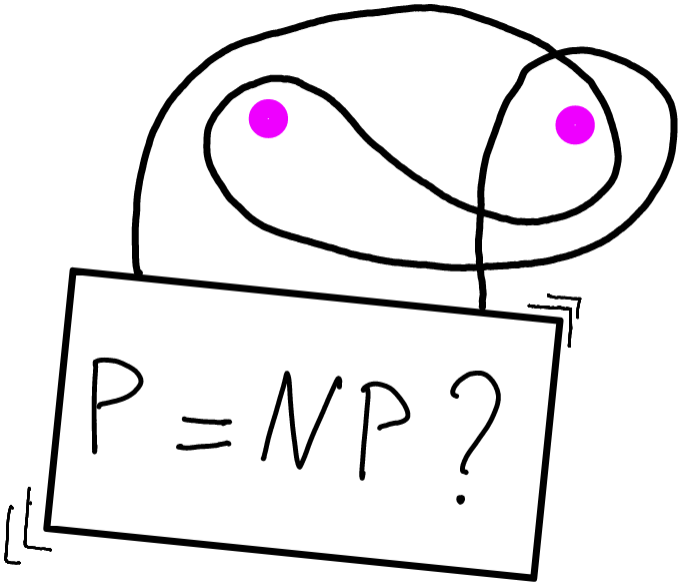
MATHEMATICS OF THE IMPOSSIBLE

THE COMPLEXITY OF COMPUTATION

Working draft compiled on December 3, 2025

Chapters 1 up to including 9 are meant to be in a decent state

Emanuele Viola



Contents

About this book

This is a book about *computational complexity theor*y. However, it is perhaps *sui generis* for various reasons:

1. The presentation is also *geared towards an algorithmic audience*. Our default model is that of *word programs* (a.k.a. word RAM) (Chapter 1), the standard model for analyzing algorithms. This is in contrast with other texts which focus on tape machines. I reduce computation directly to quasi-linear 3Sat using sorting algorithms, and cover the relevant sorting algorithm. Besides typical reductions from the theory of NP completeness, I also present a number of other reductions, for example related to the 3SUM problem and the exponential-time hypothesis (ETH). This is done not only to showcase the wealth of settings, but because these reductions are central to algorithmic research. Also, I include a chapter on *data structures*, which are typically studied in algorithms yet omitted from complexity textbooks. I hope this book helps to reverse this trend; impossibility results for data structures squarely belong to complexity theory. Finally, a recurrent theme in the book is the power of restricted computational models. I expose *surprising algorithms which challenge our intuition* in a number of such models, including space-bounded algorithms, boolean and algebraic circuits, tape machines, and communication protocols.

2. The book contains a number of recent, exciting results which are not covered in available texts, including: space-efficient simulations (Chapter 6), connections between various small-depth circuit classes (section §8.3), catalytic computation (section §6.9), cryptography in NC^0 (section §7.7), doubly-efficient proof systems (which have formed the basis of some deployed cryptographic systems) (section §10.4), simple constructions of expanders avoiding iterated recursion (Chapter 12), recent number-on-forehead communication protocols (section §13.4), succinct data structures (section 15.1.2), impossibility results for constant-depth algebraic circuits (Chapter 14), and natural-proof barriers that are informed by deployed cryptosystems (Chapter 18).

3. I also present several little-known but important results. This includes factoring efficiently using large words(Theorem 1.2), cosmological bounds (Theorem 2.7), the complexity of computing integers and its connections to factoring (section §14.2), and several results on pointer chasing section 13.4.1) and tape machines (Theorem 16.9, section §16.7).

4. A number of well-known results are presented in a different way. Why? To demystify them and expose illuminating connections. Some of this was discussed above in 1. In addition, unlike other texts where they appear later, here I present *circuits* and *randomness* right away, and weave them through the narrative henceforth. For example, I introduce BPP and showcase its power via intuitive problems which require minimal mathematical background. Key results such as $\mathrm{BPP} \subseteq \Sigma_2\mathrm{P}$ and $\mathrm{PH} \subseteq \mathrm{BP} \cdot \oplus \cdot \mathrm{P}$ are presented through the lens of the circuit model and are connected to pseudorandom generators. In Chapter 11 the BIG-HIT generator is used to give a streamlined construction of pseudorandom generators from hard functions, avoiding some of the steps of previous constructions. Reductions are presented *before* completeness rather than later as in most texts. This, I believe, demystifies their role and leads to a transparent exposition as stand-alone results. Also, reductions are presented first as *implications*, then specialized in various ways. This clashes with most texts and affects a few things, for example the definition of NP-intermediate problems, see Exercise 5.4. On the other hand, implications are the most flexible type of reductions, and I think understanding implication is more fundamental than understanding more constrained types of reductions.

5. This book *challenges traditional assumptions and viewpoints*. For example, I discuss my reasons for not believing $\mathrm{P} \neq \mathrm{NP}$. In particular, I catalog and contextualize for the first time conjectures in complexity lower bounds which were later disproved (Chapter 19). Also, I emphasize that several available impossibility results may be "strong" rather than "weak" as commonly believed because they fall just short of proving major separations (e.g. section §6.3), and I expose the limitations of standard tools such as the hard-core set lemma (section 11.2.2). Finally, the notes put key results in perspective.

I made several other choices to focus the exposition on the important points. For example I work with partial (as opposed to total) functions by default. This affects many things, for example hierarchy theorems, see section §3.5. The following quote from *[107]* captures a common mindset towards partial functions, a.k.a. promise problems: "Recall that *promise problems offer the most direct way of formulating natural computational problems. [...]* In spite of the foregoing opinions, we adopt the convention of focusing on standard decision and search problems." To put results in context I routinely investigate what happens under slightly different assumptions. Finally, I present proofs in a "top down" fashion rather than "bottom up," starting with the main high-level ideas and then progressively opening up details, and I try to first present the smallest amount of machinery that gives most of the result. I decided to relegate references to the notes of each chapter and not spell out names of authors. Central results, such as the PCP theorem, are co-authored by many people and span several papers, so that their history is better explained in the notes. To save some of the thrill of name-splashing, names appear in select portions which bend to the historical. Names also appear in the index, so one can for example look up "Markov's inequality" there.

This book is intended both as a textbook and as a reference book. The intended audience includes students at all levels, and researchers, in both computer science and related areas

such as mathematics, physics, data science, and engineering. The text is interspersed with exercises which serve as quick concept checks, for example right after a definition. More advanced problems are collected at the end of each chapter. Solutions or hints for both exercises and problems are provided as separate manuals. I assume no background in theory of computation, only some "mathematical maturity" as can arise for example from typical introductory courses in discrete mathematics. All other mathematical background is covered in Appendix B.

The book can be used in several different types of courses.

- For an introductory course in complexity theory, suitable for an undergraduate student who has had some prior exposure to algorithms and mathematical thinking, one can cover Chapters 16 to 5. At the same time the text can expose the interested students to more advanced topics, and stimulate their critical thinking.

- For a broader course in complexity, suitable for advanced undergraduate students or for graduate students, one can add Chapters 5.5 to 10. Such a course can be supplemented with isolated topics from Chapters 11 to 18. For example, in my offerings of cross-listed undergraduate/graduate PhD complexity theory, I typically cover Chapters 16 to 10 and then one or two select chapters from 11 to 18. The pace is about one chapter a week, and I ask the students to attempt all exercises.

- For a special-topics course or seminar one can use Chapters 11 to 18. One possibility, which I tested, is covering all these chapters.

Chapters 16 to 10 are best read in order. Chapters 11 to 18 have fewer dependencies and can be read more or less in any order.

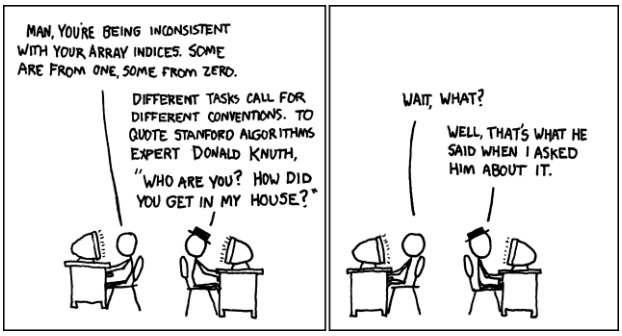
I hope this text will keep the reader engaged and serve as an invitation and guide to the mysterious land of complexity, until the reader stands at its frontier, gazing into the vast unknown.

Acknowledgments

Obviously, this work would not have been possible had I not had the privilege of interacting with many (most?) of the leading complexity theorists over the years. Many people helped in various ways specifically with this book, including: Eric Allender, Noga Alon, Ravi Boppana, Sam Buss, Peter Bürgisser, Bill Gasarch, Songhua He, Valentine Kabanets, Michal Koucký, Nutan Limaye, Bruno Loff, Oded Schwartz, Srikanth Srinivasan, Justin Thaler, Salil Vadhan, Avi Wigderson. I am also indebted to the students in my classes and the readers of my blog, where some of this material appeared first.

Chapter -1

Conventions, choices, and caveats



https://xkcd.com/163/

The c notation. The mathematical symbol c has a special meaning in this text. Every *occurrence* of c denotes a real number > 0. There exist choices for these numbers such that the claims in this book are (or are meant to be) correct. Let us illustrate via few examples:

– "For all sufficiently large n" can be written as $n \geq c$.

– "It is an open problem to show that some function in NP requires circuits of size cn." This is a correct statement at the moment of this writing: one can replace this occurrence with 5.

– "$c > 1 + c$" is correct if we assign 2 to the first occurrence, 1 to the second.

– "$100n^{15} < n^c$" is correct for all large enough n. Assign $c = 16$.

– "$c < 1/n$ for every n". This is not true: No matter what we assign c to, we can pick a large enough n. Note the assignment to c is absolute, independent of n.

More generally, when subscripted this notation indicates a function of the subscript. For example:

– "For every ϵ and all sufficiently large n" can be written as $n \geq c_\epsilon$.

This notation replaces the big-Oh notation. For readers who replace the latter, a quick and dirty fix is to replace every occurrence of c in this book with $O(1)$.

Logarithm. log denotes logarithm with base 2 by default.

Polynomial. It is customary in complexity theory to bound quantities by a polynomial, as in polynomial time, when in fact only one monomial matters. It seems to me this makes some statements cumbersome, and lends itself to confusion since polynomials with many terms are useful for many other things. I use *power* instead of polynomial, as in power time (similarly to say "power law").

The asymptotics of Time. In some texts, the complexity class Time(t) corresponds to at steps for a constant a. Here it's t steps. So linear time is the union over a of Time(at). This is perhaps more in line with the way times are stated in algorithms, and works well with some results, e.g. the hierarchy theorem. Also, I tie the asymptotic input length to the program size, see TBD.

Cardinality. For a set A I also write A for its cardinality $|A|$.

Definition 6.3 of branching programs, and cycles. Some texts require that the graph of a branching program, ignoring all labels, is acyclic. This can be thought of as a *syntactic* condition which makes it easy to decide if a graph corresponds to a branching program. By contrast, my definition is *semantic*: I ask that the path induced by an input does not contain a cycle. One would *a priori* need to examine all $x \in X$ to decide if this is the case. The semantic definition appears more in line with the way we defined other resources. For example, given a program it is not immediate to decide if it runs in a specific time bound. Moreover, it makes the proof of Theorem 6.3 slightly easier. For a comparison of the two definitions see Problem 6.1.

The alphabet of TMs. I define TMs with a fixed alphabet. This choice slightly simplifies the exposition (one parameter vs. two), while being more in line with common experience (it is more common experience to increase the length of a program than its alphabet). This choice affects the proof of Theorem **??**; but the details don't seem any worse.

Summary of some terminological and not choices

Some other sources	this book
$O(1)$, $\Omega(1)$	c
$\|A\|$ for the size of a set A	A
polynomial time	power time
superpolynomial	superpower
mapping reduction (sometimes)	A reduces to B in P means $B \in \mathrm{P} \Rightarrow A \in \mathrm{P}$
NP-complete	complete for P vs NP
Extended Church-Turing thesis	Power-time computability thesis
pairwise independent	pairwise uniform
Turing machine	tape machine
TM with any alphabet	TM with fixed alphabet
classes have total functions	classes have partial functions
promise-P, promise-BPP, ...	P, BPP, ...
P/poly	CktP
L/poly	BrL
$\{0,1\}$	$[2]$
Time$(t) \to at(n)$ steps for $n \geq n_0$	$t(n)$ steps for $n \geq \|P\|$
PH $\subseteq \mathrm{P}^{\#\mathrm{P}}$	PH $\subseteq$ SymP

Unindexed mathematical notation and symbols

bit-len(x)	Minimum number of bits to write $x \in \mathbb{N}$(Definition 1.3)
weight(x)	number of bits set to 1 in x
f_n	The restriction of the function $f : X \subseteq [2]^* \to [2]^*$ to inputs of length n
$[P]$	1 if P is true, 0 otherwise
$[i..j]$	$\{i, i+1, i+2, \ldots, j\}$
$[i]$	$[0..i-1] = \{0, 1, 2, \ldots, i-1\}$
$[2]^n$	binary strings of length n
$[2]^*$	binary strings of any length
$i \mid j$	i divides j
$\mathbb{C}$	complex numbers
$\mathbb{E}$	expectation
$\mathbb{F}_q$	field with q elements
$\mathbb{N}$	natural numbers $\{0, 1, 2, \ldots\}$
$\mathbb{P}$	probability
$\mathbb{Q}$	rational numbers (from *quotient*)
$\mathbb{R}$	real numbers
$\mathbb{Z}$	integer numbers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (from *Zahlen*)
$\mathbb{Z}_m$	integers $[m]$ with operations modulo m
§	section

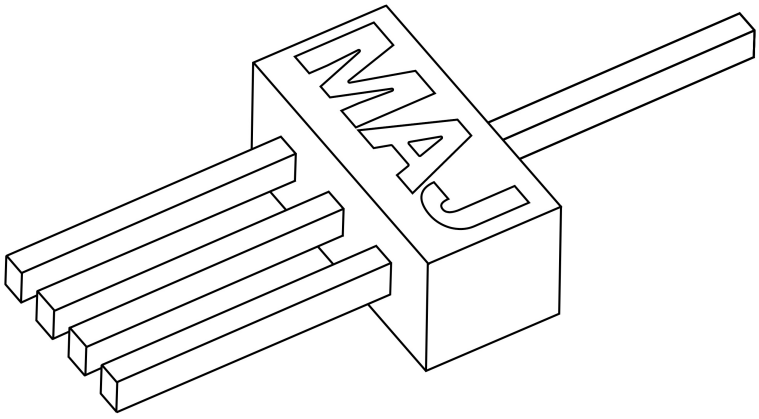
The $\{0,1\}$ notation is cumbersome for people and compilers. What I really would like is use $2 = \{0,1\}$ as in $f : 2^n \to 2$, but I fear it's pushing it a little.

Abbreviations

a.k.a.	also known as
e.g.	as an example (*exempli gratia*)
i.e.	that is (*id est*)
iff	if and only if
lhs	left-hand side
prob.	probability
rhs	right-hand side
r.v.	random variable
s.t.	such that
w.h.p.	with high prob.
w.l.o.g.	without loss of generality
cf	compare

Chapter 0

A teaser



Consider a computer with *three* bits of memory. There's also a clock, beating $1, 2, 3, \ldots$ In one clock cycle the computer can read one bit of the input and update its memory, or stop and return a value. These actions depend only on the clock, the three memory bits, and the length of the input.

Let's give a few examples of what such computer can do.

First, it can compute the And function on n bits:

```
Computing And of (x_1,x_2,...,x_n)

For i = 1, 2, ... until n
   Read x_i
   If x_i = 0 return 0
Return 1
```

We didn't really use the memory. Let's consider a slightly more complicated example. A word is *palindrome* if it reads the same both ways, like *racecar*, *non*, *anna,* and so on. Similarly, example of palindrome bit strings are $11, 0110$, and so on.

Let's show that the computer can decide if a given string is palindrome quickly, in n steps

Deciding if $(x_1, x_2, \ldots, x_n)$ is palindrome:

For $i = 1, 2, \ldots$ until $i > n/2$
  Read x_i and write it in memory bit m
  If $m \neq x_{n-i}$ return 0
Return 1

That was easy. Now consider the Majority function on n bits, which is 1 iff the sum of the input bits is $> n/2$ and 0 otherwise. Majority, like any other function on n bits, can be computed on such a computer in time *exponential* in n.

Exercise 0.1. Prove that any function $f : [2]^n \to [2]$ can be computed on such a computer in time 2^{cn}.

So this works for any function, but it's terribly inefficient. Can we do better for Majority? Can we compute it in time which is just a power of n?

> Convince yourself that this is impossible. Hint: If you start counting bits, you'll soon run out of memory.

If you managed to convince yourself, you are not alone.

And yet, we will see the following shocking result:

Theorem 0.1. Majority can be computed on such a computer in time n^c.

And this is not a trick tailored to majority. Many other problems, apparently much more complicated, can also be solved in the same time.

But, there's something possibly even more shocking.

Shocking situation:
It is consistent with our state of knowledge that every "textbook algorithm" can be solved in time n^c on such a computer! Nobody can disprove that. (Textbook algorithms include sorting, maxflow, dynamic programming algorithms like longest common subsequence etc., graph problems, numerical problems, etc.)

The **Shocking theorem** gives some explanation for the **Shocking situation**. It will be hard to rule out efficient programs on this model, since they are so powerful and counterintuitive. In fact, we will see later that this can be formalized. Basically, we will show that the model is so strong that it can compute functions that provably escape the reach of current mathematical proofs... if you believe certain things, like that it's hard to factor numbers. This now enters some of the *mysticism* that surrounds complexity theory, where different beliefs and conjectures are pitted against each other in a battle for ground truth.

Proofs Above is one way in which complexity theory has contributed to the ancient concept of "proof." But the impact is much more widespread, and also affects many computer systems currently deployed.

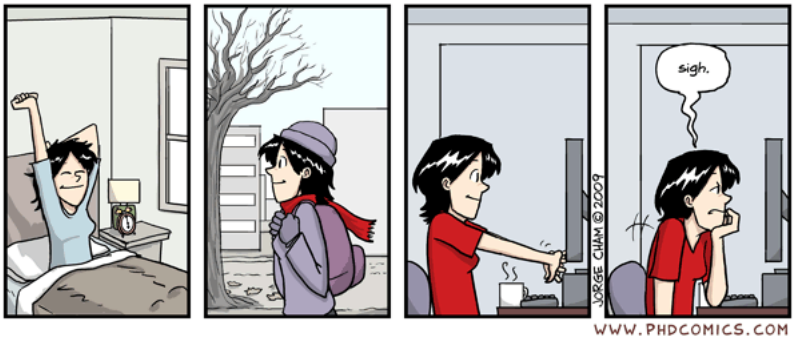
Suppose I claim I do have a program as above that computes majority. The program is fairly short, but it runs in time n^{10}. Naturally, you want to check it on an input x. But even if the input has length $|x| =5$ bits, this would take forever. Can I convince you that the program does compute majority correctly quickly, much faster than it would take you to run it? This sounds impossible, I could be cheating in any way, you could only be sure if you checked each step of the computation. But in fact, it is possible, and your computation time would be essentially linear in $|x|$. The proof will be *interactive*, you will ask me a question, I will give a reply, and so on a few times, and all your computation time (but not mine) will be very small.

OK, it works on one input, but how can you check if it works on *every* input of length say $n = 100$? Now, this gotta be really impossible to do efficiently, after all there are 2^{100} possible inputs. Turns out we will again see how you can verify this in time power, not exponential, in n. And in fact this works not just for these simple programs, but for *any* program.

This is just a glimpse of the fascinating world of complexity we are about to enter.

Chapter 1

Time



In this chapter we introduce our first – and a main – model of computation. In general, computation has two fundamental features:

- **Locality.** Computation proceeds in small, quick, and local steps. Each step only depends on and affects a small amount of "data." For example, in the grade-school algorithm for addition, each step only involves a constant number of digits.

- **Generality.** The computational process is general in that it applies to many different inputs. At one extreme, we can think of a single algorithm which applies to an infinite number of inputs. This is called *uniform* computation and is explored in this chapter. Or we can design algorithms that work on a finite set of inputs. This makes sense if the description of the algorithm is much smaller than the description of the inputs that can be processed by it. This setting is usually referred to as *non-uniform* computation and is explored in Chapter 2.

We seek a model that is realistic: it should be close enough to how computers work and are programmed. At the same time, it should not be overly complicated to define and to reason about.

1.1 Word programs

Our model can be seen as a simple machine language, or assembly, not unlike that used by microprocessors. We think of computing over *words* of w bits. Memory is organized as an array M of such words. Also, there is an array R of words corresponding to a constant number of *registers*. We allow basic arithmetic operations among registers, and read from and write to memory. We have a basic control-flow instruction, GOTO...IF which allows us to create loops similar to the *for* or *while* loops in many programming languages. In addition, we include the instruction MALLOC that increases the number of memory words.

Definition 1.1. A *word program* of size ℓ is a sequence of ℓ instructions of the following type:

- $R[i] := R[j]$, where $i, j \in [\ell]$ (copying a register)
- $R[i] := b$, where $i \in \ell$ and $b \in [2] = \{0, 1\}$ (setting a register to constant)
- $R[i] := E$, where $i \in [\ell]$ and E is a *basic expression,* which is one of the following operations or relations, among two registers or one register and the constant 1:
  - $+, -, *, \%$ addition, subtraction, multiplication, and modular remainder
  - $<<, >>$, left and right bit shift
  - $\wedge, \vee, \neg$ bit-wise and, or, and negation
  - $>, <, \leq, \geq, =$, order relations;
- $R[i] := M[R[j]]$, where $i, j \in [\ell]$ (reading from memory)
- $M[R[i]] := R[j]$, where $i, j \in [\ell]$ (writing to memory)
- GOTO i IF E, where $i \in [\ell]$, and E is a basic expression as above
- MALLOC (increasing memory)
- STOP

Word programs are sometimes called word RAMs.

Note the above definition is redundant. For example, we can implement $R[i] := R[j]$ as $R[i] := R[j] * 1$. Many other simplifications are possible. However, we keep the above syntax for clarity. Later we will consider various minimalistic programs, including some that only have two instructions (see section §7.5).

An important concept that we will see again and again is that of a *configuration*. A configuration is a "snapshot" of the execution of a program at a certain point that contains all the information required to carry on the computation after that point, assuming the program is known.

Definition 1.2. The *configuration* C of a program of size ℓ is a tuple

$$C = (i, m, w, R, M)$$

where $i \in [\ell]$ is the *program counter*, $m \in \mathbb{N}$ is the *space*, $w \in \mathbb{N}$ is the word length, $R \in [2^w]^\ell$ is the array of registers, and $M \in [2^w]^m$ is the memory.

As indicated earlier, a computation step makes *local changes*. To indicate such changes, the following notation is useful. For a vector V (e.g., R, M) we write $V[i \leftarrow x]$ for the vector V' where $V'[i] = x$ and $V'[j] = V[j]$ for all $j \neq i$. Regarding memory, we shall maintain the invariant that the word length is sufficient to write the number m of memory words, so w is at least the *bit-length* of m. In particular this allows us to index the memory words (for which $w \geq \text{bit-len}(m-1)$ suffices).

Definition 1.3. The bit-length of $a \in \mathbb{N}$ is denoted bit-len(a) and is the minimum number of bits needed to write a. We have bit-len$(a) = \lceil \log(a+1) \rceil$, with the exception bit-len$(0) = 1$.

We can now formally define computation.

Definition 1.4. Let P be a word program of size ℓ. Let $C = (p, m, w, R, M)$ be a configuration, and let instruction p of P be I. Then C *yields* configuration C' as follows:

- If $I =$ "$R[i] := E$" then $C' = (p+1, m, w, R[i \leftarrow E], M)$. Here the operations in E are performed modulo 2^w: The operation is performed over naturals, but only the least significant w bits are stored in $R[i]$. Subtraction $R[j] - R[k]$ is defined as 0 if $R[j] < R[k]$. Relations (e.g., $R[i] < R[j]$) take value 1 if true, 0 otherwise.
- If $I =$ "$R[i] := M[R[j]]$" then $C' = (p+1, m, w, R[i \leftarrow M[R[j]]], M)$, where $M[R[j]]$ is defined to be 0 if $R[j] \geq m$.
- If $I =$ "$M[R[i]] := R[j]$" then $C' = (p+1, m, w, R, M[R[i] \leftarrow R[j]])$ if $i < m$, otherwise M is unchanged.
- If $I =$ "MALLOC" then $C' = (p+1, m+1, \max\{w, \text{bit-len}(m+1)\}, R, M[m \leftarrow 0])$.
- If $I =$ "GOTO i IF E" then $C' = (i, m, w, R, M)$ if $E \geq 1$, and $C' = (p+1, m, w, R, M)$ if $E = 0$.

A *computation* is a sequence of configurations corresponding to a program.

Definition 1.5. A *computation* of a word program P is a sequence of configurations $C_0, C_1, \ldots, C_t$ such that C_i yields C_{i+1} for $i \in [t]$. We say that P computes C_t from C_0.

In general, configuration C_0 starts with some information, or input, and C_t contains some additional information, written in the registers, or memory, or some combination.

Example 1.1. We give a program for *CountingSort* starting from suitable configurations. The space is $m := ct$ (recall the "c" notation from Chapter -1), the first t memory words $M[0..t-1]$ contain numbers in $[t]$, the other memory words are 0, the word size is $c\lceil \log m \rceil$, and $R[0] := t$. The following program will place the sorted sequence of numbers into words $M[2t..3t-1]$. The words $M[t..2t-1]$ are auxiliary words used to count the number of occurrences. Typically, these two sub arrays of M would be given different names; this example illustrates how adding suitable shifts to memory indexes, we can use a single array to simulate many arrays.

/* Loop to set $M[t+i]$ to the number of input words equal to i:
  For $(R[1] = 0; R[1] < R[0]; R[1]++)$
    $M[R[1] + R[0]]++;$
*/
0: $R[2] := M[R[1]]$
1: $R[2] := R[2] + R[0]$
2: $R[3] := M[R[2]]$
3: $R[3] := R[3] + 1$
4: $M[R[2]] := R[3]$
5: $R[1] := R[1] + 1;$
6: GOTO 0 IF $R[1] < R[0]$
/* Loop to set $M[t+i]$ to the number of input words equal or less than i:
  For $(R[1] = 1; R[1] < R[0]; R[1]++)$
    $M[R[1] + R[0]] += M[R[1] + R[0] - 1];$
*/
7: $R[1] := 1$
8: $R[2] := R[1] + R[0]$
9: $R[3] := R[2] - 1$
10: $R[4] := M[R[2]]$
11: $R[5] := M[R[3]]$
12: R[6]:=$R[4] + R[5]$
13: $M[R[2]] := R[6]$
14: $R[1] := R[1] + 1;$
15: GOTO 8 IF $R[1] < R[0]$
/* Loop to place each number at right location.
  For $(R[1] = 0; R[1] < n; R[1]++)$ {
    $M[M[M[R[1]] + R[0]] + 2 * R[0]] = M[R[1]];$
    $M[M[R[1]] + R[0]]--;$
  }
*/
16: $R[1] := 0$
// $M[M[M[R[1]] + R[0]] + 2 * R[0]] = M[R[1]];$
17: $R[3] := M[R[1]]$
18: $R[2] := R[3] + R[0]$

19: $R[2] := M[R[2]]$
20: $R[2] := R[2] + R[0]$
21: $R[2] := R[2] + R[0]$
22: $M[R[2]] := R[3]$
// $M[M[R[1]] + R[0]] - -;$
23: $R[3] := R[3] + R[0]$
24: $R[4] := M[R[3]]$
25: $R[4] := R[4] - 1$
26: $M[R[3]] := R[4]$
27: $R[1] := R[1] + 1;$
28: GOTO 17 IF $R[1] < R[0]$

The purpose of this example was to convince ourselves that standard algorithms can be implemented as word programs. Going forward, we will rarely need to write down programs explicitly; a high-level description suffices.

1.2 Complexity classes

In Example 1.1 we have started from a somewhat *ad hoc* configuration: The input was given in words, we had enough memory for the scratch array, and the output was written after that. If we didn't have enough memory, we would have needed to use MALLOC to allocate it. If the input was given in bits, we would have needed to translate it into words. To talk about the complexity of computing arbitrary functions, we now fix some input-output conventions.

The input is going to be a bit-string $x = (x_0, x_1, \ldots, x_{n-1}) \in [2]^n$. We use $|x|$ to denote n. We often need to work with *more structured* objects, like tuples, graphs, matrices, programs, etc. One can always encode such objects in binary, and often ignore the details of such encodings. For a simple example, we can encode a pair (x, y) where $x, y \in [2]^*$ by inserting a 0 in front of each bit of x except the last which has a 1 in front:

$$(x, y) \to 0x_0 0x_1 \cdots 0x_{|x|-2} 1x_{|x|-1} y. \tag{1.1}$$

Such a string uniquely specifies x and y. The length of this representation is $2|x|+|y|$. We are doubling the bits in x, but this blow-up hardly concerns us. Still, succincter encodings exist, see Problem 1.1. At the beginning of the computation, the input is in words $M[0..n-1]$, one bit per word. As mentioned earlier, the word length w is set to the minimum that can write the number m of memory words, and at the beginning $m = n$, so $w :=$ bit-len(n). Other choices for the word size are discussed in section §1.7. For some problems (e.g., sorting, see Example 1.1) it is actually natural to given the input in words, not bits. Typically, one can quickly convert between the two representations, so this distinction will not make a difference in most settings, while working with bits suffices for and simplifies most of the presentation.

Exercise 1.1. Extend the encoding above (Equation (1.1)) to tuples; then give a high-level description of a word program that converts an input tuple $x \in [m]^m$ (given as above one bit per memory word) into the format expected in Example 1.1.

Registers are initialized at 0, except $R[0]$ containing n. At the end, the output is written starting in cell $M[0]$, again one bit per cell, and $R[0]$ contains the length of the output.

Definition 1.6. We say that a word program P computes $y \in [2]^*$ on input $x \in [2]^*$ in *time* t if there is a computation $C_0, C_1, \ldots, C_t$ such that:

- C_0 is the start configuration $(0, n, w, R, M)$ where $n = |x|$, $w :=$ bit-len(n), $R[0] = n$ and the other $R[i]$ are 0, and $M[i] = x_i$ for $i \in [n]$ while $M[i] = 0$ for $i \in [n..2^w - 1]$.

- C_t is a configuration (p, m, w, R, M) where instruction p of P is STOP, $R[0] = |y| \leq m$, and $M[0] = y_0, M[1] = y_1, \ldots, M[|y| - 1] = y_{|y|-1}$.

Example 1.2. The following program on input $x \in [2]^*$ outputs xx, i.e., two copies of x. We use $R[1]$ to loop from 0 to $n - 1$, and $R[2]$ to loop from n to $2n - 1$; each time we copy $M[R[1]]$ into $M[R[2]]$ using scratch register $R[3]$. We call MALLOC to allocate memory words for the output.

0: $R[2] := R[1] + R[0]$ // $R[2]$ points to next symbol of 2nd copy
1: $R[3] := M[R[1]]$
2: MALLOC
3: $M[R[2]] := R[3]$
4: $R[1] := R[1] + 1$
5: GOTO 0 IF $R[1] < R[0]$
6: $R[0] := R[0] + R[0]$ //Double $R[0]$ to indicate output length
7: STOP

Next we define complexity classes. First, some remarks.

- We allow *partial functions,* i.e., functions with a domain X that is a strict subset of $[2]^*$, as opposed to *total functions* which are defined over the entire $[2]^*$. Partial functions are a natural choice for many problems.

- We measure the running time of the machine in terms of the *input length,* denoted n. Running times can be complicated expressions such as $t(n) = 2n^3 \log(n + 1) + 3n + 15$ which are hard to make sense of and depend on irrelevant details of the model, e.g., is the constant 2 allowed or do we need to build it via the expression $1 + 1$? We'd like to focus on the leading terms only, and say for example that the former expression is $\leq 3n^3 \log n$. This is true for large enough n, but not for small n. For example, if $n = 1$ then $\log n = 0$, so the expressions $3n^3 \log n$ would be 0, but $t(n)$ is not 0. As these details do not add to our understanding, we adopt the convention that the running time bound only has to hold for inputs larger than a constant.

- Depending on the context, it is convenient to work with complexity classes of *boolean* functions, i.e., range [2], or with functions with multi-bit output, i.e., range $[2]^*$. The multi-bit variant is denoted with an "F" in front. More generally, we will be interested in computing not just functions but *relations*. That is, given an input x there will be a *set* $f(x)$ of several possible outputs, and we just want to compute any y in $f(x)$. The case of functions is when $f(x)$ is a singleton set $\{y\}$, in which case we simply write $y = f(x)$. The "F" variant will be defined for this general case of relations. Many natural problems naturally give relations, one example being factoring (see 1.8).

Definition 1.7. [Time complexity classes] Let $t : \mathbb{N} \to \mathbb{N}$ be a function. Time(t) is the set of functions that map a subset $X \subseteq [2]^*$ to [2] for which there exists a word program P that computes $f(x)$ in time $\leq t(|x|)$ on any input $x \in X$ of length $\geq |P|$.

We also define

$$\text{Linear-Time} := \bigcup_{d \geq 1} \text{Time}(dn),$$

$$\text{Quasi-Linear-Time} := \bigcup_{d \geq 1} \text{Time}(n \log^d n),$$

$$\text{Power-Time P} := \bigcup_{d \geq 1} \text{Time}(n^d),$$

$$\text{Exponential-Time Exp} := \bigcup_{d \geq 1} \text{Time}(2^{n^d}).$$

Correspondingly, FTime(t) is the set of functions that map a subset $X \subseteq [2]^*$ to subsets of $[2]^*$ for which there exists a word program P that computes some $y \in f(x)$ in time $\leq t(|x|)$ on any input $x \in X$ of length $\geq |P|$. We simply write $y = f(x)$ for the case of functions (where $f(x) = \{y\}$). Also,

$$\text{FP} := \bigcup_{d \geq 1} \text{FTime}(n^d).$$

Remark 1.1. Here $|P|$ is the length of the program. This Definition 1.7 does not change if "length $\geq |P|$" is relaxed to "length $\geq n_0$" for another constant n_0 depending on P. Indeed, we can easily pad the length of P so that it's larger than n_0, for example by adding n_0 useless instructions $R[0] := R[0]$ at the end. The relaxed definition can be more useful when writing programs, as we wouldn't want to write the padding. But this book is mostly about *analyzing* programs, and the choice $n_0 = |P|$ shortens some definitions and proofs (e.g., time hierarchies). Similarly, Definition 1.7 does not change if we insist that P computes f correctly even on inputs of length $< |P|$ (in some time). Indeed, we can modify the program so that it contains the value of f for all those inputs. Depending on the circumstance, we will use the variant that is most convenient and makes the arguments more transparent.

An analysis of the program in Example 1.2 shows that duplicating a string is in FTime(cn). As mentioned earlier, we usually do not write programs but give higher-level descriptions, as illustrated by the next examples.

Example 1.3. Computing the Or of n bits is in Time(cn). Indeed, we can scan the input once, keeping track of the Or of the bits seen so far in a register, and in the end write that register to memory. This takes time $cn + c$, which is $\leq cn$ for large enough n.

Example 1.4. Addition of 2 naturals is in FTime(cn). We can implement the grade-school algorithm. Given x, y, we compute their sum bit-by-bit, starting with the least-significant bit and keeping track of the carry. This requires constant time per bit. This sum is written in a new memory array, then copied in the output. This takes linear time. Overall, the time is $\leq cn$ for large enough n.

Exercise 1.2. Prove that multiplication of 2 naturals in in FP.

We note that the definitions of complexity classes allow for arbitrary constants (e.g., the exponent of log in the definition of Quasi-Linear-Time). In practice, natural problems falling within these classes tend to have reasonable constants leading to algorithms that can be deployed. On the other hand, the constants can lead to unpractical results. This point is well illustrated by Theorem 1.1 and Problem 1.2.

For many important problems, whether they are in FP or not is not known. We will encounter many such problems in this book, e.g. in Chapter 4. For the discussion in this chapter, it suffices to consider factoring:

Definition 1.8. The Factoring problem: Given $x \in \mathbb{N}$, compute a prime factor of x.

It is not known if Factoring is in FP. A common conjecture is that it is not; moreover many deployed cryptographic systems (for electronic commerce, etc.) rely on the infeasibility of factoring numbers, in fact products of two primes, of about 1000 digits.

Conjecture 1.1. Factoring is not in FP.

The fastest algorithm that we have runs in exponential time 2^{n^c}.

1.3 You don't need much to have it all

How powerful are word programs? Perhaps surprisingly, they are all-powerful.

Power-time computability thesis. Power-Time, P, is the same for word programs as for any "realistic" model of computation.

This is a *thesis*, not a *theorem*. The meaning of "realistic" is a matter of debate, and one challenge to the thesis is discussed in Chapter 3. However, the thesis can be proved for many standard computational models, which include all modern programming languages. The proofs aren't hard, and amount to designing suitable *compilers*. Essentially, one just goes through each instruction or construct in the language and gives a implementation in a word program. In Example 1.1 we have basically done this: We have shown how to write "for" loops and complicated expressions as word programs. Less obvious is that even models

that appear much more restricted are equivalent to word programs. We will see this in Chapter 16.

A main source for our interest in word programs is that one can formulate a bolder thesis, namely that word programs capture efficient computation up to lower-order factors:

Quasi-Linear-Time computability thesis. Quasi-Linear-Time is the same for word programs as for any "realistic" model of computation.

Example 1.1 supports the Quasi-Linear-Time computability thesis, because the word-program implementation of quick sort runs in linear time.

Because of these theses, word programs will be often identified with and simply called "algorithms," and we can appeal to "algorithmic intuition" when asserting the existence of a program. To develop such intuition, next we show how some intuitive composition properties of algorithms work for word programs.

1.4 Composing programs

Programs are usually made by composing many simpler programs. In this section we prove some composition results which will be used frequently, often implicitly. This serves as an illustration of the model and increase our confidence in the computability theses from section 1.3.

The next result illustrates a basic property of FP: Closure under composition. We mostly use this for functions, where the composition $g \circ f$ is simply defined as $x \to g(f(x))$. But it works just the same for relations, where the notation $g(f(x))$ stands for $\bigcup_{y \in f(x)} g(y)$.

Claim 1.1. If f and g are in FP then the composition $h(x) := g(f(x))$ is in FP.

Proof. Let $f \in \text{FTime}(n^a)$ and $g \in \text{FTime}(n^b)$ and let P_f and P_g be corresponding programs as in Definition 1.7. The idea is simply to run P_f and then P_g. The output length of f on an input of length n is $\leq n^a + n$, because P_f can only execute so many MALLOC instructions, and the output length is bounded by the memory available, see Definition 1.6. Hence $P_g(f(x))$ will run in time $\leq (n^a + n)^b$ if the $|f(x)| \geq |P_g|$. Otherwise, P_g will take at most a number s of steps, the maximum over all inputs of length $\leq |P_g|$. Overall, the running time of the combined program on an input of length n is

$$n^a + (n^a + n)^b + s \leq n^{ab+1}$$

for large enough n, as desired.

However, P_f may stop in a configuration with more memory than the start configuration of P_g, and we do not know how P_g behaves in that case. To address this, we modify P_g as follows. We use registers to keep track of a new word size w' and space m', initialized to the values bit-len($|f(x)|$) and $|f(x)|$ as in the starting configuration on input $f(x)$. After each instruction we truncate the registers to word size w', using bit-wise And. When accessing

memory, the index is compared to m' to determine if it is out of boundary. When MALLOC is executed, we increase m', and adjust w' accordingly. These changes allows us to simulate the behavior of P_g on the input $f(x)$, and only increase the time by a constant factor. **QED**

Typical programs (including the ones in Example 1.1 and Example 1.2) work just as well when started from a configuration with larger memory and word size. We call such programs *compatible*. For compatible programs, the above proof can be simplified by skipping the last paragraph, i.e., we can simply run the programs one after the other as they are. Alternatively, we could have used a FREE instruction that is the opposite of MALLOC; but the approach above is more flexible.

In addition, we often need to run a program within the context of another computation, i.e., as a *subroutine*. That is, we are in some configuration and want to compute a function f on some input x which is in memory, while keeping the memory intact (so we can run other programs). This situation is captured in the following claim.

Claim 1.2. Let $f \in \text{FP}$. Then the function $f'(x, y) := (x, y, f(x))$ is in FP.

Proof. We begin by copying x into $|x|$ new memory words, starting at word $n = |(x, y)|$. Now we run a program P for f on this copy, but modify it by adding n to all memory addresses accessed. This would be the end of the proof if P was compatible. Otherwise, as in Claim 1.1 we need to address the fact that P was designed to start with $|x|$ words of memory; we can use the same solution. **QED**

Example 1.5. Given a natural x, computing x^4+x^2 is in FP. Indeed, we know that squaring is in FP by Exercise 1.2. Using Claim 1.2 we can compute $x \to (x, x^2)$ in FP. Composing this via Claim 1.1 with squaring and addition (Example 1.4) we obtain $x \to x^4 + x^2$.

1.5 Universal programs

Universal programs can simulate any other program on any input. These programs play a critical role in some results we will see shortly. They also have historical significance: before them machines were tailored to specific tasks. One can think of universal programs as epitomizing the victory of *software* over *hardware:* A single machine can be programmed to simulate any other machine. A universal program will thus take as input both another program Q and an input x for Q (in a standard encoding such as equation (1.1)).

Lemma 1.1. There is a word program U such that for any word program $Q \in [2]^*$ and input $x \in [2]^*$ we have: If Q computes $y \in [2]$ from x in time t then U on input Q and x computes y in time $c(t + |Q| + |y|)$.

Proof. To achieve the claimed fast simulation, it is convenient to represent word programs in a specific format. A program of size ℓ will be represented as a list of 4ℓ words. For each

instruction we have 4 words. The first has a constant range and denotes the type of the instruction. The other 3 words contain numbers in $[\ell]$ which represent the indices or the program counter in the instruction. E.g., for the instruction GOTO 15 IF $R[7] < R[2]$ the three numbers would be $15, 7, 2$.

The program U first computes this representation for Q. This can be done in linear time, by scanning the representation of Q in a read-once fashion, as in Exercise 1.1.

After that, U arranges the memory so that the first words contain the above representation of Q, followed by the starting configuration of Q on input x. Then U simulates the instructions of Q, one at the time. Each instruction of Q takes constant time to simulate. U first reads from the configuration the program counter, then reads the corresponding instruction of Q and the corresponding indices, if any. Then it reads or writes the corresponding words from the configuration of Q.

The remaining details are the same as for Claim 1.2. We map memory word i of Q to memory word $i + s$ of U, where s is the index of the first memory word in the configuration of Q. Also, while simulating instructions of Q, the instructions of U use the larger word size, but then U truncates the output to the word size of Q.

E.g., if the instruction was the memory-read $R[5] := M[R[3]]$ then U reads the content of $R[3]$ from the configuration. Let this be v. It then reads the memory at location $s + v$, and finally updates the content of $R[5]$ in the configuration, truncating it to the word size of Q.

At the end, U can copy the output y of Q in the first $|y|$ words. **QED**

Some variants of this result will be used. First, we will often use a "clocked" version of universal machines, where the machine starts with a time bound t in a register, and only carries the simulation for t steps. This can be easily implemented within the same time, by decreasing the clock at every instruction. Second, we can insist that U is a total function, that is, it takes as input any bit string. Not all strings would immediately correspond to valid programs, but it is easy to tell which ones are, and we can interpret any instruction which does not parse as, say, STOP. This way every string corresponds to a program.

