation to Max-Clique is an algorithm that, given a graph G, outputs a clique of size at least $s \cdot \omega(G)$.

And it turns out that approximating Max-Clique — in fact, even getting a *remotely* close approximation — is also very hard.

Theorem 11.21 (Håstad 1999, Zuckerman 2007)
If there is a polynomial-time $n^{0.999}$-approximation to MaxClique, then $\mathsf{P} = \mathsf{NP}$.

(Here n denotes the number of vertices of the graph.)

This is an astonishing result — what's behind it is a statement about a decision version of this problem (similar to s-Gap-k-CSP) that says we can't even distinguish graphs with $\omega(G) = n^{\varepsilon}$ from graphs with $\omega(G) = n^{1-\varepsilon}$.

We're not going to prove this, but we will prove a simpler version.

Proposition 11.22
If there is a polynomial-time $O(1)$-approximation to MaxClique, then $\mathsf{P} = \mathsf{NP}$.

Proof. We'll use the PCP theorem for this — suppose we've got a PCP verifier $\mathcal{V}$ for 3SAT that makes k queries (where k is a constant), tosses $c(n)$ coins, and makes queries to a proof of length $\ell(n)$; and suppose it has soundness δ. Then given a 3CNF φ, we want to turn the PCP for φ into a graph. (We'll write c and ℓ instead of $c(n)$ and $\ell(n)$ for convenience.)

We're going to make a graph that corresponds to all possible accepting runs of the verifier — so the graph $\mathcal{G}_\varphi$ will have vertices of the form $(r, q_1, \ldots, q_k, a_1, \ldots, a_k)$, where each $q_i \in [\ell]$ represents a query, each $a_i \in \{0, 1\}$ represents the answer to that query, and $r \in \{0, 1\}^c$ represents the sequence of coin tosses. And we put in all vertices of this form such that $\mathcal{V}^{\mathcal{P}}(\varphi, r)$ makes the queries q_i and accepts if it receives the answers a_i — so we can think of $\mathcal{G}_\varphi$ as a graph representing all the ways for the verifier to accept.

(We can construct this graph in a similar way to how we construct the k-CSP in Theorem 11.16 — if $c(n) = O(\log n)$ and k is constant, then we can just enumerate over all $\mathsf{poly}(n)$ strings r and check what queries $\mathcal{V}$ makes on each, and how it responds to each of the 2^k possible answers.)

And we draw an edge between two configurations if they're consistent — more precisely, we draw an edge between $(r, \{q_i\}, \{a_i\})$ and $(r', \{q_i'\}, \{a_i'\})$ if $r \neq r'$, but for all i and j, if $q_i = q_j'$ then $a_i = a_j'$. (This essentially states that if the first vertex (corresponding to running the verifier on randomness r) involves receiving a certain answer a to some query q, then if we make the same query q in the second vertex (with randomness r'), we should receive the same answer.)

Then a clique in this graph corresponds to a consistent proof, and vice versa; so we can show that

$$\max_{\mathcal{P}}[\mathbb{P}[\mathcal{V}^{\mathcal{P}} \text{ accepts}]] = \frac{\omega(\mathcal{G}_\varphi)}{2^c}.$$

In other words, finding the maximum clique size actually tells us the maximum acceptance probability we could have (over the choice of proof). (The point is that for any proof, the set of runs of randomness r on which $\mathcal{V}$ accepts corresponds to a clique in the graph, where we set each a_i to be the answer $\mathcal{P}$ would give to the query q_i. And the converse is true as well, because we've drawn edges in a way that ensures the proof is consistent.)

And then if φ is satisfiable, then there exists some proof $\mathcal{P}$ such that $\mathcal{V}^{\mathcal{P}}$ accepts with probability 1, which means $\omega(\mathcal{G}_\varphi) = 2^c$. Meanwhile, if φ is unsatisfiable, then for every proof $\mathcal{P}$ the probability $\mathcal{V}^{\mathcal{P}}$ accepts is at most s, which means $\omega(\mathcal{G}_\varphi) < \delta \cdot 2^c$. □

§11.3.5 Hardness of approximation for Max-3SAT

One example of a problem where we know the *exact* optimal approximation ratio is Max-3SAT. We'll actually consider Max-E3SAT, where each clause is required to have *exactly* three distinct literals.

Theorem 11.23
The problem Max-E3SAT has a polynomial-time $\frac{7}{8}$-approximation.

In other words, given an instance of Max-E3SAT, there's a polynomial-time algorithm that outputs an assignment satisfying at least a $\frac{7}{8}$-fraction of the optimum number. The idea of the proof is that we can just choose a uniform random assignment; this will satisfy any clause with probability $\frac{7}{8}$, so by linearity of expectation it's expected to satisfy at least $\frac{7}{8}$ of all clauses (and $\frac{7}{8}$ of all clauses is certainly at least $\frac{7}{8}$ of the optimum). This is a randomized algorithm, but it's possible to derandomize it — for example, one way to do so is by using a 3-wise independent assignment instead of a completely random assignment (this is all we need for the proof to work, and such distributions can be generated with short seeds).

And there's more sophisticated algorithms based on semidefinite programming (which we are not going to discuss) that show we don't need the assumption that all clauses have three distinct literals — i.e., we can get an $\frac{7}{8}$-approximation even for Max-3SAT.

And what's remarkable is that assuming $\mathsf{P} \neq \mathsf{NP}$, we know this is the best we can do!

Theorem 11.24 (Håstad 2001)
If there exists $\varepsilon > 0$ such that there is a polynomial-time $(\frac{7}{8} + \varepsilon)$-approximation to Max-3SAT, then $\mathsf{P} = \mathsf{NP}$.

This is quite striking — we've got a very silly algorithm achieving a $\frac{7}{8}$-approximation (we just choose a random assignment), but even doing slightly better than this silly algorithm would imply $\mathsf{P} = \mathsf{NP}$.

§11.3.6 Vertex-Cover and the unique games conjecture

We'll now discuss a few major open problems in approximation. The first concerns the problem Vertex-Cover.

Definition 11.25 (Vertex-Cover)
- **Input:** a graph $G = (V, E)$.
- **Output:** the minimum $S \subseteq V$ such that for all $\{u, v\} \in E$, either $u \in S$ or $v \in S$.

In other words, we're given a graph and we're trying to find the smallest possible subset S of vertices such that every edge has at least one endpoint in S. (Such a subset is called a vertex cover, which is the reason for the problem's name.)

Theorem 11.26
There is a polynomial-time 2-approximation to Vertex-Cover.

Here we're talking about a *minimization* problem rather than a maximization one, so a 2-approximation means that the vertex cover the algorithm outputs is *at most twice* the size of the minimum one.

Proof sketch. The idea is that we start by taking any edge $e = \{u, v\}$; then we know at least one of its endpoints has to be in any vertex cover S, so we just take *both* its endpoints. And then we remove e as well as all edges incident to u and v (which we've already covered). And then we keep doing this — we find another edge and place both its endpoints in the vertex cover, and so on.

In the end, the collection of edges we chose are vertex-disjoint by construction, so any vertex cover has to have at least one vertex from each of them; and we've taken both the endpoints, so our vertex cover is at most twice the size of the optimum. □

Open question 11.27. Is 2 the best possible approximation factor? Or is there a 1.999-approximation?

We don't know the answer — we can get a hardness result from the assumption $\mathsf{P} \neq \mathsf{NP}$, but the best result we know has a different constant.

Theorem 11.28 (Dinur)
If there is a polynomial-time 1.3-approximation to Vertex-Cover, then $\mathsf{P} = \mathsf{NP}$.

However, there's something called the *unique games conjecture* which *would* imply that there's no 1.999-approximation to VertexCover (more precisely, that 2 really is the best we can do).

We'll describe this conjecture in a way that doesn't have anything to do with uniqueness or games; the name is because the original conjecture was formulated in a very different way.

First, we'll think of p as a very large prime, and consider a certain optimization problem over $\mathbb{F}_p$. As a starting point, we'll consider the following problem.

Definition 11.29 ($\mathsf{2LIN}_p$)

- **Input:** a system of linear equations over $\mathbb{F}_p$, where each equation is of the form $x_i + x_j = a$.
- **Output:** a solution to the system.

Solving this problem is easy — we can just use Gaussian elimination (for example) to solve the system of equations in polynomial time. So that's not what the unique games conjecture is about; but it's the starting point. Instead, we consider the following question.

Question 11.30. Imagine we've got a system of equations that's not completely satisfiable. How well can we *approximately* solve it?

So for a $\mathsf{2LIN}_p$ instance $\mathcal{C}$, we let $\mathsf{OPT}(\mathcal{C})$ denote the maximum number of equations that can be simultaneously satisfied; and our goal is to find an assignment such that the number of equations it satisfies is a decent fraction of $\mathsf{OPT}(\mathcal{C})$.

One way we could try to get an approximation is by picking a uniform random assignment; then we have a $\frac{1}{p}$ probability of satisfying each equation, so we'd expect the assignment to satisfy a $\frac{1}{p}$-fraction of all of them. But we're thinking of p as very large, so this is tiny; we'd like to know if we can do better.

Conjecture 11.31 (Unique games) — For every $\varepsilon > 0$, there is a prime p such that it is NP-hard to distinguish between a $\mathsf{2LIN}_p$ instance $\mathcal{C}$ with $\mathsf{OPT}(\mathcal{C}) \leq \varepsilon$ vs. $\mathsf{OPT}(\mathcal{C}) \geq 1 - \varepsilon$.

In particular, this conjecture would mean there's no constant-factor approximation to $\mathsf{Max\text{-}2LIN}_p$ (as $p \to \infty$).

§11.3.7 Approximation in graph coloring

Another area where there's major open problems in approximation is graph coloring — graph coloring is simple to describe, but we don't know very much about how well we can approximate it.

Definition 11.32. The chromatic number of a graph G, denoted $\chi(G)$, is the minimum number of colors needed to color the vertices of G such that no edge has two endpoints with the same color.

We do know the following statement about the hardness of approximating $\chi(G)$ in general.

Theorem 11.33 (FK 1998, Z 2007)
If there is a polynomial-time $n^{0.999}$-approximation algorithm for graph coloring, then $\mathsf{P} = \mathsf{NP}$.