1.6 The fastest algorithm for Factoring

A curious fact about some problems is that we know of an algorithm which is, in an asymptotic sense to be discussed now, essentially the fastest possible algorithm. This algorithm proceeds by simulating every possible program. When a program stops and outputs the answer, we can *check it* efficiently. Naturally, we can't just take any program and simulate it until it ends, since it may never end. So, we will use a clocked simulation. There is a particular simulation schedule which leads to efficient running times. For concreteness we state this for Factoring, but the same ideas will apply to other problems.

Theorem 1.1. There is a word program U that computes Factoring and has the following property: For any word program P for Factoring, and every input x, if P runs in time t on x then U runs in time $c_P t + c_P |x|^c \log t$.

This is a striking result. There is a single algorithm that does nearly as well as any other. In particular, Factoring is in FP iff U runs in power time. Moreover, the algorithm U is explicitly given: we can program it and run it. Naturally, the catch is that it is not practical. Indeed, the result nicely illustrates how "constant factors" can lead to impractical results because, of course, the problem is that the constant in front of t can be enormous, see Problem 1.2.

Proof. For $i = 1, 2, \ldots$ we simulate program i for 2^i steps. As discussed in section 1.5, a modification of Lemma 1.1 guarantees that for each i the simulation takes time $c2^i$. If program i stops and outputs y, then U checks in time $|x|^c$ if y is prime and divides x. For this we use that deciding primality is in P (see the notes).

Now let P be a word program computing Factoring. In the enumeration of programs, each program appears infinitely often. For example, one can add useless STOP instructions. So, if P has a description of length c_P, it also has descriptions of length $c_P + j$ for any j. (Note that adding a STOP instruction may take more than 1 bit, but again as discussed in section 1.5 anything which doesn't parse can be interpreted as STOP, so indeed we can have descriptions of length $c_P + 1, c_P + 2$,etc.) Let us take the shortest description which has length $\geq \log t$, which suffices to terminate the simulation. Let ℓ be the length of this description, and note $\ell \leq c_P + c \log t$.

The time spent by U for a fixed i is $\leq c \cdot 2^i + |x|^c$. Hence the total running time of U is

$$\leq \sum_{i=1}^{\ell} \left(c2^i + |x|^c \right) \leq c_P 2^{\ell+1} + \ell |x|^c \leq c_P t + c_P |x|^c \log t.$$

QED

1.7 On the word size

Having defined and illustrated the model, we return to an issue related to its definition: the word size. While the MALLOC operation allows to increase the word length, the latter does not get too big. Specifically, if a word program runs in time t it can only use $\leq t$ MALLOC instructions, and hence $\leq n + t$ memory words. In particular, the word length w will remain $\leq$ bit-len$(n + t)$. For some problems it is natural to start with slightly larger word size, like 10bit-len(n). (Think bit-len(n) as being approximately $\log n$.) This allows for example to compute an integer like n^2 in a register. However, one can simulate such a register using word size $\log n$ with only a constant factor increase in number of registers and running time. We now show this. For concreteness let us first define a model with larger word size.

Definition 1.9. We say that a word program computes y on input x *with word size* $b(n)$ if it computes y as in Definition 1.6, except that the start configuration in Definition 1.6 has word size $w := b(n)$ (as opposed to $w :=$ bit-len(n) in Definition 1.6). The classes FTime$(t(n))$ with word size $b(n)$ are defined correspondingly.

The next claim shows that increasing the word size by any constant only impacts the running time by a constant factor.

Claim 1.3. FTime($t(n)$) with word size $a \cdot$ bit-len$(n) \subseteq$ FTime($c_a t(n)$), for every $a \in \mathbb{N}$.

Proof. We only sketch this proof for $a = 2$. We use 4 w-bit registers to represent 1 register R' of $2w$ bits, where each w-bit register is *half-empty*: has non-zero bits only in the least significant $w/2$ bits:

$$R' = 2^{3w/2}R[3] + 2^w R[2] + 2^{w/2}R[1] + R[0].$$

To simulate addition between $2w$-bit registers, we first add the corresponding 4 w-bit registers. The results will fit in our w-bit registers, since the registers are half-empty. To maintain half-emptiness, we pick the most significant $w/2$ bits from the $R[0]$, add them to $R[1]$ and remove them from $R[0]$. Now $R[0]$ is half-empty. We do the same for $R[1]$, and so on.

A similar approach works for the other instructions, see Problem 1.3. **QED**

So we can typically assume we have such longer words when designing algorithms, but restrict to the smaller word size when analyzing.

What about larger words? One is tempted to brush aside details and consider *unbounded* word sizes, where all computation is performed over integers. This model is sometimes a useful abstraction when writing algorithms. However, some care is needed because the ability to perform arithmetic with unbounded (or very large) integers gives surprising power. As we now show, this allows us to factor integers efficiently!

Theorem 1.2. There is a function $b(n)$ s.t. factoring $\in$ FTime(n^c) with word size $b(n)$.

1.7.1 Factoring with large words: proof of Theorem 1.2

The proof showcases and ties together in an unusual but not overly complicated way many fundamental algorithms, including repeated squaring, recursion, binary search, and greatest common divisor. We break it up in a series of steps, interesting in their own right. We show that given an n-bit integer x we can compute the following in time n^c, *into a register* (i.e., we reach a configuration where the desired quantity is in a register):

(0) x (i.e., x is "loaded" into a register).

(1) Exponentiation y^x, given any integer y in a register (with any number of bits),

(2) Factorial $x!$

(3) A prime factor of x.

We note that (1) and (2) involve numbers with $\geq 2^n$ bits. Here is where we leverage the large word size. Next we prove these points in turn.

Exercise 1.3. Prove (0).

(1) This is done via *repeated squaring*. Namely, write

$$x = \sum_{i=0}^{b} 2^i x_i$$

and note

$$y^x = \prod_{i=0:x_i=1}^{b} y^{2^i},$$

where a product of no terms is defined to be 1. The numbers y^{2^i} can be computed by repeated squaring in time cb, because $(y^{2^i})^2 = y^{2^{i+1}}$. Then we can just multiply together those corresponding to $x_i = 1$.

(2) We give a recursive algorithm.

If x is odd we reduce to the case of x even using $x! = x \cdot (x-1)!$.

If x is even we use

$$x! = \binom{x}{x/2} (x/2)!^2.$$

The factorial in the rhs is computed recursively. To compute the binomial we use the binomial theorem to write, for any ℓ:

$$(2^\ell + 1)^x = \sum_{j=0}^{x} \binom{x}{j} 2^{\ell \cdot j}.$$

Note the binary representation of the rhs contains all the binomials $\binom{x}{j}$ spaced out. Specifically, $\binom{x}{j}$ starts at bit $\ell \cdot j$. Since each binomial is $\leq 2^x$ and thus takes $\leq x$ bits, picking $\ell := x$ ensures that the binary representations of the binomials do not overlap in the binary representation of the lhs. Hence, the bits for the binomial $\binom{x}{x/2}$ are an interval of the bits of the lhs.

For example, for $\ell = x = 6$, the binary representation of $(2^\ell + 1)^x$ is

$$1\ 000110\ 001111\ 010100\ 001111\ 000110\ 000001$$

and you can verify that block i of 6 bits is the binary representation of $\binom{6}{i}$. For example, $\binom{6}{4} = 1111$ in binary.

To compute these bits, note the lhs can be written y^x for $y = 2^\ell + 1$. Now, y can be computed efficiently by (1), and so again by (1) one can compute y^x efficiently. To extract the bits of y^x we divide by $2^{\ell x/2}$ (implemented using the bit-shift operation $>>$) then take the ℓ least significant bits (implemented as bit-wise And with $2^\ell - 1$).

Exercise 1.4. Prove (3). Guideline: The main idea is that computing the *greatest common divisor* (gcd) of x and $y!$ tells us whether x has a factor $\leq y$ or not. Use that the gcd of x and y can be computed in time linear in the bit length of $\min\{x, y\}$, see Fact B.2 and recall word programs have the modular remainder operation.

From (3), the factor can also be output to memory in linear time, concluding the proof of Theorem 1.2.

1.8 The grand challenge

The grand challenge of computational complexity theory is to prove tight lower bounds on the running times required to solve "natural" computational problems on general computational models, such as word programs. As mentioned in Chapter 0, our ability to prove such impossibility results appears limited. Indeed, it is consistent with our knowledge that all such problems are in FTime(cn). To point to specific problems, we can say that it is consistent with our knowledge that FTime(cn) contains Factoring, and all the problems in chapters 3 and 4.

For these problems, essentially only trivial impossibility results are known. For example, if a function depends on all the input bits, we need time $\geq n$ to read all the input. One can optimize the constants, see Problem 1.4. But even a bound of cn is unknown.

In this section we present some impossibility results for somewhat artificial problems. We will cover a variety of techniques which will be used later also for more natural problems, at the price of some extra restrictions on the computational model. For example, these techniques will be used in Theorem 6.20, stating that any algorithm solving the natural "sat" problem requires either much time or space.

1.8.1 Undecidability and diagonalization

First, we briefly discuss *undecidable* problems: problems that cannot be solved by computers, regardless of time. The main technique for showing this is known as *diagonalization* and will feature later as well. This technique can be illustrated informally via the following riddle:

Can there be a computer that answers every question correctly with a "yes" or "no"?

We claim that such a computer cannot exist. The argument is as follows. Suppose such a computer exists and call it M. Then ask it the following question:

Does M answer "no" to *this* question?

We claim that M cannot produce an answer to this question. Indeed, if M answers "yes," then since M is correct it should have answered "no" which is a contradiction. While if M answers "no" then it is not true that M answers "no" to the question, hence M answers "yes" which is again a contradiction. Hence, there is a contradiction either way, and we conclude that the computer M does not exist.

Perhaps surprisingly at first sight, this argument can be formalized. The main idea is simply that if we have a program, we can give it as input its own description.

Theorem 1.3. [An undecidable language] There is $f : [2]^* \to [2]$ that cannot be computed by any word program, regardless of time.

Proof. The problem f takes as input a description P of a program. The output is defined as follows. If P run on its own description outputs 1, then $f(P) := 0$. Otherwise (including if P isn't a valid description, or if P does not stop), $f(P) := 1$.

Assume towards a contradiction that a program Q exists that computes f. Consider the output $Q(Q)$ of Q on its own description. If $Q(Q) = 0$ then $f(Q) = 0$, which means that Q run on its own description outputs 1, i.e., $Q(Q) = 1$. This is a contradiction. Similarly, if $Q(Q) = 1$ then $f(Q) = 1$. Since Q is a valid program that stops, this means that Q run on its own description does not output 1. This is again contradiction. Hence, Q cannot exist. **QED**

Using similar ideas, one can prove that several other problems are undecidable. For example, the generic problem of *program verification*, i.e., does this program produce this output, is undecidable.

To understand why this technique is called diagonalization consider the matrix M where the rows are programs and the columns inputs, and entry P, x of M is 1 if $P(x)$ outputs 1, 0 if it outputs 0, and is blank otherwise. Then, the function f in the proof is designed to be different from the diagonal. That is, $f(Q) \neq M_{Q,Q}$.

1.8.2 The hierarchy of Time

Can you solve more problems if you have more time? For example, can you write a program that runs in time $t'(n) = n^2$ and computes something that cannot be computed in time $t(n) = n^{1.5}$? To set the stage, we note that the answer is yes for trivial reasons if we allow for functions with long output lengths. Indeed, in time n^2 we can output a string with $\geq cn^2$ ones, but in time $n^{1.5}$ we can output at most $n^{1.5}$ ones. This shows $\text{FTime}(n^2)$ is not contained in $\text{FTime}(n^{1.5})$.

The answer is more interesting and useful if the functions are boolean. Such results are known as *time hierarchies*, and a generic technique for proving them is *diagonalization.* The argument allows us to show that even increasing running time by a constant factor yields more computational power. There is however a technicality due to the fact that for some pathological running times $t(n)$ the result is not true:

Claim 1.4. There exists a function $t(n)$ such that for every function $t'(n) \geq t(n)$ we have $\text{Time}(t'(n)) = \text{Time}(t(n))$.

Proof. Let $t(n)$ be the so-called *busy-beaver* function: the maximum over all word programs P of length $\leq n$ and over all $x \in [2]^n$ of length n of the number of steps it takes for P to stop on input x. If P does not stop on x we define this number to be 0, so that the maximum is well defined.

Now let $f \in \text{Time}(t'(n))$ and let P be a corresponding program as in Definition 1.7. Note that P on inputs of length n stops within $t(n)$ steps by definition of $t(n)$. Hence $f \in \text{Time}(t(n))$. **QED**

The time bound $t(n)$ in Claim 1.4 is strange; in particular it is hard to compute. Next we show that for time bounds that are easy to compute, including all "standard" time bounds like $n, n \log n, n^2, 2^n$, etc., the time hierarchy does hold.

Definition 1.10. A function $t : \mathbb{N} \to \mathbb{N}$ is *time-constructible* if $t(n) \geq n$ and there is a word program that does not use memory reads or writes, and starting in a configuration with register $R[0] = n$ and all other registers equal to 0, reaches a configuration where $R[1] = t(n)$ in time $\leq a \cdot t(n)$ for some constant $a \in \mathbb{N}$.

Most time bounds of interest are time constructible. We illustrate a few examples.

Claim 1.5. The following functions of n are time-constructible:

1. an for any $a \in \mathbb{N}$
2. $n \cdot \text{bit-len}(n)$.
3. n^a for any $a \in \mathbb{N}$.
4. 2^n.

Proof. 1. The program loops from 0 to $n-1$. In each iteration, it increases $R[1]$ by a, and also calls MALLOC a times. Calling MALLOC is necessary to have word size large enough to store the value an.

4. Write 1 in $R[1]$. The program loops n times; at each iteration it doubles the value in $R[1]$ using the same approach in 1. for $a = 2$. Iteration i takes $c2^i$ steps. The total number of steps is thus $\sum_{i \in [n]} c2^i \leq c2^n$, as desired. **QED**

Exercise 1.5. Prove 2. and 3.

We can now state and prove the time-hierarchy theorem that for every time-constructible function t there is a constant a depending only t, and a function in $\text{Time}(at)$ that is not in $\text{Time}(t)$.

Theorem 1.4. [Time hierarchy] $\text{Time}(c_t t) \not\subseteq \text{Time}(t)$, for any time-constructible t.

Proof. We define f by giving a program for it. The input is itself a program Q. On that input, first use time-constructibility to compute $t(|Q|)$ into a register called *counter*. Then, make two copies of Q, e.g. using the program from Example 1.2 (but adding a delimiter as required by the next step). Now run the universal machine from Lemma 1.1 to simulate Q on its own description, but modify the simulation as follows. For every step simulated, decrease the counter. If Q outputs 1 before the counter stops, output 0 instead. Otherwise, including the cases where Q is not a correct program description, or the counter reaches 0 before Q stops, output 1. (The universal machine in Lemma 1.1 works in the same way even

if started with larger word length, as may arise after computing t; alternatively we can use the technique in Claim 1.1.)

The running time of this program is

$$at(n) + cn + ct(n)$$

where the terms correspond to computing t, duplicating Q, and running the simulation. In all, the program takes time $at(n)$ for a constant a.

We now show that such an f is not in Time($t(n)$). Suppose towards a contradiction that $f \in$ Time($t(n)$) and let P be a corresponding program as in Definition 1.7. Consider running P on its own description, as in Theorem 1.3. We have $P(P) = f(P)$, but the simulation in the definition of f stops on input P, and so $f(P)$ is the complement of $P(P)$, which is a contradiction. Hence $f \notin$ Time($t(n)$). **QED**

Corollary 1.1. Exp $\not\subseteq$ P.

Proof. Let $t(n) = 2^n$, which is time constructible by Claim 1.5. Then use that P $\subseteq$ Time($2^{n/2}$) $\subsetneq$ Time(2^n) $\subseteq$ Exp, where the first and last inclusion are by definition, and the middle separation is by Theorem 1.4. **QED**

1.9 Problems

Problem 1.1. [Encoding pairs] An encoding of pairs is a 1-1 map $f : [2]^* \times [2]^* \to [2]^*$. The encoding is *efficient* if both f and the inverse of f are in FP. Describe efficient encodings with the following upper bounds on $|f(x, y)|$ (recall Definition 1.3 of bit-len):

1. $2|x| + |y|$.
2. $2\text{bit-len}(|x|) + |x| + |y|$.
3. $2\text{bit-len}(\text{bit-len}(|x|)) + \text{bit-len}(|x|) + |x| + |y|$.

Prove that for any efficient encoding, efficient or not, and for all arbitrarily large n there are strings x, y with $|x| + |y| = n$ such that $|f(x, y)| \geq n + \log(n + 1) - 1 \geq n + \text{bit-len}(n) - c$, nearly matching the above.

Problem 1.2. Suppose there exists a such that Theorem 1.1 holds with the running time of U replaced with $(|P| \cdot t \cdot |x|)^a$. (That is, the dependence on the program description improved to power; on the other hand we allow even weaker dependence on t and $|x|$.) Prove that Factoring $\in$ FP.

Problem 1.3. Explain how to simulate multiplication between registers with $2w$ bits using registers with w bits.

Problem 1.4. Recall from Example 1.3 that Or is in Time(cn). Show that Or is not in Time($2n - c$).

1.10 Notes

> *Concluding, I view the mystery of the difficulty of proving (even the slightest non-trivial) computational difficulty of natural problems to be one of the greatest mysteries of contemporary mathematics. [305]*

The fundamental work on complexity is [103]. That work formalized computation for the first time, and discovered its self-referential ability, essentially inventing universal machines, the diagonalization technique, and proving Theorem 1.3. (The focus of [103] was slightly different, so the theorem is not quite stated there.) Of course, [103] did not come out of nowhere, but was in fact a reaction to a program of automating mathematics, and it built on logical formalizations of mathematics; and diagonalization has its roots in (and takes the name from) the proof that the real numbers are uncountable. Also, there are several previous works aimed at formalizing computation in various branches of science. See [206] for an account of this compelling history. Still, if a fundamental work must be picked, [103] seems appropriate, for it can be considered the first work on impossibility results about general computation.

The formalization of computation in [103] is in terms of recursive functions, not unlike modern functional programming languages. Many other equivalent formalizations came about later. The model of *tape machines,* discussed in Chapter 16, was introduced in [279]. This model is closer to computer hardware or imperative programming languages, and makes it a little more intuitive how to measure time and space in computation. Historically, complexity theory was developed over such tape machines, with complexity classes defined in [129]. This paper also proves a first hierarchy result, later improved, see Chapter 16.

The fact that the hierarchy theorem does not hold for some (non time-constructible) function $t(n)$ was proved in [274, 52]. Their result is stronger than Claim 1.4 because it yields a computable $t(n)$.

Word programs with unbounded word size are from [74], where a time hierarchy is also proved. The first paper to consider word programs (with bounded word size) seems to be [94], which has: "*What seems to be needed is a computational model that avoids the potential abuses of the unit cost random access machine, but allows for unit cost operations among operands of "reasonable" size, i.e., operands of size commensurate with the sizes of the numbers to be sorted.*" Several other works have put forth variants of word programs.

The power-time computability thesis is an efficient version of the computability thesis from logic. For a proof of a formalization of the latter, and related discussion, see [121].

Theorem 1.2 is from [246]. The algorithm for the greatest common divisor is very old, see [85], but there was no bound on its efficiency until [180], which provides the bound we need. Primality was placed in P in [10].

Problem 1.2 is from [277].

A brief history of the impossible. Historically, impossibility results have been some of the most consequential. An early example is the proof that the diagonal of a square is irrational, about 2500 years ago. This can be seen as a statement about computation, where

the computational model are rational numbers. The target object is "natural" or occurs "in the wild." Another famous example is that there is no closed-form algebraic expression for polynomial equations of degree 5. In the first half of the 20th century undecidability results in logic and mathematics such as Theorem 1.3 began to appear. Complexity theory takes a more quantitative angle on impossibility. It focuses on a resource such as time, considers problems that can be solved with enough of the resource, and tries to *bound from below the amount of resource that's needed*. Because of this, impossibility results are also known as "*lower bounds*." The first such results were proved in the second half of the 20th century, and many have not been improved since. The first results on tape machines are from the 60s, and so are the first results on circuits such as [211, 204], though circuit complexity started already in [249]. (Circuits are introduced in Chapter 2.) [275] provides a regional survey of research on lower bounds. A fresh wave of mathematical ideas and results in circuit complexity came in the 80's and 90's, discussed in Chapter 7. This wave soon "hit the wall," for example the "wall" of constant-depth circuits with mod 6 gates, see section §9.5, and stalled. Still the results we do know are very consequential. For example, a lower bound for small-depth circuits from [204] is often cited as a key factor in ushering the "AI winter" of the 1970s and 80s, see the quote from [222] in Chapter 7.

Two lines of work have moved more or less parallel to developments in boolean circuit complexity. The first is *algebraic* complexity, see Chapter 14. The second is a line of works that has devised increasingly sophisticated ways to leverage uniformity and diagonalization, producing results such as [225, 88, 90, 308] (see Theorem 6.20, Theorem 6.21, and Theorem 9.8) that do not have a non-uniform counterpart.

Another active line of research starting in the 70's has proposed several "barriers" to explain the lack of progress, see Chapter 18.

Chapter 2

Circuits

In this chapter we introduce *circuits*, a *non-uniform* model of computation which is close to the hardware of an actual computer. We can think of a circuit as a graph, or as a restricted program without control-flow operation, called straight-line program.

Definition 2.1. A *circuit* or *straight-line program* of size s with n input bits and m output bits is a sequence of s instructions where instruction $i \in [s]$ is of one of the following types:

- $g_i := b$, for $b \in [2]$,
- $g_i := t \wedge t'$, where each of the terms t and t' is either a previous instruction g_h with $h < i$ or a literal (an input bit x_i or its negation $\neg x_i$ for $i \in [n]$)
- $g_i :=t \vee t'$, where t and t' are as for $g_i := t \wedge t'$
- $g_i := \neg g_h$ where $h < i$.

The g_i are called *gates*. The *fan-out* of a gate is the number of occurrences in other instructions. A circuit computes a function on $[2]^n$ in the natural way, executing the instructions in order. The last m gates constitute the output.

We can also visualize a circuit as an acyclic directed graph with nodes representing gates and directed edges representing wires from the output of one gate to the input of the next.

As for word programs, the instruction set in Definition 2.1 is somewhat arbitrary and redundant. For example, we can write $g_i = 0$ as $g_i = x_j \wedge \neg x_j$, etc. Also, a single gate type, like NAnd which computes the negation of And, suffices to simulate all others (since we can write $\neg x$ as $x\mathrm{NAnd}x$). Less obviously, gates of the type $g_i = \neg g_j$ can be removed at little cost in size (Problem 2.2). Other types of gates are often useful. In fact, it is often useful to allow *any* gate on two inputs. Again, this comes at little cost in size, as we shall prove shortly in Theorem 2.1. The basis we picked is however natural in some contexts so we picked it for concreteness.

Example 2.1. Here's a circuit computing the Xor of two bits:

$$(x_0 \wedge \neg x_1) \vee (\neg x_0 \wedge x_1).$$

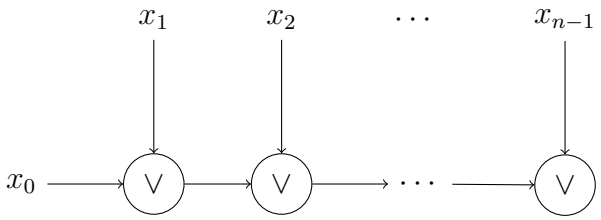
This circuit has size 3 and one output gate. The corresponding sequence of gates is:
$g_0 := x_0 \wedge \neg x_1$
$g_1 := \neg x_0 \wedge x_1$
$g_2 := g_0 \vee g_1$.

Definition 2.2. Let $g : \mathbb{N} \to \mathbb{N}$ be a function. We denote by $\text{CktSize}(g(n))$ the set of functions $f : X \subseteq [2]^* \to [2]^*$ for which there exists $n_0 \in \mathbb{N}$ s.t. for every $n \geq n_0$ there is a circuit of size $g(n)$ that computes f on every input in X of length n. We also define

$$\text{CktP} := \bigcup_d \text{CktSize}(n^d).$$

Note we only gave a definition for non-boolean functions, as the distinction between boolean and non-boolean will not be too important for circuits.

Example 2.2. The function Or on n bits, which outputs 1 if one of the bits is 1, and 0 otherwise, is in $\text{CktSize}(cn)$. A corresponding circuit can be drawn as



The same result holds for And.

Often we will consider computing functions on *small inputs*. In such cases, we can often use the following general result. In a way, the usefulness of the result goes back to the locality of computation. The result will be used extensively.

Theorem 2.1. Every function $f : [2]^n \to [2]^m$, for $n, m \geq 1$, can be computed by a circuit of size $\leq cm2^n/n$.

Proof. It suffices to consider $m = 1$. The case of larger m follows by applying the solution for $m = 1$ to each output bit of f. So let $f : [2]^n \to [2]$. We give several different proofs showcasing a wealth of ideas and yielding circuits for f of various sizes.

Size $n2^n$: Write

$$f(x) = \bigvee_{a \in [2]^n : f(x)=1} [x = a],$$

where recall $[x = a]$ is 1 if $x = a$ and 0 otherwise. In turn we can write

$$[x = a] = \left(\bigwedge_{i \in [n]: a_i = 1} x_i \right) \wedge \left(\bigwedge_{i \in [n]: a_i = 0} \neg x_i \right).$$

Using the circuit for And from Example 2.2, $[x = a]$ has circuits of size $\leq n$. Plugging these circuits in the expression for f above, and again using the circuit for Or from Example 2.2 we obtain a circuit of size $n2^n$.

Size $c2^n$:

Let f_0 and f_1 be functions on $n-1$ bits corresponding to setting the last input bit x_{n-1} of f to 0 or 1. Then we have

$$f(x) = (f_0(x_0, x_1, \ldots, x_{n-2}) \wedge \neg x_{n-1}) \vee (f_1(x_0, x_1, \ldots, x_{n-2}) \wedge x_{n-1}). \tag{2.1}$$

This shows that the maximum circuit size $S(n)$ of functions on n bits satisfies

$$S(n) \leq 2S(n-1) + 3,$$

also, $S(0) = 1$ as a function on 0 bits is just a constant. Opening up the recursion we have $S(n) \leq 2^n + 32^{n-1} + 32^{n-2} + \ldots + 3 \leq c \cdot 2^n$ (see Fact B.4).

Size $c2^n/n$: Perform the recursion in equation (2.1) until we have functions on k bits, for a k to be determined. This gives a "circuit" C' for f of size $c2^{n-k}$ where some of the gates compute functions on k bits corresponding to fixings of the last $n-k$ input bits of f. The savings in size comes from this cool observation: While there are 2^{n-k} gates computing functions on k bits, for a small k there are much fewer functions on k bits, so many of these gates actually computed the same function. So we can re-use the same circuit for many gates, leading to a better bound on the circuit size than we would get by continuing the recursion all the way to the base case. Here we critically use that the circuit model allows large fan-out, which is also cool (in other models this can't be done).

Specifically, there are 2^{2^k} functions on k bits. By an approach above we can compute each of them by a circuit of size $c2^k$. So overall all these functions can be computed by a circuit of size $c2^{2^k+k}$. Combining this circuit with C', we see that for any k there is a circuit for f of size

$$\leq c\left(2^{n-k} + 2^{2^k+k}\right).$$

To minimize this quantity we pick k that balances the two summands. In particular, for $k = \lceil \log n \rceil - c$ and $n \geq c$, the quantity is

$$\leq c\left(2^{n-\lceil \log n \rceil + c} + 2^{n/4 + \log n}\right) \leq c\left(2^n/n + 2^{n/2}\right) \leq c2^n/n.$$

QED

Exercise 2.1. [Simple indexing circuit] The Indexing problem: Given $x \in [2]^n$ and $i \in [2]^{\log n}$, where n is a power of 2, compute bit i of x. Prove that Indexing is in CktSize($cn \log n$). Use this to give another proof of the $cn2^n$ bound in Theorem 2.1.

Exercise 2.2. Prove that the sum of two n-bit integers is in CktSize(cn). Hint: Consider a circuit for the sum of three bits $f : [2]^3 \to [2]^2$, $f(x, y, z) := x + y + z$. Apply this result repeatedly.

2.1 The grand challenge for circuits

With regard to the the grand challenge, the situation for circuits is analogous to that for word programs. Even a bound of cn is unknown, e.g. for the problems mentioned in section §1.8, and we can only prove modest-looking bounds for smaller constants. The arguments are a little more complicated and useful for circuits. For example, Or has circuits of size $\leq n$ as we saw in Example 2.2, whereas it requires time $\geq 2n - c$ (Problem 1.4). So proving a circuit-size bound $\geq (1 + c)n$ requires a different function. We illustrate these techniques with a simple example; a line of works has optimized constants with increasingly complicated case-analyses, see the notes.

Theorem 2.2. Let $\text{Thr}_2 : [2]^n \to [2]$ output 1 iff at least two input bits are 1. Then $\text{Thr}_2 \notin \text{CktSize}(2n - c)$.

The proof uses the far-reaching *restrict-and-simplfy method.* A *restriction* is just a fixing of some input variables to constants. The method shows that *after applying a suitable restriction the circuit simplifies.* One then iterates the argument – fixing more input variables and simplifying more the circuit – until the circuit trivializes. The important point is that the circuit trivializes *faster* than the target function; in the end, the restricted circuit has become say a constant, but the restricted target function is not, concluding the proof. In this first, basic instantiation of the method, both the restriction and the simplification are modest: We show that we can fix *one* input variable and remove *two* gates. Albeit modest, this simplification is not trivial. This method will be studied in Chapter 7 for more specialized types of circuits exhibiting dramatic simplifications.

Proof. We prove it for the stronger circuit model where we allow as gates *any* functions on two bits. To apply the restrict-and-simplify method, suppose there was a circuit of size $\leq 2n - c$ computing Thr_2 on $n \geq 2$ bits. Let g be a gate connected to two literals ℓ_i, ℓ_j of distinct variables $x_i \neq x_j$. Such a gate exists when $n \geq 2$, for else the output is either a constant or a literal, which isn't true. We claim that either x_i or x_j appears in another gate. This is because ranging over the 4 possible fixings $x_i = b_i \in [2]$ and $x_j = b_j \in [2]$, the function Thr_2 ranges over 3 different functions of the remaining $n - 2$ variables (depending on the value $b_i + b_j \in [3]$), for $n \geq 2$. On the other hand, if both x_i and x_j were connected only to g, there would be only 2 possible resulting circuits, because g only takes two values. Hence, fixing either $x_i = 0$ or $x_j = 0$ allows us to remove two gates, while still computing Thr_2 on $n - 1$ bits. Any gate that took as input one of those two removed gates is changed to compute the resulting function. The exception is when we are removing the output gate, in which case we simplify the circuit to a single gate. We iterate this argument $\leq n-2$ times until the restricted circuit is a single gate. Since there are still ≥ 2 input variables unfixed, this is not possible. **QED**

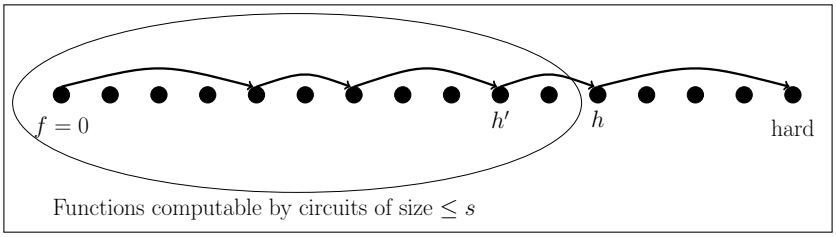


Figure 2.1: Illustration of the proof of Theorem 2.4.

As we said, we don't know how to prove that natural functions require large circuits. However, we can prove the *existence* of functions requiring large circuits, using counting arguments. (Counting arguments can be used for word programs as well, but Theorem 1.3 gave a more concrete problem.)

Theorem 2.3. There are functions $f : [2]^n \to [2]$ that require circuits of size $\geq c2^n/n$, for every n.

This bound matches Theorem 2.1.

Proof. First we claim that a circuit $C : [2]^n \to [2]$ of size $s \geq n$ can be described by $cs \log s$ bits. On the other hand there are 2^{2^n} functions on n bits. Since each circuit computes at most one function, if every function is computable by a circuit of size s we have

$$cs \log s \geq 2^n.$$

This is not possible if $s \leq c2^n/n$, as the lhs is $\leq c(2^n/n) \cdot n < 2^n$. **QED**

Exercise 2.3. Prove the claim.

We now turn to proving the analogue of the Time Hierarchy (Theorem 1.4) in the circuit model. Specifically, we'd like to show that increasing the size of circuits allows us to compute more functions. One can prove such a result by combining Theorem 2.3 and Theorem 2.1. But a stronger and more enjoyable argument exists.

Theorem 2.4. [Hierarchy for circuit size] For every n and $s \leq c2^n/n$ there is a function $f : [2]^n \to [2]$ that can be computed by circuits of size $s + n + c$ but not by circuits of size s.

Proof. Refer to figure 2.1. Consider a path from the all-zero function $f = 0$ to a hard function which requires circuits of size $\geq s$, guaranteed to exist by Theorem 2.3, changing the output of the function on one input at each step of the path. Let h be the first function that requires size $> s$, and let h' be the function right before that in the path. (Note $h \neq f$, since f has circuit size 1.) By construction, h' has circuits of size $\leq s$, and h can be computed from h' by changing the value on a single input. The latter can be implemented by circuits of size $n + c$. **QED**

Compared to the Time Hierarchy Theorem 1.4, the above circuit hierarchy is finer. The gap in the latter is of the order of the input length, whereas in the former it is of the order of the time bound. In fact, a surprising, much finer hierarchy for circuit size exists, where the gap is *just one gate*, see Problem 2.3.

2.2 Circuits vs. Time

In this section we explore the relationship between circuit and time classes. First, we show that circuits can simulate word programs. It isn't quite true that FP $\subseteq$ CktP, since a function in FP can have different output lengths for inputs of the same input length, whereas a circuit has a fixed number of output bit. But the containment is true for boolean functions (or more generally for functions whose output length depends only on the input length).

Theorem 2.5. P $\subseteq$ CktP.

Proof. Let $f \in \text{Time}(n^a)$ for a natural $a \geq 1$, let P be a corresponding word program of length ℓ, and let $n \geq c_{a,\ell}$. Because P runs in time $\leq n^a$, it uses $\leq n + n^a \leq 2n^a$ memory words, and the word length is $\leq c_a \log n$. Therefore, we can represent a configuration of P using $\leq c_{a,P} n^a \log n \leq n^{c_a}$ bits.

It suffices to build a circuit S that given as input a configuration $C = (i, m, w, R, M)$ computes the configuration C' that C yields. Indeed, we can then stack together n^a copies of such a circuit, and output the first bit of $M[0]$ in the last configuration. To build S we first construct a *memory-read circuit* that given a configuration and a pointer i to a memory word, computes $M[i]$. This can be done with size n^{c_a} using $c_a \log n$ copies of the indexing circuit from Exercise 2.1, one for each bit of $M[i]$.

Returning to the construction of S, we first apply ℓ such memory-read circuits to fetch the ℓ memory words $M[R[i]]$, $i \in [\ell]$ indexed by the ℓ registers. At this point, each bit of the next configuration C' can be computed from the fetched values and the information in C *excluding* the memory M. Overall, each bit of C' is a function of just $\ell c_a \log n$ bits. By Theorem 2.1, this bit of C' can be computed by a circuit of size $n^{c_{\ell,a}}$. And so all of C' can be computed by in size $n^{c_{\ell,a}}$. **QED**

When starting with a function in Time(n^a), this argument produces circuits of size n^b for a somewhat large b. This is mainly due to the use of the brute-force circuit from Theorem 2.1. On the other hand, we used no knowledge of the specific instructions of the word program; the same proof applies to any instructions as long as they are local. By contrast, one can argue that the specific instructions from Definition 1.1 have efficient circuit implementations and optimize the argument to obtain b about $2a$. But $b < 2a$ is open.

It is not known if Exp $\subseteq$ CktP. It is widely conjectured that Exp is not in CktP, and in fact that Exp requires circuits of exponential size. But all that we can prove is:

Theorem 2.6. For every k, Exp $\not\subseteq$ CktSize(n^k).

Proof. On an input x of length n, the function f is defined to be a function h applied to only the first $m := (k+1)\lceil \log n \rceil$ input bits. By Theorem 2.3, there is a such function h on m bits that requires circuits of size $\geq c2^m/m > n^k$ for $n \geq c_k$. In exponential time we can enumerate over all 2^{2^m} functions h on m bits. For each such function, we also enumerate over all circuits of size $\leq n^k$ to determine if it can be computed by such a circuit. If it can't, we have found h and hence f. Otherwise, we try the next h. Recall that, as in the proof of Theorem 2.3, a circuit of size n^k can be described with $\leq ckn^k \log n \leq n^{k+1}$ bits, for $n \geq c_k$. So the the entire enumeration is feasible in exponential time.

Note we have found f without even looking at the input x, only its length. Finally, we output $f(x)$. **QED**

2.3 Cosmological, non-asymptotic impossibility

Most of the results in this book are *asymptotic*, i.e., they only apply to sufficiently large inputs. As stated, these results don't say anything for inputs of a fixed length. For example, any function on 10^{100} bits (or any other constant-size set) is in Time(0), according to Definition 1.6. (The time bound only needs to apply to large enough inputs.)

However, it is important to note that all the proofs are *constructive* and one can work out non-asymptotic results. We state next one representative example when this was done. It is a problem in logic, an area which often produces very hard problems. On an alphabet of size 63, we consider formulas with first-order variables $x, y, \ldots$ that range over $\mathbb{N}$, second-order variables $S, T, \ldots$ that range over finite subsets of $\mathbb{N}$, the predicates "$y = x+1$" and "$x \in S$," and standard quantifiers, connectives, constants, and order relations, and set equality. For example one can write things like "every finite set has a maximum:" $\forall S \exists x \in S \forall y \in S, y \leq x$.

Deciding the truth of such formulas is extremely hard:

Theorem 2.7. Any circuit deciding the truth of logical formulas of length at most 610 in the language above requires $\geq 10^{125}$ gates.

So even if each gate were the size of a proton, the circuit would not fit in the known universe. The proof, referenced in the notes, is a variant of the techniques we saw in this chapter and Chapter 1.

2.4 Problems

Problem 2.1. Prove that Indexing (as defined in Exercise 2.1) is in CktSize(cn).

Problem 2.2. [Pushing negations to the input] Show that for any circuit $C : [2]^n \to [2]$ of size s there is an equivalent circuit C' of size $2s$ without Not gates $g_i := \neg g_j$.

Problem 2.3. [Finer hierarchy for circuit size]. Prove that for all n and $s \leq c2^n/n$, there is a function $f : [2]^n \to [2]$ that can be computed by circuits of size $s+1$ but not by circuits of size s.

2.5 Notes

The existence of hard functions for circuits via counting arguments, Theorem 2.3, goes back to [249], Theorem 7, "*Are most functions simple or complex?*" So does the brute-force computation of functions via circuits. The bound in Theorem 2.1 is from [194]. Further works have optimized the constants.

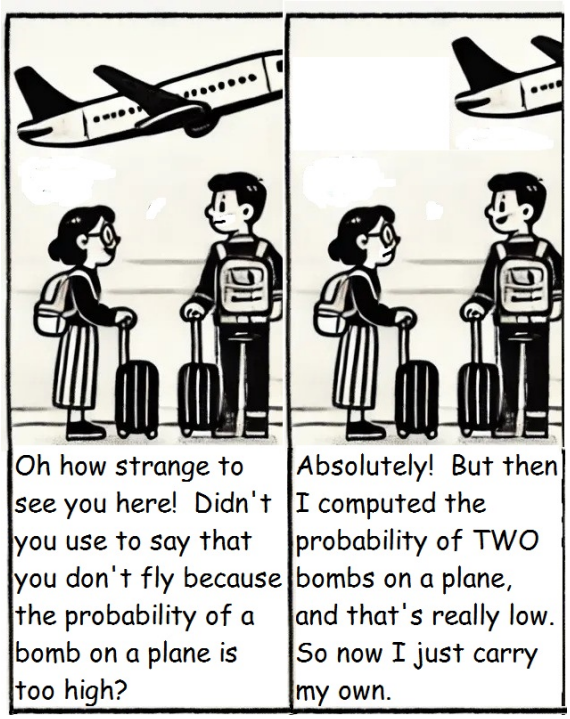
Theorem 2.7 is from [262], see the reference for the history of the result.

A number of papers have proved circuit lower bounds of the form an for various constants a, for functions in P. Over the full basis, i.e., when the computation gates can compute arbitrary functions (the model of Theorem 2.2), see [185]; over the basis $\neg, \vee, \wedge$, see [157] for slightly better constants.

The fine hierarchy for circuit size, Problem 2.3, is from [276].

Chapter 3

Randomness



Today, there's a significant challenge to the computability thesis. This challenge comes from... I know what you are thinking: *Quantum computing, superposition, factoring.* Nope. *Randomness.*

The last century or so has seen an explosion of randomness affecting much of science, and computing has been a leader in the revolution. Today, randomness permeates computation. Except for basic "core" tasks, using randomness in algorithms is standard, and considered "feasible." So let us augment our model with randomness.

Definition 3.1. A *randomized word program* is a word program with the extra instruction

- $R[i] :=$ Rand, where $i \in [\ell]$ and the ℓ is the number of registers of the program.

This instruction sets $R[i]$ to a uniform value in $[2]^w$ independent of all previous values, where w is the current word length.

We denote by *BPTime*$(t(n))$ *with error* $\epsilon(n)$ the set of functions f that map a subset $X \subseteq [2]^*$ to $[2]$ for which there exists a randomized word program P that, on any input $x \in X$ of length $\geq |P|$, stops within $t(|x|)$ steps and has

$$\mathbb{P}[P(x) = f(x)] \geq 1 - \epsilon(|x|),$$

where $\mathbb{P}$ denotes probability, taken over the values given by Rand.

The set FBPTime with error $\epsilon(n)$ is defined in the same way except that f outputs subsets and the condition is $\mathbb{P}[P(x) \in f(x)] \geq 1 - \epsilon(|x|)$.

If the error ϵ is not specified then it is assumed to be $1/3$. We also define

$$\text{BPP} := \bigcup_a \text{BPTime}(n^a),$$
$$\text{FBPP} := \bigcup_a \text{FBPTime}(n^a),$$

where BP stands for *bounded (error) probability.*

Exercise 3.1. Does the following algorithm show that deciding if a given integer x is prime is in BPP? "Pick a uniform integer $y \in [2..x-1]$. If y divides x output NOT PRIME, else output PRIME."

The introduction of randomness in our model raises several fascinating questions. Does randomness allow us to solve problems faster? Is BPP = P? Does randomness exist "in nature?" Do we need "perfect" randomness for computation? We begin to explore these questions in this chapter, and will return to them later.

First, in Definition 3.1 we have assumed that we have "perfect" randomness, i.e., that Rand returns a *uniform* value. "Imperfect" sources of randomness could be a better model for the randomness sources available "in nature." Problem 3.2 asks you to prove that a simple imperfect source, where the bits are biased, suffices for BPP. A large body of research has been devoted to greatly generalize this setting to pinpoint the imperfect sources of randomness that suffice for simulating BPP.

Nest, we begin by investigating the error bound in the definition of BPTime. This bound is somewhat arbitrary, because it can be reduced.

3.1 Error reduction for one-sided algorithm

We first discuss how to reduce the error in the important special case of *one-sided errors*. Suppose we have a randomized algorithm for a boolean f with the extra condition that it never makes mistakes when $f(x) = 0$. In other words:

- if $f(x) = 0$ then $\mathbb{P}[P(x) = 0] = 1$, while

- if $f(x) = 1$ then $\mathbb{P}[P(x) = 1] \geq 1 - \epsilon(|x|)$.

The corresponding classes are called RTime and RP.

The error probability of such a one-sided algorithm can be reduced by running it r times independently, and taking the Or of the outcomes. In the case $f(x) = 0$, all r runs will give 0, and so the Or will be 0. In the case that $f(x) = 1$, the prob. of a mistake is the prob. that all the r runs give 0. Because the runs are independent, this equals

$$\mathbb{P}[P(x) = 0]^r \leq \epsilon(|x|)^r.$$

For example, if ϵ is a constant, the probability decays exponentially fast with r.

In this one-sided case we can in fact even allow the error parameter ϵ to be very close to 1: If $\epsilon = 1 - p$ then the error bound is (by Fact B.7)

$$(1 - p)^r \leq e^{-pr}.$$

In particular, we can start with error prob. ϵ as large as $1 - 1/n^a$ and still drive it to 2^{-n^b}, for any constants a and b, at the price of running the algorithm $r = n^{a+b}$ times.

3.2 Error reduction for BPTime

We now discuss error-reduction for (two-sided) BPP algorithms. We have the following result.

Theorem 3.1. [Error reduction for BPP] Definition 3.1 of BPP remains the same if the error 1/3 is replaced with $1/2 - 1/n^a$ or $1/2^{n^a}$, for any constant a.

The idea in the proof is natural: You repeat the algorithm r times for a large r, but instead of computing Or as in section §3.1, you take a majority vote. The analysis of this is slightly more involved than for the case of one-sided algorithms. The math fact that powers the proof is this:

$$(1 - \epsilon)(1 + \epsilon) = 1 - \epsilon^2. \tag{3.1}$$

It's a non-trivial fact: We are multiplying a quantity less than 1 by another which is bigger than 1, and the product is less than 1. In the context of randomized algorithms, this will allow us to show that making $r/2$ mistakes is unlikely.

Proof of Theorem 3.1. Suppose that f is in BPP with error $\epsilon \leq 1/2 - 1/n^a$ and let P be a corresponding randomized word program. On an input x, let $1/2 - p$ be the error probability on x, where $p \geq 1/n^a$. Let us run P for $r := 8n^{2a} \cdot n^b$ times, each time with fresh randomness, and take a majority vote. This new algorithm makes a mistake only if at least $r/2$ runs of P make a mistake. (We consider a tie a mistake, so it doesn't matter how majority is defined for ties.)

To bound this error probability, consider a sequence $s \in [2]^r$ of possible outcomes, where $s_i = 1$ means that run i of the algorithm made a mistake, and denote by $w(s)$ the total

number of mistakes (i.e., the number of ones in s, or the weight). Let p_s be the probability of the sequence of outcomes, and note

$$p_s = (1/2 - p)^{w(s)}(1/2 + p)^{r - w(s)}.$$

Our goal is to bound

$$\sum_{s \in [2]^r : w(s) \geq r/2} p_s.$$

Each term p_s in the sum is $\leq (1/2 - p)^{r/2}(1/2 + p)^{r/2}$, because $w(s) \geq r/2$ and $p > 0$. The total number of possible outcomes is 2^r. So the above sum is

$$\leq 2^r (1/2 - p)^{r/2}(1/2 + p)^{r/2} = (1 - 2p)^{r/2}(1 + 2p)^{r/2} = (1 - 4p^2)^{r/2} \leq e^{-4p^2 r/2} \leq 2^{-n^b},$$

as desired. The second equality in the derivation is equation (3.1) and then we use Fact B.7 and our choice of r. **QED**

The probabilistic analysis in the proof is a special case of deviation bounds for the sum of random variables, a far-reaching topic which we discuss in section §B.6.1. The analysis above is more self-contained and elementary.

3.3 The power of randomness

In this section we explore various computing paradigms that are enabled by randomness and that allow us to solve problems faster than the best known deterministic algorithm. The techniques we see will be used many times later. We present two problems, both involving checking identities over various domains, a setting where the power of randomness really shines.

3.3.1 Verifying matrix multiplication

We denote by $\mathbb{F}_2$ the set $[2]$ equipped with addition and multiplication modulo 2. This is the finite field $\mathbb{F}_2$; finite fields are discussed more in section §B.7.

Definition 3.2. The MatrixMultiplicationVerification (MMV) problem: Given 3 square matrices A, B, C over $\mathbb{F}_2$, is $AB = C$?

This problem can be solved (deterministically) simply by performing the matrix multiplication AB and checking equality with C. But the fastest known algorithm to multiply two $d \times d$ matrices runs in time $\geq d^{2+c}$, leading to a running time $\geq n^{1+c}$ for MMV. By contrast, a simple randomized algorithm runs in linear time.

Theorem 3.2. MMV $\in$ BPTime(cn).

The proof uses a fact which is as simple as it is useful:

Fact 3.1. [*Random subset* or *random parity*] Suppose $x \in [2]^n$ is non-zero. Then the parity of a uniformly-selected subset of x is a uniform bit. Equivalently, for any non-zero $x \in \mathbb{F}_2^n$, if $A \in \mathbb{F}_2^n$ is uniform then $\sum_i A_i x_i$ is uniform in $\mathbb{F}_2$.

Proof of Theorem 3.2. Let $d = c\sqrt{n}$ be the dimension of the matrices. Pick U uniformly in $\mathbb{F}_2^d$. Compute the matrix-vector products $(AB)U$ and CU in time cn; and check if they are equal. To compute $(AB)U$ exploit the associativity rule $(AB)U = A(BU)$: compute BU first, then multiply by A.

If $AB = C$ the check always passes. In case $AB \neq C$, one of the rows is different. Say $(AB)_i \neq C_i$, where M_i is row i of M. The check only passes if $(AB)_i U = C_i U \iff ((AB)_i - C_i)U$. By the random parity Fact 3.1, the check passes with prob. $\leq 1/2$. We can reduce the error as in section §3.1. **QED**

3.3.2 Checking if a circuit represents zero

We now present a deceptively simple problem which is in BPP but is not known to be in P. This problem asks if a given integer is zero; the twist is that the circuit is not given explicitly, but succinctly, via a circuit.

Definition 3.3. An *integer circuit* of size s is a sequence of s instructions (or gates) $g_0, g_1, \ldots$ of the following types:

- $g_i := b$, with $b \in \{-1, 1\}$
- $g_i := g_j + g_k$, where $j, k < i$
- $g_i := g_j \times g_k$, where $j, k < i$.

All operations are over $\mathbb{Z}$. The circuit represents the integer computed by the last instruction.

Note an integer circuit has no inputs. It does not compute a function but represents a single integer.

Definition 3.4. The *Zero-Integer-Circuit* (ZIC) problem: Given an integer circuit, decide if it represents 0.

It is not known if ZIC $\in$ P. One cannot simply evaluate the circuit, since the integers computed at the gates can be doubly exponential in n, and thus require an exponential number of bits to represent in binary.

Exercise 3.2. Give an integer circuit with s gates representing an integer $\geq 2^{2^{s-c}}$.

Nevertheless, we have:

Theorem 3.3. ZIC $\in$ BPP.

The technique in the proof is to pick a random prime number p in a suitable range, compute the circuit modulo p, and check if the output is 0. In other words, we are computing in the finite field $\mathbb{F}_p$ consisting of the set of integers $[p]$ with operations modulo p, see section §B.7 for more on fields. Because we are working modulo p, the integers represented by the gates stay small and we can compute them in FP. If the original circuit represents 0 then it will also represent 0 modulo any prime. Less obviously, if for many primes p the circuit represents 0 modulo p then in fact it also represents 0 as an integer. This technique is called *fingerprinting* or *modular hashing*.

For the analysis we need to bound the number of primes in a specific interval. The so-called *Prime Number Theorem* gives tight bounds, but the next bound suffices and has a significantly simpler proof, given below.

Lemma 3.1. [Weak prime number theorem] The number $\pi(t)$ of primes in $[t]$ is $\geq ct/\log t$, for $t \geq c$.

We also need the following bound on the maximum integer represented by a circuit.

Claim 3.1. Prove that an integer circuit with s gates represents an integer whose absolute value is $\leq 2^{2^s}$.

Proof. By induction on s. For $s = 0$ the circuit consists of a constant only, whose absolute value is $1 \leq 2^1$. For the inductive step, if the output gate is $\times$, by induction its children represent integers $\leq 2^{2^{s-1}}$ in absolute value, so the output represents an integer $\leq 2^{2^{s-1}} \cdot 2^{2^{s-1}} = 2^{2 \cdot 2^{s-1}} = 2^{2^s}$ in absolute value. Ditto for the + gate. **QED**

We can now solve ZIC in BPP.

Proof of Theorem 3.3.. The algorithm picks a uniform prime p less than 2^{2n}, using the process explained below, then evaluates the circuit modulo p, and if the latter is 0 it outputs that the represented integer is zero; otherwise it outputs it is non-zero.

If the circuit represents 0 the algorithm is always correct.

If the circuit does not represent 0, then by Claim 3.1 it represents an integer in absolute value $\leq 2^{2^n}$. Such a value can be divisible by at most 2^n primes, since each prime is ≥ 2. Since by Lemma 3.1 the number of primes in the chosen range is $\geq 2^{2n}/n^c$, the probability of selecting a p that divides y will be $\leq 2^n \cdot n^c/2^{2n}$.

There only remains to pick a uniform prime. This can be done in FBPP as follows. Consider picking a uniform integer at most 2^{2n}. By the bound on the number of primes, Lemma 3.1, we have at least a $1/n^c$ chance of getting a prime, and if we do get a prime it will be a uniform prime by construction. We can test if a number is prime, and if it is not we can repeat. By the same analysis in section §3.1, if we try n^c times the chance of never getting a prime would be

$$(1 - 1/n^c)^{n^c} \leq 2^{-n}.$$

Summing the two error bounds above, we obtain that the overall error probability is (much) less than 1/3. **QED**

Finally, we prove the bound on the number of primes. The proof is a mathematical gem, and completely elementary. As we saw in section §1.7, factorials and binomials tell us a great deal about primes. So let us follow the lead.

Proof of Lemma 3.1. Consider

$$T := \binom{2t}{t}.$$

On the one hand

$$T \geq 2^{2t}/(2t+1) \tag{3.2}$$

because this is the largest binomial coefficient, and there are $2t+1$ such coefficients. A stronger bound holds (see Fact B.5) but we don't need it.

On the other hand we can prove an upper bound in terms of π. The main claim for this is that *any prime power dividing T is $\leq 2t$*. Let us first assume this and conclude the proof of Lemma 3.1. Denote by $e_p(x)$ the exponent of p in the prime decomposition of x, i.e., the largest i s.t. p^i divides x. Then

$$T = \prod_{p \leq 2t} p^{e_p(T)} \leq \prod_{p \leq 2t} 2t = (2t)^{\pi(2t)}.$$

Taking logarithms and using equation (3.2) we obtain

$$\pi(2t)(\log 2t) \geq 2t - \log(2t+1)$$

from which Lemma 3.1 follows.

There remains to prove the claim. For this we note that for $k \in \mathbb{N}$ and a prime p:

$$e_p(k!) = \sum_{i \geq 1} \left\lfloor \frac{k}{p^i} \right\rfloor. \tag{3.3}$$

The proof of this is left as exercise. Applying this to $T = \frac{(2t)!}{(t!)^2}$ we get

$$e_p(T) = \sum_{i \geq 1} \left\lfloor \frac{2t}{p^i} \right\rfloor - 2 \left\lfloor \frac{t}{p^i} \right\rfloor.$$

Now, if $x := t/p^i$ is of the form $j + \epsilon$ where $j \in \mathbb{N}$ and $\epsilon \in [0, 0.5)$ then $2\lfloor x \rfloor = \lfloor 2x \rfloor = 2j$ and each difference in the sum is 0; while if $\epsilon \in [0.5, 1)$ then $2\lfloor x \rfloor = 2j$, and $\lfloor 2x \rfloor = 2j+1$ and each difference is 1. In particular,

$$e_p(T) \leq \sum_{i \geq 1 : p^i \leq 2t} 1 \leq \log_p 2t.$$

And so $p^{e_p(T)} \leq p^{\log_p 2t} = 2t$. **QED**

Exercise 3.3. Prove Equation (3.3).

3.4 Does randomness really buy time? Is P=BPP?

In this section we discuss the relationships between Time and BPTime. We have the following basic inclusions.

Theorem 3.4. $\mathrm{P} \subseteq \mathrm{BPP} \subseteq \mathrm{Exp}$.

Proof. The first inclusion is by definition. The idea for the second inclusion is to brute-force the random choices in exponential time. A randomized word program M running in time n^a uses words of $\leq c_a \log n$ bits. Each Rand operation gives a uniform word, so the program uses $\leq n^a \cdot c_a \log n \leq n^{c_a}$ random bits. We can enumerate over all $2^{n^{c_a}}$ choices for these bits, run M for each of them, and output the majority of the outcomes. This takes time $2^{n^{c_a}}$. **QED**

Exercise 3.4. Generalize Theorem 3.4 by proving $\mathrm{Time}(t) \subseteq \mathrm{BPTime}(t) \subseteq \mathrm{Time}(c^{t \log t})$, for any function $t = t(n)$.

Now, two surprises. First, $\mathrm{BPP} \subseteq \mathrm{Exp}$ is the fastest deterministic simulation we can *prove*. On the other hand, what may come as a bigger surprise, despite the examples in section §3.3 it is widely believed that in fact P = BPP! And not only that, it is even believed that the overhead to simulate randomized computation deterministically is very small. The gap between our ability and common belief is abysmal.

One of the exciting developments of complexity theory has been the connection between the $\mathrm{P} =^? \mathrm{BPP}$ question and the "grand challenge" from section §1.8. At a high level, it has been shown that explicit functions that are hard for circuits can be used to *de-randomize* computation. In a nutshell, the idea is that if a function is hard to compute then its output is "random," so can be used instead of randomness. Quantitatively, the harder the function the more randomness it provides. At one extreme, we have the following striking connection:

Theorem 3.5. Suppose for some $a > 0$ there is a function in $\mathrm{Time}(2^{an})$ which on inputs of length n cannot be computed by circuits with $2^{n/a}$ gates, for all large enough n. Then $\mathrm{P} = \mathrm{BPP}$.

In other words, either randomness is useless for power-time computation, or else circuits can speed up exponential-time uniform computation! We will prove this in Chapter 11, Exercise 11.15.

While we don't know if P = BPP, we can prove that, like P, BPP has power-size circuits.

Theorem 3.6. $\mathrm{BPP} \subseteq \mathrm{CktP}$.

The proof is a nice application of the *probabilistic method,* where a mathematical object with a certain property is proved to exist by showing that a random object satisfies the property with non-zero probability. In this application, the object is an outcome for the randomness of the randomized word program, and the property is that it should work for *all* the inputs of a certain length. Once we have such an outcome, we can hardwire it.

To show that a random object satisfies the property, we drive the error to exponentially small by Theorem 3.1. Once the error is smaller than the number 2^n of inputs we are considering, we can afford a *union bound*. This technique of combining tail and union bounds is often used in the probabilistic method.

To make the proof more transparent it is convenient to adopt the following notation. For a randomized program P we write $P(x, R)$ for the execution of P on input x where $R = (R_0, R_1, R_2, \ldots)$ are the values given by Rand. The j-th instruction $R[i] :=$ Rand is replaced with $R[i] := R_j$, truncated to the current word length. Recall that the word length is dynamic; for simplicity we can assume that each R_i is longer than the maximum word length.

Proof. Let $f : X \subseteq [2]^* \to [2]$ be in BPP. By Theorem 3.1 we can assume that the error is $\epsilon < 2^{-n}$, and let P be a corresponding program. Note

$$\mathbb{P}_R\left[\exists x \in [2]^n : P(x, R) \neq f(x)\right] \leq \sum_{x \in [2]^n} \mathbb{P}_R[P(x, R) \neq f(x)] \leq 2^n \cdot \epsilon < 1,$$

where the first inequality is a union bound.

Therefore, there is a fixed choice for R that gives the correct answer for every input $x \in [2]^n$. This choice can be hardwired in the circuit, and the rest of the computation can be written as a circuit by Theorem 2.5. **QED**

3.5 The hierarchy of BPTime

In this section we prove the analogous of the Time Hierarchy Theorem 1.4 for BPTime.

Theorem 3.7. $\mathrm{BPTime}(c_t t(n+1)) \not\subseteq \mathrm{BPTime}(t(n))$, for any time-constructible non-decreasing t.

The proof uses a variant of the diagonalization technique which is known as *delayed diagonalization*. To explain how it fits in, recall that one of the issues that arises in the proof of Theorem 1.4 is that the input program may not run in a bounded amount of time. The solution is to use a *clocked simulation* of the program: We simulate the program, but if it runs for too long, we stop it. A more serious complication arises for randomized programs: The program may have acceptance probability very close to 1/2, whereas for BPTime we want probability $\geq 2/3$ (or bounded away from 1/2). No clocking mechanism works to fix this issue. Instead, we are going to compute the acceptance probability in *brute-force*, similarly to the proof of Theorem 3.4, and then "round it." This takes exponential time, so we can only afford it on very small inputs. This is where delayed diagonalization kicks in. It allows us to spread the effort over many different inputs, so that the expensive brute-force step becomes feasible.

Proof. We shall pad programs to increase the length of their description (this should not be confused with the padding mentioned in 1.1, which serves a different purpose). We write P_i

for a program P which is padded with i extra bits (as we did, say, in the proof of Theorem 1.1).

We define the following set of inputs X and function $f : X \to [2]$. The set X is a subset of padded programs P_i. The definition depends on (1) how i compares with $L(|P|)$, where $L(k) := 2^{t^c(k)}$, and (2) whether P is *bounded on an input x,* which means it runs in $\leq t(|x|)$ steps on x, and moreover $\mathbb{P}[P(x) = b] \geq 2/3$ for some $b \in [2]$.

If $i < L(|P|)$ then $P_i \in X$ if $P(P_{i+1})$ is bounded, and we define $f(P_i) := b$ if $\mathbb{P}[P(P_{i+1}) = b] \geq 2/3$.

If $i = L(|P|)$ then $P_i \in X$ always and $f(P_i)$ is intuitively defined as $\neg P(P)$. More precisely, if P on input P accepts with prob. $\geq 2/3$ within $t(|P|)$ steps, then $f(P_i) := 0$. In all other cases, including if P is not bounded on P, $f(P_i) := 1$.

If $i > L(|P|)$ then $P_i \notin X$.

First we claim that $f \in \text{BPTime}(c_t t(n))$. On input P_i of length n we first decide if $i = L(|P|)$. For this we compute $\lfloor \log^c i \rfloor$; this can be done in linear time . Then we use time-constructibility of t to compare this value with $t(|P|)$. Computing $t(|P|)$ takes time $\leq c_t t(|P|) \leq c_t t(n+1)$ because t is non-decreasing.

If $i < L(|P|)$ we run P on P_{i+1} and output its answer. This uses the universal program Lemma 1.1, and we don't even need to clock the simulation, since P is bounded on P_{i+1}. This takes $ct(|P_{i+1}|) = ct(n+1)$ steps.

If $i = L(|P|)$ we run P on P for $t(|P|)$ steps, for any possible choice of the coin tosses. This takes $\leq L(|P|)$ steps, which is less than n, and hence less than $t(n+1)$.

Next we claim that $f \notin \text{BPTime}(t(n))$. Suppose otherwise and let P be a corresponding program. Now consider the behavior of P on inputs $P_0 = P, P_1, P_2, \ldots, P_{L(|P|)}$. There are a couple of cases. First suppose P is bounded on all these P_i. Then all the P_i are in X we have the inequalities

$$\begin{array}{ccccccc} P(P_0) & & P(P_1) & & & & P(P_{L(|P|)}) \\ \| & \not\parallel & \| & \not\parallel & \ldots & \not\parallel & \| \\ f(P_0) & & f(P_1) & & & & f(P_{L(|P|)}) \end{array} .$$

These hold by correctness of P and the definition of f. However, also by definition of f we have $f(P_{L(|P|)}) \neq P(P_0)$, which is a contradiction.

Otherwise, let i be the largest s.t. P is not bounded on P_i. It cannot be that $i = L(|P|)$, because $P_{L(|P|)} \in X$ by definition and so P would not correctly compute f. In the other case, P is bounded on P_{i+1}. But then $P_i \in X$ by definition, and again this means P is not computing f on P_i correctly. **QED**

Whereas Theorem 1.4 gave a separation for total functions, here we only get a separation for partial functions; it is an open problem to prove a time hierarchy for total functions.

3.6 Problems

Problem 3.1. Prove $\text{BPTime}(10) \not\subseteq \text{Time}(n/10)$. Hint: Recall we allow partial functions.

Problem 3.2. Consider *biased* randomized word programs where the operation Rand returns one bit which, independently from all previous calls to Rand, is 1 with probability 1/3 and 0 with probability 2/3. Show that BPP does not change for sucbiased word programs.

Problem 3.3. [ZPP] Show that the following three conditions for $f : X \subseteq [2]^* \to [2]$ define the same class of boolean functions, called ZPP for *zero (error) probability*:

1. There exists a randomized algorithm that always computes f correctly and whose *expected* running time is n^a for some $a \in \mathbb{N}$ and large enough n.

2. There exists a randomized algorithm that on every input x outputs either $f(x)$ or '?' in time n^a for some $a \in \mathbb{N}$ and large enough n, and the probability of outputting '?' is $\leq 1/2$.

3. There exists a randomized *one-sided* algorithm for f and also a randomized one-sided algorithm for $1 - f$, both of which run in time n^a for some $a \in \mathbb{N}$ and large enough n. Here "one-sided" is as in section §3.1.

Problem 3.4. For a circuit C on n bits denote by p_C the probability $\mathbb{P}_x[C(x) = 1]$.

1. Show that the CAP (Circuit Acceptance Probability) problem is in FBPP: Given a circuit C and $\epsilon > 0$ written in unary (for example, as a string of $1/\epsilon$ ones) compute p s.t. $|p - p_C| \leq \epsilon$. Hint: Use Lemma B.1.

2. Show that the following decision version of CAP is in BPP: Given a circuit C, a number p (written in binary), and $\epsilon > 0$ written in unary, such that $|p_C - p| \geq \epsilon$, decide if $p_C \geq p$.

Problem 3.5. Assume P = BPP. Prove (see Problem 3.4 for the definition of CAP and p_C):

1. CAP is in FP.

2. Given a circuit C and ϵ written in unary s.t. $p_C \geq \epsilon$ we can compute $x : C(x) = 1$ in FP.

3. Given n in unary we can compute an n-bit prime in FP.

3.7 Notes

Randomized tape machines were studied already in [80]. Randomized complexity classes, including BPP, originate in [148].

Theorem 3.6 is from [8].

Theorem 3.2 is from [95].

Theorem 3.3 is from [243].

Theorem 3.5 is from [154].

The hierarchy for BPTime is from [138], where it is attributed to folklore. See [138] for further discussion.

Chapter 4

Reductions

One can relate the complexity of functions via *reductions*. This concept is so ingrained in common reasoning that giving it a name feels, at times, strange. For in some sense pretty much everything proceeds by reductions. In any algorithms textbook, the majority of algorithms can be cast as reductions to algorithms presented earlier in the book, and so on. And it is worthwhile to emphasize now that, as we shall see below, reductions, even in the context of computing, have been used for millennia. For about a century reductions have

been used in the context of undecidability in a modern way, starting with the incompleteness theorem in logic, whose proof reduces questions in logic to questions in arithmetic.

Perhaps one reason for the more recent interest in complexity reductions is that we can use them to relate problems that are tantalizingly close to problems that today we solve routinely on somewhat large scale inputs with computers, and that therefore appear to be just out of reach. By contrast, reductions in the context of undecidability tend to apply to problems that are completely out of reach, and in this sense remote from our immediate worries.

4.1 Types of reductions

Informally, a reduction from a function f to a function g is a way to compute f given that we can compute g. One can define reductions in different ways, depending on the overhead required to compute f given that we can compute g. The most general type of reduction is simply an *implication*.

General form of reduction from f to g:
If g can be computed with resources X **then** f can be computed with resources Y.

A common setting is when $X = Y$. In this case the reduction allows us to stay within the same complexity class.

Definition 4.1. We say that f reduces to g in X (or under X reductions) if

$$g \in X \Rightarrow f \in X.$$

A further special and noteworthy case is when $X = \mathrm{P}$; in these cases the reduction can be interpreted as saying that if g is easy to compute than f is too. But in general X may not be equal to Y. We will see examples of such implications for various X and Y.

If $g \notin X$ this definition trivializes, since then everything reduces to g. But the point of this definition is that it allows us to draw connections between problems *whose complexity is not known*. Still, it is sometimes useful to be more specific about how the implication is proved. For example, this is useful when inferring various properties of f from properties of g, something which can be obscured by a stark implication.

A more constrained type of reduction which is sometimes more suitable for a fine-grained analysis is the following:

Definition 4.2. We say that f *map reduces* to g in X (or via a map in X) if there is $M \in X$ such that $f(x) = g(M(x))$ for every x.

Exercise 4.1. Suppose that f map reduces to g in X.
(1) Suppose $X = \mathrm{FP}$. Show f reduces to g in X.
(2) Suppose $X = \bigcup_d \mathrm{FTime}(d \cdot n^2)$. Can you still show that f reduces to g in X?

In general, the harder the problems we are reducing the more constrained the type of reduction and the class X can be. In several interesting cases, map reductions with an extremely constrained class X suffice. But in other settings, the reductions we shall see are not even mapping reductions. In fact, our first example is not a mapping reduction.

4.2 Multiplication

Summing two n-bit integers is in CktSize(cn) (Exercise 2.2). But the smallest circuit known for multiplication has $\geq cn \log n$ gates. (For multiplication on word programs see the notes.) It is a long-standing question whether we can multiply two n-bit integers with a linear-size circuit.

What about squaring integers? Is that harder or easier than multiplication? Obviously, if we can multiply two numbers we can also square a number: simply multiply it by itself. This is a trivial example of a reduction. What about the other way around? We can use a reduction established millennia ago by the Babylonians. They employed the equation

$$a \cdot b = \frac{(a+b)^2 - (a-b)^2}{4} \tag{4.1}$$

to reduce multiplication to squaring, plus some easy operations like addition and division by four. In our terminology we have the following.

Definition 4.3. The Multiplication problem: Given two n-bit integers, output their product. The Squaring problem: Given an n-bit integer, output its square.

Theorem 4.1. Multiplication reduces to Squaring in linear circuit size, i.e., in $\bigcup_a \text{CktSize}(an)$.

Proof. Suppose C is a linear-size circuit computing Squaring. Then we can multiply using equation (4.1). Specifically, given a and b we use Exercise 2.2 to compute $a + b$ and $a - b$. (We haven't seen subtraction or negative integers, but it's similar to addition.) Then we run C on both of them. Finally, we again use Exercise 2.2 for computing their difference. It remains to divide by four. In binary, this is accomplished by ignoring the last two bits – which costs nothing on a circuit. **QED**

4.3 3Sum

Definition 4.4. The 3Sum problem: Given a list of integers, are there three that sum to 0?

It is easy to solve 3Sum in time $n^2 \log^c n$ with a word program, at least if the numbers have $\leq \log^c n$ bits: We can first sort the integers and then for each pair (a, b) we can do a binary search to check if $-(a+b)$ is also present. Let's convince ourselves that the bit length of the integers does not affect this algorithm:

Exercise 4.2. Prove 3Sum$\in$ Time$(n^2 \log^c n)$. Hint: Consider the case where each integer takes w bits.

3Sum has been conjectured to require quadratic time.

Definition 4.5. SubquadraticTime $:= \bigcup_{\epsilon>0}$ Time$(n^{2-\epsilon})$.

Conjecture 4.1. 3Sum $\notin$ SubquadraticTime.

One can reduce 3Sum to a number of other interesting problem to infer that, under Conjecture 4.1, those problems require quadratic time too.

Definition 4.6. The Collinearity problem: Given a list of points in the plane, are there three points on a line?

Theorem 4.2. 3Sum reduces to Collinearity in SubquadraticTime.

Proof. We give a proof assuming that the input to 3Sum consists of distinct integers, and using the fact that multiplication is in quasilinear time (which we do not prove). The case where some integer is repeated is left as exercise. Refer to Figure 4.1 for an illustration. We map instance $a_1, a_2, \ldots, a_t$ of 3Sum to the points

$$(a_1, a_1^3), (a_2, a_2^3), \ldots, (a_t, a_t^3),$$

and solve Collinearity on those points. Computing this map takes quasi-linear time.

Let $(x, x^3), (y, y^3)$, and (z, z^3) be three points with x, y, z all distinct. Note these points are on a line iff

$$\frac{y^3 - x^3}{y - x} = \frac{z^3 - x^3}{z - x}.$$

Because $y^3 - x^3 = (y - x)(y^2 + yx + x^2)$, this condition is equivalent to

$$y^2 + yx + x^2 = z^2 + zx + x^2 \Leftrightarrow (x + (y + z))(y - z) = 0.$$

This is equivalent to $x + y + z = 0$ because $y \neq z$.

Note that the Collinearity instance has length linear in the 3Sum instance, and the result follows. **QED**

We now give a reduction in the other direction: We reduce a problem to 3Sum.

Definition 4.7. The 3Cycle problem: Given the adjacency list of a directed graph, is there a cycle of length 3?

The fastest known algorithm for 3Cycle runs in time $\geq n^{1.4}$. We can show that an improvement follows if 3Sum $\in$ Time$(n^{1+\epsilon})$ for a small enough ϵ. We give a randomized reduction. While a deterministic reduction is also possible, the randomized reduction is simpler.

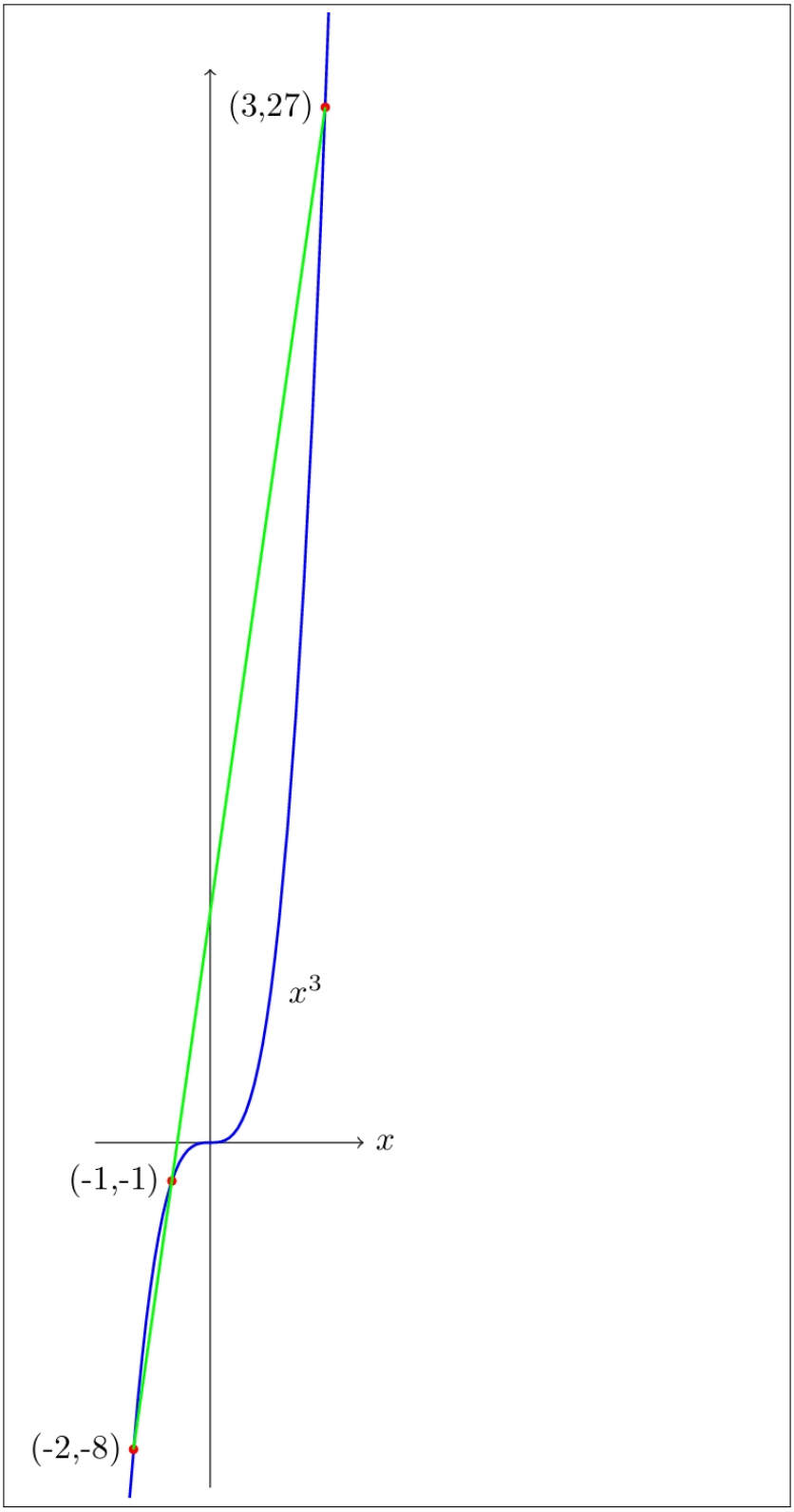


Figure 4.1: Illustration of the proof of Theorem 4.2 for the 3Sum instance $(-2,-1,3)$. Because $-2-1+3=0$, there is a straight line through the points $(-2,-2^3),(-1,-1^3),(3,3^3)$.

Theorem 4.3. 3Sum $\in$ Time$(t(n)) \Rightarrow$ 3Cycle $\in$ BPTime$(ct(n))$, for any $t(n) \geq n$.

Proof. We assume we are given a list of edges where each node is represented using $\log n$ bits. The reduction assigns random numbers r_x with $4 \log n$ bits to each node x in the graph. The 3Sum instance consists of the integers $r_x - r_y$ for every edge $x \to y$ in the graph. Its size is linear in the input length.

To verify correctness, suppose that there is a cycle

$$x \to y \to z \to x$$

in the graph. Then we have $r_x - r_y + r_y - r_z + r_z - r_x = 0$, for any random choices.

Conversely, suppose there is no cycle, and consider any three numbers $r_{x1} - r_{y1}, r_{x2} - r_{y2}, r_{x3} - r_{y3}$ from the reduction and its corresponding edges. Some node xi has unequal in-degree and out-degree in those edges. This means that when summing the three numbers, the random variable r_{xi} will not cancel out. When selecting uniform values for that variable, the probability of getting 0 is at most $1/t^4$.

By a union bound, the probability there there are three numbers that sum to zero is $\leq t^3/t^4 < 1/3$. **QED**

Many other clusters of problems exist, for example based on matrix multiplication or all-pairs shortest path.

4.4 Satisfiability

In this section we begin to explore an important cluster of problems not known to be in P. What's special about these problems is that in Chapter 5 we will show that we can reduce *arbitrary computation* to them, while this is unknown for the problems in the previous section.

The basic problem in this class is to determine, given a circuit C, if there is an input x s.t. $C(x) = 1$. In fact, for this chapter it suffices to work with circuits which are made of just a couple of layers of And, Or gates, with *unbounded* fan-in. This special case is defined next and linked to the general problem in the next Chapter 5.

Definition 4.8. A *literal* is a variable x or its negation $\neg x$. A *clause* is an Or of literals. A *conjunctive normal form formula (CNF)* is an And of clauses. A *kCNF* is a CNF where each clause has k literals. A CNF is *satisfiable* if there exists a boolean assignment to the variables on which the circuit outputs 1. We also call 0 *false* and 1 *true* in this context.

The 3Sat problem: Given a 3CNF ϕ, is there an assignment x s.t. $\phi(x) = 1$?

Example 4.1. The formula $(x \vee y \vee z) \wedge (z \vee \neg y \vee \neg x)$ is a 3CNF. This CNF is satisfiable because the assignment $x = 1, y = 0, z = 0$ gives $(1 \vee 0 \vee 0) \wedge (0 \vee 1 \vee 0) = 1 \wedge 1 = 1$.

On the other hand, the 3CNF $(x \vee x \vee x) \wedge (\neg x \vee \neg x \vee \neg x)$ is not satisfiable: no matter how x is set, one of the two clauses will be 0.

There are two main reasons why 3Sat is important. First, a number of important problems are routinely reduced to 3Sat and then solved using sat solvers. Second, we will show in the next Chapter 5 that is captures *arbitrary computation*.

One can solve 3Sat in exponential time via *brute-force*: Try all assignments to the variables. Despite much effort no significantly faster algorithm is known. In particular, the following is a leading conjecture of complexity theory.

Conjecture 4.2. 3Sat $\notin$ P.

In fact, stronger conjectures have been made, parameterized by the number of variables, basically asserting that brute-force search is the best that we can do.

Conjecture 4.3. [Exponential time hypothesis (ETH)] There is $\epsilon > 0$ such that there is no algorithm that on input a 3CNF ϕ with v variables and cv^3 clauses decides if ϕ is satisfiable in time $2^{(\epsilon+o(1))v}$.

Conjecture 4.4. [Strong exponential-time hypothesis (SETH)] For every $\epsilon > 0$ there is k such that there is no algorithm that on input a kCNF ϕ with v variables and cv^k clauses decides if ϕ is satisfiable in time $2^{(1-\epsilon+o(1))v}$.

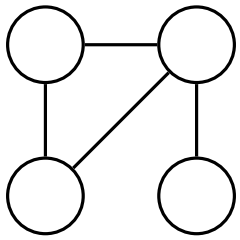
It is known that SETH $\Rightarrow$ ETH, but the proof is not immediate.

We now give reductions from 3Sat to several other problems. The reductions are in fact mapping reductions. Moreover, the reduction map can be extremely restricted, see Problem 4.4. In this sense, therefore, these reductions can be viewed as direct translations of the problems, and maybe we shouldn't really be thinking of the problems as different, even if they at first sight refer to different objects (formulas, graphs, numbers, etc.). More on this perspective is in Chapter 19.

4.4.1 3Sat to Clique

Definition 4.9. The Clique problem, given a graph G and an integer t, are there t nodes in G that are all connected? The latter is called a *clique* of size t.

Example 4.2. The following graph has a clique of size 3 but not of size 4:



Theorem 4.4. 3Sat reduces to Clique in P.

Proof. Given a 3CNF φ with k clauses, we construct a graph G with $3k$ nodes where we have a node for each literal occurrence. We then connect all except

(A) Nodes in same clause, and

(B) Contradictory nodes, such as x and $\neg x$.

The construction is in FP.

We claim that φ is satisfiable iff G has a clique of size k.

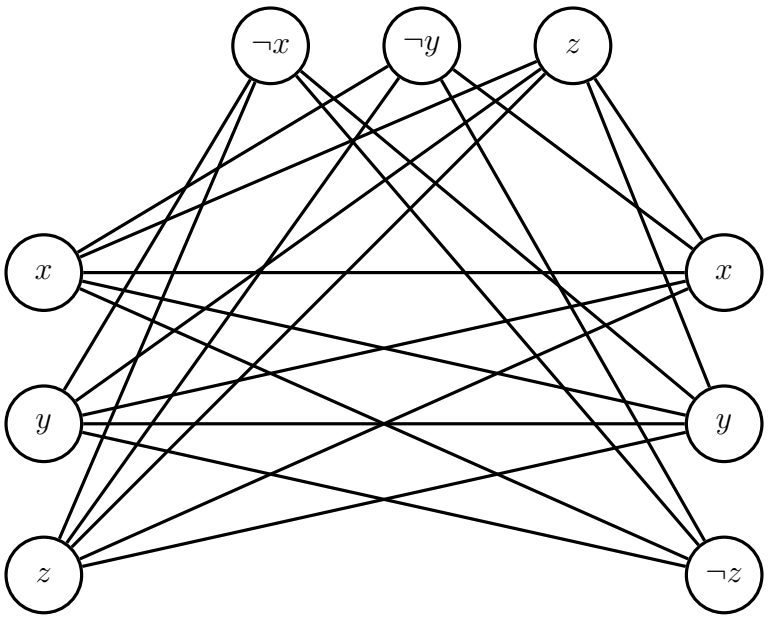
Only if: Given a satisfying assignment, collect exactly one node which is satisfied in each clause. This makes $t = k$ nodes. For any pair of such nodes, (A) does not hold by construction, and (B) because they correspond to an assignment.

If: Given a clique of size t, pick any assignment that makes the corresponding literals true. This is a valid definition by (B). Also, because of (A), there is at least one true literal in each clause. **QED**

Example 4.3. Consider

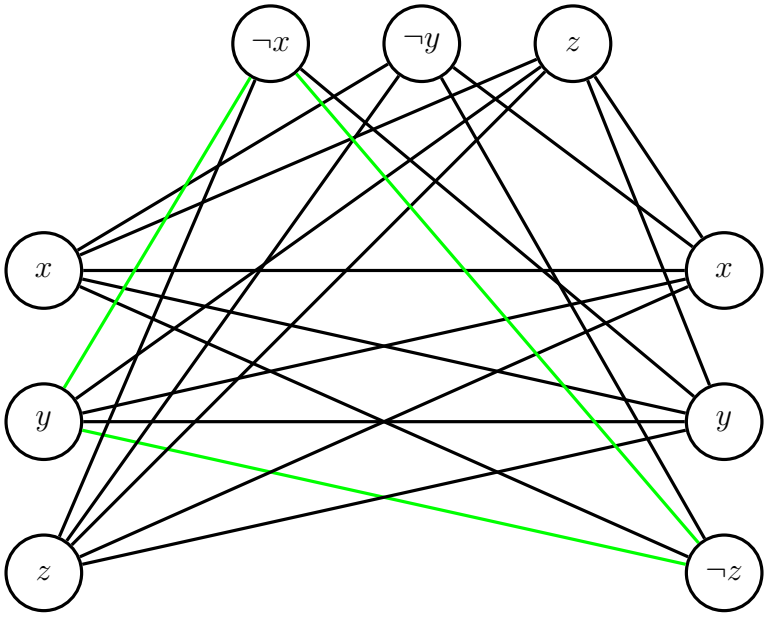
$$\varphi = (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z).$$

The corresponding graph G is:

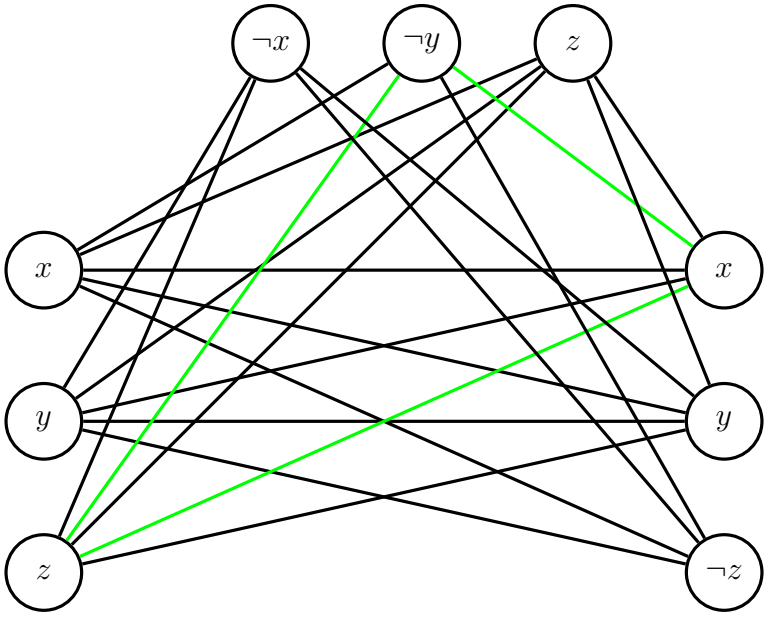


We seek cliques of size $t = k = 3$, a.k.a. triangles.