In other words, if there's an algorithm which, given a graph with chromatic number n^{ε}, finds a coloring with even $n^{1-\varepsilon}$ colors, then we'd have $\mathsf{P} = \mathsf{NP}$.

But this is for graphs with chromatic number roughly n^{ε}; what if we promise that the graph has much smaller (i.e., constant) chromatic number?

Open question 11.34. Is there a polynomial-time algorithm that, given any 3-colorable graph, finds a coloring with at most $n^{1/10}$ colors?

This is a major open problem — we know that there *is* a polynomial-time algorithm achieving slightly less than $n^{1/5}$ colors, and we know there isn't one achieving 5 colors (unless $\mathsf{P} = \mathsf{NP}$). But there's a big gap between 5 and $n^{1/5}$, and shrinking it is a major open problem.

§11.4 A weaker PCP theorem

Last class we stated the PCP theorem (Theorem 11.6) — that it's possible to take any NP verifier and make a new verifier that tosses $O(\log n)$ coins and only makes a *constant* number of queries to the proof (and which always accepts **YES** instances, and is fooled on **NO** instances with probability less than δ). We're not going to prove this, but today we'll prove a slightly weakened form.

Theorem 11.35
We have $\mathsf{NP} \subseteq \mathsf{PCP}_{1/3}[O((\log n)^2), O((\log n)^2)]$.

(The soundness parameter $\frac{1}{3}$ is arbitrary, and can be amplified to any value we want.)

This is already quite interesting — if we've got $O((\log n)^2)$ randomness, this corresponds to proofs of length $n^{O(\log n)}$. But we're only making $O((\log n)^2)$ queries, which is much smaller than reading the whole proof (it's also much smaller than reading the whole proof for the *original* NP verifier, which has length $\mathsf{poly}(n)$).

§11.4.1 The high-level idea

We want to construct a PCP verifier for every NP problem where we've got a proof of length $n^{O(\log n)}$ (since we've got $O((\log n)^2)$ bits of randomness), but on any run of randomness we only make $O((\log n)^2)$ queries. We're going to construct such a verifier just for 3SAT; this suffices because 3SAT is NP-complete.

The idea is that we'll start with a formula φ in $n = 2^k$ variables, which we'll think of as being indexed by k-bit strings, and we'll imagine that there's a satisfying assignment A. And our goal is to find a way to encode φ and A such that checking satisfiability can be done with just $(\log n)^2$ queries.

The main idea is to use the sum-check protocol from Theorem 10.32 (where we saw an interactive proof for #SAT). In the sum-check protocol, we've got a prover trying to convince a verifier that $\sum_a P(a) = t$, where P is some polynomial and we're summing a over some set of assignments (e.g., $\{0,1\}^k$). At a super high level, we're going to set up our satisfiability check so that it involves a sum of this form — specifically, so that A is a satisfying assignment for φ if and only if a certain sum is 0 — and then we'll run a sum-check protocol to check this.

And we'll set this up so that there's only $O(\log n)$ variables. Then we can imagine building a tree of what the prover needs to respond with on each possible run of the randomness; and if we put that whole prover tree in the PCP, then the verifier can simulate the interactive proof themselves. (So we're sort of taking the interactive proof and giving you (the verifier) the entire prover strategy in one shot.)

§11.4.2 Algebrization

The first idea is *algebrization* — we're going to algebrize the assignment A (which means we think of A as some sort of polynomial), algebrize the formula φ (so we think of φ as a polynomial as well), and convert the claim that A satisfies φ into a *system* of polynomial equations.

First we'll define what it means to algebrize a Boolean function in general.

Definition 11.36. For a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ and a field $\mathbb{F}$, the algebrization of f over $\mathbb{F}$ is the n-variate polynomial $\widetilde{f}(x_1, \dots, x_n)$ over $\mathbb{F}$ such that:

- $\widetilde{f}$ is multilinear (i.e., it has no squared variabes) — in other words, we can write $\widetilde{f}(x_1, \dots, x_n) = \sum_{S\subseteq[n]} c_S \prod_{i\in S} x_i$ for some coefficients $c_S \in \mathbb{F}$.
- For all $a \in \{0,1\}^n$, we have $f(a) = \widetilde{f}(a)$.

(We think of $\mathbb{F}$ as $\mathbb{F}_p$ for some large prime p.)

Note that there's 2^n subsets $S \subseteq [n]$, so the algebrization of f contains 2^n monomials.

Example 11.37

The **AND** function $x_1 \wedge x_2 \wedge \cdots \wedge x_n$ has algebrization $x_1 x_2 \cdots x_n$, and the **OR** function $x_1 \vee x_2 \vee \cdots \vee x_n$ has algebrization $1 - (1 - x_1)(1 - x_2)\cdots(1 - x_n)$.

We've seen examples for the **AND** and **OR** functions, but it turns out that *every* Boolean function can be algebrized.

Theorem 11.38

For all Boolean functions f and all $\mathbb{F}$, there is a unique algebrization $\widetilde{f}$ of f over $\mathbb{F}$.

This is an extremely fundamental result — lots of theorems since the 1990s have been proved using this.

Proof. We can actually prove this using a simple linear algebra argument — given f, we need to choose the coefficients c_S that define $\widetilde{f}$ such that we end up with $\widetilde{f}(a) = f(a)$ for all $a \in \{0,1\}^n$. This defines a system of 2^n linear equations in 2^n variables — where the variables are c_S for each $S \subseteq [n]$, and we get one equation for each $a \in \{0,1\}^n$.

And we can show that these equations are all linearly independent, so since there's the same number of variables as equations, there's always a unique solution. □

Remark 11.39. The fact that we require $\widetilde{f}$ to be multilinear is necessary for this property to hold — otherwise there would be multiple possible polynomials, as $x^2 = x$ for $x \in \{0, 1\}$.

Now we want to algebrize A and our formula φ. First, algebrizing A is fairly simple — we think of φ as being on $n = 2^k$ variables indexed by $\{0, 1\}^k$, so we can view A as a function $A\colon \{0, 1\}^k \to \{0, 1\}$. And now A is a Boolean function, so we can algebrize it.

(We're going to take $\mathbb{F} = \mathbb{F}_p$ for some $p \in [n^4, 2n^4] = [2^{4k}, 2^{4k+1}]$.)

Algebrizing φ is trickier — we need some way of taking our 3CNF φ and viewing it as a function. The idea is to consider the function that takes each possible clause and records whether or not it's in φ. So we consider the function $\varphi\colon \{0, 1\}^{3k} \times \{0, 1\}^3 \to \{0, 1\}$ where the first $3k$ bits correspond to variable indices $i_1, i_2, i_3 \in \{0, 1\}^k$ (representing which variables the clause involves) and the last three bits $b_1, b_2, b_3 \in \{0, 1\}$ correspond to whether or not each variable is negated — so

$$\varphi(i_1, i_2, i_3, b_1, b_2, b_3) = \begin{cases} 1 & \text{if } (x_{i_1} = b_1) \vee (x_{i_2} = b_2) \vee (x_{i_3} = b_3) \text{ is a clause in } \varphi \\ 0 & \text{otherwise} \end{cases}$$

(where we think of as x_i as $(x_i = 1)$ and $\overline{x_i}$ as $(x_i = 0)$). (In a slight abuse of notation, we're going to use φ to refer to both the formula and its encoding as a function.)

Example 11.40

If φ contains the clause $x_{01} \vee \overline{x_{10}} \vee x_{11}$, then $\varphi(01, 10, 11, 1, 0, 1) = 1$.

So far, we've just performed a 'functionization' — we've found a way to encode a 3CNF as a function $\{0, 1\}^{3k+3} \to \{0, 1\}$. But now we've got a Boolean function, so we can algebrize it to get a polynomial $\widetilde{\varphi}$.

So now we've got a function φ that encodes our formula (by encoding, for every possible clause, whether it's in the formula or not), and another function A that encodes the assignment (by encoding, for each variable, whether it's true or false); and we've algebrized them into polynomials $\widetilde{\varphi}$ and $\widetilde{A}$. The next thing we want to do is encode the claim that A satisfies φ (in a way that we'll later be able to algebrize).

Claim 11.41 — An assignment A satisfies φ if and only if for all indices $i_1, i_2, i_3 \in \{0, 1\}^k$ and bits $b_1, b_2, b_3 \in \{0, 1\}$, we have

$$0 = \varphi(i_1, i_2, i_3, b_1, b_2, b_3) \cdot (A(i_1) - b_1)^2 (A(i_2) - b_2)^2 (A(i_3) - b_3)^2.$$

We don't need the squares for this claim to be true; but the reason we have them is so that the right-hand side is always 0 or 1, which will be useful later (in the sum-check protocol).

Proof. The point is that for any $(i_1, i_2, i_3, b_1, b_2, b_3)$, if the corresponding clause isn't in the formula φ, then $\varphi(i_1, i_2, i_3, b_1, b_2, b_3) = 0$ (and this equation is satisfied). Meanwhile, if it *is* in φ, then this term is 1. So this equation holds if and only if $A(i_j) = b_j$ for some $j \in \{1, 2, 3\}$ — and this is exactly what it means for A to satisfy the clause $(x_{i_1} = b_1) \vee (x_{i_2} = b_2) \vee (x_{i_3} = b_3)$. □

So far, we've taken the notion that an assignment A satisfies a formula φ and written it in a weird way in terms of functions (we haven't gotten to polynomials yet). And now we want to algebrize this — we want to convert the condition in Claim 11.41 into a system of polynomial equations. We'll do this by taking each of the equations in Claim 11.41 and sticking in the algebrizations of φ and A — so we define the polynomial

$$P(i_1, i_2, i_3, b_1, b_2, b_3) = \widetilde{\varphi}(i_1, i_2, i_3, b_1, b_2, b_3) \prod_{j=1}^{3} (\widetilde{A}(i_j) - b_j). \tag{12}$$

We think of each of i_1, i_2, and i_3 as consisting of k variables and b_1, b_2, and b_3 as consisting of 1 variable, so P is a polynomial in $m = 3k+3$ variables. And we have $\deg \widetilde{F} \leq 3k+3$ (since it's a multilinear polynomial in $3k+3$ variables) and $\deg \widetilde{A} \leq k$, so $\deg P \leq 9k+3$. Furthermore, the multilinearity of $\widetilde{F}$ and $\widetilde{A}$ means that every variable has degree at most 7 in P (it has degree at most 1 in $\widetilde{F}$, and degree at most 2 in each term $(\widetilde{A}(i_j) - b_j)^2$).