A satisfying assignment to φ is $x = 0; y = 1; z = 0$. The corresponding clique is shown next in green:



Another satisfying assignment is $x = 1; y = 0; z = 1$. The corresponding clique is shown next in green:



4.4.2 3Sat to Subset-Sum

Definition 4.10. The Subset-sum problem: Given n integers a_i and a target t, is there a subset of the a_i that sums to t?

Example 4.4. There is a subset of $5, 2, 14, 3, 9$ summing to $t := 25$ $(2 + 14 + 9 = 25)$. But there is no subset of $1, 3, 4, 9$ summing to $t := 15$.

Subset-sum is also a very interesting problems. If the numbers are small it can be solved in power time via dynamic programming. Hence the next reduction capitalizes on the magnitude of the numbers.

Theorem 4.5. 3Sat reduces to Subset-sum in P.

Proof. On input φ with v variables and k clauses we produce a list of numbers with $v + k$ digits. The most significant v correspond to variables; the other k to clauses. For each variable x include number a_x^T which has 1 in the digit corresponding to x, and a 1 in every digit of a clause where x appears without negation. Similarly, include number a_x^F which also has a 1 in the digit corresponding to x, and now a 1 in every digit of a clause where x appears negated.

Also, for each clause C, include twice the number a_C which has a 1 in the digit corresponding to C, 0 in others.

Set t to be 1 in first v digits, and 3 in rest k digits.

This construction is in FP.

Now suppose φ has satisfying assignment. Pick a_x^T if x is true, a_x^F if x is false. The sum of these numbers yield 1 in first v digits by construction. It also yields 1, 2, or 3 in each of the last k digits because each clause has a true literal. By picking appropriate subset of the numbers a_C we can reach t.

Conversely, given a subset, note that there is no carry in sum, because there are only 3 literals per clause. So digits behave "independently." For each pair a_x^T, a_x^F exactly one is included, otherwise would not get 1 in that digit. Define x true if a_x^T included, false otherwise. For any clause C, the a_C contribute ≤ 2 in that digit. So each clause must have a true literal otherwise sum would not get to 3 in that digit. **QED**

Example 4.5. Let $\varphi := (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$. The subset-sum instance is:

	var x	var y	var z	clause 1	clause 2	clause 3
$a_x^T =$	1	0	0	1	0	1
$a_x^F =$	1	0	0	0	1	0
$a_y^T =$	0	1	0	1	0	1
$a_y^F =$	0	1	0	0	1	0
$a_z^T =$	0	0	1	1	1	0
$a_z^F =$	0	0	1	0	0	1
$a_{c1} =$	0	0	0	1	0	0
$a_{c2} =$	0	0	0	0	1	0
$a_{c3} =$	0	0	0	0	0	1
$t =$	1	1	1	3	3	3

A satisfying assignment is $x = 0, y = 1, z = 0$. The corresponding subset is:

		var x	var y	var z	clause 1	clause 2	clause 3	
	$a_x^T =$	1	0	0	1	0	1	
	$a_x^F =$	1	0	0	0	1	0	
	$a_y^T =$	0	1	0	1	0	1	
	$a_y^F =$	0	1	0	0	1	0	
	$a_z^T =$	0	0	1	1	1	0	
	$a_z^F =$	0	0	1	0	0	1	
(2x)	$a_{c1} =$	0	0	0	1	0	0	(choose twice)
(2x)	$a_{c2} =$	0	0	0	0	1	0	(choose twice)
(2x)	$a_{c3} =$	0	0	0	0	0	1	
	$t =$	1	1	1	3	3	3	

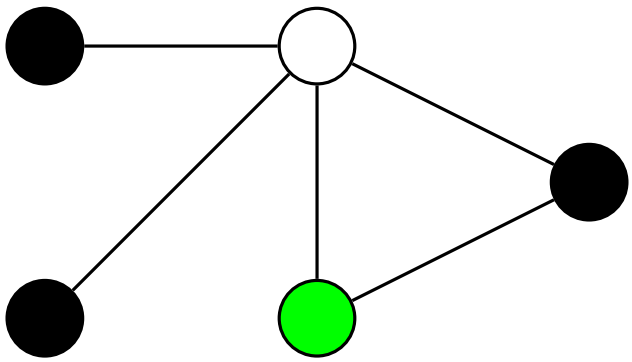
Another satisfying assignment is $x = y = z = 1$ with corresponding subset

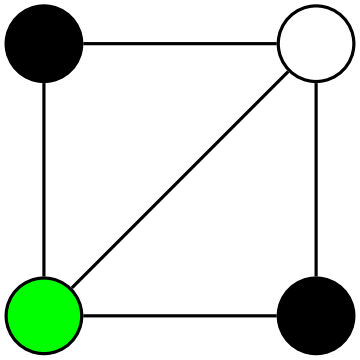
		var x	var y	var z	clause 1	clause 2	clause 3	
	$a_x^T =$	1	0	0	1	0	1	
	$a_x^F =$	1	0	0	0	1	0	
	$a_y^T =$	0	1	0	1	0	1	
	$a_y^F =$	0	1	0	0	1	0	
	$a_z^T =$	0	0	1	1	1	0	
	$a_z^F =$	0	0	1	0	0	1	
(2x)	$a_{c1} =$	0	0	0	1	0	0	
(2x)	$a_{c2} =$	0	0	0	0	1	0	(choose twice)
(2x)	$a_{c3} =$	0	0	0	0	0	1	
	$t =$	1	1	1	3	3	3	

4.4.3 3Sat to 3Color

Definition 4.11. For $k \in \mathbb{N}$, a *k-coloring* of a graph is a coloring of each node using at most k colors, such that no adjacent nodes have the same color. The kColor problem: Given a graph G, does it have a k coloring?

Example 4.6. The following graphs have a 3-coloring, shown:





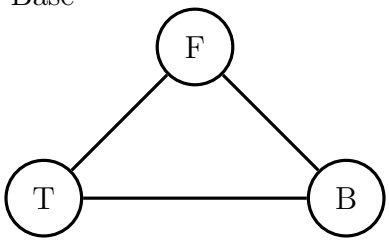
An example of a graph that cannot be 3-colored is a clique of size 4.

Theorem 4.6. 3Sat reduces to 3Color in P.

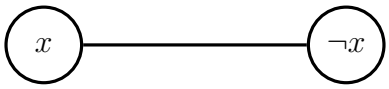
Proof. Given a 3CNF ϕ, we construct a graph G as follows.

Add 3 special nodes called the "palette" in a clique:

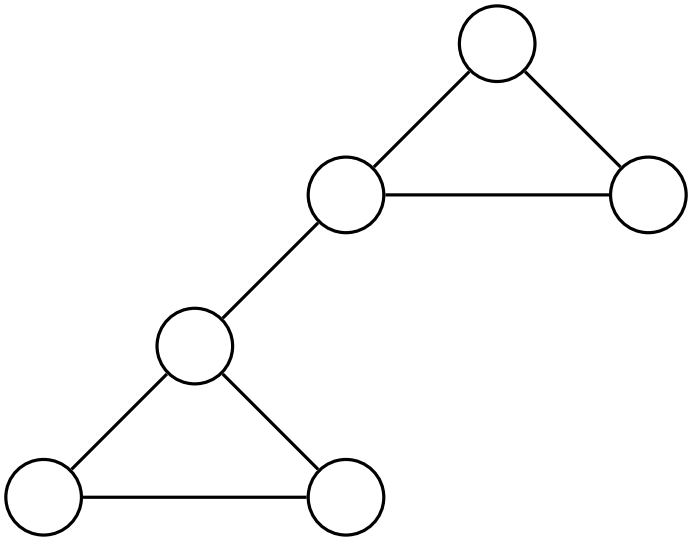
T = True
F = False
B = Base



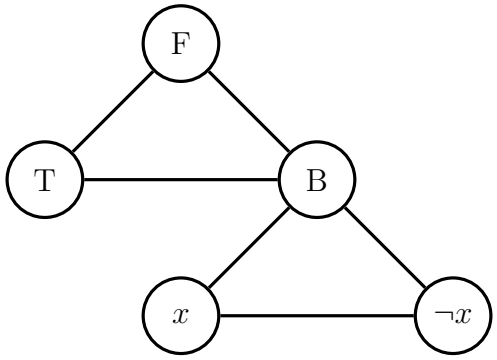
For each variable add 2 literal nodes with an edge between them



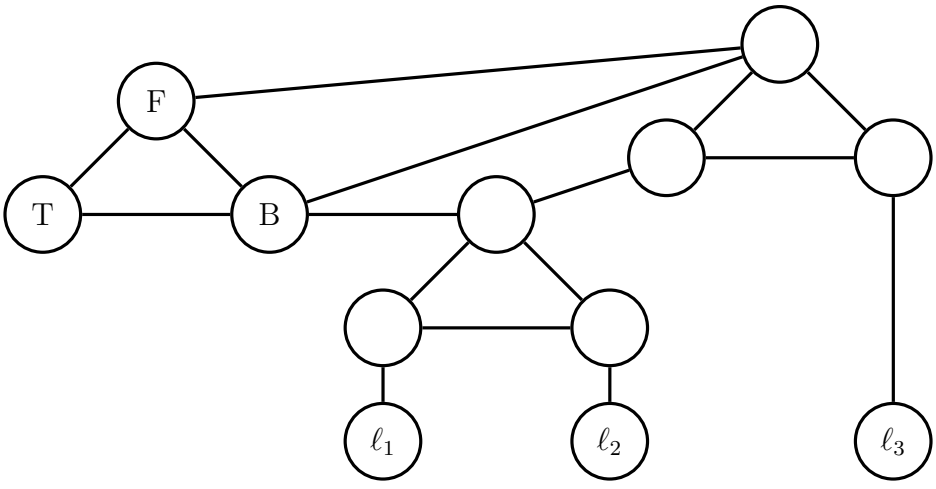
For each clause add the following gadget with 6 nodes



Connect each literal node to node B in the palette



For each clause (ℓ_1, ℓ_2, ℓ_3) connect the clause gadget to the palette and to the nodes ℓ_i as follows:

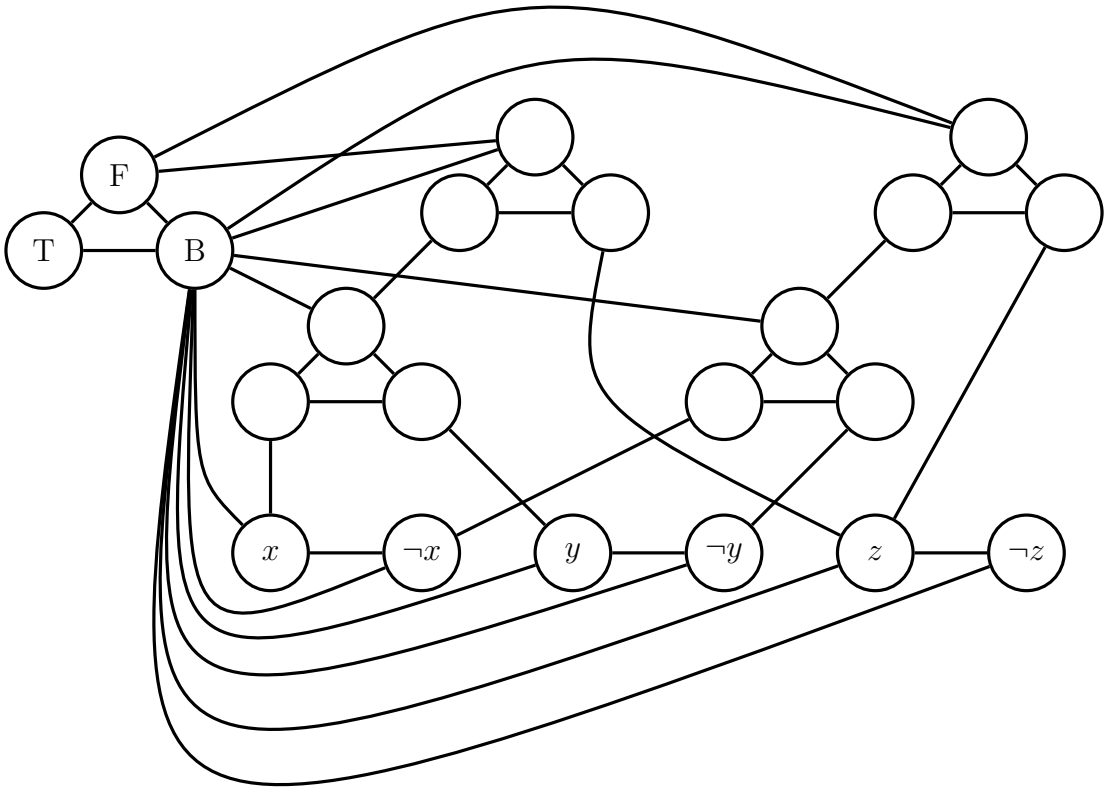


The construction of G is in P. We now prove that φ is satisfiable iff G is 3 colorable. We begin with some preliminary remarks. In the palette, T's color represents TRUE, and F's color represents FALSE. Note in a 3-coloring, all variable nodes must be colored T or F because they are connected to B. Also, x and $\neg x$ must have different colors because they are connected. So we can "translate" a 3-coloring of G into a true/false assignment to variables of φ.

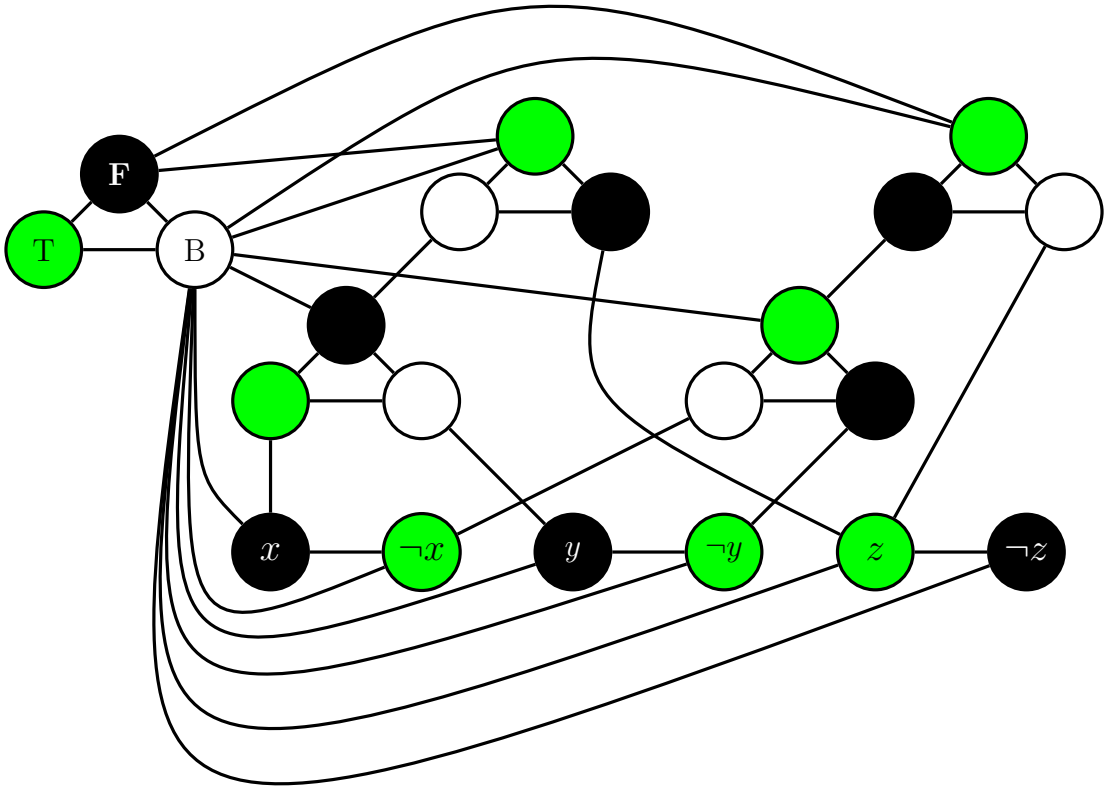
The important claim is that a clause gadget can be 3-colored iff any of the literals connected to it is colored True. This holds because each of the two triangles in a the clause gadget is computing "Or:" In a triangle, the top node is colored according to the Or of the two literals connected to the bottom two nodes in the triangle. For example, if the literals are both F, then the bottom nodes in the triangle must be colored T and B, and so the top is F.

The result follows. Given a satisfying assignment, we can pick the corresponding coloring of the literal nodes and extend it to a 3 coloring of the entire graph. Vice versa, given a 3 coloring of the graph we can infer an assignment to the variables and note that each clause has a true literal since each clause gadget is 3 colored. **QED**

Example 4.7. Let $\varphi := (x \vee y \vee z) \wedge (\neg x \vee \neg y \vee z)$. Then G is



A satisfying assignment is $x = y = 0$ and $z = 1$. The corresponding coloring is



Exercise 4.3. The problem System: Given a systems of linear inequalities, does it have an integer solution?

For example,

$$(x + y \geq z, x \leq 5, y \leq 1, z \geq 5) \in \text{System};$$
$$(x + y \geq 2z, x \leq 5, y \leq 1, z \geq 5) \notin \text{System}.$$

Reduce 3Sat to System in P.

Exercise 4.4. Reduce 3-Color to 4-Color in P.

Reductions in the opposite directions are possible, and so in fact the problems in this section form a cluster of *power-time equivalent* problems: Anyone of the problems is in P iff all the others are. We will see a generic reduction in the next chapter. For now, we illustrate this equivalence in a particular case.

Exercise 4.5. Reduce 3Color to 3Sat in P, following these steps:

1. Given a graph G, introduce variables $x_{i,d}$ representing that node i has color d, where d ranges in the set of colors $C = \{g, r, b\}$. Describe a set of clauses that is satisfiable if and only if for every i there is exactly one $d \in C$ such that $x_{i,d}$ is true.
2. Introduce clauses representing that adjacent nodes do not have the same color.
3. Briefly conclude the proof.

4.5 Power hardness from SETH

Assuming SETH it has been shown that for a number of prominent problems, such as *longest common subsequence,* or *edit distance*, the classical algorithms that run in quadratic time are the best possible: The problems cannot be solved in subquadratic time. In this section we prove a result of this flavor, which is in fact the building block of the results we just mentioned.

Definition 4.12. The Or-Vector problem: Given two lists A and B of strings of the same length, determine if there is $a \in A$ and $b \in B$ such that the bit-wise Or $a \vee b$ equals the all-one vector.

Similarly to 3Sum, the Or-Vector problem is in Time(cn^2), in fact the setting is slightly easier as it's more natural to assume that the bit vectors have the same length here, see Exercise 4.2. We can show that a substantial improvement would disprove SETH.

Theorem 4.7. Or-Vector $\in$ SubquadraticTime $\Rightarrow$ SETH is false.

Proof. Given a kCNF ϕ of size n with v variables and $d \leq n$ clauses, divide the v variables in two blocks of $v/2$ each. For each assignment to the variables in the first block construct the vector in $[2]^d$ where bit i is 1 iff clause i is satisfied by the variables in the first block. Call A the resulting set of vectors. Let $N := 2^{v/2}$ and note $|A| = N$. Do the same for the other block and call the resulting set B.

Note that ϕ is satisfiable iff Or-Vector(A, B) is true, i.e., $\exists a \in A, b \in B$ such that $a \vee b = 1^d$.

Constructing these sets takes time cNn: For each of the N assignments, we scan the input to construct the vector.

The instance length of Or-Vector is Nd. If Or-Vector $\in$ SubquadraticTime we can then solve the kCNF instance in time $cNn + (Nd)^{2-\epsilon} \leq c2^{n/2}n + 2^{n(2-\epsilon)/2}n^{2-\epsilon}$, for some $\epsilon > 0$. Taking $k = c_\epsilon$ then rules out SETH. **QED**

This proof is an interesting example of how we can connect different parameter regimes. SETH is stated in terms of exponential running times, but we can get an improvement by "scaling up" an algorithm for Or-Vector. In general, "scaling" parameters is a powerful technique in the complexity toolkit.

4.6 Search problems

Most of the problems in the previous sections ask about the *existence* of solutions. For example 3Sat asks about the existence of a satisfying assignment. It is natural to ask about computing such a solution, if it exists. Such non-boolean problems are known as *search problems*.

Next we show that in some cases we can reduce a search problem to the corresponding boolean problem.

Definition 4.13. The Search-3Sat problem: Given a satisfiable 3CNF formula, output a satisfying assignment.

Theorem 4.8. Search-3Sat reduces to 3Sat in FP.

Proof. We construct a satisfying assignment one variable at the time. Given a satisfiable 3CNF, set the first variable to 0 and check if it is still satisfiable with the assumed algorithm for 3Sat. If it is, go to the next variable. If it is not, set the first variable to 1 and go to the next variable. **QED**

Exercise 4.6. The Search-Clique problem: Given a graph G and an integer t s.t. G has a clique of size t, output one such clique. Show that Search-Clique reduces to Clique in FP.

Recall that in Theorem 1.1 we gave the "fastest" algorithm for Factoring. Similar algorithms exist for other search problems. The next exercise asks you to verify this for 3Sat.

Exercise 4.7. There is a word program U that computes Search-3Sat and has the following property: For any other word program P for Search-3Sat, and every input x, if P runs in time t on x then U runs in time $c_P t + c_p |x| \log^c |x| \log t$. Note and explain why the time bound is better than Theorem 1.1.

4.7 Gap-Sat: The PCP theorem

> *Furthermore, most problem reductions do not create or preserve such gaps. There would appear to be a last resort, namely to create such a gap in the generic reduction [...]. Unfortunately, this also seems doubtful. The intuitive reason is that computation is an inherently unstable, non-robust mathematical object, in the sense that it can be turned from non-accepting by changes that would be insignificant in any reasonable metric – say, by flipping a single state to accepting.*

One of the most exciting, consequential, and technical developments in complexity theory of the last few decades has been the development of reductions that create *gaps.*

Definition 4.14. The γ-Gap-3Sat problem: Solve 3Sat on formulas that are either satisfiable or are such that any assignment satisfies at most a γ fraction of clauses.

Note that 3Sat is equivalent to γ-Gap-3Sat for $\gamma = 1 - 1/n$, since a formula of size n has at most n clauses. At first sight it is unclear how to connect the problems when γ is much smaller. Surprisingly, and contrary to the quote above, it is possible to obtain a constant γ. This result is known as the PCP theorem, where PCP stands for probabilistically-checkable-proofs. The connection to proof systems will be discussed in Chapter 10.

Theorem 4.9. [PCP] 3Sat map reduces to $(1-c)$-Gap-3Sat in P.

This result has several exciting consequences. As mentioned before, we shall prove in Chapter 5 that arbitrary computation can be reduced to 3Sat. Hence the PCP theorem gives us a way to *robustify* computation and make it more stable. This is discussed more in Chapter 5.

Another consequence of the PCP Theorem 4.9 is that, if we believe Conjecture 4.2, not only we cannot determine if a given 3CNF is satisfiable, but we can't even distinguish the case in which every clause can be satisfied from the case in which at most $1-c$ fraction of clauses can be satisfied. A consequence of this is that we cannot compute an *approximation* to the maximum number of clauses that can be satisfied. In this sense, 3Sat is *inapproximabile.* It has been a major line of research to obtain tight inapproximability results for a variety of problems. This is well-motivated by the fact that we often don't need to compute an optimal solution, but a good enough solution will do. Similar inapproximability results can be established for other problems such as 3Color and Clique. For some problems such as Clique this follows from the reduction we saw, as the next exercise asks you to verify.

Definition 4.15. The γ-Gap-Clique problem: Solve Clique on pairs (G, t) where G either has a clique of size $\geq t$ or has no clique larger than γt.

Exercise 4.8. Assuming Theorem 4.9, reduce 3Sat to γ-Gap-Clique for a constant $\gamma < 1$.

In general, however, reductions among problems may not preserve gaps.

4.8 Problems

Problem 4.1. The problem H: The input is a directed graph with a special source node s, m destination nodes $t_1, t_2, \ldots, t_m$, and a subset B of *collapsing nodes*. Decide whether there are m paths from s to each of the destination nodes. The paths can share edges, but any two paths entering a collapsing node must leave through the same outgoing edge.

Reduce 3Sat to H in P.

Problem 4.2. The Quad-Sys problem: Given a system of quadratic equations over $\mathbb{F}_2$, decide if it has a solution. Reduce 3Sat to Quad-Sys in P.

Problem 4.3. Reduce Search-3Color to 3Color in P.

Problem 4.4. Specify suitable encodings of 3Sat and 3Color and prove that 3Sat map reduces to 3Color in ProjectionsP (see Definition 5.3).

4.9 Notes

Circuits of size $cn \log n$ for multiplication were obtained in [130]. It has been shown in [9] that this size is tight based on a conjecture in network coding. The paper [244] gives a linear-time algorithm for multiplication on a "storage modification machine." It *seems* this algorithm can be written as a word program, but I don't know this has been verified. See also discussion in section 4.3.3.C in [173].

Following [73] (discussed in the next chapter), [165] established reductions from satisfiability of (general) boolean formulas to 21 problems, including 3Sat, Clique, Cover-by-vertexes, 3Color, and Subset Sum. This opened the floodgates: The web of reductions from 3Sat is immense, see [101] for a starter, or the list on wikipedia: https://en.wikipedia.org/wiki/List_of_NP-complete_problems. Amusingly, among these problems are (generalized versions of) several popular videogames, including *Tetris*, *Lemmings*, etc. For an exposition of this type of results see the video https://www.youtube.com/watch?v=oS8m9fSk-Wk.

For videos covering the reductions in section §4.4, watch videos 29, 30, 31, and 32 from https://www.ccs.neu.edu/home/viola/classes/algm-generic.html (the videos use the terminology "polynomial time" instead of "power time" here).

The ETH and the SETH are from [151] and [153]. Again, a large number of reductions involving these hypotheses exists. In particular, tight hardness results based on SETH have been established for several well-studied problems, including longest-common subsequence [6] and edit distance [36]. See also [35].

The web of reductions of 3Sum, including Theorem 4.2, was first spun in [98] and has grown ever since. Theorem 4.3 is from [295].

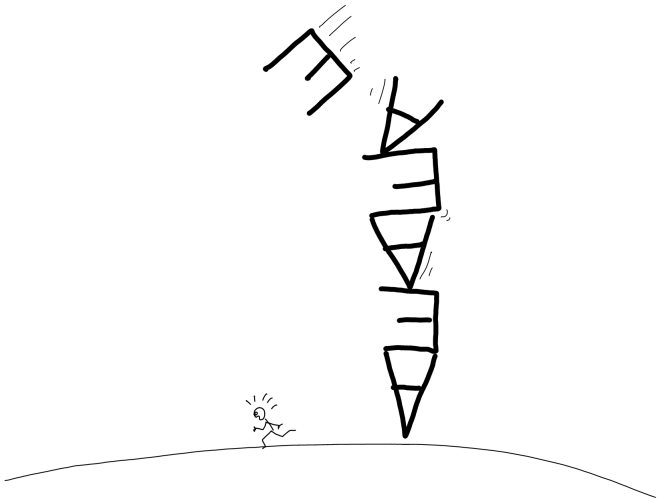
Theorem 1.1 is from [184].

The quote at the beginning of section §4.7 is from [220]. The PCP theorem as stated in Theorem 4.9 is from [26]. A sequence of exciting works preceded and followed it. For an account, as well as a proof of the PCP theorem, see [25].

Problem 1.2 is from [277].

Chapter 5

Nondeterminism



In this chapter we show how to reduce *arbitrary computation* to 3Sat (and hence to the other problems we showed in Section §4.4 3Sat reduces to). While in Chapter 4 we already showed that some problems like 3Color can be reduced to 3Sat, we now show how to reduce *any* problem computable in a certain class to 3Sat (the class includes 3Color and many other problems). This is a *generic, model-based* form of reducibility.

What powers everything is the following landmark and, in hindsight, simple result which reduces circuit computation to 3Sat. Specifically, given a circuit we can construct a formula whose satisfiability is equivalent to the output of the circuit.

Theorem 5.1. Given a circuit $C : [2]^n \to [2]$ with s gates we can compute in FP a 3CNF formula φ_C in $n + s$ variables such that for every $x \in [2]^n$:

$$C(x) = 1 \Leftrightarrow \exists y \in [2]^s : \varphi_C(x, y) = 1.$$

The key idea to *guess computation and check it efficiently, using that computation is local.* The additional s variables one introduces contain the values of the gates during the computation of C on x. We simply have to check that they all correspond to a valid

computation, and this can be written as 3CNF because each gate depends on at most two other gates.

Proof. Introduce a variable y_i for each computation gate g_i in C. The value of y_i is intended to be the value of gate g_i during the computation. Whether the value of a gate g_i is correct is a function γ of 3 variables: y_i and the ≤ 2 gates that input g_i, some of which could be input variables. This γ can be written as a 3CNF by the first constructions in the proof of Theorem 2.1. (We can apply that construction to write $\neg\gamma$ as a 3DNF, then complement that to obtain a 3CNF for γ.) Finally, take an And of all these 3CNFs, and add clause y_o for the output gate g_o. **QED**

Exercise 5.1. Write down the 3CNF for the circuit in Example 2.1, as given by the proof of Theorem 5.1.

We can combine Theorem 5.1 with the simulation of word programs by circuits (Theorem 2.5) to reduce time to 3Sat. We present this in Theorem 5.3. However, this simulation has a power loss, and using it we end up reducing time t to 3CNFs of size $\geq t^2$. Later in section §5.3 we obtain a quasi-linear simulation using an enjoyable argument which bypasses Theorem 2.5.

In fact, these simulations apply to a more general, *non-deterministic*, model of computation. We define this model next.

5.1 Nondeterministic computation

In the concluding equation in Theorem 5.1 there is an $\exists$ quantifier on the right-hand side, but there isn't one on the left, next to the circuit. However, because the simulation works for every input, we can "stick" a quantifier on the left and have the same result. The resulting circuit computation $C(x, y)$ has two inputs, x and y. We can think of it as a *non-deterministic* circuit, which on input x outputs 1 iff $\exists y : C(x, y)$. The message here is that – if we allow for an $\exists$ quantifier, or in other words consider nondeterministic computation – efficient computation is *equivalent* to 3CNF! This is one motivation for formally introducing a *nondeterministic* computational model.

Definition 5.1. NTime($t(n)$) is the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a word program P that for every $x \in X$ of length $\geq |P|$ satisfies:

- $f(x) = 1$ iff $\exists y \in [2]^{t(n)}$ such that $P(x, y) = 1$, and
- $P(x, y)$ stops within $t(n)$ steps for every $y \in [2]^{t(n)}$.

We also define

$$\text{Quasi-Linear-NTime} := \bigcup_{d \geq 1} \text{NTime}(n \log^d n),$$

$$\text{Non-deterministic-Power-Time, NP} := \bigcup_{d \geq 1} \text{NTime}(n^d),$$

$$\text{Non-Deterministic-Exponential-Time, NExp} := \bigcup_{d \geq 1} \text{NTime}(2^{n^d}).$$

Note that the running time of P is a function of $|x|$, not $|(x, y)|$. This difference is inconsequential for NP, since the composition of two powers is another power. But it is important for a more fine-grained analysis.

We refer to a word program as in Definition 5.1 as a *nondeterministic program*, and to the y in $P(x, y)$ as the *nondeterministic choices,* or *guesses,* of the machine on input x.

We can also define NTime in a way that is similar to BPTime, Definition 3.1. That is, we could use randomized word programs, and say that the output is 1 if the probability of outputting 1 is > 0. The instruction Rand in this case should be better be called Guess. The two definitions are equivalent for typical time bounds (but $t(n)$ is always easy to compute from the input length in Definition 5.1, whereas it could be hard to compute in the other definition). My choice for BPTime is motivated by the identification of BPTime with computation that is actually run. For example, in a programming language one uses an instruction like Rand to obtain random values; one does not think of the randomness as being part of the input. By contrast, NTime is a more abstract model, and the definition with the nondeterministic guesses explicitly laid out is closer in spirit to a 3CNF.

All the problems we studied in Section §4.4 are in NP.

Fact 5.1. 3Sat, Clique, SubsetSum, and 3Color are in NP.

Proof. For a 3Sat instance f, the variables y correspond to an assignment. Checking if the assignment satisfies f is in P. This shows that 3Sat is in NP. **QED**

Exercise 5.2. Finish the proof by addressing the other problems in Fact 5.1

We can think of NP as the problems which admit a solution that can be verified efficiently, namely in P. For example for 3Sat it is easy to verify if an assignment satisfies the clauses, for 3Color it is easy to verify if a coloring is such that any edge has endpoints of different colors, for SubsetSum it is easy to verify if a subset has a sum equal to a target, and so on.

The theory of NP completeness shows that *the proof verification can be implemented in a restricted model*, namely a 3CNF. So we don't have to think of the verification step as using the full power of P: Any proof can be turned into another proof that can be verified by a 3CNF.

The PCP theorem takes this one step further: *The proof verification can be implemented in constant randomized time*. For Gap-3Sat, you pick a uniform clause, and check if it's satisfied.

P vs. NP

The flagship question of complexity theory is whether P = NP or not. This is a young, prominent special case of the grand challenge (section §1.8). Contrary to the belief that BPP = P, see section 3.4, the general belief seems to be that P ≠ NP. Similarly to BPP, see Theorem 3.4, the best deterministic simulation of NP runs in exponential time by trying all nondeterministic guesses. This gives the middle inclusion in the following fact; the other two are by definition.

Fact 5.2. P ⊆ NP ⊆ Exp ⊆ NExp.

Recall that a consequence of the Time Hierarchy Theorem 1.4 is that P ≠ Exp (Corollary 1.1). From the inclusions above it follows that

$$\text{P} \neq \text{NP} \text{ or } \text{NP} \neq \text{Exp}, \text{ possibly both.}$$

Thus, we are not completely clueless, and we know that at least one important separation is lurking somewhere. Most people appear to think that *both* separations hold, but we are unable to prove *either*.

Randomness vs. NP

Essentially by definition, NP contains the one-sided version of BPP, called RP, see Section §3.1.

Exercise 5.3. Prove this.

The relationship with BPP is not clear. However, we have the following result.

Theorem 5.2. P = NP ⇒ P = BPP.

Thus, if nondeterminism can be dispensed with, so can (double-sided) randomness. Theorem 5.2 is the combination of two results discussed later: Theorem 5.9 and Theorem 5.8.

5.2 Completeness

We now go back to the question at the beginning of this chapter about reducing arbitrary computation to 3Sat. We shall reduce all of NP to 3Sat. Problems admitting such reductions deserve a definition. Since we will encounter similar reductions in a variety of contexts, we give a general definition.

Definition 5.2. Let X, Y, R be sets of functions. We call a function f:
hard for X vs Y if $f \in X \Rightarrow X = Y$;
complete for X vs Y if $f \in Y$ and f is hard for X vs Y;
hard for Y under map reductions in R if every $g \in Y$ map reduces to f in R;
complete for Y under map reductions in R if $f \in Y$ and f is hard for Y under map reductions in R.

Just like for reductions (see section §4.1) the definitions with map reductions are more constrained, and they imply the others as as long as X is closed under R. For example, if f is hard for NP under map reductions in P then f is hard for P vs NP. Many problems we encounter (including 3Sat) are in fact hard under surprisingly constrained reductions, basically just duplicating or fixing bits. We define one such type of reductions next.

Definition 5.3. ProjectionsP is the subset of CktP where each output bit is either 0, 1, or an input literal (a variable x_i or its negation).

See Problem 4.4 for an example of a reduction under projections. However for simplicity we often state the results for more general classes of reductions, like FP.

Complete problems are the "hardest problems" in the class, as formalized in the following fact.

Fact 5.3. Suppose f is complete for P vs NP. Then $f \in \mathrm{P} \Leftrightarrow \mathrm{P} = \mathrm{NP}$.

Proof. ($\Leftarrow$) This is because $f \in \mathrm{NP}$.

($\Rightarrow$) Let $f' \in \mathrm{NP}$. Because f is hard we know that $f \in \mathrm{P} \Rightarrow f' \in \mathrm{P}$. **QED**

Exercise 5.4. Suppose $\mathrm{P} = \mathrm{NP}$. Prove that any problem in NP is complete for P vs NP.

Suppose instead $\mathrm{P} \neq \mathrm{NP}$. Let $f \in \mathrm{NP}$. Prove f is complete for P vs NP iff $L \notin \mathrm{P}$.

Fact 5.3 points to an important interplay between problems and complexity classes. We can study complexity classes by studying their complete problems, and vice versa.

The central result in the theory of NP completeness is the following.

Theorem 5.3. 3Sat is complete for P vs NP.

Proof. 3Sat is in NP by Fact 5.1. Next we prove hardness. As mentioned earlier, the main idea is to combine Theorem 5.1 and Theorem 2.5; the rest of the proof mostly amounts to opening up definitions.

Let $f \in \mathrm{NP}$ and let P be a corresponding word program which runs in time n^d on inputs (x, y) where $|x| = n$ and $|y| = n^d$, for some constant d. By the simulation of word programs by circuits, Theorem 2.5, we can compute in FP a circuit $C(x, y)$ of size $n^{c_{d,P}}$ such that for any x, y we have $P(x, y) = 1 \Leftrightarrow C(x, y) = 1$.

Now, suppose we want to compute f on an input w. This is equivalent to deciding if $\exists y : C(w, y) = 1$ by what we just said. We can "hard-wire" w into C to obtain the circuit $C_w(y) := C(w, y)$ only on the variables y. Here by "hard-wire" we mean replacing the input gates x with the bits of w. The size of C_w is at most the size of C plus 2, in case we need to add the constant gates. Now we can apply Theorem 5.1 to this new circuit to produce a 3CNF φ_w on variables y and new variables z such that $C_w(y) = 1 \Leftrightarrow \exists z : \varphi_w(y, z) = 1$, for any y. The size of φ_w and the number of variables z is power in the size of the circuit.

We have obtained:

$$f(w) = 1 \Leftrightarrow \exists y : P(w, y) = 1 \Leftrightarrow \exists y : C_w(y) = 1 \Leftrightarrow \exists y, z : \varphi_w(y, z) = 1 \Leftrightarrow \varphi_w \in \mathrm{3Sat}.$$

Hence if 3Sat is in P we can compute $f(w)$ efficiently by deciding if φ_w is satisfiable. **QED**

In sections §4.4 we reduced 3Sat to other problems which are also in NP by Fact 5.1. This implies that all these problems are NP-complete. Here we use that if problem A reduces to B in P, and B reduces to C, then also A reduces to C. This is because if $C \in \mathrm{P}$ then $B \in \mathrm{P}$, and so $A \in \mathrm{P}$.

Corollary 5.1. Clique, Subset-sum, and 3Color are complete for P vs NP.

It is important to note that there is nothing special about the *existence* of complete problems. The following is a simple such problem that does not require any of the machinery in this section.

Exercise 5.5. The Simu problem: Given a word program P, an input x, and $t \in \mathbb{N}$, where t is written in unary, decide if there is $y \in [2]^t$ such that $P(x, y) = 1$ in t steps.

Prove that Simu is complete for P vs NP.

What if t is written in binary?

The interesting aspect of complete problems such as 3Sat and those in Corollary 5.1 is that they are very simple and structured. This makes them suitable for reductions, and for inferring properties of their complexity class which are less evident from a model-based definition.

5.3 From programs to 3Sat in quasi-linear time

The framework in the previous section is useful to relate membership in P of different problems in NP, but it is not suitable for a more fine-grained analysis. For example, under the assumption that 3Sat is in $\mathrm{Time}(cn)$ we cannot immediately conclude that other problems in NP are solvable in this time or in about this time. We can only conclude that they are in P. In particular, the complexity of 3Sat cannot be related to that of other central conjectures, such as whether 3Sum is in subquadratic time, Conjecture 4.1.

The culprit is the power loss in reducing word programs to circuits, mentioned at the beginning of the chapter. We now remedy this situation and present a quasi-linear reduction. As we did before, see Theorem 5.1 and Theorem 5.3, we first state a version of the simulation for (deterministic) computation which contains all the main ideas, and then we note that a completeness result follows.

Theorem 5.4. Given an input length $n \in \mathbb{N}$, a time bound $t \in \mathbb{N}$, and a word program P that runs in time t on inputs of n bits, we can compute in time $t' := c_P t \log^c t$ a 3CNF φ on variables (x, y) where $|y| \leq t'$ such that for every $x \in [2]^n$:

$$P(x) = 1 \iff \exists y : \varphi(x, y) = 1.$$

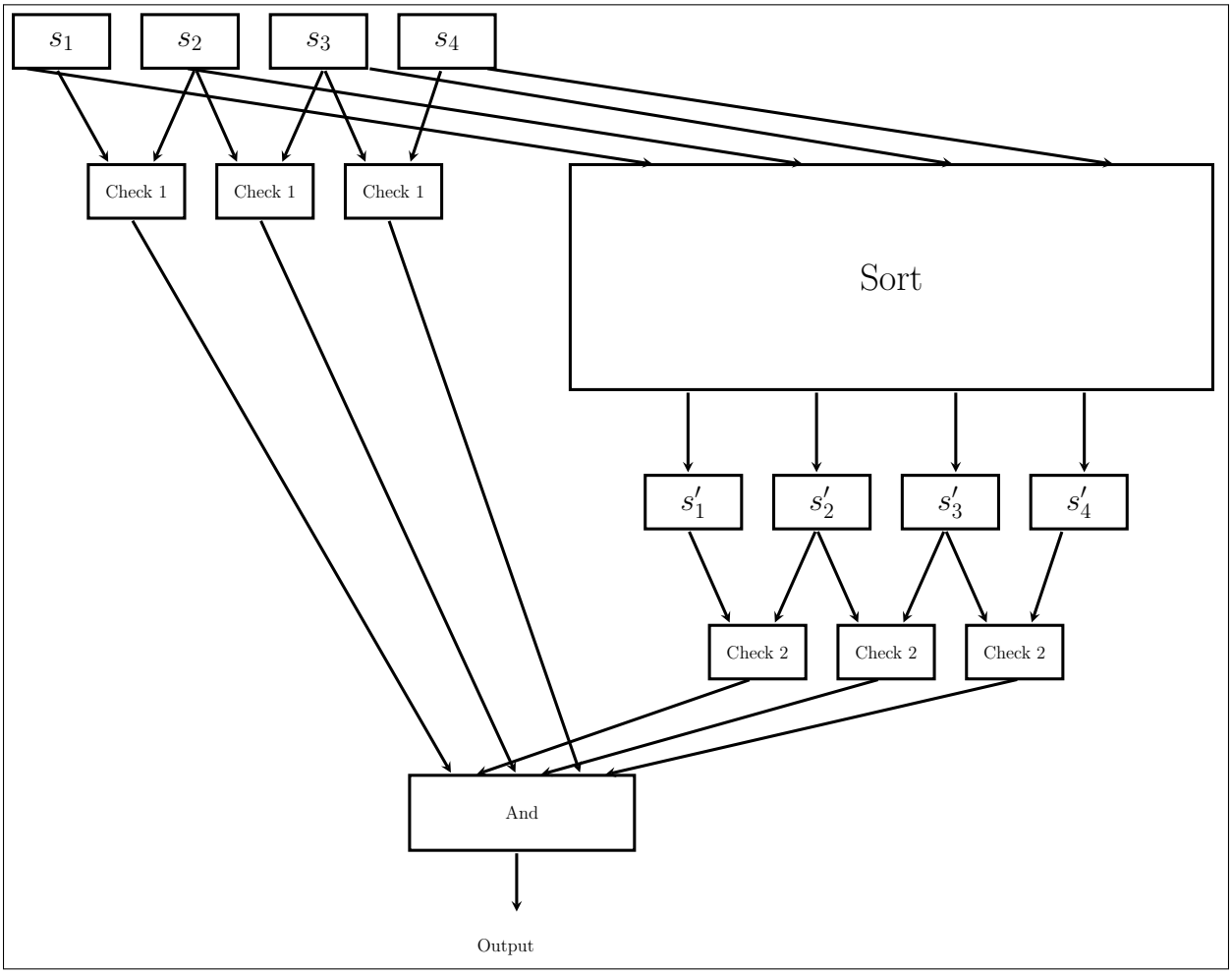


Figure 5.1: Circuit in the proof of Theorem 5.4.