So now in order to check that A satisfies φ, we just need to check

$$P(c) = 0 \quad \text{for all } c \in \{0,1\}^m. \tag{$*$}$$

Note that $m = 3k+3 = O(\log n)$, so there's only $\mathsf{poly}(n)$ equations in $(*)$.

§11.4.3 A sum-check

So far, we've written the statement that A satisfies φ as a *system* of polynomial equations $(*)$. But what we really want is *one* polynomial (specifically, one polynomial which is some sort of sum-check, so that we can prove the equation is satisfied via a sum-check protocol). So the next step is to reduce $(*)$ to just a single equation.

Claim 11.42 — The system $(*)$ is equivalent to the single equation $\sum_{c\in\{0,1\}^m} P(c) = 0$.

The point is that we defined P so that $P(c) \in \{0,1\}$ for all c (specifically, $P(c)$ is 0 if the corresponding clause is satisfied, and 1 otherwise). And we chose p big enough that there's no overflow (there's $2^m = 2^{3k+3} < 2^{4k} \leq p$ terms in the sum — this is why we chose $p \geq n^4$).

And now this is something we can run a sum-check protocol on! We first saw a sum-check protocol in the interactive proof protocol for #SAT (Theorem 10.32); there we took a Boolean formula and turned it into a polynomial P, and we wanted to check a statement that $\sum_{c\in\{0,1\}^n} P(c)$ was equal to a certain value. Abstracting away the details of that argument, we get the following more general statement.

Theorem 11.43 (Sum-check theorem)

Let P be a polynomial in m variables which can be evaluated in $\mathsf{poly}(n)$ time, where the degree of each variable is at most d. Then there is an IP protocol for the statement $\sum_{c\in\{0,1\}^m} P(c) = 0$ (over a field $\mathbb{F}$) with $O(m)$ rounds and messages of length $O(d \log |\mathbb{F}|)$.

In our application, we'll have $m = O(\log n)$, $d = 7$, and $\log |\mathbb{F}| = O(\log n)$.

The idea of the proof is that given the statement $\sum_{c_1,\ldots,c_m\in\{0,1\}} P(c_1, \ldots, c_m) = a$, we peel off the first variable c_1 and replace it with a variable z; and the prover claims that $\sum_{c_2,\ldots,c_m\in\{0,1\}} P(z, c_2, \ldots, c_m)$ is a certain polynomial $Q(z)$. And the verifier does a sanity check that $Q(0) + Q(1) = a$ (i.e., that this does justify the original claim), then chooses a random value r_1 to evaluate Q at, and asks the prover to show that $\sum_{c_2,\ldots,c_m\in\{0,1\}} P(c_2, \ldots, c_m) = Q(r_1)$ — which is a sum-check in one fewer variable. So we're peeling off one variable at a time and substituting it with a random value. In particular, all the messages that the prover sends are polynomials in a single variable of degree at most d (and all the messages the verifier sends are uniform random values of $\mathbb{F}$).

§11.4.4 The correct proof $\mathcal{P}$

Now we've got almost all the ingredients in place to describe the correct proof $\mathcal{P}$ — i.e., the proof that makes $\mathcal{V}$ accept when our formula φ really is satisfiable.

The proof will consist of $n^{O(\log n)}$ bits. First, we'll encode the satisfying assignment A, but we'll do so in an extremely redundant way — we're going to do this by evaluating the polynomial $\widetilde{A}$ at *all* vectors $v \in \mathbb{F}^k$

(note that $\widetilde{A}$ is a polynomial in k variables). There's $k \approx \log n$ variables and each takes on $|\mathbb{F}| \approx n^4$ possible values, so this means we've got $n^{O(\log n)}$ vectors v to plug in; and we write down $\widetilde{A}(v)$ for all such v. This is enormous — not only do we include all the bits of the satisfying assignment A, but we also have all the values its algebrization would give when we plug in other values in $\mathbb{F}$. (Normally A would just take n bits to encode, but we've got an extremely redundant encoding that takes $n^{O(\log n)}$ bits and includes all these other values of $\widetilde{A}$.)

The second thing we want to include in $\mathcal{P}$ is a way to simulate the interactive sum-check protocol. In that protocol, the prover sends a polynomial, the verifier sends randomness, the prover sends another polynomial, and so on. So in $\mathcal{P}$ we'll include a whole strategy tree of all the messages that $\mathcal{P}$ would send over all the randomness. The IP protocol has messages of length $O(\log n)$, so there's $\mathsf{poly}(n)$ choices for the randomness in each round (i.e., each node in the tree has $\mathsf{poly}(n)$ children); and the protocol runs for $O(\log n)$ rounds, so the tree has depth $O(\log n)$. So this tree has size $n^{O(\log n)}$.

§11.4.5 Verifying the proof

Now suppose our verifier wants to check that A satisfies φ (given the extremely redundant encoding of A, together with the IP prover's strategy tree). First, we've written the statement that A satisfies φ as a single polynomial equation (in Claim 11.42). So the verifier can try to run the sum-check protocol for this equation (using the given tree) — they can pick all the randomness $r_1, \ldots, r_m \in \mathbb{F}$ for the sum-check protocol up front, and then simulate the sum-check protocol step by step (so they make a query with this randomness and receive a bunch of polynomials $Q_1, \ldots, Q_m$ from the strategy tree, and they check that $Q_i(0) + Q_i(1) = Q_{i-1}(r_{i-1})$ for each i).

But we've got a problem — in the sum-check protocol, in the end the verifier is supposed to accept if $P(r_1, \ldots, r_m) = t$ (where t is some value determined by the prover's polynomials — specifically, $Q_m(r_m)$ — and $r_1, \ldots, r_m \in \mathbb{F}$ are all the random values we've plugged in). But the definition of P in (12) involves both $\widetilde{\varphi}$ and $\widetilde{A}$. It's fine that it involves $\widetilde{\varphi}$ — the verifier has access to φ and can simply compute $\widetilde{\varphi}$ themselves. But what's *not* fine is that it involves $\widetilde{A}$ — because the verifier doesn't actually know $\widetilde{A}$. (And it can't get the assignment A and construct $\widetilde{A}$ itself, because it's only got $O((\log n)^2)$ queries.)

As a first attempt, we've set up the proof $\mathcal{P}$ to include the values of $\widetilde{A}$ on all inputs, so we could imagine just grabbing the values of $\widetilde{A}(v)$ from $\mathcal{P}$ that we need to evaluate $P(r_1, \ldots, r_m)$. In the case where φ is satisfiable, this will work perfectly fine (i.e., this protocol does satisfy completeness). The problem is soundness — what happens if φ is *not* satisfiable? The analysis of the sum-check protocol shows that $\mathcal{V}$ will reject with high probability *assuming* that it can evaluate $P(r_1, \ldots, r_n)$ correctly in the end. But we don't have the actual polynomial P; we just have the values of $\widetilde{A}$ sitting on the proof tape. And if we were guaranteed that these values really are the outputs of a multilinear polynomial $\widetilde{A}$, then this would work (because then having the values of $\widetilde{A}(v)$ for all v is good enough to evaluate P, and everything works out).

But the issue is that $\mathcal{P}$ could be completely cheating with $\widetilde{A}$ — they could've taken $\widetilde{A}$ to be some arbitrary function (meaning that the values $\widetilde{A}(v)$ aren't consistent with any multilinear polynomial); and then the proof breaks down. To fix this, we'll need to use the following statement as a black box.

Theorem 11.44 (BFL, Rubinfeld–Sudan)

There is a randomized algorithm $\mathcal{C}(v_1, \ldots, v_k)$ with oracle access to a function $B \colon \mathbb{F}^k \to \mathbb{F}$ (with $|\mathbb{F}| \geq k^4$) that makes $O(k^2)$ queries to B and runs in polynomial time, and such that:

- If there is a multilinear polynomial Q of degree at most k such that $B(v) = Q(v)$ for all v, then $\mathcal{C}^B(v_1, \ldots, v_k) = Q(v_1, \ldots, v_k)$ with probability 1.
- If for every multilinear polynomial Q of degree at most k we have $B(v) \neq Q(v)$ for at least a $\frac{1}{100}$-fraction of vectors v, then $\mathcal{C}$ rejects with high probability.

This is essentially a multilinearity test — this means if the given oracle B encodes a multilinear polynomial, then we can recover the value of that polynomial at a given input; but if it's *far* from any polynomial, then we realize this and reject.

And the point is that now we can use $\mathcal{C}$ to obtain the values of $\widetilde{A}$ using $O((\log n)^2)$ queries — and we'll use the values of $\widetilde{A}$ we get from this to evaluate P. This fixes the issue because now if the prover is cheating (by sticking in something that's not a multilinear polynomial for $\widetilde{A}$), then we'll catch them with high probability.

§12 Communication complexity

Today we'll talk about communication complexity. Interestingly, there's a lot of things in communication complexity that are well-understood, unlike many other topics we've seen.

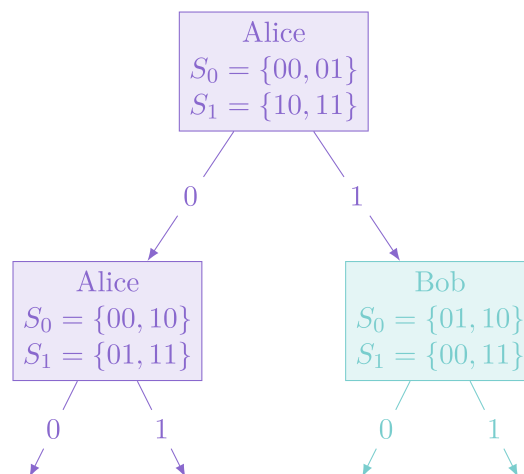
§12.1 Setup and protocol trees

Communication complexity is more of an information-theoretic notion than one about computing. We imagine we have two parties Alice and Bob, both of whom have some input — Alice has some input $x \in \{0,1\}^n$ and Bob has some input $y \in \{0,1\}^n$. And together, they want to compute some function $F: \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ that depends on both x and y. They do this by communicating bits — Alice sends some $\{0,1\}$-bit about her input, and then Bob sends some bit about his input, and so on. (We don't require them to alternate — maybe Alice speaks multiple times in a row.)