We now present the proof of this amazing result.

At the high level, the approach is like in Theorem 5.1: We are going to *guess* computation and check it efficiently. However, we can't quite guess t configurations (Definition 1.2) of a word program, since each configuration contains the memory as well and so can have size about t. This would result in a quadratic blow-up, which we seek to avoid. Instead, we will guess *internal configurations* which are like configurations *except* that we don't include the entire memory, but only one word in case of Read/Write operations. This is similar to the data we have after fetching memory locations in the simulation of P by circuits, Theorem 2.5.

Definition 5.4. The *internal configuration,* abbreviated IC*,* of a word program specifies:

- the program counter,
- the word length w,
- its registers, and
- if the current instruction is a Read $R[i] := M[R[j]]$ or Write $M[R[j]] := R[i]$ then the IC includes the word $M[R[j]]$.

If a program P runs in time $t \geq n$ on an input of length n, its ICs can be described by $c_P \log t$ bits. Again, this is as in the proof of Theorem 2.5.

As we mentioned, we are going to guess the ICs, and then we need to check them efficiently by a circuit. This is not immediate, since, again, the word program can read and write in memory at arbitrary locations, something which is not easy to do with a circuit. It is convenient to view the checking as two separate checks, only one of which involves memory. If both checks pass, then the computation is correct.

More precisely, a sequence of internal configurations $s_1, s_2, \ldots, s_t$ corresponds to the computation of the program on input x iff for every $i < t$:

1. If s_i does not access memory, then s_{i+1} has its registers, program counter, and word length updated according to the instruction executed in s_i,
2. If s_i is computing a read operation $R[i] := M[R[j]]$ then in s_{i+1} register $R[i]$ contains *the most recent value written in memory* word $R[j]$. In case this word was never written, then $R[i]$ should contain $x_{R[j]}$ if $R[j] \in [n]$, and 0 otherwise. The program counter in s_{i+1} also points to the next instruction.

Rather than directly constructing a 3CNF that implements these checks, we construct a circuit and then appeal to Theorem 5.1. The circuit is illustrated in Figure 5.1. To construct a circuit for Check 1, note that for each i the check between ICs s_i and s_{i+1} can be implemented by a circuit of size $c_P \log^c t$. For example, this circuit may check that $R[0] := R[1] + R[2]$. This can be done via the circuit in Exercise 2.2. And the same holds for other operations. Alternatively, we can argue that these check are in P (note the input length is the length

of two ICs), and then appeal to Theorem 2.5 to obtain a circuit. Taking an And of these circuits over the choices of i gives a circuit of the desired size for Check 1.

The difficulty lies in Check 2, because the circuit needs to find "the most recent value written." The clever solution is to *sort the ICs by the memory address in Read/Write operations*. After sorting, we can implement Check 2 as easily as Check 1, since we just need to check adjacent pairs of ICs.

The emergence of sorting in the theory of NP-completeness cements the pivotal role this operation plays in computer science.

To implement this idea we need to be able to sort the ICs by their memory addresses with a quasi-linear size circuit. Several quasi-linear time sorting algorithms are well known (such as CountingSort, see Example 1.1), but implementing them as quasi-linear-size circuits is not immediate. Still, such sorting circuits exist:

Lemma 5.1. Given t and m we can compute in time $t' := t \cdot (m \log t)^c$ a circuit (of size $\leq t'$) that sorts t integers of m bits.

Because this reduction is so fundamental, for completeness we give a proof of Lemma 5.1 in section §5.3.1.

We summarize the key steps in the proof.

Proof of Theorem 5.4. We construct a circuit C as in Figure 5.1 (for $t = 4$) and then appeal to Theorem 5.1. The extra variables y correspond to t ICs $s_1, s_2, \ldots, s_t$. An IC takes $c_P \log t$ bits to specify, so we need $\leq c_P t \log t$ variables y. The circuit C first performs Check 1 above for each adjacent pair (s_i, s_{i+1}) of ICs. This takes size $c_P \log^c t$ for each pair, and so size $c_P t \log^c t$ overall.

Then C sorts the ICs by memory addresses, producing sorted ICs $s'_1, s'_2, \ldots, s'_t$. This takes size $t \cdot \log^c t$ by Lemma 5.1, using that the memory addresses have $m \leq c \log t$ bits. Then the circuit performs Check 2 for each adjacent pair (s'_i, s'_{i+1}) of ICs. The circuit size required for this is no more than for Check 1.

Finally, the circuit takes an And of the results of the two checks, and also checks that s_t outputs 1.

This concludes the proof, except for some low-level details related to sorting which I discuss now. The sorting Lemma 5.1 refers to integers, whereas as common in algorithms we need to sort *unstructured objects* (the ICs) by *keys* (the memory addresses). However, we can just view each IC as an integer with the memory address written in the most significant digits, to have the same effect. Also, the sorting should be *stable*: The relative order of ICs with the same key should be maintained in the output. This can again be easily accomplished by including a timestamp, i.e., the original order as a number in $[t]$, written next in significance after the memory address. (So if two ICs address the same memory location, their order will be decided based on the timestamp.) This timestamp increases the key by $\leq c \log t$ bits and so can be afforded. **QED**

We can now prove completeness in a manner similar to Theorem 5.3, with a relatively simple extension of Theorem 5.4.

Theorem 5.5. 3Sat is complete for NTime($t(n)$) under map reductions in Time($t \log^c t$), for every time-constructible $t(n) \geq n \log^c n$.

Recall the assumption on t is satisfied by all standard functions – see discussion in section 1.8.2.

Proof. First let us show that 3Sat is in NTime($t(n)$). Since $t(n) \geq n \log^c n$ it is enough to show that 3Sat is in quasilinear-NTime. Consider a 3CNF instance φ of length n. This instance has at most n variables, and we can guess an assignment y to them within our budget of non-deterministic guesses. There remains to verify that y satisfies φ. For this, we can do one pass over the clauses. For each clause, we access the bits in y corresponding to the 3 variables in the clause, and check if the clause is satisfied. This takes constant time per clause, and so time cn overall.

Now let us prove hardness. Let $f \in$ NTime(t) and P be a corresponding word program. Given an input $w \in [2]^n$ we have to compute in time $t \log^c t$ a 3CNF φ such that

$$\exists y \in [2]^{t(n)} : P(x, y) = 1 \iff \exists y \in [2]^{t(n) \log^c t(n)} : \varphi(y) = 1.$$

First we compute $t(n)$, using the assumption. We now apply Theorem 5.4, but on a new input length $n' := c(n + t) \leq ct$, to accommodate for inputs of the form (x, y). This produces a formula φ of size $c_P t \log^c$ in variables (x, y) and new variables z. We can now set the variables x' to the input w to conclude the proof. The running time is dominated by that of the reduction in Theorem 5.4, which is $t \log^c t$. **QED**

We can now give the following quasi-linear version of Fact 5.3. The only extra observation for the proof is again that the composition of two quasi-linear functions is quasi-linear.

Corollary 5.2. 3Sat $\in$ Quasi-Linear-Time $\Leftrightarrow$ Quasi-Linear-NTime = Quasi-Linear-Time.

Exercise 5.6. Prove that Theorem 5.5 holds with 3Color instead of 3Sat using the reduction in section §4.4. Can you obtain a similar result for Clique and Subset-sum using the reductions in section §4.4?

Exercise 5.7. Prove that 3Sum reduces to 3Sat in Subquadratic time.

5.3.1 Efficient sorting circuits: Proof of Lemma 5.1

We present an efficient sorting algorithm for an array $A[0..n-1]$ which enjoys the *CE property*: *the only way in which the input is accessed is via* Compare-Exchange *operations*. Compare-Exchange takes two indexes i and j and swaps $A[i]$ and $A[j]$ if they are in the wrong order. It has the following code:

```
Compare-Exchange(Array A[0..n-1] and indexes i and j with i < j):
if A[i] > A[j]
 swap A[i] and A[j]
```

Why care about this property? It makes the comparisons *independent from the input,* and this allows us to implement the algorithm with a network – a *sorting network* – of fixed Compare-Exchange operations. In particular, we will get a circuit.

The algorithm is called CE-Sort. It has a basic recursive structure similar to MergeSort:

```
CE-Sort(A[0..n − 1]):
if n ≥ 1 {
 CE-Sort(A[0..n/2 − 1])
 CE-Sort(A[n/2..n − 1])
 CE-Merge(A[0..n − 1])
}
```

Throughout, we assume that n is a power of 2.

Algorithm CE-Merge(A) merges the two already sorted halves $A[0..n/2-1]$ and $A[n/2-1..n-1]$ of A, resulting in a sorted output sequence. CE-Merge replaces the Merge operation in Mergesort which does not have the CE property. It works in a remarkable and startling way. First it merges the *odd* subsequence of the entire array A, then the *even*, and finally it makes a series of Compare-Exchange operations.

```
CE-Merge(A[0..n − 1]):
if n = 2
 Compare-Exchange(A, 0, 1)
else {
 CE-Merge(A[0, 2, 4, . . . , n − 2]) //the even subsequence
 CE-Merge(A[1, 3, 5, . . . , n − 1]) //the odd subsequence
 for i ∈ {1, 3, 5, 7, . . . , n − 3}
  Compare-Exchange(A, i, i + 1)
}
```

We shall now argue that this algorithm is correct.

Lemma 5.2. If $A[0..n/2-1]$ and $A[n/2..n-1]$ are sorted, then CE-Sort($A[0..n-1]$) outputs a sorted array.

Proof. To prove this lemma we invoke the 0-1 *principle*. This principle says that CE algorithms sort integers correctly iff they sort correctly integers in [2]. In this particular case, we can apply this principle to say that it suffices to prove the lemma when each $A[i] \in [2]$ for every i.

For completeness we prove this principle in now. Let A be an input and B an output produced by a CE algorithm S. Suppose B is not sorted let k be the smallest such that $B[k] > B[k+1]$. Define a function f such that $f(x) = 1$ if $x \geq b_k$ and $f(x) = 0$ otherwise. For an array X let $f(X)$ denote the array obtained by applying f to every entry of X. Observe

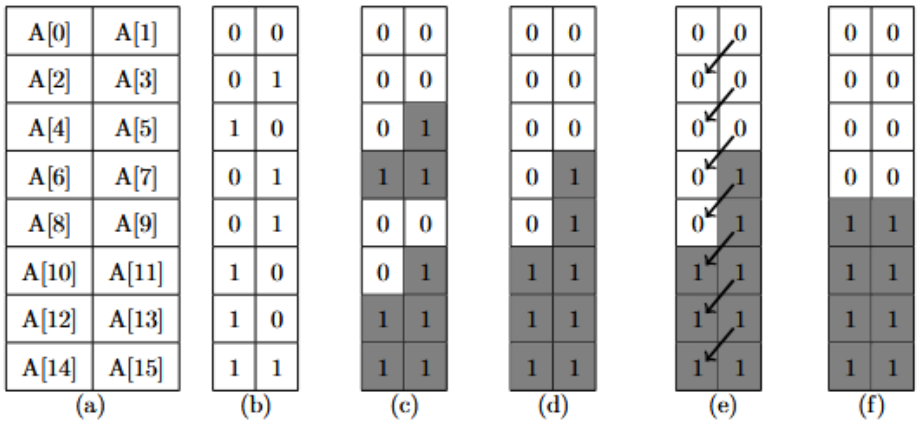


Figure 5.2: CE-Merge

that $f(B)$ is not sorted. However (see Exercise 5.8) f commutes with any Compare-Exchange operation applied to any sequence X:

$$f(\text{Compare-Exchange}(X, i, j)) = \text{Compare-Exchange}(f(X), i, j). \tag{5.1}$$

If the algorithm S is just a sequence of Compare-Exchange, we have

$$f(B) = f(S(A)) = S(f(A)).$$

So the algorithm fails to correctly merge the 0-1 sequence $f(A)$.

Having established this principle, we now prove the lemma for $A \in [2]^n$ by induction on n, following the recursive definition of CE-Merge. Refer to figure 5.2.

The base case $n = 2$ is clear. Assume that CE-Merge correctly merges any two sorted 0-1 sequences of size $n/2$. We view an input sequence of n elements as an $n/2 \times 2$ matrix, with the left column corresponding to elements at the even-indexed positions $0, 2, \ldots, n-2$ and the right column corresponding to elements at the odd-indexed positions $1, 3, \ldots, n-1$ (Figure 5.2(a)). 5.2(b) shows a corresponding 0-1 input, which we can assume w.l.o.g. because of the zero-one principle. Figure 5.2(c) shows the matrix after the recursive calls to the sorting. Since the upper half of the matrix is sorted by assumption, the right column in the upper half has the same number or exactly one more 1 than the left column in the upper half. The same is true for the lower half. Because each (length-$(n/4)$) column in each half of the matrix is also individually sorted by assumption, the induction hypothesis guarantees that after the two calls to CE-Merge both the left and right (length-$(n/2)$) columns are sorted (Figure 5.2(d)).

At this point only one of 3 cases arises:

1) The odd and even subsequences have the same number of 1s.

2) The odd subsequence has a single 1 more than the even subsequence.

3) The odd subsequence has two 1s more than the even subsequence.

In the first two cases, the sequence is already sorted. In the third case, the Compare-Exchange operations (Figure 5.2(e)) yield a sorted sequence (Figure 5.2(f)). **QED**

Exercise 5.8. Prove Equation (5.1).

To conclude the proof of Lemma 5.1 about sorting circuits, it only remains to argue efficiency. Let $S_M(n)$ denote the number of Compare-Exchange operations for CE-Merge for an input sequence of length n. We have the recurrence

$$S_M(n) = 2 \cdot S_M(n/2) + n/2 - 1,$$

which yields $S_M(n) \leq cn \log n$.

Finally, let $S(n)$ denote the number of calls to Compare-Exchange for CE-Sort on an input sequence of length n. Then we have the recurrence

$$S(n) \leq 2 \cdot S(n/2) + cn \log n$$

which yields $S(n) = cn \cdot \log^2 n$.

To conclude the proof, note that Compare-Exchange for inputs with m bits can be implemented by a circuit of size m^c.

5.4 Power from completeness

The realization that arbitrary computation can be reduced to 3Sat and other problems is powerful and liberating. In particular in this section wee demonstrate that it allows us to significantly widen the net of reductions.

5.4.1 Max-3Sat

The 3Sat asks if all the clauses can be satisfied. The Max-3Sat asks to compute the maximum number of clauses that can be satisfied.

Definition 5.5. The Max-3Sat problem: Given a 3CNF, compute the maximum number of clauses that can be satisfied by an assignment to the variables.

3Sat trivially reduces to Max-3Sat in FP. The converse will be shown next.

Theorem 5.6. Max-3Sat reduces to 3Sat in FP.

Proof. Consider the problem Atleast-3Sat: Given a 3CNF formula and an integer t, is there an assignment that satisfies at least t clauses? This is in NP and so can be reduced to 3Sat in P. This is the step that's not easy without "thinking completeness:" given an algorithm for 3Sat it isn't clear how to use it directly to solve Atleast-3Sat.

Hence, if 3Sat is in P so is Atleast-3Sat. On input a 3CNF ϕ for Max-3Sat, we can use binary search and Atleast-3Sat to find the largest t s.t. $(\phi, t) \in$ Atleast-3Sat. We then output t. **QED**

Having found the maximum, one can also compute a corresponding satisfying assignment fixing one variable at the time as in the proof of Theorem 4.8.

5.4.2 NP is as easy as detecting unique solutions

In this section we present a beautiful randomized reduction. A satisfiable 3CNF can have multiple satisfying assignments. On the other hand some problems and puzzles have unique solutions. It is natural to ask if the hardness of 3Sat stems from multiple assignments – perhaps a unique satisfying assignment would be easy to find. In this section we show that, in fact, deciding the satisfiability of 3CNFs with a unique satisfying assignment is no easier, if randomized reductions are allowed.

We will be working with satisfiability of general circuits, and infer the result for 3CNFs as a consequence via completeness. We now define these problems and then state the main result.

Definition 5.6. The Unique-CktSat problem: Given a circuit C s.t. there is at most one input x for which $C(x) = 1$, decide if such an input exists.

Unique-3Sat is the Unique-CktSat problem restricted to 3CNF circuits.

Theorem 5.7. 3Sat reduces to Unique-3Sat in BPP.

The proof is another example of the power of randomness, and introduces an important paradigm in randomized computing: *hashing*. We will use hash functions to "isolate" assignments. First we define the type of hash functions we will need, then we construct them, and finally we state and prove the isolation lemma.

Definition 5.7. A distribution H on functions mapping $S \to T$ is called *pairwise uniform* if for every distinct $x, x' \in S$ and every $y, y' \in T$ one has

$$\mathbb{P}_H[H(x) = y \wedge H(x') = y'] = 1/|T|^2.$$

This is saying that on every pair of distinct inputs H behaves like a uniform function. Yet unlike completely uniform functions, the next lemma shows that pairwise uniform functions can have a short description, which makes them suitable for use in algorithms.

Lemma 5.3. Let $H : [2]^n \to [2]^t$ be the random function $H(x) := Ax + B$ where A is a uniform $n \times t$ matrix over $\mathbb{F}_2$ and B is a uniform vector in $\mathbb{F}_2^t$. Then H is pairwise uniform.

Exercise 5.9. Prove that the lemma follows from the lemma with $t = 1$.

Proof. Let $t = 1$ and consider any $x \neq x'$. Let $i \in [n]$ be a coordinate s.t. $x_i \neq x'_i$. We aim prove that the pair $(H(x), H(x'))$ is uniformly distributed over $\mathbb{F}_2^2$. In fact, we will prove the stronger result that this is true even for any fixing of all the coordinates in A except i.

Assuming without loss of generality that $x_i = 0$ (and $x'_i = 1$), we have that

$$(H(x), H(x')) = \left(\sum_{j \neq i} A_j x_j + B, \sum_{j \neq i} A_j x'_j + A_i x'_i + B \right) = (a + B, b + A_i + B)$$

where a and b are fixed values in $\mathbb{F}_2$. This distribution is uniform over $\mathbb{F}_2^2$ iff the distribution $(B, A_i + B)$ is; and the latter is indeed uniform. **QED**

There are constructions of pairwise-uniform with smaller support, see Problem 5.1.

The next result shows how we can use pairwise uniformity to "isolate" an element.

Lemma 5.4. [Isolation] Let H be a pairwise uniform function mapping $S \to T$, and let $1 \in T$. The probability that there is a unique element $s \in S$ such that $H(s) = 1$ is

$$\geq \frac{|S|}{|T|} - \frac{|S|^2}{|T|^2}.$$

In particular, if $|T|/8 \leq |S| \leq |T|/4$ this prob. is $\geq \frac{1}{8} - \frac{1}{16} \geq 1/8$.

Proof. For fixed $s \in S$, the probability that s is the unique element mapped to 1 is at least the prob. that s is mapped to 1 minus the prob. that both s and some other $s' \neq s$ are mapped to 1. This is

$$\geq \frac{1}{|T|} - \frac{|S| - 1}{|T|^2}.$$

These events for different $s \in S$ are disjoint; so the target probability is at least the sum of the above over $s \in S$. **QED**

We can now present the reduction from 3Sat to Unique-3Sat.

Proof of Theorem 5.7. Given a 3Sat instance ϕ with $\leq n$ variables x, pick a uniform $i \in [n + c]$. We then sample a pairwise uniform function mapping $[2]^n$ to $[2]^i$, say from Lemma 5.3, and consider a circuit computing

$$C(x) := \phi(x) \wedge H(x) = 0^i.$$

Computing H as in Lemma 5.3 yields a circuit of size n^c.

If ϕ is not satisfiable, C is not satisfiable, for any choice for H.

Now suppose that ϕ has $s \geq 1$ satisfying assignments. With prob. $\geq c/n$ over the choice of i we will have $2^{i-3} \leq s \leq 2^{i-2}$, in which case Lemma 5.4 guarantees that C has a unique satisfying assignment with prob. $\geq c$ over the choice of H. Overall, a Unique-CktSat algorithm on C outputs 1 with prob. $\geq c/n$. By the same analysis of section §3.1, if we repeat this process cn times, with independent random choices, the Or of the outcomes gives the correct answer with prob. $\geq 2/3$. **QED**

Exercise 5.10. Conclude the proof reducing Unique-CktSat to Unique-3Sat.

5.5 Alternation

We placed one quantifier "in front" of computation and got something interesting: NP. So let's push the envelope and place more. As we will see the corresponding classes turn out to be extremely useful, with deep ties to impossibility results, the P vs. BPP question, circuits, and much more. The proofs of several results that do not *prima facie* involve multiple quantifiers excurse into multiple quantifiers. Examples include new reductions to 3Sat (Exercise 5.12), impossibility results for 3Sat established in Theorem 6.20 and Theorem 6.21, and P = NP ⇒ P = BPP (see Theorem 5.2 and Exercise 5.13). The first of these will be proved in this chapter.

The next definition is like the Definition 5.1 of NTime, except there are multiple quantifiers.

Definition 5.8. $\Sigma_i\mathrm{Time}(t(n))$ is the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a word program P that for every $x \in X$ of length $\geq |P|$ satisfies:
- $f(x) = 1$ iff $\exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q_i y_i \in [2]^{t(|x|)} : P(x, y_1, y_2, \ldots, y_i) = 1$, and
- $P(x, y_1, y_2, \ldots, y_i)$ stops within $t(n)$ steps for every $y_1, y_2, \ldots, y_i \in [2]^{t(n)}$.

$\Pi_i\mathrm{Time}(t(n))$ is defined similarly except that we start with a $\forall$ quantifier. We also define

$$\Sigma_i\mathrm{P} := \bigcup_d \Sigma_i\mathrm{Time}(n^d),$$

$$\Pi_i\mathrm{P} := \bigcup_d \Pi_i\mathrm{Time}(n^d), \text{ and}$$

$$\text{the Power Hierarchy, PH} := \bigcup_i \Sigma_i\mathrm{P} = \bigcup_i \Pi_i\mathrm{P}.$$

We refer to such computation and the corresponding programs as *alternating,* since they involve an alternation of quantifiers. The PH is the power-time analogue of the older arithmetical hierarchy from computability theory or logic in which nondeterministic time plays the role of listable (a.k.a. computably enumerable, etc.).

Similarly to NP, we have PH ⊆ Exp. We leave this as an exercise; we'll prove a stronger fact later.

Exercise 5.11. The Min-Ckt problem: Given a circuit C, decide if there is no smaller circuit equivalent to C. Prove that Min-Ckt is in $\Pi_2\mathrm{P}$.

The following key result says that alternations can simulate randomness. More precisely, two simulations. The first optimizes the number of quantifiers, the second the time. This should be contrasted with various *conditional* results (such as Theorem 3.5) suggesting that in fact a quasilinear deterministic simulation (with no quantifiers) is possible.

Theorem 5.8. BPP ⊆ PH.

More in detail, for every function t we have:
(1) $\mathrm{BPTime}(t) \subseteq \Sigma_2\mathrm{Time}(t^2 \log^c t)$, and
(2) $\mathrm{BPTime}(t) \subseteq \Sigma_3\mathrm{Time}(t \log^c t)$.

The proof is best understood in terms of constant-depth circuits and is therefore postponed to Chapter 9.

5.5.1 Does the hierarchy collapse?

We refer to the event that $\exists i : \Sigma_i \mathrm{P} = \mathrm{PH}$ as "the PH collapses." It is unknown if the PH collapses. A common belief appears to be that it does not. According to this belief, statements of the type

$$X \Rightarrow \text{PH collapses}$$

constitute evidence that X is false. Examples of such statements are discussed next.

Theorem 5.9. $\mathrm{P} = \mathrm{NP} \Rightarrow \mathrm{P} = \mathrm{PH}$.

The idea in the proof is simply that if you can remove one quantifier then you can remove more.

Proof. We prove by induction on i that $\Sigma_i \mathrm{P} \bigcup \Pi_i \mathrm{P} = \mathrm{P}$.

The base case $i = 1$ follows by assumption and the fact that P is closed under complement.

Next we do the induction step. We assume the conclusion is true for i and prove it for $i + 1$. We will show $\Sigma_{i+1}\mathrm{P} = \mathrm{P}$. The result about $\Pi_{i+1}\mathrm{P}$ follows again by complementing.

Let $f \in \sum_{i+1} \mathrm{P}$, so $\exists a \in \mathbb{N}$ and a power-time program P such that for any input x of length n:

$$f(x) = 1 \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \forall y_4 \ldots Q_{i+1} y_{i+1} \in [2]^{n^a} : P(x, y_1, y_2, \ldots, y_{i+1}) = 1.$$

(As discussed after Definition 5.8 we don't need to distinguish between time as a function of $|x|$ or of $|(x, y_1, y_2, \ldots, y_{i+1})|$ when considering power times as we are doing now.)

Now the creative step of the proof is to consider f' defined as

$$f'(x, y_1) = 1 \iff \forall y_2 \in [2]^{n^a} \ldots Q_{i+1} y_{i+1} \in [2]^{n^a} : P(x, y_1, y_2, \ldots, y_{i+1}) = 1.$$

Note $f' \in \Pi_i \mathrm{P}$. By induction hypothesis $f' \in \mathrm{P}$. So let P' compute f' in power time. We have $f(x) = 1 \iff \exists y_1 \in [2]^{n^a} : P'(x, y_1) = 1$. And so $f \in \mathrm{NP} = \mathrm{P}$, again using the hypothesis. **QED**

Exercise 5.12. Prove that Min-Ckt reduces to 3Sat in P.

Exercise 5.13. Prove that $\mathrm{P} = \mathrm{NP} \Rightarrow \mathrm{P} = \mathrm{BPP}$ (Theorem 5.2).

Exercise 5.14. Prove the following variant of Theorem 5.9: Suppose $\forall \epsilon > 0$ $\mathrm{NTime}(n^{1+\epsilon}) \subseteq \mathrm{Time}(n^{1+2\epsilon})$. Then $\Sigma_i \mathrm{Time}(n^{1+\epsilon}) \subseteq \mathrm{Time}(n^{1+4^i\epsilon})$ for every $i \in \mathbb{N}, \epsilon > 0$.

We have shown that if you can remove one quantifier you can remove all of them – Theorem 5.9. Next we show that if you can *swap* a quantifier you can swap and then collapse all of them. For later uses we prove this in the general case where the swap happens at some level of the hiearchy – a similar extension of Theorem 5.9 holds as well.

Theorem 5.10. $\Sigma_i\mathrm{P} = \Pi_i\mathrm{P} \Rightarrow \mathrm{PH} = \Sigma_i\mathrm{P}$.

Proof. We prove that for every $j \geq i$ we have $\Sigma_j\mathrm{P} \bigcup \Pi_j\mathrm{P} = \Sigma_i\mathrm{P} \bigcap \Pi_i\mathrm{P}$. We proceed by induction on j. The base base $j = i$ is given by hypothesis. For the inductive step let $f \in \sum_{j+1}\mathrm{P}$, the case $f \in \Pi_{j+1}\mathrm{P}$ being symmetrical. By definition, $\exists a \in \mathbb{N}$ and a power-time program P such that for any input x of length n,

$$f(x) = 1 \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \forall y_4 \dots Q_{j+1} y_{j+1} \in [2]^{n^a} : P(x, y_1, y_2, \dots, y_{j+1}) = 1.$$

Similarly to the proof of Theorem 5.9 consider

$$f'(x, y_1) = 1 \iff \forall y_2 \in [2]^{n^a} \dots Q_{i+1} y_{i+1} \in [2]^{n^a} : M(x, y_1, y_2, \dots, y_{i+1}) = 1.$$

Note $f' \in \Pi_j\mathrm{P}$. By induction hypothesis $f' \in \Sigma_i\mathrm{P}$. So there is a power-time program P' and $a' \in \mathbb{N}$ such that

$$f(x, y_1) = 1 \Leftrightarrow \exists z_1 \forall z_2 \exists z_3 \forall z_4 \dots Q_i z_i \in [2]^{n^{a'}} : P'(x, y_1, z_1, z_2, \dots, z_i) = 1.$$

W.l.o.g. $a' \geq a$. Hence we have

$$f(x) = 1 \Leftrightarrow \exists y_1 \exists z_1 \forall z_2 \exists z_3 \forall z_4 \dots Q_i z_i \in [2]^{n^{a'}} : P'(x, y_1, z_1, z_2, \dots, z_i) = 1.$$

We can now merge or collapse the two quantifiers $\exists y_1$ and $\exists z_1$ into a single one, which shows that $f \in \Sigma_i\mathrm{P} = \Pi_i\mathrm{P}$, where the equality holds by assumption. **QED**

It is not known if NP has power-size circuits. However, the following result shows that if it does then the PH collapses. This is an interesting connection between non-uniform and uniform computational models.

Theorem 5.11. $\mathrm{NP} \subseteq \mathrm{CktP} \Rightarrow \mathrm{PH} = \Sigma_2\mathrm{P}$.

Proof. We'll show $\Pi_2\mathrm{P} \subseteq \Sigma_2\mathrm{P}$ and then appeal to Theorem 5.10. Let $f \in \Pi_2\mathrm{Time}(n^d)$ and P be a corresponding program s.t.

$$f(x) = 1 \Leftrightarrow \forall y_1 \in [2]^{n^d} \exists y_2 \in [2]^{n^d} : P(x, y_1, y_2) = 1.$$

We claim the following equivalent expression for the right-hand side:

$$\forall y_1 \in [2]^{n^d} \exists y_2 \in [2]^{n^d} : P(x, y_1, y_2) = 1 \Leftrightarrow \exists C \, \forall y_1 \in [2]^{n^d} : P(x, y_1, C(x, y_1)) = 1,$$

where C ranges over circuits of size $|x|^{d'}$ for some d'. If the equivalence is established the result follows, since evaluating a circuit can be done in power time.

To prove the equivalence, first note that the $\Leftarrow$ direction is obvious, by setting $y_2 := C(x, y_1)$. The interesting direction is the $\Rightarrow$. Consider the problem Search-CktSat which is like Search-3Sat but for general circuits rather than 3CNFs. Now, because CktSat is in NP, by assumption it has power-size circuits. By the reduction in Theorem 4.8, Search-CktSat has power-size circuits S as well. Hence, the desired circuit C may, on input x and y_1 produce a new circuit W mapping an input y_2 to $P(x, y_1, y_2)$ (as in Theorem 2.5), and run S on W to produce y_2. **QED**

Exercise 5.15. Prove that $\mathrm{PH} \not\subseteq \mathrm{CktSize}(n^k)$, for any $k \in \mathbb{N}$. (Hint: Theorem 2.6.)

Improve this to $\Sigma_2\mathrm{P} \not\subseteq \mathrm{CktSize}(n^k)$.

5.6 Problems

Problem 5.1. Let $\mathbb{F}_q$ be a finite field. Define the random function $H : \mathbb{F}_q \to \mathbb{F}_q$ as $H(x) := Ax + B$ where A, B are uniform in $\mathbb{F}_q$. Prove that H is pairwise uniform. Explain how to use H to obtain a pairwise uniform function from $[2]^n$ to $[2]^t$ for any given $t \leq n$.

Problem 5.2. In this problem you'll prove non-deterministic time hierarchies. We focus on computation with one quantifier only, but similar results hold for any level of the hierarchy. Note we don't even need $t(n)$ to be time-constructible.

(1) Prove $\Sigma_1\text{Time}(t(n)) \not\subseteq \Pi_1\text{Time}(ct(n))$, for any $t(n) \geq n$. Hint: Follow the Time Hierarchy Theorem 1.4.

(2) Prove $\text{NTime}(ct(n+1)) \not\subseteq \text{NTime}(t(n))$, for any non-decreasing $t(n) \geq n$. Hint: Follow the randomized Time Hierarchy Theorem 3.7.

Problem 5.3. Prove $\text{Exp} \subseteq \text{CktP} \Rightarrow \text{Exp} = \Sigma_2\text{P}$.

5.7 Notes

NP-completeness and Theorem 5.3 originates in the fundamental works [73, 184]. The first paper proves a version of Theorem 5.1 for tape machines, for a more recent and similar exposition see [256]. Theorem 5.4 is from [122, 236]. The first work focuses on an equivalence between computational models, while the second explicitly constructs a 3CNF formula. We presented the proof in a slightly different way, using the sorting circuits from [39] and following the exposition in [213].

The reduction to unique-3Sat, Theorem 5.7, is from [284] from which we borrowed the section title which, interestingly, emphasizes how easy NP could be, see Chapter 19.

Pairwise uniformity was studied as least since [228] and [60]. For background see [280, 136].

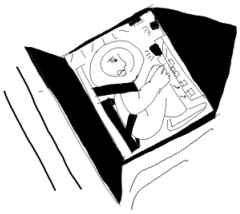
The PH was identified in [263], where Theorem 5.9 is also proved. Theorem 5.11 is from [166].

For a compendium of problems complete for various levels of the PH, in the style of [101], see [242].

The first item in the simulation of BPP Theorem 5.8 is from [255]; the second is from [292].

Chapter 6

Space



Time is only one of many resources we wish to bound. Another important one is *space*, which is another name for the *memory* of the machine. Computing under space constraints could be slightly less familiar to us than computing under time constraints, and many surprises lay ahead in this chapter that challenge our intuition of space-efficient computation.

We shall consider both space bounds bigger than the input length and smaller. For the latter, we allow the program to *read* the input words, but not *write* on them. We also want to compute functions f whose output is more than 1 bit. One option is to have some memory words which are *write-only*. I prefer instead to reduce this to boolean functions by requiring that given x, i output bit i of $f(x)$ is computable efficiently. To make sense of this we'll also require that we can decide if i is out of range, i.e., $i \geq f(x)$. Given this choice, there isn't much use in writing the output in memory, so we'll simply compute it in a register. Finally, we will follow the custom of measuring space in *bits*; measuring in terms of words would also be natural, as discussed right after the definition.

With this in mind, we give the following space analogue of Definition 1.6 of time-bounded computation.

Definition 6.1. We say that a word program P computes $y \in [2]$ on input $x \in [2]^*$ in space s if for some t it computes y into register $R[0]$ in time t as in Definition 1.6 (except the output is written in $R[0]$), and in addition:

- P never writes in words $0..n-1$.
- The total number of bits in the memory words used by P, excluding the input, is $\leq s$. In other words, every configuration (p, m, w, R, M) occurring in the computation has $(m-n)w \leq s$.

We now define the corresponding classes.

Definition 6.2. We denote by $\text{Space}(s(n))$ the set of functions $f : X \subseteq [2]^* \to [2]$ for which there is a word program P that for every $x \in X$ of length $\geq |P|$ computes $f(x)$ in space $s(|x|)$.

We define:

$$\text{logarithmic space, L} := \bigcup_d \text{Space}(d \log n),$$

$$\text{power space, PSpace} := \bigcup_d \text{Space}(n^d).$$

We denote by $\text{FSpace}(s(n))$ the set of functions $f : X \subseteq [2]^* \to [2]^*$ s.t.

1. given $x \in X$ and $i < |f(x)|$ computing bit i of $f(x)$ is in $\text{Space}(s(n))$,
2. given $x \in X$ and i computing $[i < f(x)]$ is in $\text{Space}(s(n))$, and
3. $|f(x)| \leq 2^{s(|x|)}$ for every $x \in X$.

FL and FPspace are defined similarly.

Note that functions in FL are restricted to have power-length output (like functions in FP have). We did not define FSpace for relations, only functions, as that will be sufficient.

The letter L stands for *logarithmic* which is the number of *bits* of space. If we measure space in words, the number is *constant*. Such programs do not need to use memory at all, they can simulate it using registers only:

L is what's computable *without writing in memory, only in registers.*

It is not clear how to make sense of sub-logarithmic space on word programs. But a surprisingly rich theory will be presented over a more restricted model in section §16.7.

As for time, a central notion is that of a *configuration* (Definition 1.2). Once a program P and an input x are fixed, for space-bounded computation we do not need to include the read-only memory words $M[0..|x|-1]$, since they cannot change. This leads to an accurate count of the number of possible configurations which will be useful several times starting now. We investigate next the relationship between space and time. We begin with some basic simulations; a slight improvement to 1. is known, see the notes.

Theorem 6.1. For every functions $t = t(n) \geq n$ and $s = s(n) \geq \log n$:

1. $\text{Time}(t) \subseteq \text{Space}(ct \log t)$,
2. $\text{Space}(s) \subseteq \bigcup_{a \in \mathbb{N}} \text{Time}(a^s)$.

Proof. 1. The space-bounded simulation will first copy the input in read-write memory words, and then simulate the time-bounded computation. Because in time t we can only allocate t extra memory words, and $t(n) \geq n$, we only need ct words of space, for a total of $\leq ct \log t$ bits. (As in Claim 1.1, one detail is that the time-bounded and space-bounded programs start off with different word size; the solution is the same as before.)

2. Let P be a word program computing an output from an input x using space s. To describe a configuration we need the contents of all the read/write memory words, which by assumption is $\leq s$ bits. We also need the program counter, the word length, and the contents of the registers. This is at most c_P times the maximum word length. The word length is always $\leq c \log(m)$ where m is the total number of memory words, and $m \leq n + s$. In total, we need $\leq c_P(s + \log(n + s))$ bits to write down a configuration. When $s \geq \log n$ this is $\leq c_P s$ bits, for a total of $\leq c_P^s$ configurations. Since the machine ultimately stops on valid inputs, configurations cannot repeat, hence the same machine (with no modification) will stop in the desired time. **QED**

Exercise 6.1. Show $\text{FL} \subseteq \text{FP}$.

Similar results hold for alternating time as well. For simplicity we only state it for PH.

Theorem 6.2. $\text{PH} \subseteq \text{PSpace}$.

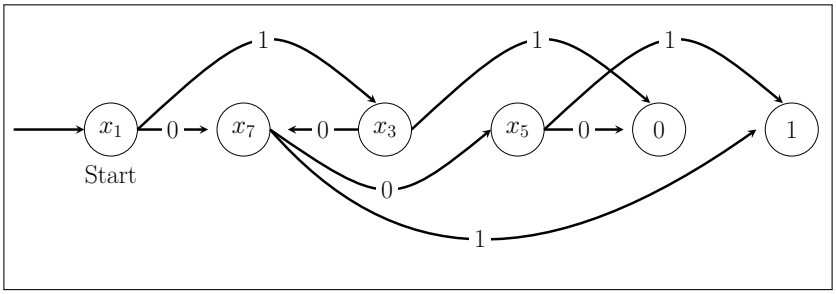


Figure 6.1: Illustration of branching program (Definition 6.3).

The proof is our first example of a general philosophy:

Unlike time, space can be reused.

Proof. Let us first illustrate why NP $\subseteq$ PSpace. So let $f \in$ NP and P be a corresponding program as in De 5.1. On input x, we have to determine if there is $y \in [2]^{n^a} : P(x, y) = 1$, for a constant a depending on P. We shall enumerate over all choices of y, run $P(x, y)$ on all of them, and accept if any run outputs 1. The key point is that we can *reuse* the same space for different runs.

To prove the more general result in the theorem statement, PH $\subseteq$ PSpace, one can proceed by induction similar to Theorem 5.9. The details are left as exercise. **QED**

Gathering this information, we have:

$$\text{L} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PH} \subseteq \text{PSpace} \subseteq \text{Exp}.$$

Just like for Time, for space one has universal programs and a hierarchy theorem. The proofs are very similar to the ones in Chapter 1 and are omitted. The main observation is that the universal program in Lemma 1.1 is space efficient, which can be verified by inspection. The hierarchy theorem implies L $\neq$ PSpace. Hence, analogously to the situation for Time and NTime (section §5.1), we know that at least one of the inclusions above between L and PSpace is strict. Most people seem to think that all are strict, but nobody can prove that any specific one is.

6.1 Branching programs

Branching programs are the non-uniform counterpart of Space, just like circuits are the non-uniform counterpart of Time.

Definition 6.3. A *branching program* is a directed graph. Each node is labeled by either an input variable, in which case it has two outgoing edges labeled 0 and 1, or else it is labeled

with a 0 or a 1, in which case it has no outgoing edges. One special node is the *start* node. A branching program computes a function $f : X \to [2]$ if for every $x \in X$, starting from the start node and following edge labels corresponding to x we reach $f(x) \in [2]$. The *size* of the program is the number of nodes, the *space* is the logarithm of the size.

We denote by $\text{BrSize}(s(n))$ the set of functions $f : X \subseteq [2]^* \to [2]^*$ s.t. there is n_0 and for every $n \geq n_0$ there are m branching programs b_i s.t. $f(x) = (b_0(x), b_1(x), \ldots, b_{m-1}(x))$, and the sum of the sizes of the b_i is $\leq s(n)$.

Finally, $\text{BrL} := \bigcup_{d \in \mathbb{N}} \text{BrSize}(n^d)$.

Just like CktP is the non-uniform analogue of P in that circuits can simulate word programs (Theorem 2.5), BrL is the non-uniform analogue of L in that branching programs can simulate space-bounded computation.

Theorem 6.3. $\text{L} \subseteq \text{BrL}$.

Proof. Each node in the program corresponds to a configuration – that's the 1-line proof.

Let us add some details. Let P be a word program for f that runs in space $s(n) = a \log(n)$ for a constant a. On inputs of length n, we create a branching program B whose nodes are the configurations of P as described in the proof of Theorem 6.1. The number of configurations is $\leq c_P^s$, so the space of B is as desired.

The start node of B is the start configuration of P.

Configurations with instruction STOP are labeled 0 or 1 depending on the value of $R[0]$ (which recall from Definition 6.2 contains the output) , and have no outgoing edges.

Configurations with instruction $R[i] := M[R[j]]$ with $R[j] \in [n]$, i.e., reading an input bit, query bit $R[j]$ of the input, and have two outgoing edges, leading to configurations where $R[i]$ is modified accordingly.

The other configurations don't need to query any input variable and can simply proceed to the next configuration (or can be shortcut altogether). **QED**

A similar result holds for FL, as long as the output length is bounded by a power. We leave this formulation as an exercise.

Definition 6.4. The branching program given by Theorem 6.3 is called the *configuration graph* of P on inputs of length n.

A somewhat space-efficient simulation of circuits by branching programs is known (unlike for Time, see Theorem 6.1 and the notes). Essentially, circuits of size s can be simulated by branching programs using space $\sqrt{s}$.

Theorem 6.4. $\text{CktSize}(s(n)) \subseteq \text{BrSize}(c^{\sqrt{s(n) \log s(n)}})$.

See section 7.8.1 for the proof.

6.2 The power of L

Computing without memory, as in L, seems quite difficult. It turns out that L is a powerful class capable of amazing computational feats that challenge our intuition of efficient computation. Moreover, these computational feats hinge on far-reaching mathematical techniques. To set the stage, we begin with a composition result. In the previous sections we used several times the intuitive fact that the composition of two maps in FP is also in FP, Claim 1.1. This is useful as it allows us to break a complicated algorithm in small steps to be analyzed separately – which is a version of the *divide et impera* paradigm. A similar composition result holds and is useful for space, but the argument is somewhat less obvious.

Lemma 6.1. Let $f_1, f_2 \in \text{FL}$. Suppose $|f_1(x)| \leq |x|^a$ for a constant a. Then $f_2 \circ f_1 \in \text{FL}$.

The basic idea is to run the program for f_2, but whenever it reads a bit from the input, run the program for f_1 to compute this bit "on the fly," reusing the same space for each computation of f_1.

Proof. We are given as input x and i and we wish to compute bit i of $f_2(f_1(x))$. First we determine $|f_1(x)|$. This is done by running the program that decides if $|f_1(x)| < j$ for $j = 1, 2, \ldots$. Each run of f_1 re-uses the same registers. (Let us discuss some details of this step. Note we can't write down $|f_1(x)|$ in a register, but by our assumption on $|f_1(x)|$, we only need $\leq c_a \log n$ bits to represent j. Hence we shall use c_a registers to store j, as in Claim 1.3. Also, when j becomes larger than i, the starting configuration on input (x, j) may have larger word length than the starting configuration on our input, which is (x, i). To address this, we use two registers to simulate one register in the program for f_1. The important thing is that each run of f_1 re-uses the same registers, so we can run all these simulations for various j using c_a registers. At the end, we have computed $|f_1(x)|$, in c_a registers.)

Then we simulate the program for f_2 as if it was run on input $(f_1(x), i)$. Recall the starting configuration has $|f_1(x)|$, which we computed above. Here we again use c_a registers for each register in the program for f_2, as above. Whenever the program for f_2 reads bit j from $f_1(x)$, we supply it "on the fly" by simulating f_1 on input (x, j), re-using the same space for each evaluation of f_1. **QED**

6.2.1 Arithmetic

A first example of the power of L is given by its ability to perform basic arithmetic. Grade school algorithms use a lot of space, for example they employ space $\geq n$ to multiply two n-bit integers.

Theorem 6.5. The following arithmetic problems are in FL:

1. Addition of two naturals.

2. Iterated addition: Addition of any number of naturals.

3. Multiplication of two naturals.

4. Iterated multiplication: Multiplication of any number of naturals.

5. Division of two naturals.

Iterated multiplication is of particular interest because it can be used to compute "pseudorandom functions." Such objects shed light on our ability to prove impossibility results via the "Natural Proofs" connection which we will see in Chapter 18.

Proof of 1. in Theorem 6.5. We are given as input $x, y \in [2]^*$ and an index i and need to compute bit i of $x+y$. Starting from the least significant bits, we add the bits of x and y, storing the carry of 1 bit in memory. Output bits are discarded until we reach bit i, which is output. We only need memory for pointers to x and y, and the carry. Recall from Definition 6.2 we also need to decide if i is less than the output length $|x+y|$. This can be done in a similar way. We do a pass on the input keeping track of the carry and determine how many output bits are needed. **QED**

Exercise 6.2. Prove 2. and 3. in Theorem 6.5.

Proving 4. and 5. is more involved and requires some of those far-reaching mathematical techniques we alluded to before. Division can be reduced to iterated multiplication. In a nutshell, the idea is to use the expansion

$$\frac{1}{x} = \sum_{i \geq 0} (-1)^i (x-1)^i.$$

We omit details about bounding the error. Instead, we point out that this requires computing powers $(x-1)^i$ which is an example of iterated multiplication (and in fact is no easier). So for the rest of this section we focus on iterated multiplication. Our main tool for this is the *remaindering representation* of integers, abbreviated RR.

Theorem 6.6. Let $p_1, ..., p_\ell$ be distinct primes and $m := \prod_i p_i$. Then $\mathbb{Z}_m$ is isomorphic to $\mathbb{Z}_{p_1} \times \ldots \times \mathbb{Z}_{p_\ell}$.

The forward direction of the isomorphism is given by the map

$$x \in \mathbb{Z}_m \to (x \mod p_1, x \mod p_2, ..., x \mod p_\ell) \in \mathbb{Z}_{p_1} \times ... \times \mathbb{Z}_{p_\ell}.$$

For the converse direction, there exist integers $e_1, ..., e_\ell \leq m^c$, depending only on the p_i such that the converse direction is given by the map

$$(x \mod p_1, x \mod p_2, ..., x \mod p_\ell) \in \mathbb{Z}_{p_1} \times ... \times \mathbb{Z}_{p_\ell} \to x := \sum_{i=1}^{\ell} e_i \cdot (x \mod p_i).$$

Each integer e_i is 0 mod p_j for $j \neq i$ and is 1 mod p_i.

Example 6.1. $\mathbb{Z}_6$ is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_3$. The equation $2+3=5$ corresponds to $(0,2)+(1,0)=(1,2)$. The equation $2 \cdot 3 = 6$ corresponds to $(0,2)+(1,0)=(0,0)$. Note how addition and multiplication in RR are performed in each coordinate separately; how convenient.

To compute iterated multiplication the idea is to move to RR, perform the multiplications there, and then move back to standard representation. A critical point is that each coordinate in the RR has a representation of only $c \log n$ bits, which makes it easy to perform iterated multiplication one multiplication at the time, since we can afford to write down intermediate products.

The algorithm is as follows:

Computing the product of input integers $x_1, \ldots, x_t$.

1. Let $\ell := n^3$ and compute the first ℓ prime numbers $p_1, p_2, \ldots, p_\ell$.
2. Convert the input into RR: Compute $(x_1 \mod p_1, \ldots, x_1 \mod p_\ell), \ldots, (x_t \mod p_1, \ldots, x_t \mod p_\ell)$.
3. Compute the multiplications in RR: $(\Pi_{i=1}^t x_i \mod p_1), \ldots, (\Pi_{i=1}^t x_i \mod p_\ell)$.
4. Convert back to standard representation.

Exercise 6.3. Prove the correctness of this algorithm.

Now we explain how steps 1, 2, and 3 can be implemented in FL. Step 4 can be implemented in FL too, but showing this is somewhat technical due to the computation of the numbers e_i in Theorem 6.6. However these numbers only depend on the input length, and so we will be able to give a self-contained proof that iterated multiplication is in BrL. The composition of the steps is then in FL by Lemma 6.1.

Step 1

By Lemma 3.1, the primes p_i have magnitude $\leq n^c$ and so can be represented with $c \log n$ bits. We can enumerate over integers with $\leq c \log n$ bits in logarithmic space. For each integer x we can test if it's prime by again enumerating over all integers y and z with $\leq c \log n$ bits and checking if $x = yz$, say using the space-efficient algorithm for multiplication in Theorem 6.5. (The space required for this step would in fact be $c \log \log n$.)

Step 2

We explain how given $y \in [2]^n$ we can compute $(y \mod p_1, \ldots, y \mod p_\ell)$ in FL. If y_j is bit j of y we have that

$$\begin{aligned} y \mod p_i &= \left[\sum_{j=0}^{n-1} (2^j y_j)\right] \mod p_i \\ &= \left[\sum_{j=0}^{n-1} (2^j \mod p_i) y_j\right] \mod p_i. \end{aligned}$$

Note that the values $a_{i,j} := 2^j \mod p_i$ can be computed in FL and only take $c \log n$ bits. Multiplying by y_j is also in FL by Theorem 6.5. Hence the problem reduces to iterated addition of n numbers which is in FL by Theorem 6.5.

Step 3

This is a smaller version of the original problem: for each $j \leq \ell$, we want to compute $(\Pi_{i=1}^{t} x_i \mod p_j)$ from $x_1 \mod p_j, \ldots, x_t \mod p_j$. However, as mentioned earlier, each $(x_i \mod p_j)$ is at most n^c in magnitude and thus has a representation of $c \log n$ bits. Hence we can just perform one multiplication at the time.

Step 4

By Theorem 6.6, to convert back to standard representation from RR we have to compute the map

$$(y \mod p_1, \ldots, y \mod p_\ell) \to \sum_{i=1}^{\ell} e_i \cdot (y \mod p_i).$$

Assuming we can compute the e_i, this is just multiplication and iterated addition, which is in FL by Theorem 6.5.

Putting the steps together

Combining the steps together we can compute iterated multiplication in FL as long as we are given the integers e_i in Theorem 6.6.

Theorem 6.7. Given integers $x_1, x_2, \ldots, x_t$, and given the integers $e_1, e_2, \ldots, e_\ell$ as above, we can compute $\prod_i x_i$ in FL.

In particular, because the e_i only depend on the input length, but not on the x_i they can be hardwired in a branching program.

Corollary 6.1. Iterated multiplication $\in$ BrL.

Exercise 6.4. Show that given integers $x_1, x_2, \ldots, x_t$ and $y_1, y_2, \ldots, y_t$ one can decide if

$$\prod_{i=1}^{t} x_i = \prod_{i=1}^{t} y_i$$

in L. You cannot use the fact that iterated multiplication is in FL, a result which we stated but not completely proved.

Exercise 6.5. Show that for every $d \in \mathbb{N}$ iterated multiplication of $d \times d$ matrices over the integers $\in$ BrL.

By contrast, when d is not constant this problem is not believed to be in BrL; see TBD.

6.2.2 Graphs

We now give another example of the power of L.

Definition 6.5. The undirected reachability problem: Given an undirected graph G and two nodes s and t, determine if there is a path from s to t.

Standard time-efficient algorithms to solve this problem *mark* nodes in the graph. In logarithmic space we can keep track of a constant number of nodes, but it is not clear how we can avoid repeating old steps forever. Surprisingly, this is possible:

Theorem 6.8. Undirected reachability is in L.

The idea behind this result is that a *random walk* on the graph will visit every node, and can be computed using small space, since we just need to keep track of the current node. Then, one can *derandomize* the random walk and obtain a deterministic walk, again computable in small space. I give a self-contained proof in Chapter 12.

6.2.3 Linear algebra

Our final example comes from linear algebra. Familiar methods for solving a linear system

$$Ax = b$$

require a lot of space. For example using elimination we need to rewrite the matrix A. Similarly, we cannot easily compute determinants using small space. However, a different method exists.

Theorem 6.9. Deciding if an integer linear system has a solution $\in \text{Space}(c \log^2 n)$.

For the proof see the notes.

6.3 Checkpoints

The *checkpoint* technique is a fundamental tool in the study of space-bounded computation. Let us illustrate it at a high level. Let us consider a graph G, and let us write $u \rightsquigarrow^t v$ if there is a path of length $\leq t$ from u to v. The technique allows us to *trade the length of the path with quantifiers*. Specifically, for any parameter b, we can break down paths from u to v in b smaller paths that go through $b-1$ checkpoints. The length of the smaller paths needs be only t/b (assuming that b divides t). We can guess the breakpoints and verify each smaller path separately, at the price of introducing quantifiers but with the gain that the path length got reduced from t to t/b. The checkpoint technique is thus an instantiation of the general paradigm of guessing computation and verifying it locally, introduced in Chapter 5. One difference is that now we are only going to guess *parts* of the computation.

The checkpoint technique

$u \rightsquigarrow^t v \Leftrightarrow \exists p_1, p_2, \ldots, p_{b-1} : \forall i \in [b] : p_i \rightsquigarrow^{t/b} p_{i+1}$,

where we denote $p_0 := u$ and $p_b := v$.

Two aspects are important of this technique. First, it can be applied *recursively* to the problems $p_i \rightsquigarrow^{t/b} p_{i+1}$. We need to introduce more quantifiers, but we can reduce the path length to t/b^2, and so on. Second, it goes hand in hand with the philosophy of re-using space: We can re-use space for different i.

We will see several instantiations of this technique, for various settings of parameters, ranging from $b = 2$ to $b = n^c$. As a first example, we use the checkpoint technique to speed up space-bounded computation with alternations.

Theorem 6.10. $\mathrm{L} \subseteq \bigcup_{i \in \mathbb{N}} \Sigma_i \mathrm{Time}(n^\epsilon)$, for any $\epsilon > 0$.

Proof. Let $f \in \mathrm{L}$. For any given $\epsilon > 0$ we will show that $f \in \Sigma_i \mathrm{Time}(n^{c\epsilon})$ for some i. Since this holds for any ϵ, the result follows. Let G be the configuration graph (Definition 6.4) corresponding to (a program for) f, with $t := n^a$ nodes for inputs of length n, some $a \in \mathbb{N}$. On input $x \in [2]^n$ we need to decide if the start node reaches the node labeled 1 within t steps (w.l.o.g. we assume there is a unique node labeled 1). We apply the checkpoint technique recursively, with parameter $b := n^\epsilon$ (we assume w.l.o.g. this and later quantities are integer). Each application reduces the path length by a factor b. Hence with a/ϵ applications we can reduce the path length to

$$\frac{t}{b^{a/\epsilon}} = \frac{t}{n^a} = 1.$$

Each quantifier ranges over $\leq cb \log n^a$ bits, which is sufficient to guess b configurations. The total number of quantified bits is then $m := c(a/\epsilon) \cdot b \log n^a \leq n^{c\epsilon}$, the latter holding for n large enough.

There remains to check a path of length 1, i.e., an edge. The endpoints of this edge are two configurations u and v which depend on the quantified bits. It suffices to say that they can be computed in time $\leq m^c \leq n^{c\epsilon}$. Once we have computed u and v we can check if u leads to v in one step, which may involve reading one bit of the input x. This check is even faster, can be done in time depending on the program only. **QED**

6.4 The grand challenge for space

We now discuss impossibility results for space, paralleling Chapter 1. Again, we can use diagonalization to prove a space hierarchy. The statement and the proof follow closely Theorem 1.4 without adding new ideas, so they are omitted. However, we mention that, again, from the space hierarchy it follows that $\mathrm{L} \neq \mathrm{PSpace}$. Hence in particular either $\mathrm{P} \neq \mathrm{L}$ or $\mathrm{P} \neq \mathrm{PSpace}$. Thus, we know that at least one important separation between time and space holds. Most people appear to think that *both* hold, but we are unable to prove *either*.

We now turn to non-uniform impossibility results. Whereas it is consistent with our knowledge that P or even NP have linear-size circuits, for branching program we can rule this our and establish a nearly quadratic lower bound for branching programs. A quadratic bound remains open, even for NP.

Definition 6.6. The Element-Distinctness problem: Given n/w vectors of length $w := 2 \log n$, decide if they are all distinct.

Exercise 6.6. Prove that Element-Distinctness $\in \text{BrSize}\left(cn^2/\log n\right)$.

Theorem 6.11. Element-Distinctness $\notin \text{BrSize}\left(n^2/\log^c n\right)$.

The proof is another example of the restrict and simplify method, see section §2.1. A suitable restriction reduces the size of the branching program; at the same time, we can guarantee that the restricted branching program still computes many functions. Comparing the number of functions to the size of the restricted program completes the argument.

Proof. Let B be a branching program computing Element-Distinctness with S variable nodes (and 2 nodes for 0 and 1). Partition its variable nodes in n/w sets depending on which (bit in a) vector they query. One of the sets has size $S' \leq Sw/n$. For any fixing a of the other $n/w - 1$ vectors, we obtain a branching program B_a with S' variable nodes, as the nodes corresponding to fixed variables can be shortcut. The critical observation is that any fixing a to distinct vectors can be recovered, up to permutation, from B_a: The vectors in a are those s.t. $B_a(a) = 1$. In particular, B_a encodes any subset of $[2]^w$ of size $n/w - 1$.

On the other hand, as in the proof of Theorem 2.3 (see Exercise 2.3), we can bound the number of branching programs of size $S' + 2$ by

$$\leq 2^{cS' \log S'} \leq 2^{(S/n) \cdot \log^c n}.$$

Indeed, for each node it suffices to indicate which variable it queries, or if it is a constant, and the two neighbors among $S' + 2$ possible nodes. For each node this takes $\leq c(\log n + \log(S' + 2)) \leq c \log S'$ (assuming w.l.o.g. $S' \geq n$). Overall, we can describe a branching program with $cS' \log S'$ bits.

Combining these two facts and taking logs we get

$$(S/n) \cdot \log^c n \geq \log \binom{2^w}{n/w - 1}.$$

Recalling $w = 2 \log n$, the binomial in the right-hand side is $\geq \binom{n^2}{n/c \log n} \geq n^{cn/\log n} \geq 2^{cn}$ (see Fact B.6) and the result follows. **QED**

No quadratic bound is known for NP.

6.5 Reductions

As in chapters 4 and 5, we can use reductions to relate the space complexity of problems, and we can identify complete problems. We discuss this in turn for PSpace and L.

6.5.1 P vs. PSpace

Definition 6.7. A *quantified boolean formula* (QBF) is a boolean formula where we allow both $\exists$ and $\forall$ quantifiers. The QBF problem: Given a QBF, is it true?

Example 6.2. $\exists x \forall y \exists z : (x \vee y) \wedge (\neg x \vee z)$ is a true QBF.

Theorem 6.12. QBF is complete for P vs PSpace.

Proof. To prove that QBF is in PSpace we use a natural recursive algorithm. Given a QBF, we consider one quantifier Qx and we recursively determine the truth of the formula with $x = 0$ and $x = 1$, reusing the space among recursive calls. If $Q = \exists$ we return *true* if at least one assignment resulted in a true formula, if $Q = \forall$ if both. The space $S(n)$ satisfies

$$S(n) \leq n^c + S(n-1)$$

with solution $S(n) \leq n^c$.

To prove hardness, we use the checkpoint technique to recursively halve the computation time, and we note that we can express this as a QBF. Details follow. Let $f \in \text{Space}(n^a)$ and P be a corresponding program. Let x be an input. As follows from the proof of Theorem 6.1, the configurations of P take $\leq cn^a$ bits and so the running time of P is $\leq t := c^{n^a}$.

To know if P goes from configuration C to C' in $\leq t$ steps, we guess a middle configuration C_m and check if it goes from C to C' in $t/2$ steps, and from C_m to C_2 in $t/2$ steps.

This introduces an $\exists$ quantifier over cn^a bits, and a $\forall$ quantifier on 1 bit. Repeating $\log(t) \leq cn^a$ times, we have reduced the time to 1. The final check to do is to verify if a configuration C goes to a configuration C' in 1 step. This (including computing C and C') can be computed in power time from the quantified bits and the input x. Hence by Theorem 2.5 and Theorem 5.1 we can introduce another $\exists$ quantifier on n^{c_a} bits and write this computation as a 3CNF. **QED**

As for NP, many natural problems are known to be PSpace complete. Among them are many popular games. Recall from Chapter 5 that popular *single-player* games and puzzles are NP-complete (when suitably generalized). By contrast, PSpace complete problems include many popular *two-player* games. This is not surprising, as QBF itself can be seen as a game between player $\forall$ and $\exists$ who alternate setting variables. The game instance consists of the QBF, and player $\exists$ wins if they can make the formula true. For a list of PSpace complete problems, including many games, see the notes.

6.5.2 L vs. P

A very basic problem is complete for L vs P. We are given a circuit *with no variables*, just constants, and we want compute its output. This is in the same spirit of the zero-integer circuit problem from Definition 3.4. But here the setting is even simpler, as we work with plain boolean circuits, as in Chapter 2.

Definition 6.8. The circuit value problem: Given a circuit C with no variables, compute $[C]$.

We can easily solve this problem in power time by computing one gate at the time. The straightforward way of doing this would require much space to store the values of intermediate gates. The next result shows that if we could dramatically decrease the space then L = P would follow.

Theorem 6.13. Circuit value is complete for L vs P.

Proof. Circuit value is in P since we can evaluate one gate at the time. Now let $f \in$ P. We can reduce computing f on input x to a circuit value instance, as in Theorem 2.5. The important point is that this reduction is in FL: given an index to a gate in the circuit, we can compute the type of the gate and index to its children in FL. This can be verified by inspection of the circuit in the proof of Theorem 2.5. The details may be easier to visualize using a different computational model, so they are postponed to Chapter 16. **QED**

Definition 6.9. The monotone circuit value problem: Given a circuit C with no variables and no negations, compute $[C]$.

Exercise 6.7. Prove that monotone circuit value is complete for L vs P. Hint: Problem 2.2.

Recall from section 6.2.3 that finding solutions to linear systems

$$Ax = b$$

has space-efficient algorithms. Interestingly, if we generalize equalities to inequalities the problem becomes P complete. This set of results illustrates a sense in which "linear algebra" is easier than "optimization."

Definition 6.10. The linear inequalities problem: Given a $d \times d$ matrix A of integers and a d-dimensional vector of integers, determine if the system $Ax \leq b$ has a solution over the reals.

Theorem 6.14. Linear inequalities is complete for L vs P.

Proof. The ellipsoid algorithm shows that Linear inequalities is in P, but we will not discuss this classic result; for a reference see the notes. Instead, we focus on showing how given a circuit C with no variables and one output gate we can construct a set of inequalities that are satisfiable iff C outputs 1. We shall have as many variables v_i as gates g_i in the circuit. The inequalities are as follows (recall Definition 2.1 of circuit):

- For a constant gate $g_i := b \in [2]$ add $v_i = b$ (which can be written as $v_i \geq b$ and $v_i \leq b$).
- For a Not gate $g_i := \neg g_j$ add $v_i = 1 - v_j$.

- For an And gate $g_i := g_j \wedge g_k$ add

$$\begin{aligned} & v_i \in [0,1] \\ & v_i \leq v_j \\ & v_i \leq v_k \\ & v_i \geq v_j + v_k - 1. \end{aligned}$$

- For an Or gate we can have a similar set of inequalities, or we can write it using Not and And (Fact B.1).

- For the output gate g_i add $v_i = 1$.

We claim that in every solution to the system above the value of v_i is the value in [2] of gate g_i on input x. This can be proved by induction. For constant and Not gates this is immediate. For an And gate $g_i = g_j \wedge g_k$, note that if $v_j = 0$ then $v_i = 0$ as well because of the equations $v_i \geq 0$ and $v_i \leq v_j$. The same holds if $v_k = 0$. If both v_j and v_k are 1 then v_i is 1 as well because of the equations $v_i \leq 1$ and $v_j + v_k - 1 \leq v_i$. **QED**

6.6 Nondeterministic space

Because of the insight we gained from considering non-deterministic time-bounded computation in section §5.1, we are naturally interested in non-deterministic space-bounded computation. In fact, perhaps we will gain even more insight, because this notion will really challenge our understanding of computation.

How to define non-deterministic space-bounded computation? A naive approach is to do a local change to Definition 5.1 of NTime. This is an ill-fated choice:

Exercise 6.8. Define $\exists \cdot \mathrm{L}$ as (NP Definition 5.1), except that the program P uses logarithmic space. Prove that $\exists \cdot \mathrm{L} = \mathrm{NP}$.

Instead, the definition of non-deterministic space is a variant of the *alternative* definition of NTime with Guess instructions (see discussion in section §5.1).

Definition 6.11. A *nondeterministic* word program is a word program with the extra instruction

- $R[i] :=$ Guess, where $i \in [\ell]$ and ℓ is the number of registers of the program.

This instruction sets $R[i]$ to a value in $[2]^w$, where w is the current word length.

A function $f : X \subseteq [2]^* \to [2]$ is computable in $\mathrm{NSpace}(s(n))$ if there is a nondeterministic word program P that for every $x \in X$ of length $\geq |P|$ satisfies:

- If $f(x) = 1$ then there exists a sequence of values for the Guess instructions such that $P(x)$ outputs 1 in space s, and

- If $f(x) = 0$ then for every sequence of values for the Guess instructions $P(x)$ outputs 0 in space s.

We define

$$\text{NL} := \bigcup_d \text{NSpace}(d \log n),$$
$$\text{NPSpace} := \bigcup_d \text{NSpace}(n^d).$$

How can we exploit this non-determinism? Recall from section 6.2.2 that reachability in *undirected* graphs is in L. It is unknown if the same holds for directed graphs. However, we can solve it in NL.

Definition 6.12. The directed reachability problem: Given a directed graph G and two nodes s and t, decide if there is a path from s to t.

Theorem 6.15. Directed reachability is in NL.

Proof. The proof simply amounts to guessing a path in the graph. The pseudocode is as follows.

```
On input G, s, t:
v := s
For i = 0 to |G|:
  If v = t then ACCEPT
  Guess a neighbor w of v
  v := w
REJECT
```

The space needed is $c(|v| + |i|) = c \log n$. **QED**

We can investigate completeness for NL similarly to NP, and have the following result.

Theorem 6.16. Directed reachability is complete for L vs NL.

Exercise 6.9. Prove this.

The simulation of space by time in Theorem 6.1 holds even for non-deterministic space.

Theorem 6.17. $\text{NSpace}(s(n)) \subseteq \bigcup_a \text{Time}(a^s)$, for any $s(n) \geq \log n$.

Proof. On input x, we compute the configuration graph of the program on input x. The nodes are the configurations, and there is an edge from u to v if the machine can go from u to v in one step. Unlike the deterministic case, this graph now has large out-degree (corresponding to the Guess instruction). Still, the size of this graph is a^s for a constant a depending on the program. This graph can be written down in power time in its size. Once we have the graph we solve reachability on this graph in power time, using say breadth-first-search.

A detail is that we may not know what s is. So we try to write down the graph with $s = \log n$. If we can't, because some configuration points to a larger configuration, we increase s by 1, and repeat. We continue until the graph can be computed; this doesn't affect the time bound. **QED**

The next theorem shows that non-deterministic space is not much more powerful than deterministic space: it buys at most a square. Contrast this with the P vs. NP question! The best deterministic simulation of NP that we know is the trivial NP $\subseteq$ Exp. Thus the situation for space is entirely different.

Theorem 6.18. $\text{NSpace}(s) \subseteq \bigcup_{d\in\mathbb{N}} \text{Space}(ds^2)$, for every $s = s(n) \geq \log n$. In particular, NPSpace = PSpace.

Proof. We use the checkpoint technique with parameter $b = 2$, and re-use the space to verify the smaller paths. Let $f \in \text{NSpace}(s(n))$ and N a corresponding non-deterministic word program. We aim to construct a (deterministic) word program that on input x computes

$$\text{Reach}(C_{\text{start}}, C_{\text{accept}}, c_N^{s(n)}),$$

where $\text{Reach}(u, v, t)$ decides if v is reachable from u in $\leq t$ steps in the configuration graph of N on input x, C_{start} is the start configuration, C_{accept} is the configuration outputting 1, and $c_N^{s(n)}$ is the number of configurations of N, as in the proof of Theorem 6.1.

The key point is how to implement Reach.

```
Computing Reach(u, v, t)
If t = 1 then output [u yields v] and stop.
\\Otherwise, t > 1
For all "middle" configurations m
  If both Reach(u, m, t/2) = 1 and Reach(m, v, t/2) = 1 then Accept.
Reject
```

Let $S(t)$ denote the space needed for computing $\text{Reach}(u, v, t)$. We have $S(1) = c_N s(n)$, while for $t > 1$:

$$S(t) \leq c_N s(n) + S(t/2).$$

This is because we can re-use the space for two calls to Reach. Therefore, the space for $\text{Reach}(C_{\text{start}}, C_{\text{accept}}, c_N^{s(n)})$ is

$$\leq c_N s(n) + c_N s(n) + \ldots + c_N s(n) \leq c_N s^2(n).$$

As in the proof of Theorem 6.17, a detail is that s may be hard to compute. Again, we can run the above procedure with incremental space bounds $s = \log n, \log n + 1, \ldots$ without affecting the final space bound. **QED**

To set the stage for the next result, recall that we do not know if NP is closed under complement. It is generally believed not to be, and if it is then the PH collapses (Theorem 5.10).

What about space? Theorem 6.18 shows $\text{NSpace}(s) \subseteq \text{Space}(cs^2)$. Because the latter is closed under complement, up to a quadratic loss in space, non-deterministic space is closed under complement.

Can we avoid squaring the space? Yes! This is weird!

Theorem 6.19. The complement of Directed Reachability is in NL. In particular, NL is closed under complement.

Proof. We want a non-deterministic word program that given a graph G with $\leq n$ nodes , and s and t accepts iff there is *no* path from s to t. For starters, assume the program has somehow computed the number C of nodes reachable from s. *The key idea is that there is no path from s to t iff there are C nodes different from t reachable from s.* Thus, knowing C we can solve the problem as follows:

```
Algorithm for deciding if there is no path from s to t, given C:
Count:= 0
For all nodes v ≠ t :
  Guess a path from s of length n.
  If path reaches v, increase Count by 1
If Count=C Accept else Reject.
```

There remains to compute C. Let A_i be the nodes at distance $\leq i$ from s, and let $C_i := |A_i|$. Note $A_0 = \{s\}, c_0 = 1$. We seek to compute $C = C_n$. To compute C_{i+1} from C_i, enumerate nodes v (candidates in A_{i+1}). For each v, enumerate over all nodes w in A_i, and check if $w \to v$ is an edge. If so, increase C_{i+1} by 1. The enumeration over A_i is done guessing C_i nodes and paths from s. If we don't find C_i nodes, we reject.

If this procedure runs all the way to C_n (without ever rejecting) then we have computed C_n correctly. **QED**

Exercise 6.10. Given a graph G and nodes s, t, and the count C of the number of nodes reachable from s, explain how to compute in FL a graph G' and nodes s', t' s.t. there is no path from s to t in G iff there is a path from s' to t' in G', and $|G'| \leq |G|^c$.

Give a direct "algorithmic" proof, based on the algorithm in Theorem 6.19, but without using the result as a black-box, or mentioning configuration graphs or completeness.

6.7 An impossibility result for 3Sat

We should turn back to a traditional separation technique – diagonalization.

We put together many of the techniques we have seen to obtain a remarkable impossibility results for 3Sat:

Theorem 6.20. Either $\text{3Sat} \notin \text{L}$ or $\text{3Sat} \notin \text{Time}(n^{1+\epsilon})$ for some constant ϵ.

Note that we don't know if 3Sat $\in$ L or if 3Sat $\in$ Time($n \log^{10} n$). But Theorem 6.20 implies that one of the two is false. One can optimize the methods to push ϵ close to 1, but even establishing $\epsilon = 1$ seems out of reach, and there are known barriers for current techniques (see the notes).

Proof. We assume that what we want to prove is not true and derive the following striking contradiction with the hierarchy Theorem 1.4:

$$\begin{aligned}\text{Time}(n^2) &\subseteq \text{L} \\ &\subseteq \bigcup_d \Sigma_d \text{Time}(n) \\ &\subseteq \text{Time}(n^{1.9}).\end{aligned}$$

The first inclusion holds by the assumption that 3Sat $\in$ L and the fact that any function in Time(n^2) can be reduced to 3Sat in FL, by Theorem 5.1 and the discussion after that, and the composition result for space, Lemma 6.1.

The second inclusion is Theorem 6.10.

For the third inclusion, we start similarly to the first. We first claim that NTime($n^{1+\epsilon}$) $\subseteq$ Time($n^{1+2\epsilon}$) for every ϵ. Then the inclusion follows from Exercise 5.14.

The claim follows from the completeness of 3Sat, Theorem 5.5. More in detail, we can reduce NTime($n^{1+\epsilon}$) to a 3Sat instance of length $m := n^{1+\epsilon} \log^c n$. Then applying the algorithm from 3Sat in the assumption, with running time $n^{1+\epsilon/2}$, we can solve this instance in time $m^{1+\epsilon} = n^{(1+\epsilon)(1+\epsilon/2)} \log^{c(1+\epsilon/2)} n$. For large enough n, this is $\leq n^{1+2\epsilon}$. **QED**

6.8 TiSp

So far in this chapter we have focused on bounding the space usage. It is natural to consider algorithms that operate in little time *and* space.

Definition 6.13. TiSp(t, s) denotes the functions computable by a word program that uses space s as in Definition 6.1 and simultaneously time t.

To illustrate the relationship between TiSp, Time, and Space, consider undirected reachability. It is solvable in Time($n \log^c n$) by breadth-first search, and in logarithmic space by Theorem 6.8. But it isn't known if it is in TiSp($n \log^a n, a \log n$) for some constant a.

Exercise 6.11. Prove the following version of Theorem 6.10: TiSp($n^a, n^{1-\alpha}$) $\subseteq \Sigma_{ca/\alpha}$Time($n$) for any $a \geq 1$ and $\alpha > 0$.

The following is a non-uniform version of TiSp.

Definition 6.14. A branching program of length t and width W is a branching program where the nodes are partitioned in t layers $L_1, L_2, \ldots, L_t$ where nodes in L_i only lead to nodes in L_{i+1}, and $|L_i| \leq W$ for every i.

Thus t represents the time of the computation, and $\log W$ the space. Recall that Theorem 9.3 gives bounds of the form $\geq cn^2/\log n$ on the size of branching program (without distinguishing between length and width). For branching programs of length t and width W this bound gives $t \geq cn^2/W \log n$. Note this gives nothing for power width like $W = n^2$. The state-of-the-art for power width is $t \geq \Omega(n\sqrt{\log n/\log\log n})$ (in fact the bound holds even for subexponential width), see the notes.

We now prove an impossibility result for 3Sat similar to Theorem 6.20, but for TiSp. We seek to rule out algorithms for 3Sat that simultaneously use little space and time, whereas in Theorem 6.20 we even ruled out the possibility that there are two distinct algorithms, one optimizing space and the other time. The main gain is that we will be able to handle much larger space: power rather than log.

Theorem 6.21. $3\text{Sat} \notin \text{TiSp}(n^{1+c_\epsilon}, n^{1-\epsilon})$, for any $\epsilon > 0$.

Proof. We follow closely the proof of Theorem 6.20. We assume that what we want to prove is not true and derive the following contradiction with the time hierarchy Theorem 1.4:

$$\begin{aligned}\text{Time}\left(n^{1+\epsilon}\right) &\subseteq \text{TiSp}\left(n^{1+c_\epsilon}, n^{1-\epsilon^2/2}\right)\\ &\subseteq \Sigma_{c_\epsilon}\text{Time}(n)\\ &\subseteq \text{Time}\left(n^{1+\epsilon/2}\right).\end{aligned}$$

We only sketch the proof of the first inclusion. It follows from the fact that 3Sat is complete under reductions s.t. each bit is computable in time (and space) $\log^c n$; this follows from an inspection of the proof of Theorem 5.4 whose details we omit.

The second inclusion is Exercise 6.11.

The last is the same as in the proof of Theorem 6.20. **QED**

6.9 Computing with a full memory: Catalytic space

> Imagine the following scenario. You want to perform a computation that requires more memory than you currently have available on your computer. One way of dealing with this problem is by installing a new hard drive. As it turns out you have a hard drive but it is full with data, pictures, movies, files, etc. You don't need to access that data at the moment but you also don't want to erase it. Can you use the hard drive for your computation, possibly altering its contents temporarily, guaranteeing that when the computation is completed, the hard drive is back in its original state with all the data intact? [...] Can you still make good use of this additional space?

Turns out you can. We illustrate this in a simple scenario. First, let us define the model.

Definition 6.15. Catalytic log-space (CL) is the class of problems that can be solved in logarithmic space and power time by a machine equipped with an extra power-size memory. For every input and any possible setting of the extra memory, the machine needs to compute the output. At the end of the computation, the extra memory must be in the same setting it was at the beginning.

Theorem 6.22. NL $\subseteq$ CL.

Proof. We are given a graph G with n nodes, and two nodes s and t and we'd like to know if there is a path from s to t. By adding self-loops on t, this is equivalent to asking if there is a path of length exactly n.

Let $W_{u,i}$ for $u \in [n], i \in [n+1]$ be a memory word of $n^c = \log q$ bits, which we view as an element of $\mathbb{Z}_q$, so sums will be modulo q. These words are in the large memory, which is initialized to values we don't know. Consider the procedure Propagate:

```
Propagate:
For i = 0, 1, ..., n − 1
  For all edges u → v in G
    W_{v,i+1} += W_{u,i}
```

Note Propagate can be easily reversed, by subtracting values in reverse order. We call the corresponding operation ReversePropagate

At the end of Propagate, the value $W_{t,n}$ contains the number of paths leading to t modulo q, *plus something which depends on the initial values of the* W. To get rid of the latter, consider running propagate after increasing $W_{s,0}$ by 1. The change in $W_{t,n}$ will be the number of paths from s to t modulo q. Since the number of such paths is $\leq n^n$, by our choice of q this this change will be 0 iff there is a path from s to t. We can check if this change is 0 by comparing the two values bit by bit.

More specifically, we can decide connectivity as follows

```
For i ∈ [log q] {
  Propagate
  Store bit i of W_{t,n} in b
  ReversePropagate
  W_{s,0} += 1
  Propagate
  Xor bit i of W_{t,n} with b
  ReversePropagate
  W_{s,0} −= 1
  If b = 1 then output CONNECTED and STOP
}
Output NOT CONNECTED
```

When the program stops the words W are repristinated. By inspection the program runs in power time. **QED**

In fact, CL is believed to be larger than NL, see Problem 7.4.

6.10 Problems

Problem 6.1. [Acyclic branching programs] Give an example of a branching program that computes a non-trivial function but whose graph (obtained by removing all labels) is not acyclic. Prove that any branching program of size s has an equivalent branching program of size s^2 whose graph is acyclic.

Problem 6.2. Prove that L map reduces in quasi-linear time to QBF where instances of length n have $\leq \log^c n$ variables.

Problem 6.3. Consider the class $\Sigma_{a(n)}\text{Time}(t(n))$ where the number of alternations is $a(n)$ on inputs of length n (as opposed to being fixed to i for every input as in $\Sigma_i\text{Time}(t(n))$.

Prove $\text{L} \neq \Sigma_{a(n)}\text{Time}(n)$ for any growing $a(n)$.

Problem 6.4. A beautiful illustration of the power of L. Consider strings over the alphabet $\{u, v, u^{-1}, v^{-1}\}$ (a.k.a. words in the free group with 2 generators). A string can be simplified by removing adjacent pairs of the type $uu^{-1}, u^{-1}u, vv^{-1}, v^{-1}v$. For example, $uv^{-1}vu^{-1}$ simplifies to uu^{-1} and then to the empty string. On the other hand, the string $uv^{-1}u^{-1}$ cannot be simplified to the empty string. The Simplify problem: Given a string, does it simplify to the empty string?

Prove Simplify is in L. Guideline:

(1) For integers i consider the matrices

$$U_i := \begin{bmatrix} 1 & 2i \\ 0 & 1 \end{bmatrix}, V_i := \begin{bmatrix} 1 & 0 \\ 2i & 1 \end{bmatrix}.$$

Show that $U_iU_j = U_{i+j}$ and so in particular $U_i^{-1} = U_{-i}$; and show the same for the V_i.

(2) Let $\begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{Z}^2$ be a vector and let $i \neq 0$. Show that if $|x| < |y|$ then $\begin{bmatrix} x' \\ y' \end{bmatrix} := U_i \begin{bmatrix} x \\ y \end{bmatrix}$ has $|x'| > |y'|$. Conversely, show that if $|x| > |y|$ then $\begin{bmatrix} x' \\ y' \end{bmatrix} := V_i \begin{bmatrix} x \\ y \end{bmatrix}$ has $|x'| < |y'|$. (This is the so-called ping-pong lemma.)

(3) Show that an alternating product of matrices $U_{i_1}V_{i_2}U_{i_3}V_{i_4}\cdots U_{i_k}$ where the i_j are not zero is not equal to the identity matrix. Note that we begin and end with a U matrix, and we alternate between U and V.

(4) Show that a product of matrices $U_{i_1}V_{i_2}U_{i_3}\cdots V_{i_k}$ where the i_j are not zero is not equal to the identity matrix. Note that we begin with U but end with a V matrix, and as before we alternate U and V matrices. (Hint: Reduce to (3) by multiplying on the left by M^{-1} and on the right by M.)

(5) Show that Simplify is in L.

(6) Consider the generalization of simplify to k elements u_i (corresponding to the free group with k generators). Show that it is in L for every k. Hint: Reduce to the $k = 2$ case by mapping u_i to u^ivu^{-i} (and u_i^{-1} to $u^{-i}vu^i$).

6.11 Notes

A non-trivial simulation of time by space is in [126]. I suspect this should also save a log factor in Theorem 6.1, but I don't know it has been verified for word programs. Theorem 6.4 is from [311], see also the follow-up [245].

Theorem 6.8 is from [234]. The time must have been "ripe:" a concurrent, different proof [278] gives the only slightly weaker space bound $c \log n \log \log n$. Later a simpler proof appeared in [239], a variant of which is in presented in Chapter 12.

Theorem 6.9 follows from [77] which in fact establishes a stronger result, for a class we encounter in Chapter 7.

The use of RR for arithmetic is from [42], which also contains several reductions among arithmetical problems. Some of the steps are from the earlier work [200]. For a discussion of the complexity of division, see [17].

Theorem 6.11 is from [211].

Theorem **??** is from [261].

For a compendium of P-complete problems see [117], for PSpace see [306].

Theorem 6.10 goes back to [212].

Theorem 6.18 is from [241].

Theorem 6.19 was obtained independently in [149, 270]. An earlier surprising collapse paved the way, at least for one of the proofs, cf [181]. The proof in [149] uses a logical formalism, the proof we presented is closer to the one in [270]. The question of whether NL is closed under complement was a.k.a. the "second LBA problem," where LBA stands for linear-bounded automaton, see [178]. As usual, had the answer been different, it would have had applications to the first LBA problem, which is the question whether Theorem 6.18 is tight, and remains open.

See [14, 44] for the state-of-the-art bounds for power-width branching programs.

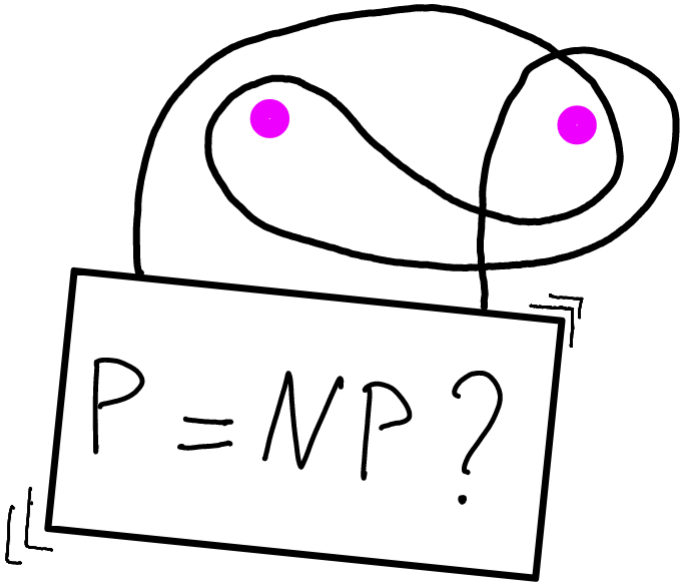
The introductory quote to section §6.7 is from [88], where Theorem 6.20 is proved. This influential work ignited a whole research area. For a survey (not up to date) see [286] or [307]. For the limitations of this type of results, see [59].

That linear inequalities is in P was shown first in [169].

Catalytic computation was introduced in [56]. The text at the beginning of section §6.9 is their introductory paragraph. They also prove that CL contains NL (Theorem 6.22) and supposedly larger classes as well. The proof of Theorem 6.22 that we presented is from [72].