Here Alice and Bob are computationally *unbounded* — both of them are all-powerful. So Alice can compute whatever she wants with x; the difficulty of this problem comes from the fact that she doesn't know y.

First, we need to define a model of computation that captures this, and we'll do this using *protocol trees*. The computation starts at the root of the tree, and at each node we specify who speaks (i.e., Alice or Bob), and two sets S_0 and S_1 (which partition $\{0,1\}^n$) such that the speaker sends b when their input is in S_b.

So for example, maybe our starting node has Alice speaking and $S_b = \{x \mid x \text{ begins with } b\}$ — this means Alice sends Bob the first bit of her input. And based on what that bit is, we go down to some new node, which also specifies who's speaking and what they send (based on their input). And so on.



And finally, in the end we'll have some leaves, each of which contains a function $g: \{0,1\}^n \to \{0,1\}$ of only x (if Alice is speaking) or only y (if Bob is speaking). And the output of this function will be the output of our protocol. (Intuitively, we imagine that after some point, one of the parties has enough information to compute $F(x,y)$ just using their input alone; and then they do this and announce the answer for free.)

Definition 12.1. Given a protocol tree $\mathcal{P}$, we define $\mathcal{P}(x,y)$ as the value of the function g at the leaf when Alice holds x and Bob holds y. We say $\mathcal{P}$ computes f if $\mathcal{P}(x,y) = f(x,y)$ for all $x, y \in \{0,1\}^n$.

Our goal in communication complexity is to minimize the number of bits that Alice and Bob have to communicate; this number corresponds to the depth of the tree.

Definition 12.2. The communication complexity of a protocol $\mathcal{P}$, denoted $\mathsf{cc}(\mathcal{P})$, is the depth of $\mathcal{P}$ as a tree — equivalently, the maximum number of bits that Alice and Bob (in total) might need to communicate over all $x, y \in \{0,1\}^n$.

Definition 12.3. We define the communication complexity of a function $f\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, denoted $\mathsf{cc}(f)$, as the minimum value of $\mathsf{cc}(\mathcal{P})$ over all protocols $\mathcal{P}$ that compute f.

So we're measuring the worst-case complexity of the best possible protocol for our function (this is very similar to the usual way we measure complexity).

§12.2 Some examples

Here are some examples of a few functions and their communication protocols.

Definition 12.4

We define $\mathsf{PAR}_n(x, y) = \sum_{i=1}^n (x_i + y_i) \pmod 2$.

In other words, PAR_n (the name stands for *parity*) computes whether the total number of 1's in x and y is odd (where $x, y \in \{0,1\}^n$).

Claim 12.5 — We have $\mathsf{cc}(\mathsf{PAR}_n) = 1$.

Proof. The idea is that $\mathsf{PAR}_n(x, y) = \mathsf{PAR}(x) \oplus \mathsf{PAR}(y)$ (where $\mathsf{PAR}(x) = \sum_i x_i \pmod 2$). So Alice simply sends Bob $\mathsf{PAR}(x)$, and then Bob computes $\mathsf{PAR}(y)$ and **XOR**s the two together.

(This shows $\mathsf{cc}(\mathsf{PAR}_n) \leq 1$; but of course it's also at least 1, since neither party knows the answer without knowing anything about the other person's input.) □

Definition 12.6

We define $\mathsf{MAJ}_n(x, y)$ to be 1 if and only if the total number of 1's in x and y is greater than n.

In other words, MAJ_n (the name stands for *majority*) computes the majority bit in the concatenation of x and y (for $x, y \in \{0,1\}^n$).

Claim 12.7 — We have $\mathsf{cc}(\mathsf{MAJ}_n) = O(\log n)$.

Proof. Alice can simply count up the number of 1's in x and send it to Bob as a binary number; this takes $O(\log n)$ bits (we can imagine a protocol tree where Alice sends these bits one at a time). And then Bob can count up the number of 1's in y and check whether the sum of the two numbers is more than n. □

We can ask whether this is optimal; we'll come back to this later.

Definition 12.8

We define $\mathsf{EQ}_n(x, y) = 1_{x=y}$.

In other words, EQ_n (the name stands for *equality*) checks whether the two inputs x and y are equal.

Of course, we have $\mathsf{cc}(\mathsf{EQ}_n) \leq n$ — there's a trivial algorithm showing $\mathsf{cc}(f) \leq n$ for *any* f, where Alice just sends her entire input to Bob (and then Bob can compute f and announce the answer for free).

Question 12.9. Can we do any better — i.e., is $\mathsf{cc}(\mathsf{EQ}_n) < n$?

It turns out that the answer is no!

Theorem 12.10

We have $\mathsf{cc}(\mathsf{EQ}_n) = n$.

There are lots of functions which have constant communication complexity. In particular, any function that can be decided by a finite automaton (with input xy) has constant communication complexity — this is because we can imagine Alice runs the finite automaton up to x, sends the final state to Bob (there's a constant number of states, so this takes a constant number of bits), and Bob can take over running the finite automaton on y.

But on the other hand, it turns out that there are functions like the equality function which are simple to describe but very hard — they take the maximum number of bits a protocol could possibly need.

This is quite striking — you'd generally expect that the really silly algorithm isn't the best you can do, but for EQ_n it really is (there's no shortcut, and we really do have $\mathsf{cc}(\mathsf{EQ}_n) = n$).

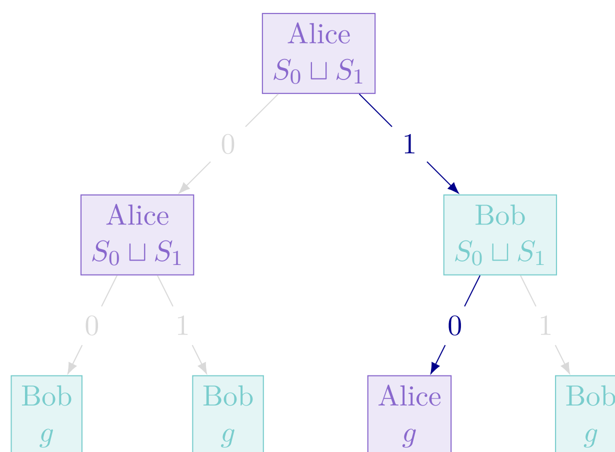
We'll now see two methods of proving this, because both are instructive — the first method is more intuitive, and the second leads to the biggest open problem in communication complexity.

§12.3 The fooling set method

The idea behind the first method (of proving Theorem 12.10) is that we'll assume there's a communication protocol using less than n bits. Then we'll be able to represent strings using short transcripts; there's 2^n strings and not too many possible transcripts, and we'll somehow use this to get a contradiction.

First, we need to formalize the notion of a transcript.

Definition 12.11. We say the transcript of $\mathcal{P}$ on (x, y) is the sequence of bits that we see along the path in the protocol tree that (x, y) corresponds to.



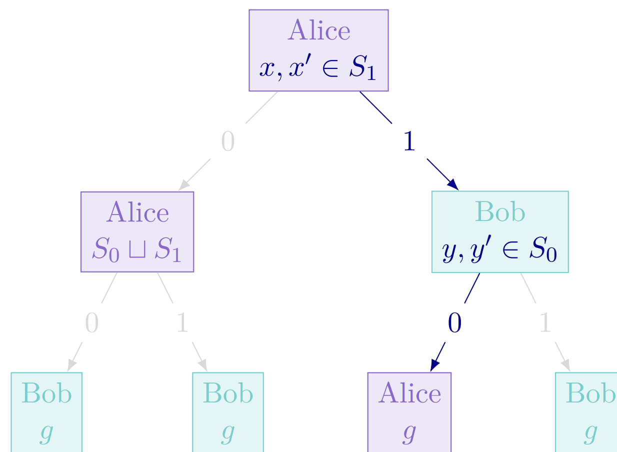
Equivalently, the transcript is the sequence of bits communicated. For example, in the above protocol, if (x, y) corresponds to the path in blue, then its transcript is 10.

Proposition 12.12

If (x, y) and (x', y') have the same transcript T, then (x, y') and (x', y) also have the same transcript T.

Proof. If (x, y) and (x', y') have the same transcript, this means they correspond to the same path down the protocol tree. And this means at each (intermediate) Alice-node on this path, x and x' are in the same subset (of the partition $S_0 \sqcup S_1$) — because Alice sends the same bit from this node whether she reads x (in the protocol on (x, y)) or x' (in the protocol on (x', y')). (The point is that once we're at this node, Alice decides what bit to send looking at only her own input; and she does the same thing in the two protocols, which means she does the same thing on x as on x'.)

And similarly, for every intermediate Bob-node on this path, y and y' are in the same subset (meaning that Bob sends the same bit on y as y').



And this means if we swap out just x' for x or y' for y (i.e., we consider (x', y) or (x, y')), the protocol will still run in the same way — we'll be at some node in the path corresponding to T; and if this is an Alice-node she'll send the same bit regardless of whether she's got x or x', while if it's a Bob-node he'll send the same bit regardless of whether he's got y or y'; so we'll remain on T. □

And already from this, we can prove that the equality function has communication complexity n.

Theorem 12.10

We have $\mathsf{cc}(\mathsf{EQ}_n) \geq n$.

Proof. This is where the idea of fooling sets comes in — we're going to come up with a big set of inputs where the equality function is always 1, but where if we switch around any of these inputs, then it's *not* 1. Specifically, we'll define

$$S = \{(x, x) \mid x \in \{0, 1\}^n\},$$

so that $|S| = 2^n$. Then $\mathsf{EQ}_n(x, x) = 1$ for all x, so EQ_n is 1 on S; but $\mathsf{EQ}_n(x, y) = 0$ for all $x \neq y$, so if we take two elements $(x, x), (y, y) \in S$ and switch between them, then EQ_n is no longer 1. And the idea is that then we must have a separate leaf for each element of S — because if two pairs $(x, x), (y, y) \in S$ corresponded to the same leaf, then they'd have the same transcript, and so if the protocol outputs 1 on (x, x) and (y, y) it'd also output 1 on (x, y).

To make this more precise, assume $\mathsf{cc}(\mathsf{EQ}_n) < n$. Then the number of possible transcripts in the protocol is at most $2^0 + \cdots + 2^{n-1} = 2^n - 1 < 2^n$ (the reason we have this sum is that we could potentially stop after i

steps for any $i \le n-1$), which means there must be distinct $(x,x),(y,y) \in S$ with the same transcript (by the pigeonhole principle).

But then by Proposition 12.12, this means (x,y) has the same transcript as well. And since $\mathsf{EQ}_n(x,x) = \mathsf{EQ}_n(y,y) = 1$, the corresponding leaf must output 1 on both (x,x) and (y,y), so it'll output 1 on (x,y) as well. (If the leaf corresponds to Alice speaking, then the output is a function g of just the first input, and from the protocol's behavior on (x,x) and (y,y) we know $g(x) = g(y) = 1$.)

And $\mathsf{EQ}_n(x,y) \neq 1$, so this contradicts the correctness of the protocol. □

The fooling set method can also be used to show that $\mathsf{cc}(\mathsf{MAJ}_n) = \Omega(\log n)$ (we take our fooling set S to consist of pairs (x,y) based on the number of 1's in x and y, such that any $(x,y) \in S$ has at least n 1's, but when we swap things out this is no longer true).

§12.4 A linear algebraic perspective

Now we'll see another method of proving lower bounds on communication complexity using linear algebra, which is instructive in a totally different way.

First, we're given a function $f\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. We can define a matrix corresponding to this function, where the rows are indexed by x (i.e., Alice's input) and the columns are indexed by y (i.e., Bob's input), and each entry of the matrix represents the output of f on the corresponding inputs.

Definition 12.13. For $f\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, we define the communication matrix of f, denoted M_f, as the $2^n \times 2^n$ matrix such that $M_f(x,y) = f(x,y)$ for all $x, y \in \{0,1\}^n$.

Example 12.14

The matrix corresponding to EQ_n is the $2^n \times 2^n$ identity matrix.

The identity matrix has lots of nice properties, but what's important for our purposes is that it has full rank — if we want to express it as a linear combination of rank-1 matrices, then we need at least 2^n of them. And it turns out that this property is already enough to ensure that EQ_n has communication complexity at least n!

§12.4.1 Combinatorial rectangles

To prove this, we'll need to introduce a new concept. The idea is that we're going to look at submatrices of M_f which correspond to sub-protocols — we imagine that we've reached a certain node in the protocol tree. Then there's some subset of x-values and y-values that led to this node, and we want to consider the corresponding submatrix of M_f.

For this, we'll define the notion of *combinatorial rectangles.*

Definition 12.15. A combinatorial rectangle of a matrix M is a submatrix M' given by a row subset $A \subseteq \{0,1\}^n$ and a column subset $B \subseteq \{0,1\}^n$ (where M' is the $|A| \times |B|$ matrix consisting of the entries indexed by $x \in A$ and $y \in B$). We denote this rectangle by $A \times B$.

So A is picking out a subset of Alice-inputs (i.e., rows) and B a subset of Bob-inputs (i.e., columns), and we're just looking at the matrix restricted to those subsets.

Example 12.16

For a 4×4 matrix, if $A = \{1, 3\}$ and $B = \{2, 4\}$, then the combinatorial rectangle $A \times B$ is the one given by the $\times$'s below.

$$\begin{bmatrix} \cdot & \times & \cdot & \times \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \times & \cdot & \times \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

We want to show that if $\mathsf{cc}(f)$ is small, then M_f is 'simple' in some way — specifically, we'll show that it can be expressed in terms of simple combinatorial rectangles.

Lemma 12.17

Every communication matrix M_f can be written as a sum of at most $2^{\mathsf{cc}(f)}$ disjoint all-1's combinatorial rectangles.

This looks like a mouthful, but it actually follows pretty directly from tracing what happens as we go down the protocol and what submatrices we get.

Proof. Let $\mathcal{P}$ be a protocol for f with depth $\mathsf{cc}(f)$. Then for all nodes v in the tree, we define $S(v)$ as the set of all pairs (x, y) such that when we run $\mathcal{P}$ on (x, y), the protocol reaches the node v (possibly going on to other nodes later). For example, if v is the root of the tree then $S(v)$ consists of all possible pairs (x, y) (since we always pass through the root).

Claim 12.18 — For all v, the set $S(v)$ is a combinatorial rectangle.

Proof. This is actually quite similar to the proof of Proposition 12.12 (in fact, Proposition 12.12 can be viewed as this statement for leaves).

Explicitly, we can prove this by induction — it's true at the root (where we can take $A = B = \{0, 1\}^n$ to consist of all possible inputs), and we'll show that if it's true for some v, then it's also true for its children.

Consider some node v, and suppose that $S(v)$ is some combinatorial rectangle $A \times B$. Assume without loss of generality that v is an Alice-node (i.e., a node where Alice speaks).

Then we can partition $A = A_0 \sqcup A_1$ where A_0 consists of the inputs $x \in A$ at which Alice says 0 at this node, and A_1 consists of the inputs $x \in A$ at which Alice says 1. And then if we let v_0 and v_1 be the corresponding children of v, we'll have $S(v_0) = A_0 \times B$ and $S(v_1) = A_1 \times B$. □

Finally, the combinatorial rectangles corresponding to the leaves partition $\{0, 1\}^n \times \{0, 1\}^n$ (because for every input (x, y), we end up reaching exactly one leaf), and there's at most $2^{\mathsf{cc}(f)}$ leaves (because the depth of the tree is $\mathsf{cc}(f)$). And for each leaf v, we can partition $S(v) = S_0(v) \sqcup S_1(v)$ where $S_b(v)$ is the set of inputs on which we reach v and output b; then $S_0(v)$ and $S_1(v)$ are both combinatorial rectangles for each v (for the same reason), and $S_0(v)$ corresponds to an all-0's matrix and $S_1(v)$ to an all-1's matrix.

So in the end, we can write M_f as the sum of the combinatorial rectangles $S_1(v)$, and we're done. □

§12.4.2 Rank vs. communication complexity

Finally, we're ready to relate the rank of M_f to the communication complexity of f.

Lemma 12.19

For all f, we have $\mathsf{cc}(f) \geq \log_2(\operatorname{rank} M_f)$.

Proof. One definition of $\operatorname{rank}(M_f)$ is the minimum number m such that we can write M_f as a sum of m rank-1 matrices. But by Lemma 12.17, we can write M_f as a sum of $2^{\mathsf{cc}(f)}$ all-1's combinatorial rectangles. And any such matrix has rank 1 — we can write the all-1's matrix on $A \times B$ as $1_B 1_A^\intercal$, where 1_A and 1_B are the indicator vectors of the sets A and B. So we get $\operatorname{rank} M_f \leq 2^{\mathsf{cc}(f)}$, as desired. □

And this immediately implies Theorem 12.10 (that $\mathsf{cc}(\mathsf{EQ}_n) \geq n$), as its communication matrix has rank 2^n.

More generally, this shows that when the rank of our communication matrix is large, the corresponding communication complexity is also large.

Question 12.20. Is the converse true — does high communication complexity imply high rank?

This is a major open problem in communication complexity.

Conjecture 12.21 (Log-rank conjecture) — For all f, we have $\mathsf{cc}(f) \leq \mathsf{polylog}(\operatorname{rank}(M_f))$.

This problem has been studied a lot. It's known there exist functions with $\mathsf{cc}(f) \geq (\log(\operatorname{rank} M_f))^4$, so if this is true, then the polynomial has to be at least quartic. Meanwhile, the best upper bound we have is $\mathsf{cc}(f) \leq \sqrt{\operatorname{rank} M_f} \log(\operatorname{rank} M_f)$, which is nontrivial but quite far from the conjectured bound.

§12.5 Randomized communication protocol

We've seen that EQ_n has communication complexity n (i.e., the best we can do is have one party send over their entire input). But in real life, this isn't how we solve equality — if you've downloaded 1000 terabytes of something and want to check that it's equal to something else, you don't go through all the bits and check that they match up. Instead, you *hash* them and check that their hashes match up; and if you do, then you're usually fine.

This motivates us to consider *randomized* protocols. Here we've got inputs x and y, and Alice and Bob get to toss some private coins; and Alice's output at each node is a function of both x and her private randomness, and likewise for Bob.

Definition 12.22. We say a randomized protocol $\mathcal{P}$ computes f if $\mathbb{P}[\mathcal{P}(x,y) = f(x,y)] \geq \frac{2}{3}$ (where the probability is over the randomness of the protocol).

And the bottom line is that randomness lets us compute EQ_n with only $O(\log n)$ bits of communication — deterministically we need n bits, so this is a setting where randomness provably does *exponentially* better.

Theorem 12.23

There is a randomized communication protocol for EQ_n with communication complexity $O(\log n)$.

Proof sketch. Let $p \in [n^4, 2n^4]$ be a prime, and let $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_n$. First, we'll have Alice and Bob make polynomials out of their input — so Alice makes a polynomial P_A such that $P_A(i) = x_i$ for each $i \in [n]$, and Bob makes a polynomial P_B such that $P_B(i) = y_i$ for each $i \in [n]$. By polynomial interpolation, they can construct such polynomials with degree at most $n - 1$ (since there's n points to interpolate on).

And now Alice chooses some random $z \in \{0, 1, \ldots, p-1\}$ and sends Bob z and $P_A(z)$, and Bob computes $P_B(z)$ and outputs 1 if and only if $P_B(z) = P_A(z)$.

This takes $O(\log n)$ bits of communication (as both z and $P_A(z)$ are elements of $\mathbb{F}_p$ with $p = \mathsf{poly}(n)$). And it works essentially because of polynomial identity testing (as in Theorem 7.10) — if $x = y$ then the two polynomials are the same, so Bob will always output 1. Meanwhile, if $x \neq y$, then P_A and P_B are different; and since we're evaluating them at a point z chosen from a field much larger than their degrees, with high probability we'll have $P_A(z) \neq P_B(z)$. □

Remark 12.24. Maybe this should make us pause — we've previously mentioned that we believe $\mathsf{P} = \mathsf{BPP}$, but this is a situation where randomness provably makes an enormous difference. So what's the difference between the two settings?

The difference is that communication complexity doesn't care about computation — it's based on information poverty. If all the input is given to us (as in P vs. BPP), we don't believe that randomness should make a difference; but when some of the input is hidden, it can.