Chapter 7

Depth



In this chapter we study circuits of small depth. Many surprises lay ahead, including a solution to the teaser in Chapter 0!

Definition 7.1. The *depth* of a circuit is the length d of a longest subsequence (or path) $g_1, g_2, \ldots, g_d$ of its gates where g_i is input to g_{i+1} for each $i \in [1..d-1]$.

For $i \in \mathbb{N}$, NC^i is the class of functions $f : X \subseteq [2]^* \to [2]^*$ for which there is $a \in \mathbb{N}$ s.t. for every $n \geq 2$ the function on inputs of length n is computable by a circuit that has depth $a \log^i n$ and size n^a.

Finally,

$$\mathrm{NC} := \bigcup_d \mathrm{NC}^d.$$

Let us begin slowly with some basic properties of small-depth circuits so as to get familiar

with them. The next exercise implies that circuits of depth $d \log n$ for a constant d also have power size, so we don't need to bound the size separately.

Exercise 7.1. A circuit of depth d with m output gates has an equivalent circuit of size $\leq mc^d$.

The next exercises shows how to compute several simple functions by log-depth circuits.

Exercise 7.2. Prove that the Or, And, and Parity functions are in NC^1.

The class NC^0 is also of great interest. It can be more simply defined as the class of functions where each output bit depends on a constant number of input bits. We will see many surprising, useful things that can be computed NC^0, see for example Theorem 11.4.

Exercise 7.3. Prove that $\mathrm{NC}^0 \neq \mathrm{NC}^1$.

7.1 Depth vs space

Now let us compare depth and space.

Theorem 7.1. $\mathrm{NL} \subseteq \mathrm{NC}^2$.

Proof. We show that directed reachability is in NC^2, from which the result follows (exercise). On input a directed graph G on $\leq n$ nodes and two nodes s and t, let M be the $n \times n$ transition matrix corresponding to G, where $M_{i,j} = 1$ iff edge $j \to i$ is in G. Transition matrices are multiplied similarly to matrices over $\mathbb{Z}$, except that "+" is replaced with "$\vee$," which suffices to determine connectivity. To answer directed reachability we compute entry t of $M^n v$, where v has a 1 corresponding to s and 0 everywhere else. (We can modify the graph to add a self-loop on node t so that we can reach t in exactly u steps iff we reach t in any number of steps.)

Computing M^n can be done by squaring $c \log n$ times M. Each squaring can be done in depth $c \log n$, by Exercise 7.2. **QED**

Conversely, we have the following.

Theorem 7.2. $\mathrm{NC}^1 \subseteq \mathrm{BrL}$.

Below in Theorem 7.7 we prove the stronger and much less obvious result that the branching programs can be taken to have width 5.

Proof. We prove by induction that circuits of depth d with 1 output gate have branching programs of size c^d, from which the result follows. The case of depth 1 is immediate. For the induction step, suppose the circuit C has the form $C_1 \wedge C_2$. By induction, C_1 and C_2 have branching programs $B_1$and B_2 each of size c^{d-1}. A branching program B for C of size 2^d is obtained by rewiring the edges leading to states labelled 1 in B_1 to the start state of B_2. The start state of B is the start state of B_1. Its size is $\leq 2 \cdot c^{d-1} \leq c^d$. **QED**

Exercise 7.4. Finish the proof by analyzing the case $C = C_1 \vee C_2$ and $C = \neg C_1$.

7.2 The power of NC^2: Linear algebra

We can informally think that *linear algebra* problems are solvable in NC^2. In particular, we can compute matrix inversion. This can be done over any field. We sketch the ideas over $\mathbb{Q}$ and refer to the notes for other fields. Rationals are represented as pairs of integers. This representation may not be unique, as it is not known how to compute the greatest common divisor in NC.

Theorem 7.3. Given an invertible matrix over $\mathbb{Q}$, we can compute its inverse NC^2.

Sketch. Let A be a $d \times d$ matrix. The key is to compute the (coefficients of the) characteristic polynomial of A,

$$p_A(x) = \det(xI - A) = x^d - s_1 x^{n-1} + s_2 x^{n-2} - \cdots \pm s_n$$

where $s_i \in \mathbb{Q}$. Once we have this polynomial we use that $p_A(A) = 0$ by Fact B.25. This gives

$$A^d - s_1 A^{n-1} + s_2 A^{n-2} - \cdots \pm s_n I = 0.$$

Multiplying this equation by A^{-1} we see that we can compute A^{-1} in NC^2 given the powers A^i and the s_i. The powers A^i can be computed in NC^2 by repeated squaring. To compute the s_i we use that

$$p_A(x) = \prod_{i=1}^{d} (x - \lambda_i)$$

where the λ_i are the eigenvalues of A. Using Fact B.27 we can compute in NC^2 the coefficients of $p_A(x)$ from the power sums

$$p_j(\lambda_1, \lambda_2, \ldots, \lambda_d) = \sum_{i=1}^{d} \lambda_i^j = \mathrm{trace}(A^j).$$

Again, A^j can be computed in NC^2 and the trace is just the sum of the diagonal elements. **QED**

7.3 Formulae

We can equivalently think of NC^1 as power-size *formulae*. A formula is a circuit that you can actually write down on a line, such as $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_1 \wedge x_3)$. By contrast, a circuit can reuse gates, so in general it is more complicated to write down.

Definition 7.2. A *formula* is a circuit where the fan-out is 1(i.e., each gate is input to at most another gate). FormulaSize and FormulaP are defined similarly to CktSize and CktP.

Formulae can have large depth, as in $x_1 \wedge (x_2 \wedge (x_3 \wedge \ldots) \ldots))$. Yet we have:

Theorem 7.4. $\mathrm{NC}^1 = \mathrm{FormulaP}$.

Proof. It suffices to prove this in the case of one output gate. Given a circuit of depth d, consider the formula where there is a gate g for every path from the output to a gate h. The type of gate g is the same as h. Connect (the gate corresponding to) path p to path p' if p' extends p by one edge. The fan-out is 1 by definition, and the number of paths is $\leq c^d$. If d is logarithmic then the formula has power size.

Conversely, we prove by induction on s that any formula f of size $s \geq c$ has circuits of depth $c \log s$. Find a subformula g of size $\in [0.3s, 0.7s]$. For $b \in [2]$ let g_b be f with g replaced by b. Then we have

$$f = (g \wedge g_1) \vee (\neg g \wedge g_0).$$

Note that g, g_0, g_1 all have size $\leq cs$, and the depth of the RHS is at most c plus the maximum depth of g, g_0, g_1. Applying the induction hypothesis completes the proof. **QED**

Exercise 7.5. Prove that g exists.

7.3.1 The grand challenge for formulae

For formulae the best known size lower bound is cubic. We present a weaker and simpler argument. It is convenient to express the bound in terms of the occurrences of variables x_i in the formula, from which a bound for FormulaSize follows (exercise).

Definition 7.3. For a function $f : [2]^n \to [2]$ we denote by $L(f)$ the minimum number of occurrences of input variables x_i in a formula computing f.

The lower bound on $L(f)$ is established via another instance of the restrict and simplify method (section §2.1). Specifically, the next lemma shows that by fixing an input bit of f we can reduce $L(f)$ somehow. Note that there is always a variable occurring $\geq L(f)/n$ times, and fixing that reduces $L(f)$ by $L(f)/n$; but this is not very helpful. The lemma improves this by a factor 1.5.

Lemma 7.1. Let $f : [2]^n \to [2]$ be a function depending on ≥ 2 input bits. We can fix an input bit so that the restricted formula $f' : [2]^{n-1} \to [2]$ has $L(f') \leq L(f)(1 - 1.5/n)$.

Proof. Let ϕ be a formula for f with $L(f)$ literals. Using Fact B.1 we can assume there are no Not gates in the formula. (In other words, the transformation in Problem 2.2 comes at no cost for formulae.) We claim that for every And gate

$$g_i = g_j \wedge \ell_k$$

where ℓ_k is a literal (x_k or $\neg x_k$) the sibling of ℓ_k is connected to a variable different from x_k. The same holds for Or gates.

For some i, there are $\geq L(f)/n$ occurrences of x_i. So there is a fixing of x_i that removes $L(f)/n$ input gates as wells as $\geq$ half of the siblings. Since the siblings contain at least another input gate, the result follows. **QED**

Exercise 7.6. Prove that the sibling contains an input gate x_j different from x_i.

Corollary 7.1. $L(\text{Parity}_n) \geq n^{1.5}$, where Parity_n is the parity function on n bits.

Proof. Note that $(1 - 1.5/n) \leq (1 - 1/n)^{1.5} = ((n-1)/n)^{1.5}$ by Fact B.8, and that the restriction of parity is parity again, possibly complemented (which does not change L). Applying Lemma 7.1 for $n - 1$ times:

$$1 \leq L(\text{Parity}_1) \leq L(\text{Parity}_n) \left(\frac{n-1}{n}\right)^{1.5} \left(\frac{n-2}{n-1}\right)^{1.5} \cdots \left(\frac{1}{2}\right)^{1.5} = L(\text{Parity}_n) \left(\frac{1}{n}\right)^{1.5}.$$

QED

A different argument gives a quadratic lower bound which is tight, see the notes.

7.4 The power of NC^1: Arithmetic

In this section we illustrate the power of NC^1 by showing that the same basic arithmetic which we saw is doable in L (Theorem 6.5) can in fact be done in NC^1 as well.

Theorem 7.5. The arithmetic problems in Theorem 6.5 are in NC^1.

Next we prove that addition, iterated addition, and iterated multiplication are in NC^1 (see Theorem 6.5 for the definitions). Multiplication is a special case; we do not address division. Addition is already non-trivial, as the computation of the carries appears sequential.

Proof that addition is in NC^1.. The approach is to compute all the carries in parallel using *carry look-ahead*. Specifically we note that the i-th carry of the sum $x + y$ is 1 iff there is some less significant position $j < i$ where the carry is generated and it is propagated up to i. This can be written as

$$c_i = 1 \iff \bigvee_{j<i} \left(x_j = 1 \wedge y_j = 1 \bigwedge_{k=j+1}^{i-1} (x_k = 1 \vee y_k = 1) \right).$$

QED

Exercise 7.7. Conclude the proof.

Iterated addition is surprisingly non-trivial. We can't use the methods from the proof of Theorem 6.5. Instead, we rely on a new and very clever technique.

Proof that iterated addition is in NC^1.. We use "2-out-of-3:" Given 3 integers X, Y, Z, we compute 2 integers A, B such that

$$X + Y + Z = A + B,$$

where each bit of A and B only depends on three bits, one from X, one from Y, and one from Z. Thus A and B can be computed in NC^0. If we can do this, then to compute iterated addition we construct a tree of logarithmic depth to reduce the original sum to a sum 2 terms, for which we can use the previous result about addition.

Here's how it works. Note $X_i + Y_i + Z_i \leq 3$. We let A_i be the least significant bit of this sum, and B_{i+1} the most significant one. Note that A_i is the XOR $X_i + Y_i + Z_i$, while B_{i+1} is the majority of X_i, Y_i, Z_i. **QED**

The following corollary will also be used to solve the teaser in Chapter 0.

Corollary 7.2. Majority is in NC^1.

Exercise 7.8. Prove it.

Next we turn to iterated multiplication. The idea is to follow the proof for L in section 6.2.1. We shall use RR again. The problem is that we still had to perform iterated multiplication, albeit only in $\mathbb{Z}_p$ for $p \leq n^c$. One more mathematical fact is useful now: $(\mathbb{Z}_p - \{0\})$ is a cyclic group (Fact B.19).

Proof that iterated multiplication is in NC^1.. We follow the proof for L in section 6.2.1. To compute iterated product of integers $r_1, r_2, \ldots, r_t$ modulo p, use Fact B.19 to compute the logarithms $\ell_1, \ell_2, \ldots, \ell_t$ s.t.

$$r_i = g^{\ell_i}.$$

Then $\prod_i r_i \mod p = g^{\sum_i \ell_i}$. We use the previous result to compute the iterated addition of the exponents in NC^1. Note that computing the exponent of a number mod p, and vice versa, can be done in log-depth since the numbers have $c \log n$ bits (as follows for example by a construction in Theorem 2.1 and Exercise 7.2). **QED**

Recall that the remaindering representation was also used to show that the iterated product of constant-dimension matrices over the naturals is in L (see Exercise 6.5). It is unknown if this problem is in NC^1, though one can make the circuit depth very close to logarithmic, see Problem 7.3.

7.5 Computing with 3 bits of memory

We now present a surprising result that in particular strengthens Theorem 7.2. For a moment, let's forget about circuits, branching programs, etc. and instead consider a new, minimalistic type of programs. Like straight-line programs (Definition 2.1), we do not have control-flow operations or tests. In addition, this new type of program will be *memoryless:* It only has *three* registers, each holding 1 bit.

Definition 7.4. A *memoryless straight-line program* (abbreviated *MSLP*) on n bits is a sequence of operations of the type

$$R_i += R_j \text{ or}$$
$$R_i += R_j x_k$$

where x_k represents an input bit and $k \in [n]$, and $i, j \in [3]$. Here $R_i+ = R_j$ means to add the content of R_j to R_i, while $R_i+ = R_j x_k$ means to add $R_j x_k$ to R_i, where $R_j x_k$ is the product (a.k.a. And) of R_j and x_k. All operations are modulo 2.

For i, j and $f : [2]^n \to [2]$ we say that the program is *for* (or *equivalent to*)

$$R_i+ = R_j f$$

if for every input x and initial values of the registers, executing the program is equivalent to the instruction $R_i+ = R_j f(x)$, where note that R_j and R_k are unchanged.

Note that if we repeat twice a program for $R_i+ = R_j f$ then no register changes (recall the sum is modulo 2, so $1 + 1 = 0$). This feature is critically exploited later to "clean up" computation. We now state and prove the surprising result.

Theorem 7.6. Suppose $f : [2]^n \to [2]$ is computable by circuits of depth d. There is a memoryless straight-line program of length $\leq c4^d$ for

$$R_i+ = R_j f,$$

for every $i \neq j$.

Once such a program is available, we can start with register values $(0, 1, 0)$ and $i = 0, j = 1$ to obtain $f(x)$ in R_0.

Proof. It is convenient to work with circuits with Xor instead of Or gates. This is without loss of generality since $x \vee y = x + y + x \wedge y$. Moreover, we assume that each Xor and Or gate take as input two previous gates (at the cost of introducing extra gates computing input literals).

We proceed by induction on d. When $d = 1$ the output gate is a constant or a literal ℓ_k. For the constant zero we can use the empty program, for the constant 1 we use $R_i+ = R_j$, for x_k we use $R_i+ = R_j x_k$ and for $\neg x_k$ we use $R_i+ = R_j x_k$ followed by $R_i+ = R_j$.

For the induction step, a program for $R_i+ = R_j(f_1 + f_2)$ is simply given by the concatenation of (the programs for)

$$\begin{aligned} R_i+ &= R_j f_1 \\ R_i+ &= R_j f_2. \end{aligned}$$

Less obviously, a program for $R_i+ = R_j(f_1 \wedge f_2)$ is given by

$$\begin{aligned} R_i+ &= R_k f_1 \\ R_k+ &= R_j f_2 \\ R_i+ &= R_k f_1 \\ R_k+ &= R_j f_2. \end{aligned}$$

QED

Exercise 7.9. Prove that the program for $f_1 \wedge f_2$ in the proof works. Write down the contents of the registers after each instruction.

A similar proof works over other fields as well.

We can now address the teaser Theorem 0.1 from Chapter 0.

Proof of Theorem 0.1. Combine Corollary 7.2 with Theorem 7.6. **QED**

We now address the complexity of computing iterated product of matrices.

Definition 7.5. The iterated product of matrices problem: Given square matrices, output the first entry of their product.

We will consider this problem over different domains and for various dimensions.

Corollary 7.3. Iterated product of 3x3 matrices over $\mathbb{F}_2$ is complete for NC^1 under map-reductions in ProjectionsP (Definition 5.3).

Exercise 7.10. Prove this.

Recall from Chapter 6 (see in particular section 6.6) that various graph reachability problems are complete for space-bounded computation. In particular, we have:

Exercise 7.11. Show that iterated multiplication of matrices (of any dimension) over $\mathbb{F}_2$ is hard for BrL under map reductions in ProjectionP.

Does your proof apply to NL as well?

Hence major open questions about computation are related to the following "purely mathematical question" that doesn't make any direct reference to computation:

Question 7.1. Can iterated product of matrices be reduced in ProjectionsP to iterated product of 3×3 matrices? (All matrices over $\mathbb{F}_2$.)

Note this means writing (any one entry of) the iterated product of matrices over variables x_i as (one entry of) a power-length product of 3×3 matrices where each entry is a literal or a constant. If the answer is positive then by above $\mathrm{BrL} \subseteq \mathrm{NC}^1$, if negative then a natural problem (in P) is not in NC^1. A breakthrough either way! More precisely, the question is equivalent to $\mathrm{NC}^1 = \oplus\mathrm{BrL}$, where the latter are "parity branching programs."

7.6 Group programs

The results in section §7.5 are relatively easy to present, but may feel a little *deus ex machina*. Now we present an alternative proof in the framework of groups. The setup may be slightly more convoluted, but the steps in the proof might be a bit more transparent. As is often the case, having two perspectives on a problem is beneficial.

> *It is better to solve one problem five different ways than to solve five problems one way.*

Definition 7.6. A group program π of length ℓ over a group G is a word of length ℓ where each element is raised to an input bit. More formally, it is given by a word $(g_1, g_2, \ldots, g_\ell)$ where $g_i \in G$, an additional element $h \in G$, and a sequence $(k_1, k_2, \ldots, k_\ell) \in [n]^\ell$ of indices to input bits. On input x the program π computes $\pi(x) = \left(\prod_{i=1}^{\ell} g_i^{x_{k_i}}\right) h \in G$. We say π α-computes $f : [2]^n \to [2]$ if $\forall x : \pi(x) = \alpha^{f(x)}$.

That is, the bits of the input x specify which subset of elements in the word to multiply. This simple formulation requires the extra element h; otherwise we can't meaningfully compute $f(0) = 1$.

Exercise 7.12. Consider the following alternative definition of group programs:

The program is given by three words in G^ℓ: $(g_1^{(b)}, g_2^{(b)}, \ldots, g_\ell^{(b)})$ for $b \in [2]$ and $(h_1, h_2, \ldots, h_\ell)$, as well as the sequence $(k_1, k_2, \ldots, k_\ell) \in [n]^\ell$ of indices to input bits, and $h \in G$; on input x the output is $\left(\prod_{i=1}^{\ell} h_i g_i^{(x_{k_i})}\right) h$.

Prove that this alternative definition is equivalent to Definition 7.6.

Computing And. Computing the And of two bits is akin to the mathematical puzzle of hanging a picture with two nails so that removing any one of them makes the picture fall (the solution is depicted at the beginning of the chapter). Actually, this works over any non-abelian group. Indeed, G being non-abelian is the same as saying that there are $a, b \in G$ s.t.

$$ab \neq ba.$$

This is equivalent to saying that the commutator

$$aba^{-1}b^{-1}$$

is not the identity. The following group program then computes the And of bits x and y:

$$a^x b^y a^{-x} b^{-y}.$$

Note that if $x = y = 1$ then we get the commutator which as we just said is not 1. Otherwise, either the as or the bs disappear, and the program evaluates to 1.

To compute circuits we naturally have to iterate this procedure. We can do so if we have non-trivial commutators that can themselves be used as elements in commutators. This approach works for any non-solvable group, a class which includes the group of matrices in Corollary 7.3. This is worked out in Problem 7.2. But now for concreteness we present a closely related construction over a specific group.

A solution over S_5. For concreteness we work with S_5, the group of permutations of 5 elements. Later we discuss other groups. Abusing notation we say that a permutation $g \in S_5$ is a *cycle* if its graph consists of exactly one cycle of length 5. For example, $1 \to 5 \to 2 \to 3 \to 4 \to 1$ is a cycle. We write it compactly as (15234).

Theorem 7.7. Let $f : [2]^n \to [2]$ be computable by a circuit of depth d. Then for any cycle $\alpha \in S_5$, f is α-computed by a group program of length c^d over S_5. In particular, NC^1 is equivalent to power-size branching programs of width 5.

Compare this to the weaker equivalence in Theorem 7.2.

Exercise 7.13. Prove the in particular part, assuming the first part.

To prove this theorem we begin with a lemma stating that the choice of the cycle is immaterial.

Lemma 7.2. Let $\alpha, \beta \in S_5$ be two cycles, let $f : [2]^n \to [2]$. Over the group S_5, f is α-computable with length $\ell \Leftrightarrow$ f is β-computable with length ℓ.

Proof. First recall that any two cycles $\alpha = (\alpha_1 \alpha_2 \dots \alpha_5)$ and $\beta = (\beta_1 \beta_2 \dots \beta_5)$ are conjugate, that is $\alpha = \rho^{-1} \beta \rho$ for

$$\rho := (\alpha_1 \to \beta_1, \alpha_2 \to \beta_2, ..., \alpha_5 \to \beta_5).$$

Hence if $\left(\prod_{i=1}^{\ell} g_i^{x_{k_i}} \right) h$ α-computes f then

$$\rho^{-1} \left(\prod_{i=1}^{\ell} g_i^{x_{k_i}} \right) h \rho$$

β-computes f. We can write this as in Definition 7.6 by Exercise 7.12. **QED**

Lemma 7.3. If $f : \{0,1\}^n \to \{0,1\}$ is α-computable by a group program of length ℓ, so is $1 - f$.

Proof. First apply the previous lemma to α^{-1}-compute f. Then replace the h in Definition 7.6 with $h\alpha$. **QED**

Lemma 7.4. If f is α-computable with length ℓ and g is β-computable with length ℓ then $(f \wedge g)$ is $(\alpha\beta\alpha^{-1}\beta^{-1})$-computable with length 4ℓ.

Proof. Concatenate 4 programs computing as follows:

- α-compute f
- β-compute g
- α^{-1}-compute f
- β^{-1}-computes g.

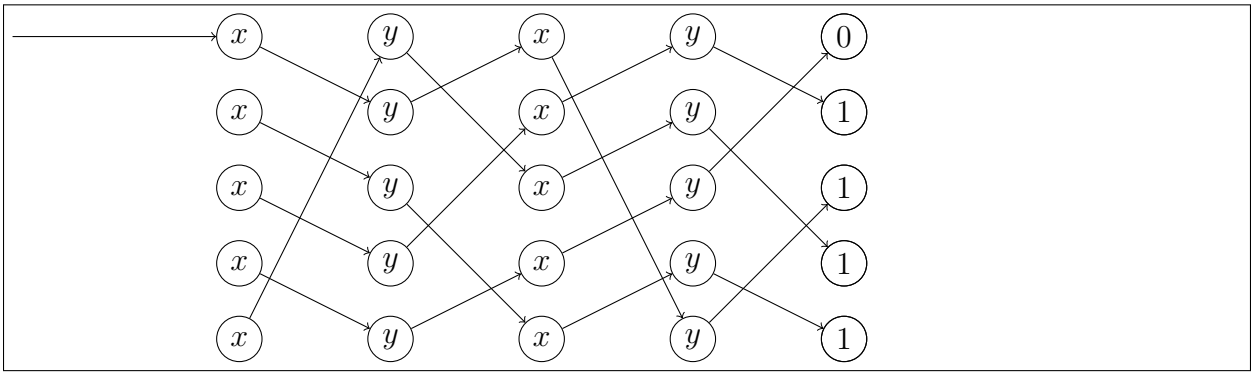


Figure 7.1: The width-5 permutation branching program for And on two bits x and y from the proof of Fact 7.1. All edges drawn have label 1. Every variable node has an edge with label 0 going to the corresponding node in the next column on the right.

If $f(x) = g(x) = 1$ then the concatenated program evaluates to $(\alpha\beta\alpha^{-1}\beta^{-1})$; otherwise it evaluates to 1. **QED**

As mentioned earlier, to iterate this lemma we need to check that the commutator itself is a cycle.

Fact 7.1. There cycles α and β in S_5 whose commutator $\alpha\beta\alpha^{-1}\beta^{-1}$ is also a cycle.

Proof. Let $\alpha := (12345)$, $\beta := (13542)$, we can check that $\alpha\beta\alpha^{-1}\beta^{-1}$ is a cycle. **QED**

Exercise 7.14. Check it.

For an illustration, see figure 7.1. Note that it is easy to compute And with a branching program, but the gain is that this is a *permutation* branching program.

Proof of first part of Theorem 7.7. By induction on d using previous lemmas. **QED**

Exercise 7.15. Give the details of the proof.

The results work over other groups as well. In particular, we can recover the results in section §7.5, see Corollary 7.3, working with the group $GL(3, 2)$ of invertible 3×3 matrices over $\mathbb{F}_2$, see Problem 7.2. This group has size 168. The smallest group size that allows for this simulation is 60, met by the alternating group $A_5 \subseteq S_5$.

7.7 The power NC^0: Cryptography

The results in this section came as a shock, and demonstrate the unsuspected power of NC^0 (and perhaps the weakness of our intuition).

Let us first illustrate the technique in the simplest possible setting. Parity is not in NC^0, since it depends on all the input bits. Nevertheless, NC^0 can do something almost as good: It can generate uniformly distributed input-output pairs of parity. In other words, this is a problem that NC^0 can't solve, but for which nevertheless NC^0 can create instances together with their solutions.

Exercise 7.16. Show that there are constant-depth circuits $C : [2]^n \to [2]^{n+1}$ whose output distribution over a uniform input is the same as $(X, \mathrm{parity}(X))$ for uniform $X \in [2]^n$.

This technique extends to any group. Given the results in the previous sections (such as Theorem 7.7) it is natural to apply it to S_5.

One-way functions are functions that are easy to compute but hard to invert. Most modern cryptography is based on the existence of one-way functions. The theory of one-way functions is rich, see the notes. We give a basic definition which suffices to make our points.

Definition 7.7. A function $f : X \subseteq [2]^* \to [2]^*$ is *one-way* if for every $g : [2]^* \to [2]^*$ in CktP there is n s.t.

$$\mathbb{P}_{x \in X, |x|=n} \left[g(f(x)) \in f^{-1}(x) \right] \leq 1/2.$$

In words, power-size circuits often fail to find a pre-image of $f(x)$ for random x, that is, a value y s.t. $f(y) = f(x)$.

Common candidates one-way functions are computable in NC^1 (for example, they compute integer multiplication and rely on the hardness of factoring). The following result shows that they can be "compiled" to be computable in NC^0.

Theorem 7.8. If there is a one-way function in NC^1 then there is also one in NC^0.

The main technique for proving this is to use that group products are complete for NC^1 (Corollary 7.3), and then use that group products enjoy *random self-reducibility*. The latter means that given a sequence of group elements

$$g_1, g_2, g_3, \ldots, g_n$$

one can easily sample a sequence of group elements that is uniform except that it has the same product as the g_i. To do this, pick $r_1, r_2, \ldots, r_{n-1}$ uniformly in G and output

$$g_1 r_1^{-1}, r_1 g_2 r_2^{-1}, r_2 g_3 r_3^{-1}, \ldots, r_{n-1}^{-1} g_n. \tag{7.1}$$

Note that the first $n-1$ elements are uniformly distributed, while the product of the n elements is the same as $\prod g_i$. Thus, we are "masking" the group program, leaving only its output unchanged. If we think of the program as encoding some "secret" (for example, the input to the one-way function) this allows us to reveal a "masked" version of the program which has the same functionality, but hides the input. Moreover, this rerandomization is done *locally*, and so can be implemented in NC^0. We will later see other applications of this re-randomization property (e.g., Problem 8.1).

Proof. nLet $f \in \mathrm{NC}^1$ be one way. Consider an input length n and let m be the corresponding output length. By Theorem 7.7, each output bit of f is α-computable by a group program of length $D := n^a$, for some $a \in \mathbb{N}$, over S_5. Note that the group program multiplies to 1 of α, so we can think of it as a boolean value. Setting $h = 1$ in Definition 7.6 without loss of generality, we can write

$$f : x \to \begin{bmatrix} \prod_{i \in [D]} g_{1,i}^{x_{k_{1,i}}} \\ \prod_{i \in [D]} g_{2,i}^{x_{k_{2,i}}} \\ \cdots \\ \prod_{i \in [D]} g_{m,i}^{x_{k_{m,i}}} \end{bmatrix}.$$

This map may not be in NC^0.

Consider instead the matrix-valued function which outputs each term in the product:

$$f'' : x \to \begin{bmatrix} \left(g_{1,1}^{x_{k_{1,1}}}, g_{1,2}^{x_{k_{1,2}}}, \ldots, g_{1,D}^{x_{k_{1,D}}}\right) \\ \left(g_{2,1}^{x_{k_{2,1}}}, g_{2,2}^{x_{k_{2,2}}}, \ldots, g_{2,D}^{x_{k_{2,D}}}\right) \\ \cdots \\ \left(g_{m,1}^{x_{k_{m,1}}}, g_{m,2}^{x_{k_{m,2}}}, \ldots, g_{m,D}^{x_{k_{m,D}}}\right) \end{bmatrix}.$$

This function is in NC^0: Each output element only depends on one input bit. However, it is clearly not one way: The output reveals the input bits.

So we simply rerandomize f'' using equation (7.1):

$$f' : x, r \to \begin{bmatrix} \left(g_{1,1}^{x_{k_{1,1}}} r_{1,1}^{-1}, r_{1,1} g_{1,2}^{x_{k_{1,2}}} r_{1,2}^{-1}, \ldots, r_{1,D-1} g_{1,D}^{x_{k_{1,D}}}\right) \\ \left(g_{2,1}^{x_{k_{2,1}}} r_{2,1}^{-1}, r_{2,1} g_{2,2}^{x_{k_{2,2}}} r_{2,2}^{-1}, \ldots, r_{2,D-1} g_{2,D}^{x_{k_{2,D}}}\right) \\ \cdots \\ \left(g_{m,1}^{x_{k_{m,1}}} r_{m,1}^{-1}, r_{m,1} g_{m,2}^{x_{k_{m,2}}} r_{m,2}^{-1}, \ldots, r_{m,D-1} g_{m,D}^{x_{k_{m,D}}}\right) \end{bmatrix},$$

where f' takes as input x as well as a matrix r of $m \times (D-1)$ group elements in S_5.

Note that for every x the output distribution of f' is a uniform matrix whose columns multiply to $f(x)$. So f' is a kind of randomized encoding of f. Now, f' is still in NC^0, yet it is no easier to invert than f. Indeed, suppose a circuit C' inverts f' with probability $1/2$:

$$\mathbb{P}_{x,r}[C'(f'(x,r)) \in f'^{-1}(x,r)] \geq 1/2.$$

Then consider the following distribution on circuits C. "On input $y = f(x) \in [2]^m$ rerandomize y to obtain a uniform matrix M_y whose columns multiply to y. Run $C'(M_y)$." For every x, the distribution of $M_{f(x)}$ is the same as the distribution of $f'(x,r)$ over uniform r. Hence C' will output a pre-image of f' with probability $\geq 1/2$. The first n bits of this preimage are a preimage of f. Finally, we can fix the randomness of C' to obtain a fixed circuit that inverts w.p. $\geq 1/2$. **QED**

Exercise 7.17. Explain which steps in the proof of one-wayness of f' breaks down for f''.

These results can be improved to show that even one-way function in FL (or other classes) imply one-way functions in NC^0, see the notes. Whether the same conclusion follows from the existence of one-way functions in FP remains open.

7.8 Word circuits

In this section we prove a powerful extension of the results in section §7.5 to computation over words. This extension is the main step in the proof of the space-efficient simulation of circuits by branching programs (Theorem 6.4).

Definition 7.8. A *word circuit* (or *word straight-line program)* with word-size w and n input words is a sequence of instructions where instruction i is of the type

$$g_i := g(t, t')$$

where $g : [2]^w \times [2]^w \to [2]^w$ is a function and each of the terms t and t' is either a previous g_j with $j < i$ or an input word x_j with $j \in [n]$. Each instruction can have a different function g. The g_i are called *gates*. A circuit computes a function from $([2]^w)^n$ to $[2]^w$ in the natural way, executing the instructions in order. The last gate is the output. The *depth* of the circuit is defined as in Definition 7.1.

Next we give the memoryless version.

Definition 7.9. A *memoryless straight line word program* on n words of size w is like a memoryless straight line program (Definition 7.4) except that the registers have w bits and we allow instructions

$$R_i += g(t, t')$$

where $g : [2]^w \times [2]^w \to [2]^w$ is a function and each of the terms t and t' is either a register R_k with $k \in [3]$ or an input word x_k with $k \in [n]$. Each instruction can have a different function g.

In the extension, we allow the straight-line program to have word size w' slightly larger than the word size w of the circuit, but we still view it as computing a function on words of w bits (e.g. by padding them with zeroes).

Theorem 7.9. Suppose $f : ([2]^w)^n \to [2]^w$ is computable by a word circuit of depth d. Then there is a memoryless straight-line word program with word size $w \log(cw)$ of length $(cw)^d$ for

$$R_i += f,$$

for every $i \in [3]$.

Proof. We can view words as elements of $\mathbb{F}_2^w$. We extend them to a larger word size $w' = wt$ by viewing each bit as an element in $\mathbb{F}_{2^t}$ and the whole word as an element of $\mathbb{F}_{2^t}^w$, for $t := \log w + c$.

We can write (each output bit of) each function f in the circuit as a polynomial over $\mathbb{F}_2$ of degree $2w$. This is done in the brute-force way as a construction in Theorem 2.1. However, now we can also view f as mapping $\mathbb{F}_{2^t}^w \times \mathbb{F}_{2^t}^w \to \mathbb{F}_{2^t}^w$ (note the value on $\mathbb{F}_2^w \times \mathbb{F}_2^w$ hasn't changed). We will give a simulation for such extended functions.

We proceed by induction on d. For $d = 1$ the output is just $g(x_i, x_j)$ which is a type of instruction we have.

For the induction step, we have programs for

$$R_2 += f_b$$

for $b \in [2]$; we seek a program for $f = g(f_0, f_1)$.

Let α be a generator of the multiplicative group of $\mathbb{F}_{2^t}$ (Fact B.19). The program is this: For each $i \in [2^t - 1]$ we perform

$$\begin{aligned}
R_0 &\times= \alpha^i R_0\\
R_0 &+= f_0\\
R_1 &\times= \alpha^i R_1\\
R_1 &+= f_1\\
R_2 &+= f(R_0, R_1).\\
R_1 &+= f_1\\
R_1 &\times= \alpha^{-i} R_1\\
R_0 &+= f_0\\
R_0 &\times= \alpha^{-i} R_0
\end{aligned}$$

At the end, registers R_0 and R_1 haven't changed, while R_2 has had added the value

$$\sum_{i \in [2^t - 1]} g(\alpha^i R_0 + f_0, \alpha^i R_1 + f_1).$$

We claim that this value equals $g(f_0, f_1)$. It suffices to prove the claim for each output bit. Specifically, we claim that for each polynomial p of degree $\leq 2w$ in $2w$ variables and every $z, y \in \mathbb{F}_{2^t}^{2w}$ we have

$$\sum_{i \in [2^t - 1]} p(\alpha^i z + y) = p(y). \tag{7.2}$$

To match parameters, set $z = (R_0, R_1)$ and $y = (f_0, f_1)$. Note that α^i is a scalar $\in \mathbb{F}_{2^t}$ multiplying the vector $z \in \mathbb{F}_{2^t}^{2w}$ component-wise. For example, we are about to use the fact that coordinate j of $(\alpha^i z)$, denoted $(\alpha^i z)_j$, equals $\alpha^i (z_j)$.

It suffices to verify equation (7.2) when p is a monomial of degree $m \leq 2w$, w.l.o.g. the product of the first m variables). In this case the LHS in equation (7.2) becomes

$$\sum_{i\in[2^t-1]} \prod_{j\in[m]} (\alpha^i z+y)_j = \sum_{i\in[2^t-1]} \sum_{S\subseteq[m]} \left(\prod_{j\in S} \alpha^i z_j\right)\left(\prod_{j\notin S} y_j\right) = \sum_{S\subseteq[m]} \left(\prod_{j\in S} z_j\right)\left(\prod_{j\notin S} y_j\right)\left(\sum_{i\in[2^t-1]} \alpha^{Si}\right).$$

When $S \neq \emptyset$, we have $\alpha^S \neq 1$ because $S \leq 2w$ and α is a generator of a group of size $2^t - 1 \geq 2w$. and by Fact B.4 the last sum equals $(1 - \alpha^{S(2^t-1)})(1 - \alpha^S)$ which is 0 since raising the element α^S to the order $2^t - 1$ of the multiplicative group of $\mathbb{F}_{2^t}$ gives 1 (Fact B.17).

When $S = \emptyset$ the right-hand sum equals $2^t - 1 = 1$. **QED**

Exercise 7.18. Extend Theorem 7.9 to the case of larger fan in. Specifically, for circuits with word size w and fan-in r give a simulation using $r + c$ registers, word size $w \log(cwr)$, and length $(cwr)^d$.

7.8.1 Simulating circuits with square-root space: proof of Theorem 6.4

In this section we use Theorem 7.9 to prove Theorem 6.4. First, we describe a straightforward proof of a result that is weaker but carries the same conceptual message that we can simulate circuits of size s using space s^{1-c}. Given such a circuit, divide its gates into blocks of consecutive b gates. Each block is only connected to $\leq 2b$ other blocks. Hence we can view this as a word circuit with $w = b$ and fan-in $\leq cw$. The depth of this circuit is $d \leq s/b + 1$. Applying Theorem 7.9 we obtain a simulation with $w + c$ registers of $cw \log w$ bits and length w^{cd}. Each instruction is computable by a branching program of size w^{cw^2}. So overall the size is $w^{cw^2} \cdot w^{cd}$. To balance the exponents, we want $w^2 = d$; so we set $w := cs^{1/3}$ and obtain a total branching-program size of $2^{cs^{2/3}\log s}$.

The better size bound can be obtained in a similar fashion but transforming the circuit into a word circuit with fan-in 2.

7.9 Problems

Problem 7.1. Show that Search-3Sat reduces to 3Sat in NC^1.

Problem 7.2. (cf Theorem 9.7) Let G be a finite non-solvable group. By Fact B.16 it has a non-trivial subgroup H whose commutator subgroup (i.e., the subgroup generated by commutators) is itself. Prove that any function $f : [2]^n \to [2]$ computable by depth-d circuits is α-computable by a program of length $\leq c_G^d$ over G, for any $\alpha \in H$.

Problem 7.3. Prove that iterated multiplication of 3×3 matrices of naturals can be computed by power-size circuits of depth $c \log n \log^* n$. Recall $\log^* n$ is the number of times we need to apply log to n to obtain a value < 2.

Guideline:

(1) Show that it suffices to compute the value mod m for an n-bit number m.

(2) Use remaindering representation to reduce this problem to the problem where m has $c \log n$ bits.

(3) Group the matrices into blocks of $c \log n$ to reduce this problem to many instances of the same problem as in (1).

(4) Formulate and solve a recursion for the depth given by this approach.

Problem 7.4. Show that the iterated product of matrices (of any dimension) over $\mathbb{F}_2$ is in CL (Definition 6.15).

7.10 Notes

Arithmetic in NC^1 was first performed in [42]. Matrix inversion in NC^2, Theorem 7.3, is from [77]. As we saw this algorithm relies on computing the characteristic polynomial. The approach in [77] and in Theorem 7.3 works over fields of large enough characteristic. Subsequently, [49] showed how to compute this (and hence matrix inversion) over any field via a different approach.

Theorem 7.4 was proved in [259] and apparently rediscovered in [54].

The lower bound for parity in Corollary 7.1 is from [267]. This is the first power lower bound for formulas, and apparently the paper that introduced the restrict and simplify method. The tight quadratic bound for parity is from [170]. A series of works has proved increasingly stronger bounds for other functions, culminating in a cubic lower bound [132]. The bounds are obtained by strengthening the restrict-and-simplify Lemma 7.1 and combining it with other ideas.

The "2-out-of-3" idea, in the proof of Theorem 7.5, is from [91].

Group programs were introduced in the 60's in [198, 177] where it was shown that any function on G^n can be computed on a simple non-abelian group. The construction involves the use of commutators to compute And, similar to the classic puzzle of hanging a picture with two nails so that removing any one nail makes it fall [260].

Group programs were rediscovered in the 80's as permutation branching programs of constant width. Using essentially the same construction in [198, 177], it was shown in [205] that NC^1 equals the set of functions computable by power-length group programs, over any non-solvable group. ([205] considers functions over the domain $[2]^n$, so can allow the more general class of non-solvable groups instead of the more restrictive simple non-abelian in [198, 177], where the domain is G^n.) After [205], several related simulations were discovered, such as [48]. section §7.6 and Problem 7.2 are from [205]. Theorem 7.6 is a variant of this result from [48]. The proof we presented follows [70]. The extension to word circuits is from [71] (they do not explicitly word circuits though), see also [109].

The proof I presented of Theorem 6.4 is based on the follow-up [245].

Theorem 7.8 is from [24]. They prove a stronger result, where f can be even in NL or $\oplus$L, and f' remains total if f is. Moreover, their techniques extend to other objects such as

cryptographic pseudorandom generators and one-way permutations. For an exposition see [293].

Problem 7.3 is from [160], the outline is from [18].

Chapter 8

Majority

> Once upon a time two daughter sciences were born to the new science of cybernetics. One sister was natural, with features inherited from the study of the brain, from the way nature does things. The other was artificial, related from the beginning to the use of computers. Each of the sister sciences tried to build models of intelligence, but from very different materials. The natural sister built models (called neural networks) out of mathematically purified neurones. The artificial sister built her models out of computer programs.
>
> In their first bloom of youth the two were equally successful and equally pursued by suitors from other fields of knowledge. They got on very well together. Their relationship changed in the early sixties when a new monarch appeared, one with the largest coffers ever seen in the kingdom of the sciences: Lord DARPA, the Defense Department's Advanced Research Projects Agency. The artificial sister grew jealous and was determined to keep for herself the access to Lord DARPA's research funds. The natural sister would have to be slain.
>
> The bloody work was attempted by two staunch followers of the artificial sister [...], cast in the role of the huntsman sent to slay Snow White and bring back her heart as proof of the deed. Their weapon was not the dagger but the mightier pen, from which came a book – *Perceptrons* – purporting to prove that neural nets could never fill their promise of building models of mind: *only computer programs could do this*. Victory seemed assured for the artificial sister. And indeed, for the next decade all the rewards of the kingdom came to her progeny, of which the family of expert systems did best in fame and fortune.
>
> But Snow White was not dead. What [they] had shown the world as proof was not the heart of the princess; it was the heart of a pig.

Can *you* kill Snow White?

After an AI winter, spectacular progress in artificial intelligence has made it even more apparent that (a type of) small-depth circuits have amazing capabilities ranging from playing chess, recognizing images, and so on. *(Artificial) neural networks* are a computing paradigm

that is inspired by the human brain and gives rise to small-depth circuits. Much of the research focus in artificial intelligence is on *training* such networks, but here we will focus only on their performance after training – a non-uniform model of computation.

Definition 8.1. A *threshold* $f : [2]^n \to [2]$ is a function of the form

$$f(x) := \left[\sum_{i \in [n]} w_i x_i \geq t \right].$$

where the $w_i \in \mathbb{R}$ are *weights* and $t \in \mathbb{R}$ is a *threshold*.