§13 Restricted circuit complexity

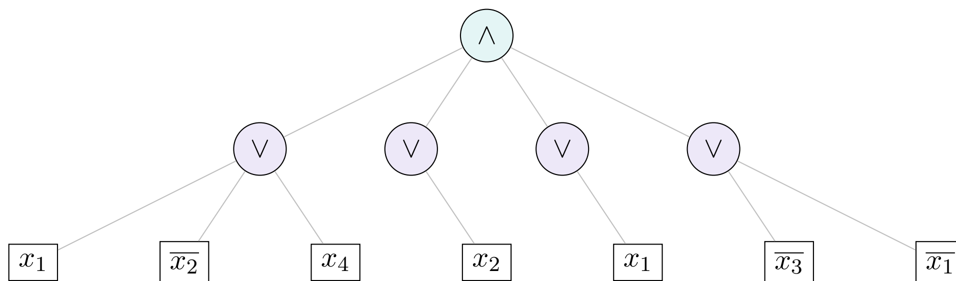
In the final lecture, we'll consider some circuit classes that are more restricted than P/poly (which we briefly mentioned in the first week of class), and we'll see a tour of some loewr bounds that people have been able to prove for such casses.

§13.1 The class AC^0

One of the most common restricted circuit classes that people study is AC^0 (AC stands for *alternating circuits*, and 0 is sort of a parameter that refers to circuit depth).

> **Definition 13.1.** We define AC^0 as the class of circuits of *bounded depth* with **AND** and **OR** gates of unbounded fan-in (and **NOT** gates at the input).

For example, with depth 2 we can have either a huge **OR** of **AND**s (i.e., a DNF) or a huge **AND** of **OR**s (i.e., a CNF). (The circuit could have nodes in the bottom layer that feed into multiple clauses, but we can just duplicate them — this only blows up the circuit size by a constant factor. It also doesn't have to alternate between **AND** and **OR**, but we might as well assume it does — for example, an **AND** of **AND**s can be turned into one **AND**).



This model can do a decent amount of things — for example, you can compute minima or add numbers (this is not obvious). But it's been shown that there's lots of things it *can't* do.

> **Theorem 13.2** (FSS 1981)
> The problem PARITY has no polynomial-size AC^0 circuits.

When we think of a AC^0 circuit family, this means we've got a circuit family $\{C_n\}$ with one circuit for each input length n, but there's some *fixed* constant depth independent of n (i.e., there's a constant d such that $\mathsf{depth}(C_n) \leq d$ for all n).

This result was sharpened by Håstad in his PhD thesis.

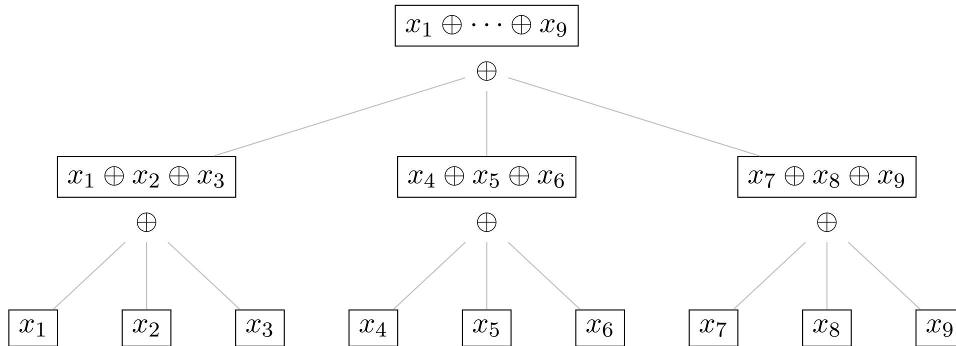
> **Theorem 13.3** (Håstad 1986)
> For all d, there is $\varepsilon > 0$ such that PARITY has no depth-d AC^0 circuits of size $2^{\varepsilon n^{1/(d-1)}}$.

There's a tradeoff between depth and size here — for example, for $d = 2$ we get the bound $2^{\Omega(n)}$, while for $d = 3$ we get $2^{\Omega(\sqrt{n})}$ (and more generally, the bound for d has an exponent of $n^{1/d-1}$).

And it can be shown that these bounds are optimal, up to the value of ε.

Claim 13.4 — For all d, there is a depth-d AC^0 circuit for PARITY of size $2^{O(n^{1/d-1})}$.

Proof sketch. The idea is that we want to compute $\sum x_i \pmod 2$, and we can split this sum into several smaller sums (i.e., we can write the parity function as a parity of parities). So we can first imagine making a depth-$(d-1)$ circuit that computes PARITY, but where we allow parity gates instead of just **OR**s and **AND**s — we make a tree of depth $d-1$ where each node has $n^{1/(d-1)}$ children, so that the number of leaves at the bottom will be n; and every one of these nodes computes the sum of its children mod 2.

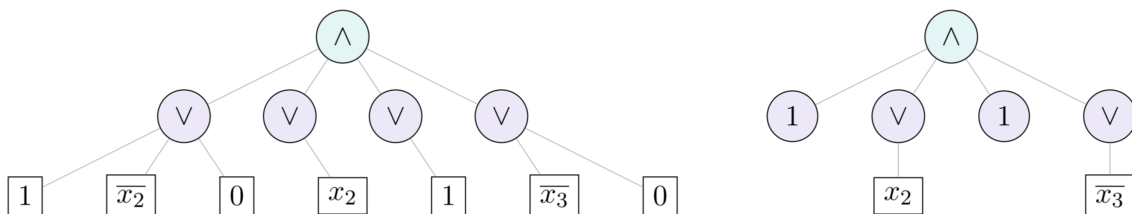


We need to turn this into an AC^0 circuit (i.e., one only involving **AND**s and **OR**s), and the idea beind how we do this is that we replace each parity gate with either a **AND** of **OR**s (i.e., a CNF) or a **OR** of **AND**s (i.e., a DNF). More precisely, *any* function on k variables can be written as a CNF or a DNF of size $2^{O(k)}$ in a pretty naive way (for example, we can imagine taking a **OR** over all inputs on which the function is 1, and then taking each clause to be an **AND** asserting that x is that input). So we replace each of the parity gates with such a CNF or DNF; and we alternate, so that the top layer is a **AND** of **OR**s, every parity gate in the next layer is a **OR** of **AND**s; and so on. And then we can collapse the consecutive layers of **OR**s into a single layer (and similarly with **AND**s), so we'll really only end up with d layers.

And each parity gate had $n^{1/(d-1)}$ inputs, so when we expand it into a CNF or DNF in the naive way, we'll end up with a circuit of size $2^{O(n^{1/(d-1)})}$. $\square$

§13.1.1 Random restrictions and Theorem 13.2

We'll now briefly discuss the proof of Theorem 13.2. The method pioneered by the authors is the idea of *random restrictions.* The idea is that we imagine we've got some low-depth circuit, and we pick some inputs and set them to values in $\{0,1\}$ (chosen uniformly at random). And then we try to simplify the circuit and see what happens — for example, if we had an **OR** gate and we've set any one of its inputs to 1, then we can get rid of the **OR** gate and replace it with a 1.



We'll imagine performing a random restriction where we pick one input x_i to keep, and we restrict all the others — so we randomly set the remaining $n-1$ variables to 0 or 1.

What happens if we perform this experiment on the function PARITY? If we set the values of all the other variables and leave x_i as it is, then PARITY is going to simplify to either $f(x) = x_i$ or $f(x) = \overline{x_i}$ (depending on whether the parity of the remaining bits is 0 or 1).

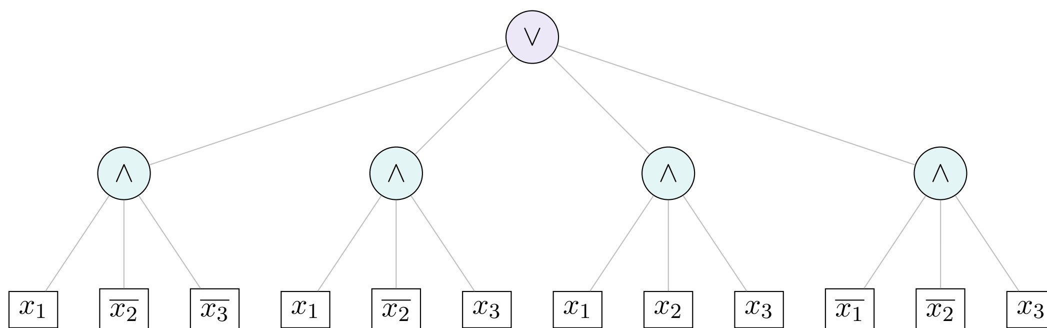
On the other hand, if we start with a small AC^0 circuit, it can be shown that with high probability (over the random restriction), the function we're left with is either the all-0's or the all-1's function.

And from these two considerations, there can't be any small AC^0 circuit that computes PARITY — because when we plug in these random values for all variables except x_i, we're left with a constant circuit, but PARITY is *not* constant (the output of the circuit won't change if we flip the value of x_i, but PARITY will).

§13.1.2 Theorem 13.3 for $d = 2$

We'll now discuss a few more details on how Håstad's proof of Theorem 13.3 works; we'll first prove it in the case $d = 2$. Suppose we have a DNF for PARITY.

Claim 13.5 — Every **AND** gate of the DNF (in the second layer) has to contain *all* n variables (possibly with negations).



Proof. Assume there's some gate g that doesn't contain some variable x_i. Then we can find two inputs x and x', differing in the ith bit and nowhere else, that both make g output 1 (we set all the literals in the clause to be true, which makes the clause true; this doesn't force the assignment of the ith bit because x_i doesn't appear in the clause, so we can set x_i to either value). And since the entire circuit is an **OR** of all these **AND** gates, this means the circuit outputs 1 on both x and x'.

But x and x' differ in the ith bit and nowhere else, so they have different parities — and that means the circuit is wrong on one of them. □

Claim 13.6 — The DNF must have at least 2^{n-1} **AND** gates.

Proof. The point is that by Claim 13.5, every single **AND** gate contains all n variables, which means it only outputs 1 on one assignment to $x \in \{0,1\}^n$. On the other hand, there are 2^{n-1} assignments to x such that $\mathsf{PARITY}(x) = 1$; and we need a separate **AND** gate for each one of these (because each **AND** gate can only handle one assignment). □

So this proves Theorem 13.3 for $d = 2$ — we've shown that any DNF computing PARITY has size at least 2^{n-1}, and the same must be true of any CNF.

§13.1.3 A sketch for general d

We'll now briefly discuss the ideas that go into proving Theorem 13.3 in the general case.

Suppose that PARITY has a size-s depth-d circuit C. At a very high level, the idea is that we'll perform a random restriction where we randomly set all but $n/(\log s)$ of the inputs to $\{0, 1\}$ — so we pick a random subset of $n/(\log s)$ of the variables and leave them unset, and for all the remaining variables, we pick a uniform assignment in $\{0, 1\}$. And then we simplify the circuit, as before (if we've got a **OR** gate with even a single 1 coming in, then we can replace it with a 1; similarly, if we've got a **AND** gate with a single 0, we can replace it with a 0).

And we can show that with high probability, C is now equivalent to some circuit C' of depth $d-1$. So we've taken a circuit with n inputs and turned it into one $n/(\log s)$ inputs and one smaller depth.

And then we can keep doing this — so we do this $d-2$ times, and we eventually get a CNF or DNF in k variables for some k; and if s is too small, then we'll get a CNF or DNF of size less than 2^{k-1}.

But our original circuit was supposed to be computing PARITY, so even if we plug in a bunch of variables, it's still computing either PARITY or its negation on the remaining k variables. But by the $d = 2$ case of Theorem 13.3, this isn't possible — our circuit has size less than 2^{k-1}, so it's too small to compute PARITY on k variables.

Remark 13.7. An interesting aspect of this proof is that it's actually algorithmic in a sense — everything we're doing here (setting some variables randomly and then simplifying) is actually some algorithm we're doing to a circuit. And you can actually use things like this to get interesting satisfiability algorithms for AC^0 circuits, by sort of randomly setting things and seeing what happens.

§13.2 AC^0 with parity gates

We've seen that AC^0 can't compute the parity function. So the next question we'll consider is, what happens if we throw in parity gates for free?

Question 13.8. What can we do with a circuit that has **OR**, **AND**, and parity gates?

More generally, we'll consider what happens if we allow mod-p gates for any prime p.

Definition 13.9. We define $\mathsf{AC}^0[p]$ as the class of circuits as in AC^0 where we also allow mod_p gates of unbounded fan-in, where

$$\mathsf{mod}_p(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } x_1 + \cdots + x_n \equiv 0 \pmod p \\ 0 & \text{otherwise.} \end{cases}$$

Of course, now $\mathsf{AC}^0[2]$ can compute the parity function; but there are still reasonably simple functions we can show it *can't* compute.

Theorem 13.10 (R 1987)

The function MAJ does not have polynomial-size $\mathsf{AC}^0[p]$ circuits for any prime p.

(Here MAJ is the *majority* function, which computes the majority bit of its input.)

Intuitively, to compute MAJ it seems like you need to count up to $\frac{n}{2}$, and this theorem says we can't do this in constant depth with **AND**s, **OR**s, and counting mod some fixed prime p. But strangely, this problem remains open if we replace p with a squarefree composite.

Open question 13.11. Does MAJ have $\mathsf{AC}^0[6]$ circuits of polynomial size?

§13.2.1 Approximating circuits with polynomials

To prove Theorem 13.10, we can't just randomly set inputs anymore — if we've got a parity gate and we randomly set some of its inputs, it's going to always simplify to a parity function of the remaining inputs, not a constant. So the random restriction methods we used to prove Theorem 13.2 and 13.3 (where we randomly set inputs and showed the circuit simplified) aren't going to work.

The proof instead uses very different ideas, which *also* turn out to be algorithmically useful.

The idea is to *approximate circuits with low-degree polynomials.* Given a circuit, we can't necessarily compute it *exactly* with a low-degree polynomial. But we can relax the notion of computing the circuit in two ways — maybe we'll only ask the polynomial to *approximate* the circuit, i.e., to get the same value as the circuit on *most* inputs. Or we can imagine we have a *distribution* of polynomials, rather than just one — and we compute on an input by choosing a polynomial from this distribution. (We call such a distribution a *probabilistic polynomial.*)

And this relaxation allows us to do much more interesting things. If we wanted a single polynomial computing an AC^0 circuit exactly, we couldn't guarantee low degree — for example, $\mathsf{AND}(x_1, \ldots, x_n) = \prod x_i$ has very high degree. But if we allow probabilistic polynomials, then we can make our degree much smaller.

§13.2.2 Two key theorems

The proof of Theorem 13.10 involves two main theorems. The first formalizes what it means to approximate a circuit by a low-degree polynomial, and states that it's always possible.

Theorem 13.12

Let $k \geq 1$, and let C be an $\mathsf{AC}^0[p]$ circuit (with n inputs) of size s and depth d. Then there exists a polynomial $Q(x_1, \ldots, x_n)$ over $\mathbb{F}_p$ of degree at most d such that

$$\mathbb{P}_{x \in \{0,1\}^n}[C(x) \neq Q(x)] \leq \frac{s}{2^k}.$$

We think of s as $\mathsf{poly}(n)$ and d as constant. We also think of p as constant. So if we take $k \approx \log s$, then the degree of Q is at most $(k(p-1))^d = \mathsf{polylog}(n)$. This is much smaller than the degree we'd need to *exactly* compute the **AND** function (which is n).

And this theorem is algorithmically useful — it gives a way of taking circuits and approximately computing them with polynomials, which can actually be used to design algorithms.

(We'll discuss a bit more about the proof soon.)

The second theorem shows there's a limitation on how well low-degree polynomials can approximately compute MAJ.

Theorem 13.13

There exists $\delta > 0$ such that for all $Q(x_1, \ldots, x_n)$ of degree at most $\sqrt{n}$ over $\mathbb{F}_p$ (for any prime p),

$$\mathbb{P}_{x \in \{0,1\}^n}[\mathsf{MAJ}(x) \neq Q(x)] \geq \delta.$$

In words, this states that for every polynomial of degree at most $\sqrt{n}$ over $\mathbb{F}_p$, the polynomial has to disagree with the majority function on some constant fraction of inputs (in fact, $\delta = \frac{1}{50}$ suffices).

The proof comes from some simulation theorem for majority — it turns out that if you generalize polynomials to allow the majority function in your basis, then you can represent *any* Boolean function with a 'majority polynomial' of degree at most $\frac{n}{2} + \sqrt{n}$.

Using Theorems 13.12 and 13.13, we can prove Theorem 13.10 — in fact, we can even prove an *exponential* lower bound on the circuit size needed to compute MAJ.

Proof of Theorem 13.10. Theorem 13.12 gives us a positive statement about approximating $\mathsf{AC}^0[p]$ circuits using low-degree polynomials, and Theorem 13.13 gives a negative statement about approximating MAJ, so we'll combine them — we'll set k so that $(k(p-1))^d \leq \sqrt{n}$. Then Theorem 13.12 says that every $\mathsf{AC}_0[p]$ circuit of size s can be approximated by a polynomial of degree at most $\sqrt{n}$ (where how good the approximation is depends on s), while Theorem 13.13 gives an upper bound on how well such a polynomial can approximate MAJ. So combining these, if we have a size-s depth-d circuit C computing MAJ, then we get that there's a polynomial Q of degree at most $\sqrt{n}$ such that

$$\delta \leq \mathbb{P}_x[\mathsf{MAJ}(x) \neq Q(x)] \leq \frac{s}{2^k}.$$

And this gives a lower bound on s — we get

$$s \geq \delta \cdot 2^k \approx \delta \cdot 2^{n^{1/2d}/(p-1)}.$$

So we've shown that MAJ needs *exponentially* large $\mathsf{AC}^0[p]$ circuits. □

§13.2.3 Theorem 13.12 and probabilistic polynomials

We'll now talk about Theorem 13.12. This theorem actually follows from something much stronger, which is super useful algorithmically — the concept of *probabilistic polynomials.*

Definition 13.14. For a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, a probabilistic polynomial for f of degree d and error ε is a distribution $\mathcal{D}$ over polynomials of degree at most d over $\mathbb{F}_p$ (for some prime p) such that for every $x \in \{0,1\}^n$, we have

$$\mathbb{P}_{Q\sim\mathcal{D}}[f(x) = Q(x)] \geq 1 - \varepsilon.$$

So if we want to evaluate f using our probabilistic polynomial $\mathcal{D}$, we draw a random Q from $\mathcal{D}$ and see what output it gives. The idea is that this acts as a randomized algorithm for computing f where the computation model is a polynomial.

There's often a tradeoff between d and ε, and in some ways there *has* to be — for example, if we consider the **AND** function, we know any polynomial that computes **AND** correctly on *all* inputs needs to have degree at least n. So if we set $\varepsilon < \frac{1}{2^n}$ (which by the union bound means there must be some polynomial that's correct on all inputs), then we need to have $d \geq n$.

And in order to get Theorem 13.12, we actually prove the following stronger statement about probabilistic polynomials.

Theorem 13.15
Any size-s depth-d $\mathsf{AC}^0[p]$ circuit has a probabilistic polynomial of degree $(k(p-1))^d$ and error $\frac{s}{2^k}$.

So if we're trying to compute some function and it happens to have an $\mathsf{AC}^0[p]$ circuit, then there's a way to represent the function algebraically with polynomials of low degree; this turns out to be super useful in the design of certain algorithms.

And once we've got Theorem 13.15, then Theorem 13.12 follows from an averaging argument — if we've got a distribution $\mathcal{D}$ over low-degree polynomials with $\mathbb{P}_{Q\sim\mathcal{D}}[f(x) = Q(x)] \geq 1-\varepsilon$ for all x, then this in particular means $\mathbb{P}_{Q\sim\mathcal{D}}\mathbb{P}_x[f(x) = Q(x)] \geq 1-\varepsilon$, and so by an averaging argument we can find some Q such that $\mathbb{P}_x[f(x) = Q(x)] \geq 1-\varepsilon$.

Finally, we'll briefly talk about some intuition for why Theorem 13.15 is true, i.e., why probabilistic polynomials exist. We'll think specifically about the **OR** function — we can write $\mathsf{AND}(x) = \prod_{i=1}^n x_i$, so by De Morgan's laws we get the naive polynomial

$$\mathsf{OR}(x) = 1 - \prod_{i=1}^{n}(1-x_i).$$

This has degree n, and we'd like to do better with a probabilistic polynomial. For simplicity, we'll assume $p = 2$, and we'll try to get an error of $\frac{1}{2}$. (If we wanted error $\frac{1}{2}$ over random x, this would be trivial — we could just output 1 on everything. But it's not so clear how to get error $\frac{1}{2}$ on each fixed input x, as in Definition 13.14.)

We're working mod 2, so we'll try to simulate **OR** mod 2. And the way we'll do this is by picking $r_1, \ldots, r_n \in \{0,1\}$ uniformly at random and taking $\sum_{i=1}^n r_i x_i \pmod 2$ — so we're trying to compute the **OR** of a bunch of variables, and the way we do this is by taking a random subset of these variables and computing their sum mod 2.

If $x_1 = \cdots = x_n = 0$, then no matter what $r_1, \ldots, r_n$ are, we're always going to get $\sum r_i x_i = 0$. On the other hand, if even one x_i is 1, then we're going to have $\sum r_i x_i = 1$ with probability $\frac{1}{2}$. So this gives us a degree-1 probabilistic polynomial for **OR** with error $\frac{1}{2}$ (where the distribution consists of the polynomials $\sum r_i x_i$ for random r_i).

So we've got a degree-1 probabilistic polynomial with error $\frac{1}{2}$. Maybe we don't like having error $\frac{1}{2}$, so we'd like to shrink it. And the idea is that we can repeat this experiment multiple times. We'll essentially run k independent trials of this experiment and take their **OR** using the formula $1 - \prod_{i=1}^k (1-y_i)$, where each y_i is an independent polynomial of the form $\sum r_j x_j$.

Now we've got a probabilistic polynomial of degree k. And if $\mathsf{OR}(x) = 0$ then it'll always output 0; meanwhile, if $\mathsf{OR}(x) = 1$, then we only get an error when each of $y_1, \ldots, y_k$ ends up being 0, which happens with probability $\frac{1}{2^k}$. So we get a probabilistic polynomial of degree k and error $\frac{1}{2^k}$.

This shows how to simulate an **OR** gate with a probabilistic polynomial. And simulating a mod-2 gate with a probabilistic polynomial over $\mathbb{F}_2$ is also easy (you just take $x_1 + \cdots + x_n$). And once we've got something for **OR**, we can turn it into something for **AND** using De Morgan's formula. So this essentially proves Theorem 13.15 for $p = 2$.

For larger values of p, you have to do other tricks to make sure things end up in $\{0,1\}$, but you can do this by raising them to the $(p-1)$th power; this is where the factor of $p-1$ in the degree comes from.

§13.3 Algorithmic methods for circuit lower bounds

So far, we've considered AC^0, and looked at what happens when we add mod_p gates for one prime p. So the next question is, what happens if we add such gates for *two* primes p?

> **Question 13.16.** What can we do with AC^0 circuits with both mod_2 and mod_3 gates?

Having both mod_2 and mod_3 gates turns out to be equivalent to having mod_6 gates — this is essentially the Chinese remainder theorem. Explicitly, if we've got mod_2 and mod_3 gates, then we can simulate a mod_6 gate by writing $\mathsf{mod}_6(x) = \mathsf{mod}_2(x) \wedge \mathsf{mod}_3(x)$. Conversely, if we've got mod_6 gates, then we can write $\mathsf{mod}_2(x) = \mathsf{mod}_6(xxx)$ and $\mathsf{mod}_3(x) = \mathsf{mod}_6(xx)$.

And we've previously mentioned that we don't know whether MAJ can be computed with $\mathsf{AC}^0[6]$ circuits. But in fact, the story is much worse than that.

Open question 13.17. Does EXP have polynomial-sized $\mathsf{AC}^0[6]$ circuits?

In fact, we don't even know whether EXP has polynomial-sized $\mathsf{AC}^0[6]$ circuits of depth 3! But we *can* show that $\mathsf{NTIME}[n^{\mathsf{polylog}(n)}]$ doesn't have such circuits.

Interestingly, the only way we know how to prove circuit lower bounds like this is through a weird paradigm of relating lower bounds to algorithms — it turns out that circuit analysis algorithms imply circuit lower bounds. First, as some intuition for why this is possible, on a problem set we saw that if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{EXP} \not\subseteq \mathsf{SIZE}[o(2^n/n)]$. This means that if we have a *perfect* circuit analysis algorithm — meaning that $\mathsf{CircuitSAT} \in \mathsf{P}$ (so we can really analyze a circuit and say whether it's satisfiable or not — and we can say tons of other things as well, because PH collapses) — then we get strong circuit lower bounds. But we don't even know whether $\mathsf{EXP} \subseteq \mathsf{P/poly}$ — the hypothesis is definitely too strong, and the conclusion seems too strong too.

Question 13.18. Are there reasonable algorithmic hypotheses we could replace $\mathsf{CircuitSAT} \in \mathsf{P}$ with (i.e., ones we might actually be able to prove) that would imply some interesting conclusion?

One circuit analysis problem we might consider besides CircuitSAT is a version of CircuitSAT for a restricted class of circuits — we can define the problem $\mathcal{C}$-SAT for any class of circuits $\mathcal{C}$ (e.g., $\mathsf{AC}^0[m]$ or formulas or unrestricted circuits).

Definition 13.19 ($\mathcal{C}$-SAT)

- **Input:** some circuit C from the circuit class $\mathcal{C}$.
- **Decide:** whether C is satisfiable (i.e., whether there exists x with $C(x) = 1$).

We can also consider a version of CAPP for $\mathcal{C}$, which is presumably an easier problem. (Previously, we saw that CAPP was the canonical problem for derandomizing BPP; the version we'll state here is actually for derandomizing RP.)

Definition 13.20 ($\mathcal{C}$-CAPP)

- **Input:** some circuit C from the circuit class $\mathcal{C}$.
- **Promise:** either C is unsatisfiable, or at least half its assignments are satisfying.
- **Decide:** which of these two cases is true.

Of course there's a simple randomized algorithm for solving $\mathcal{C}$-CAPP, so we're interested in whether there's a *deterministic* algorithm. Asking for a *polynomial*-time deterministic algorithm (when $\mathcal{C}$ is unrestricted) would imply $\mathsf{P} = \mathsf{RP}$; we believe this is true, but it's probably hard to prove, so we won't ask for algorithms this strong. Instead, we'll just ask for an algorithm slightly better than exhaustive search — and it turns out that we'll be able to get circuit lower bounds from such algorithms.

Theorem 13.21

Let $\mathcal{C}$ be 'nice' (in ways we will not define). If $\mathcal{C}$-CAPP has an $O(2^n s/n^k)$-time algorithm (on circuits of size s with n inputs) for all k, then NEXP does not have polynomial-size $\mathcal{C}$-circuits.

Exhaustive search would take us roughly $2^n \cdot s$ time (we'd have to try all 2^n inputs and evaluate the circuit on each). And this theorem states that if we can get an algorithm that does even slightly faster, then we can get circuit lower bounds for NEXP.

We believe that this hypothesis should be true (if we believe in derandomization, this would mean there's a *polynomial*-time algorithm), but it's open for most circuit classes $\mathcal{C}$. But it is known for some $\mathcal{C}$, and this lets you prove bounds like $\mathsf{NEXP} \not\subseteq \mathsf{AC}^0[6]$.

§13.3.1 Proof idea for unrestricted circuits

Finally, we'll briefly outline how we prove Theorem 13.21 when $\mathcal{C}$ is the class of all (unrestricted) circuits.

The first ingredient of the proof is the *easy witness lemma*. Roughly speaking, what it says is that if $\mathsf{NEXP} \subseteq \mathsf{P/poly}$, then every NEXP verifier $\mathcal{V}$ has a 'succinct witness' — a witness that can be encoded by polynomial-sized circuits. (For x to be a **YES** instance means that there's some witness y such that $\mathcal{V}(x, y)$ accepts; y could potentially have exponential length, but the easy witness lemma says that there's some witness y which can be described succinctly — with only a polynomial-sized circuit.)

The other ingredient we need is some sort of extremely efficient PCP — if $f \in \mathsf{NTIME}[2^n]$, then there's a PCP for f with $O(1)$ queries and $n + 5\log n$ randomness, and with completeness 1 and soundness $\frac{1}{2}$ (and where the verifier runs in polynomial time). So even though we've started with a problem that took time 2^n to solve nondeterministically, we can get a PCP for it where the verifier only tosses roughly n random coins and runs in polynomial time.

And given these two facts, the idea of the proof is to show that if we've got a good CAPP algorithm and $\mathsf{NEXP} \subseteq \mathsf{P/poly}$, then $\mathsf{NTIME}[2^n] \subseteq \mathsf{NTIME}[o(2^n)]$, which would contradict the nondeterministic time hierarchy.

To show this, we start with some $f \in \mathsf{NTIME}[2^n]$. This means we've got a nondeterministic algorithm for f that first guesses $O(2^n)$ bits, and then checks in $O(2^n)$ time. And we want to speed up both of these steps so that they take $o(2^n)$ time.

First, the easy witness lemma says that no matter what verifier we've got for checking, we can actually guess a polynomial-size circuit that encodes the witness (rather than guessing the $O(2^n)$-bit witness) — so we first guess a circuit W_x of polynomial size, whose truth table encodes some witness.

And then we're going to imagine using the crazy PCP to check — so we run the PCP verifier $\mathcal{V}$, where we toss $n + 5\log n$ coins, make some queries, and then accept or reject. And the idea is that this last part can be implemented as a call to CAPP — if $f(x) = 1$, then there exists some circuit W_x for which $\mathcal{V}$ accepts with probability 1, meaning that $\mathcal{V}^{W_x}(x, v) = 1$; while if $f(x) = 0$, then for all circuits W_x, $\mathcal{V}$ will reject with probability $\frac{1}{2}$. So we can make a circuit $C_x(r)$ that takes the randomness r as its input and outputs 1 if and only if $\mathcal{V}$ rejects; then figuring out whether $f(x)$ is 0 or 1 corresponds exactly to solving CAPP.