A *threshold circuit* is a circuit made only of threshold gates with unbounded fan-in. We denote by ${\rm TC}_{\mathbb{R}}{}^0$ the class of functions $f : X \subseteq [2]^* \to [2]^*$ for which there is $a \in \mathbb{N}$ s.t. for every $n \geq 1$ the function on inputs of length n is computable by a threshold circuit of depth a and size n^a. Finally, we denote by ${\rm TC}^0$ the same class except that the parametersw_i and t in each threshold gate are integers with n^a bis.

Exercise 8.1. Prove that any function on $n = 1$ bit is computable by a threshold circuit of size 1.

Exercise 8.2. Prove ${\rm TC}^0 \subseteq {\rm NC}^1$.

One can define ${\rm TC}^1$ etc. as in Definition 7.1, but we won't need that.

One can show that without loss of generality the weights and and the threshold in a circuit of size s can be taken to be integers with absolute value $\leq 2^{cs}$. This can even be reduced to s^c at the cost of increasing the size by a power and the depth by one. In particular:

Theorem 8.1. ${\rm TC}_{\mathbb{R}}{}^0 = {\rm TC}^0$.

A striking instantiation of the grand challenge is that it is open to prove impossibility results for threshold circuits of depth 2. Even the status of very simple-looking functions is unknown:

Question 8.1. *Is inner product mod* 2 *in* $TC^0_{\mathbb{R}}$ *(i.e., a threshold of thresholds)?*

As usual, a good explanation for this ignorance is the power of TCs, of which we give several examples next.

Exercise 8.3. A function $f : [2]^* \to [2]$ is *symmetric* if it only depends on the weight of the input. Prove that any symmetric function is in ${\rm TC}^0$.

8.1 The power of ${\rm TC}^0$: Arithmetic

Theorem 8.2. The arithmetic problems in Theorem 6.5 are in ${\rm TC}^0$.

The proof follows closely that for ${\rm NC}^1$ in section §7.4 (which in turn was based on that for FL). Only iterated addition requires a new idea.

Exercise 8.4. Prove that iterated addition of naturals is in ${\rm TC}^0$. (Hint: Write input as $n \times n$ matrix, one number per row. Divide columns into blocks.)

8.2 Neural networks

Neural networks are made of a "small" number of layers, each consisting of a large number of *(artificial) neurons*. A neuron is a generalization of a threshold: We allow real numbers as input, and rather than checking if the sum surpasses a threshold, we compute an *activation function* of the sum. Several activation functions are considered. In first approximation, we can think of σ as being simply a threshold. Another popular choice is the *ReLU* (defined next).

Definition 8.2. A *neuron* is a function mapping $(x_1, \ldots, x_m) \in \mathbb{R}^m \to \sigma(\sum_i w_i x_i) \in \mathbb{R}$, where $w_i \in \mathbb{R}$ are weights and $\sigma : \mathbb{R} \to \mathbb{R}$ is an *activation function.* The *ReLU* (rectified linear unit) activation function is $\sigma(s) := \max\{0, s\}$. A *neural network* is like a threshold circuit but with neurons instead of thresholds.

Using the fact that TC^0 can compute arithmetic, one can show that neural networks can be simulated by threshold circuits (on integer inputs, and when all the weights are integer).

8.3 TC^0 vs. NC^1

Another great question is whether $\mathrm{TC}^0 = \mathrm{NC}^1$. It is known that Parity cannot be computed by majority circuits of size $n^{1+\epsilon}$ and depth $c \log 1/\epsilon$. Note that *the size bound approaches n exponentially fast with the depth.* This tradeoff is known to be tight for parity, and a natural question is whether we can improve on it and prove stronger bounds, say for iterated multiplication of elements from a group such as S_5, see section §7.6. The following result shows that, in fact, slightly stronger bounds would already imply $\mathrm{TC}^0 \neq \mathrm{NC}^1$. We state this result for TC^0 and S_5 but the proof does not use anything specific about the types of gates or the group: similar results holds for various circuit classes and associative problems.

Theorem 8.3. Suppose $\mathrm{TC}^0 = \mathrm{NC}^1$. Then there is $a \in \mathbb{N}$ s.t. for every ϵ iterated product of n elements in S_5 can be computed by threshold circuits of depth $d' := a \log 1/\epsilon$ and size $d' n^{1+\epsilon}$.

Proof. The problem is in NC^1. By assumption, it has threshold circuits of depth d and size n^k for constants d and k. Exploiting the associativity of the problem, we compute the product recursively according to a regular tree. The root is defined to have level 0. At Level i we compute n_i products of $(n^{1+\epsilon}/n_i)^{1/k}$ elements. At the root ($i = 0$) we have $n_0 = 1$.

By the assumption, each product at Level i has threshold circuits of size $n^{1+\epsilon}/n_i$ and depth d. Hence Level i can be computed by threshold circuits of size $n^{1+\epsilon}$ and depth d.

We have the recursion

$$n_{i+1} = n_i \cdot (n^{1+\epsilon}/n_i)^{1/k}.$$

The solution to this recursion is $n_i = n^{(1+\epsilon)(1-(1-1/k)^i)}$, see below.

For $i = ck \log(1/\epsilon)$ we have $n_i = n^{(1+\epsilon)(1-\epsilon^2)} > n$; this means that we can compute a product of $\geq n$ elements, as required.

Hence the total depth of the circuit is $d \cdot ck \log(1/\epsilon)$, and the total size is $\leq$the depth times $n^{1+\epsilon}$.

It remains to solve the recurrence. Letting $a_i := \log_n n_i$ be the exponent of n_i we have:

$$\begin{aligned} a_0 &= 0 \\ a_{i+1} &= a_i(1-1/k) + (1+\epsilon)/k = a_i\beta + \gamma \end{aligned}$$

where $\beta := (1-1/k)$ and $\gamma := (1+\epsilon)/k$.

This gives

$$a_i = \gamma \sum_{j \leq i} \beta^j = \gamma \frac{1-\beta^{i+1}}{1-\beta} = (1+\epsilon)(1-\beta^{i+1}).$$

QED

8.4 The power of Majority

An impossibility result is just one of the things we can ask about a model. A natural strengthening is proving an *average-case* impossibility result. One motivation for this quest is that an impossibility result may be irrelevant to instances that occur "in nature," if the latter follow a specific distribution. Another motivation comes from cryptography, where a random secret key should give a hard cryptosystem.

So, for a distribution D on inputs we can ask how many mistakes are made by any function in a class F to compute h over D. We call this the *hardness* of h. If h is boolean, either the constant 0 or 1 compute it w.p. $\geq 1/2$. So the maximum hardness of h is $1/2$, if these constants belong to F as is typically the case.

When the hardness approaches $1/2$, which is the important setting where functions in F can't compute h much better than random guessing, it is more natural to consider the distance between the hardness parameter and $1/2$, which is called *correlation*. We now define these quantities which will be used extensively in several subsequent chapters.

Multiple occurrences of D in the same expression indicate the same sample from D.

Definition 8.3. Let D a distribution over the input set X.

We say that $h : X \to [2]$ is δ-*hard* for $f : X \to [2]$ w.r.t. D (or over D) if $\mathbb{P}[f(D) \neq h(D)] \geq \delta$, and that it is δ-hard for F w.r.t. D if it is δ-hard for every $f \in F$.

The *correlation* between h and f w.r.t. D is

$$|\mathbb{E}[(-1)^{f(D)+h(D)}]| = |\mathbb{P}[f(D) = h(D)] - \mathbb{P}[f(D) \neq h(D)]| = |1 - 2\mathbb{P}[f(D) \neq h(D)]|.$$

We also introduce the notation $\mathbb{E}e[y]$ for $\mathbb{E}[(-1)^y]$ which allows us to write correlation as

$$|\mathbb{E}e[f(D) + g(D)]|\,.$$

We say h has correlation $\leq \delta$ with F w.r.t. D if it has correlation $\leq \delta$ w.r.t. D with any $f \in F$.

If D is not specified it is assumed to be the uniform distribution.

Thus the correlation (or hardness) between two functions is a measure of how often the functions agree. To illustrate parameters, if $h = f$ or $h = 1 - f$ then the correlation is one. The hardness is no larger than 0 in the first case and is 1 in the latter. Typical complexity classes are closed under complement, in which case the fact that we take absolute values in the correlation is immaterial and hardness and correlation are equivalent. If h and f disagree on exactly one input in $[2]^n$ the correlation is $1 - 1/2^n - 1/2^n = 1 - 2/2^n$ and the hardness is $1/2^n$. If they disagree on exactly half the inputs the correlation is zero and the hardness is $1/2$. For any f, most functions h have correlation close to 0 with f.

Exercise 8.5. [Average-case/correlation version of Theorem 2.3] Prove that for every $n \geq c$ there are functions $h : [2]^n \to [2]$ that have correlation 2^{-cn} with circuits of size 2^{cn}.

At first sight, average-case hardness and correlation bounds seem stronger than impossibility. In fact, *impossibility* results are *equivalent* to strong correlation bounds! This is where TC^0 kicks in: The equivalence holds for models that can compute majority.

Theorem 8.4. [Computing $\iff$ correlating over any distribution] Let F be a set of functions mapping $[2]^n$ to $[2]$; and let h be a function.

[$\Leftarrow$] Suppose for every distribution D on $[2]^n$ there is $f \in F$ s.t. $\mathbb{E}e[f(D) + h(D)] \geq \epsilon$. Then there exist cn/ϵ^2 functions $f_i \in F$ s.t.

$$h = \mathrm{Majority}(f_1, f_2, \ldots, f_{cn/\epsilon^2}).$$

[$\Rightarrow$] Suppose $h = \mathrm{Majority}(f_1, f_2, \ldots, f_t)$ for odd t. Then for any distribution D there is i s.t. $\mathbb{E}e[f_i(D) + h(D)] \geq 1/t$.

Exercise 8.6. Prove the [$\Rightarrow$].

In particular, for classes C that include majority and are suitably closed under composition, such as CktP, NC^1, or TC^0, we get that $h \notin \mathrm{C}$ iff there is a distribution D for which any $f \in \mathrm{C}$ has correlation $\leq 1/n^a$ for any a. In other words, *superpower correlation bounds for some distribution are necessary and sufficient for superpower impossibility.*

In the above theorem we need correlation under *every* distribution. Jumping ahead, in section 11.2.2 we will study a similar connection but under the *uniform* distribution. The proofs are related, and they still rely on majority.

We now develop machinery to understand and prove Theorem 8.4. A useful viewpoint, here and elsewhere, is the *equivalence between randomness in the input and having it in the model*:

Corollary 8.1. Let h be a function, and F a set of functions. There is a distribution over F s.t. $\mathbb{E}e[F(x) + h(x)] \geq \alpha$ for every input x iff for every distribution D over inputs there is $f \in F$ s.t. $\mathbb{E}e[f(D) + h(D)] \geq \alpha$.

Exercise 8.7. Prove the "only if" direction of Corollary 8.1.

The "if" direction of Corollary 8.1 is a special case of the *min-max theorem from game theory, a.k.a. linear-programming duality, theorem of the alternatives for linear systems, etc*, stated next.

Theorem 8.5 (Linear duality)**.** Let X and Y be sets, $p : X \times Y \to \mathbb{R}$, and $\alpha \in \mathbb{R}$. Then either

(1) there is a distribution D_X on X s.t. $\mathbb{E}_{D_X} p(D_X, y) < \alpha$ for every $y \in Y$, or
(2) there is a distribution D_Y on Y s.t. $\mathbb{E}_{D_Y} p(x, D_Y) \geq \alpha$ for every $x \in X$.

Equivalently, $\neg(1) \Rightarrow (2)$: If for every distribution D_X on X there is $y \in Y : \mathbb{E}_{D_X} p(D_X, y) \geq \alpha$ then (2). In words, if for every distribution on X there is y that gets $\geq \alpha$ in expectation, then in fact there is a single distribution on Y that for every x get $\geq \alpha$ in expectation.

Exercise 8.8. Prove the "if" direction of Corollary 8.1.

Theorem 8.4 follows from Corollary 8.1 and tail bounds (Lemma B.1), similarly to the proof of the error-reduction Theorem 3.1 for BPTime.

Proof of Theorem 8.4, $\Leftarrow$. Use Corollary 8.1 to get a distribution on F. The majority of cn/ϵ^2 samples from F has error $< 2^{-n}$ (see the proof of Theorem 3.1 or Lemma B.1). By a union bound (as in the proof of Theorem 3.6) we can fix the samples to the f_i. **QED**

By Theorem 8.4 to prove impossibility results for circuits for depth-2 TC^0 it suffices to prove correlation bounds with depth-1 TC^0 circuits, i.e., a single majority gate. Such correlation bounds will be proved in Theorem 13.7 for inner product modulo 2. So it follows that inner product modulo 2 is not in depth-2 TC^0 (cf Question 8.1).

8.5 Problems

Problem 8.1. Suppose there is $f \in \mathrm{TC}^0$ s.t. $\mathbb{P}_{g_i \in S_5}[f(g_1, g_2, \ldots, g_n) = \prod_i g_i] \geq 0.51$. Show $\mathrm{NC}^1 = \mathrm{TC}^0$.

8.6 Notes

The introductory quote is from [222]. The broader history is amusing and may serve as an example of the power of impossibility results. Neural networks were studied since the 40's [199]. In the book [237] it was already pointed out that a constant-depth neural network can compute any function, though there was no good proposal for training such networks. The book referred to in [222] is [204]. What it showed is impossibility results for minimalistic neural networks, consisting of a threshold gate applied to And gates. Specifically, it showed that such models cannot compute parity (or other simple functions), unless the fan-in of the Ands is large. The term "perceptron" is sometimes used to refer to this model, though the meaning varies in the literature. Their result said nothing about networks of larger depth, yet various sources blame it for the onset of the AI winter, during which funding for research

in neural networks was particularly hard to find. Perhaps a better explanation is that the hardware, data, and the math weren't yet there. The work [46] "vindicate[d] the reputation of the much maligned perceptron" by showing that small, probabilistic perceptrons can simulate AC^0 (see Definition 9.1).

The $\mathrm{TC}^0_{\mathbb{R}} = \mathrm{TC}^0$ Theorem 8.1 is from [105].

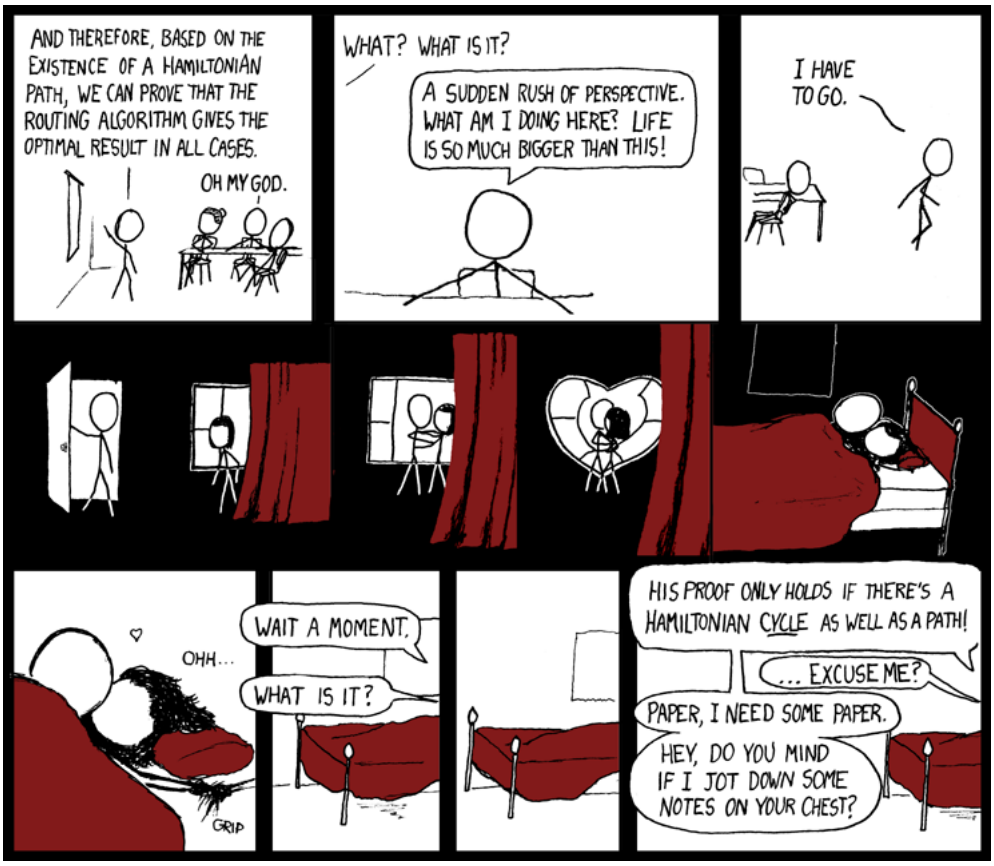
The $n^{1+c \log d}$ lower bound for computing parity by majority circuits is from [152]. A weaker version of Theorem 8.3, which however contains the main message, is from [19]. The stated version is in [66].

The easy direction (Exercise 8.6) of Theorem 8.4 is the "discriminator" lemma in [125].

Corollary 8.1 is from [314] and has been very influential.

Chapter 9

Alternation



https://xkcd.com

In this chapter we study constant-depth circuits with And, Or, and modular gates of unbounded fan-in. This is perhaps the best understood model of computation, and there is a wealth of far-reaching mathematical techniques surrounding it, as well as deceptively

simple open problems. Moreover, this chapter contains an important message regarding the grand challenge. We'll get to it shortly after we define the model.

We have already encountered CNFs in Definition 4.8. A DNF, short for *disjunctive normal form*, is the complement: the output is Or and the other gates are And and are called *terms*. In this chapter we study *alternating circuits* which are a natural generalization of CNFs and DNFs to higher depth.

Definition 9.1. An *alternating circuit* of size s with n input bits and m output bits is a sequence of s instructions where instruction $i \in [s]$ is of one of the following types:

- $g_i := \bigwedge_{j \in [r]} t_j$
- $g_i := \bigvee_{j \in [r]} t_j$

where each t_j is either a gate g_j with $j < i$ or a literal, and $r \in \mathbb{N}$. The last m gates are the output.

We denote by AC^0 the class of functions $f : X \subseteq [2]^* \to [2]^*$ for which there is $d \in \mathbb{N}$ s.t. for every $n \geq 1$ the function on inputs of length n is computable by an alternating circuit of depth d and size n^d.

The name "alternating" comes from the fact that the input gates to an And gate g_i are without loss of generality Or gates g_j, since any And gate g_j can be merged with g_i exploiting that the fan-in is unbounded. And vice versa. Therefore And and Or gates alternate.

Example 9.1. And on n bits is an alternating circuit of depth 1.

DNFs and CNFs are alternating circuits of depth 2.

An (unbounded fan-in) And of DNFs or an Or of CNFs, is an alternating circuit of depth 3.

Addition is in AC^0. We can implement the carry look-ahead construction in the proof of Theorem 7.5.

It will follow from the impossibility results that all the other arithmetic problems in Theorem 6.5 are not in AC^0. The detaisl are left as exercise.

We also consider a generalization where we also allow *modular counting* gates.

Definition 9.2. A Mod-m function $f : [2]^n \to [1]$ is of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = g\left(\sum_{i \in [n]} x_i \mod m\right)$$

for some $g : [m] \to [2]$.

An *alternating circuit with Mod-m gates* is like an alternating circuit but we also have Mod-m gates. The class AC^0-Mod-m is defined analogously to AC^0. Mod-2 functions are also called *parity* and denoted $\oplus$. We denote $\mathrm{ACC}^0 := \bigcup_{m \in \mathbb{N}} \mathrm{AC}^0$-Mod-$m$, for "alternating circuits with counters."

We have

$$\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{AC}^0\text{-Mod-2} \subseteq \mathrm{AC}^0\text{-Mod-6} \subseteq \mathrm{ACC}^0 \subseteq \mathrm{TC}^0$$

with all inclusions being by definition except that last one that holds by Exercise 8.3.

This chapter contains an important message regarding the grand challenge. For alternating circuits, even equipped with parity gates, we can prove exponential impossibility bounds. Even simple functions like Majority cannot be computed in depth d and size $2^{n^{c/d}}$ (Theorem 9.1). At the same time, we will also show (section §9.3) that alternating circuits are powerful enough to simulate L (or even larger classes) with the same tradeoff between depth and size, up to constants. Specifically, any function $f \in \mathrm{L}$ has on inputs of length n alternating circuits of depth d and size $2^{n^{c_f/d}}$. Therefore:

> The impossibility results we can prove for alternating circuits are best possible short of proving a major separation such as $\mathrm{L} \neq \mathrm{P}$.

In fact, this message is apparent even for depth 3.

> Simple functions like parity require depth 3 circuits of size $2^{c\sqrt{n}}$ (Corollary 9.3). Improving this bound implies new results for branching programs and log-depth circuits (see section §9.3).

This exciting state of affairs is discussed further in Chapter 19.

Another important message of this chapter is that alternating circuits and AC^0 closely correspond to alternating programs and the PH (section §5.5). In particular, AC^0 can be seen as a "scaled down" version of PH. This connection will illuminate the proof that $\mathrm{BPP} \subseteq \mathrm{PH}$.

We begin by presenting the impossibility results. To set the stage, let's prove strong results for depth 2, that is, DNFs or CNFs.

Exercise 9.1. Prove that Parity and Majority each require DNFs of size $\geq 2^{cn}$. Hint: What if you have a term with $< n/2$ variables?

Even depth 3 is much more challenging.

9.1 The polynomial method

In this section we introduce a new and far-reaching technique to prove impossibility results for alternating circuits. This technique departs from the restrict-and-simplify method. Using it, we obtain the following result:

Theorem 9.1. Suppose an alternating circuit of depth $d > 1$ and size s with parity gates computes Majority on n bits. Then $s \geq 2^{cn^{0.5/(d-1)}}$.

The proof uses *the polynomial method* which goes by "simulating" alternating circuits by *low-degree polynomials*. We then show that Majority does not have such simulation. The simulation of circuits by polynomials is not exact, but probabilistic. It works over various fields, and for simplicity we focus on $\mathbb{F}_2$.

Example 9.2. Recall that a polynomial in n variables over $\mathbb{F}_2$ is an object like

$$p(x_1, x_2, \ldots, x_n) = x_1 \cdot x_2 + x_3 + x_7 \cdot x_2 \cdot x_1 + x_2 + 1.$$

Because we are only interested in inputs in $\mathbb{F}_2$ we have $x^i = x$ for any $i \geq 1$ and any variable x, so we don't need to raise variables to powers bigger than one.

The And function can be written as the polynomial

$$\text{And}(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} x_i.$$

The Or function can be written as

$$\text{Or}(x_1, x_2, \ldots, x_n) = 1 + \text{And}(1 + x_1, 1 + x_2, \ldots, 1 + x_n) = 1 + \prod_{i=1}^{n} (1 + x_i).$$

For $n = 2$ we have

$$\text{Or}(x_1, x_2) = x_1 + x_2 + x_1 \cdot x_2.$$

Definition 9.3. A distribution P on polynomials over $\mathbb{F}_2$ computes a function $f : \mathbb{F}_2^n \to \mathbb{F}_2$ with error ϵ if for every x we have

$$\mathbb{P}_P[P(x) = f(x)] \geq 1 - \epsilon.$$

We call P a *probabilistic polynomial*.

The next theorem shows that AC^0-$\oplus$ has low-degree probabilistic polynomials over $\mathbb{F}_2$ (with small error). In particular, it correlates with a low-degree polynomial.

Theorem 9.2. [AC^0-$\oplus$ computable by low-degree probabilistic polynomials.] Let $C : [2]^n \to [2]$ be (the function computed by) an alternating circuit of depth d and size s with parity gates. Then there is a distribution P on polynomials over $\mathbb{F}_2$ of degree

$$(c \log s)^{d-1} \cdot \log 1/\epsilon$$

that computes C with error ϵ.

In particular, there is a polynomial p over $\mathbb{F}_2$ of the same degree such that

$$\mathbb{P}_{x \in [2]^n}[C(x) \neq p(x)] \leq \epsilon.$$

The important point in Theorem 9.2 is that if the depth d is small (e.g., constant) (and the size is not enormous and the error is not too small) then the degree is small as well. For example, for AC^0-$\oplus$ the degree is power logarithmic for constant error.

9.1.1 Proof of Theorem 9.2

The key result is the simplest case: C is just a single Or gate.

Lemma 9.1. For every ϵ and n there is a distribution P on polynomials of degree $\log 1/\epsilon$ in n variables over $\mathbb{F}_2$ that computes Or with error ϵ. The same holds for And.

Proof. For starters, pick the following distribution on linear polynomials: For a uniform $A = (A_1, A_2, \ldots, A_n) \in [2]^n$ output the polynomial

$$p_A(x_1, x_2, \ldots, x_n) := \sum_i A_i \cdot x_i.$$

Let us analyze how p_A behaves on a fixed input $x \in [2]^n$:

- If $\mathrm{Or}(x) = 0$ then $p_A(x) = 0$;
- If $\mathrm{Or}(x) = 1$ then $\mathbb{P}_A[p_A(x) = 1] \geq 1/2$.

While the error is large in some cases, a useful feature of p_A is that it never makes mistakes if $\mathrm{Or}(x) = 0$. This allows us to easily reduce the error by taking $t := \log 1/\epsilon$ polynomials p_A and combining them with an Or.

$$p_{A_1,A_2,\ldots,A_t}(x) := p_{A_1}(x) \vee p_{A_2}(x) \vee \cdots \vee p_{A_t}(x).$$

The analysis is like before:

- If $\mathrm{Or}(x) = 0$ then $p_{A_1,A_2,\ldots,A_t}(x) = 0$;
- If $\mathrm{Or}(x) = 1$ then $\mathbb{P}_{A_1,A_2,\ldots,A_t}[p_{A_1,A_2,\ldots,A_t}(x) = 1] \geq 1 - (1/2)^t \geq 1 - \epsilon$.

It remains to bound the degree. Each p_{A_i} has degree 1. The Or on t bits has degree t by Example 9.2. Hence the final degree is $t = \log 1/\epsilon$.

The case of And is analogous and is left as exercise. **QED**

Proof of Theorem 9.2. The "in particular" part follows because averaging over x we have

$$\mathbb{P}_{x,P}[C(x) \neq P(x)] \leq \epsilon.$$

Hence we can fix a particular polynomial p s.t. the probability over x is $\leq \epsilon$.

We now prove the first part of the theorem. We apply Lemma 9.1 with error $\leq \epsilon/s$ to every gate. Literals $\neg x_i$ are written as the polynomial $1 + x_i$ (recall we are working over $\mathbb{F}_2$). By a union bound, the probability that any gate makes a mistake is ϵ, as desired.

The final polynomial is obtained by composing the polynomials of each gate. The composition of a polynomial of degree d_1 with another of degree d_2 results in a polynomial of degree $d_1 \cdot d_2$. Since each polynomial has degree $\leq \log(s/\epsilon) + 1 \leq \log(cs/\epsilon)$, the final degree is $\log^d(cs/\epsilon)$.

To improve this to the stated bound, reason as follows. Apply Lemma 9.1 to every gate *except the output gate* with error c/s. Then apply Lemma 9.1 to the output gate with error c. The composition is then a polynomial of degree $(c \log s)^{d-1}$ which has error $\leq 1/3$. To shrink the error to ϵ, take $c \log 1/\epsilon$ independent samples of the polynomial and compute their majority exactly by a polynomial of degree $c \log 1/\epsilon$. **QED**

Exercise 9.2. Explain why we can handle parity gates as well (as claimed in Theorem 9.2).

9.1.2 Using the approximation to show that Majority is hard

The other step in the proof of the lower bound for majority (Theorem 9.1) is to show that Majority cannot be approximated by such low-degree polynomials. For this the key result is the following:

Lemma 9.2. Every function $f : [2]^n \to [2]$ can be written as $f(x) = p_0(x) + p_1(x) \cdot \mathrm{Maj}(x)$, for some polynomials p_0 and p_1 of degree $\leq n/2$. This holds for every odd n.

Proof. Let M_0 be the set of strings with weight $\leq n/2$, and M_1 the set of strings with weight $\geq n/2$. Assume that for any $b \in [2]$ and every function $f : M_b \to [2]$ there is a polynomial p_b of degree $\leq n/2$ s.t. p_b and f agree on M_b. From this the lemma follows, because we can write

$$f = p_1 \mathrm{Maj} + p_0(1 + \mathrm{Maj}) = p_0 + (p_0 + p_1)\mathrm{Maj}.$$

To prove the assumption, we claim that (the vectors corresponding to) their truth tables over M_0 are linearly independent. This means that any polynomial gives a different function over M_0, and because the number of polynomials is the same as the number of functions, the assumption follows. **QED**

Exercise 9.3. Prove the claim in the proof. Hint: More generally, prove that a non-zero polynomial of degree $\leq k$ is non-zero on a string of weight $\leq k$.

Proof of Theorem 9.1. Apply Theorem 9.2 with $\epsilon = 1/10$ to obtain p. Let S be the set of inputs on which $p(x) = C(x)$. By Lemma 9.2, any function $f : S \to [2]$ ca be written as

$$f(x) = p_0(x) + p_1(x) \cdot p(x).$$

The right-hand side is a polynomial of degree $\leq d' := n/2 + \log^{d-1}(cs)$. The number of such polynomials is the number of possible choices for each monomial of degree i, for any i up to the degree. This number is

$$\prod_{i=0}^{d'} 2^{\binom{n}{i}} = 2^{\sum_i^{d'} \binom{n}{i}}.$$

On the other hand, the number of possible functions $f : S \to [2]$ is

$$2^{|S|}.$$

Since a polynomial computes at most one function, taking logs we have

$$|S| \leq \sum_{i}^{d'} \binom{n}{i}.$$

The right-hand side is at most $2^n(1/2 + c\log^{d-1}(s)/\sqrt{n})$, since each binomial coefficient is $\leq c2^n/\sqrt{n}$, see Fact B.5.

On the other hand, $|S| \geq 0.9 \cdot 2^n$.

Combining this we get

$$0.9 \cdot 2^n \leq 2^n(1/2 + c\log^{d-1}(s)/\sqrt{n}).$$

This implies

$$0.4 \leq c\log^{d-1}(s)/\sqrt{n},$$

proving the theorem. **QED**

AC^0-$\oplus$ is a particularly interesting class because we can prove exponential impossibility results, but not exponential correlation bounds. The proof technique above gives a correlation bound no better than $1/\sqrt{n}$, for any function. Proving that some function in P has correlation $< 1/\sqrt{n}$ with AC^0-$\oplus$ is an open problem. This open problem is notable because it stands in the way of a number of other long-standing problems, see the notes. One candidate is the Xor of two majorities on disjoint inputs (cf Lemma 11.4).

Question 9.1. *Does AC^0-$\oplus$ have correlation $< 1/\sqrt{n}$ with the Xor of two majorities on $n/2$ bits each?*

9.2 Switching lemmas

In this section we sharpen the restrict-and-simplify method for alternating circuits, slightly improving the parameters of the bounds established via the polynomial method, Theorem 9.1. For depth d the polynomial method gave a size bound of $s \geq 2^{cn^{0.5/(d-1)}}$. For alternating circuits we can remove the 0.5. In particular, for depth 3 we obtain $s \geq 2^{c\sqrt{n}}$.

Definition 9.4. A *restriction* ρ is an assignment of the variables to $\{0, 1, \star\}$, i.e., some variables are replaced with constants, while those assigned to *star* $\star$ are left "alive." In a *random restriction*, the stars are selected at random according to some distribution, and also each unrestricted variable is set to a uniform bit. For a restriction ρ with s stars and $f : [2]^n \to [2]$ we denote by $f_\rho : [2]^s \to [2]$ the restricted function.

Two natural ways to select the locations for the stars are to select them uniformly among subsets of a fixed size, or to assign each bit to $\star$ independently with the same probability.

A *switching lemma* shows that applying a random restriction to a CNF (or DNF) greatly simplifies it. The simplification is so extreme that one can write a restricted CNF as a DNF and viceversa, whence the name "switching." This switch allows one to write a restricted circuit of depth d as a circuit of depth $d - 1$.

In fact, the restricted CNF (or DNF) can be written as a small-depth decision tree.

Definition 9.5. A function $f : [2]^n \to [2]$ is *w-local* if it depends on $\leq w$ input bits. A *decision tree* is a branching program where each node has in-degree 1. The tree has depth d if every path has $\leq d$ edges.

For example, a decision tree of depth 1 queries one bit.

Exercise 9.4. Let f be computable by a decision tree of depth d. Prove that f is also:

- $2^d - 1$ local,
- A d-DNF,
- A d-CNF.

We begin with a switching lemma which is useful in building intuition because its proof makes it evident why switching is at all possible. In terms of parameters, the lemma is sufficient to prove superpower lower bounds, for example establishing that Parity is not in AC^0, but not exponential.

Lemma 9.3. [Switching, I] Let f be an a-DNF on n bits and ρ a random restriction setting each bit to $\star$ independently w.p. $1/n^\epsilon$. Then f_ρ is a decision tree of depth $c_{a,\epsilon}$ except with prob. $\leq 1/n^{1/\epsilon}$.

Note the larger the a and the smaller the ϵ the stronger lemma (except for the depth of the tree). To apply a switching lemma to a circuit we view gates at depth 1 as 1-DNFs or 1-CNFs; we illustrate this below.

Proof. We prove it by induction on a for all ϵ.

For base case we can take $a = 0$ which is defined as a constant function.

For the inductive step, suppose f contains k disjoint terms. The prob. p that f_ρ is not constant is $\leq$ the prob. that all the terms in f_ρ are not 1 (as soon as a term is 1 the function is 1). A term is 1 if all the literals are 1. A literal is 1 if it is not mapped to $\star$ and then it is fixed to 1, this has probability $= 0.5(1 - 1/n^\epsilon) \geq 1/3$. Hence a term is 1 w.p. $\geq (1/3)^a$ and

$$p \leq (1 - 1/3^a)^k \leq e^{-k/3^a}.$$

For $k = 3^a \log n^{1/\epsilon}$ we have $p \leq 1/n^{1/\epsilon}$ as desired.

Otherwise, we claim that there are ak variables I that intersect *every* term of f. This is a basic and extremely useful combinatorial fact. To prove it, simply collect disjoint terms greedily. We collect $\leq k - 1$, since f does not contain k disjoint terms. No other term is disjoint, which means it contains ≥ 1 of the variables in the terms just collected. Since each term has $\leq a$ variables, we have proved the claim.

Now view ρ as first applying ρ_1 which assigns $\star$ with prob. $1/n^{\epsilon/2}$, and then again applying ρ_2 with the same $\star$ prob. to the remaining variables. Over the choice of ρ_1, the prob. of having $\geq 4/\epsilon^2$ stars in I is by a union bound and Fact B.6

$$\leq \binom{ak}{4/\epsilon^2} \left(\frac{1}{n^{\epsilon/2}}\right)^{4/\epsilon^2} \leq \left(\frac{ca3^a\epsilon^{-1}\log n}{4/\epsilon^2}\right)^{4/\epsilon^2} \frac{1}{n^{2/\epsilon}} \leq \frac{0.5}{n^{1/\epsilon}}$$

for $n \geq c_{a,\epsilon}$.

When there are $\leq 4/\epsilon^2$ stars in I, a decision tree for f_ρ can begin by reading all these variables. After reading them, we get an $(a-1)$-DNF, because the variables intersected every term. The number of possible DNFs thus obtained is $\leq 2^{4/\epsilon^2}$. By a union bound, and the induction hypothesis, the prob. over ρ_2 that one of these $(a-1)$-DNFs is not a decision tree of depth $\leq c_{a,\epsilon}$ is

$$\leq 2^{4/\epsilon^2} \cdot \frac{1}{n^{2/\epsilon}} \leq \frac{0.5}{n^{1/\epsilon}}.$$

Overall, the error probability is $\leq 2 \cdot 0.5/n^{1/\epsilon} \leq 1/n^{1/\epsilon}$ as desired. **QED**

Next we state a stronger switching lemma. The proof does not really benefit from the fact that the terms are simple, so we state it for arbitrary functions.

Lemma 9.4. [Switching, II] Let $C : [2]^n \to [2]$ equal the Or of functions $f_i : [2]^n \to [2]$ where each f_i is w-local. Let ρ be a random restriction with s stars. The probability that C_ρ is not a decision tree of depth d is $\leq (cws/n)^d$.

The proof of Lemma 9.4 is below in section 9.2.1.

Applying switching lemmas several times allows us to collapse an alternating circuit to a low-depth decision tree. The collapse applies even to circuits of exponential size. We state and prove this consequence trading simplicity of exposition for parameter optimization.

Corollary 9.1. Let $C : [2]^n \to [2]$ be an alternating circuit of depth d size $s \leq 2^{n^{c_d}}$. Let ρ be a random restriction with $2 \log s$ stars. The probability that C_ρ is not a decision tree of depth $\log s$ is $\leq 1/s$.

In particular, there is a restriction ρ with $\log s$ stars s.t. C_ρ is constant.

Proof. Set $w := \log s$. We view ρ as successive applications of restrictions whose number of stars is square root of the number of variables. View the circuit as having depth $d+1$ and the input gates have fan-in $1 \leq w$. The first application of Lemma 9.4 gives error $\leq (cw/\sqrt{n})^w$. In the good case, the input gates now are decision trees of depth $\leq w$. We can write this as a CNF or DNF with terms of size $\leq w$, and merge the output gate with the gates in the next level in the circuit, which are now computing Ors (or Ands) of functions on $\leq w$ bits. The next application of Lemma 9.4 gives error $\leq (cw/n^{1/4})^w$, and so on. In general, application i of the lemma gives an error of

$$(cw/n^{1/2^i})^w \leq c^{-w} = 1/s^2.$$

Taking a union bound over all gates in the circuit, the error probability is as desired.

The number of stars in the final restriction is $\geq 2 \log s$ by our assumption on s. **QED**

Exercise 9.5. Prove the "in particular."

One can use the switching lemma to prove exponential lower bounds to compute explicit functions by small-depth alternating circuits. The simplest example is parity, given next. In this case, we also prove an exponentially strong correlation bound (Definition 8.3) . The polynomial method gives weaker correlation bounds. This is a qualitative difference explored more in Chapter 11.

Corollary 9.2. The correlation between parity on n bits and an alternating circuit of depth d and size $s \leq 2^{n^{c_d}}$ is $\leq 1/s$.

Proof. The correlation between parity on m bits and decision trees of depth $< m$ is zero. View a uniform input as first picking a restriction, and then filling the stars. By Corollary 9.1, after picking the restriction the circuit is a decision tree of depth $\log s$, which is strictly less than the number of remaining stars, except with probability $1/s$. **QED**

By contrast, even a single bit has much larger correlation with Majority, as follows for example by Theorem 8.4.

By optimizing the proof of Corollary 9.1 one can prove that depth-d alternating circuits for parity have size $\geq 2^{n^{c/(d-1)}}$ which is tight up to the constants. This bound holds even if the circuits have any correlation $\geq 1/2^{n^{c/(d-1)}}$. Using yet another switching lemma, one can prove a better tradeoff between size and correlation, see the notes.

We illustrate the parameter optimization in the simple and significant case of lower bounds for depth 3. We leave the extension to arbitrary depth and correlation bounds as an exercise.

Corollary 9.3. Parity requires depth-3 alternating circuits of size $\geq 2^{c\sqrt{n}}$.

Proof. Pick a random restriction ρ which is the composition of two restrictions, the first picking cn stars, and the second picking $c\sqrt{n}$. For concreteness and w.l.o.g. let C be an And of DNFs. View the depth-1 gates as 1-CNFs. By Lemma 9.4, the first restriction collapses a depth-1 gate to a $c\sqrt{n}$-depth decision tree except with prob. $\leq 2^{-c\sqrt{n}}$. This is small enough that we can do a union bound over all gates at depth 1. When all the gates simplify to $c\sqrt{n}$-depth decision trees, we can rewrite the restricted circuit as a And of $c\sqrt{n}$-DNFs. Now, over the second restriction, the probability that one such DNF does not become a $c\sqrt{n}$-depth decision tree is again $2^{-c\sqrt{n}}$, and we can again do a union bound over all the DNFs, of which there are still $\leq 2^{c\sqrt{n}}$. We can write each such decision tree as a $c\sqrt{n}$-CNF. By merging the And gates we obtain that C_ρ is a t-CNF, where $t \leq c\sqrt{n}$. At the same time, by the choice of parameters we have $> t$ stars. To conclude, we argue that such a CNF cannot compute parity. To prove this, note that it suffices to fix a clause of the CNF to 0 to fix the entire CNF to 0. Since this is a t-CNF, we only need to fix $\leq t$ variables. However, there is still at least one star, so parity is not fixed. **QED**

9.2.1 Proof of switching Lemma 9.4

We illustrate the proof in stages.

The simplest case: Or of n bits. Here f is simply the Or of n bits $x_1, x_2, \ldots, x_n$. In the restriction some of the bits may become 0, others 1, and others yet may remain unfixed, i.e., assigned to stars. Those that become 0 you can ignore, while if some become 1 then the whole circuit C becomes 1.

We will show that the number of *bad* restrictions, those for which the restricted circuit $C|_\rho$ requires decision trees of depth $\geq d$, is small.

For this simple case, a straightforward proof of a stronger bound exists.

Exercise 9.6. Give it.

We give an alternative argument which we can then extend to the general case. We are going to encode such restrictions *using another restriction*. In this simple case, the restriction will have no stars; that is, it is just a 0/1 assignment to the variables). The gain is clear: just think of a restriction with zero stars versus a restriction with one star. Let us quantify this gain.

Definition 9.6. Denote by N_s the number of restrictions with exactly s stars.

Exercise 9.7. $N_s = \binom{n}{s} 2^{n-s}$.

Going back to the gain, we have $N_0 = 2^n$, while $N_1 = 2^{n-1} \cdot n$, so $N_0/N_1 \leq c/n$. Note that this an upper bound on the error probability.

We repeat that *we only want to encode the bad restrictions for which $C|_\rho$ requires large depth*. So ρ does not map any variable to 1, for else the Or is 1 which has decision trees of depth 0. The way we are going to encode ρ is this: *Simply replace the stars with ones*. To go back, replace the ones with stars. We are using the ones in the encoding to "signal" where the stars are. Hence, the number of bad restrictions is at most $N_0 = 2^n$, which is tiny compared to the number $N_s = \binom{n}{s} 2^{n-s}$ of restrictions with s stars (see Exercise 9.7). The error probability is then (using Fact B.6)

$$\frac{2^n}{\binom{n}{s} 2^{n-s}} = \frac{2^s}{\binom{n}{s}} \leq \left(\frac{2s}{n}\right)^s.$$

This is stronger than Lemma 9.4. (We can assume $d \leq s$, since every function on s bits has decision trees of depth s, and so for $d \geq s$ the error probability is 0.)

The medium case: Or of functions on disjoint inputs. So, again, let's take a random restriction ρ with exactly s stars. Some of the functions may become 0, others 1, and others yet may remain unfixed. Those that become 0 you can ignore, while if some become 1 then the whole circuit becomes 1.

As before, we will show that the number of restrictions for which the restricted circuit $C|_\rho$ requires decision trees of depth $\geq d$ is small. To accomplish this, we are going to encode/map such restrictions using/to a restriction with just $s - d$ stars, plus a little more information. As we saw already, the gain in reducing the number of stars is clear. In particular, saving d stars reduces the number of restrictions by a factor $(cs/n)^d$:

$$\frac{N_{s-d}}{N_s} = \frac{\binom{n}{s-d}2^{n-(s-d)}}{\binom{n}{s}2^{n-s}} = \frac{\frac{n!}{(s-d)!(n-s+d)!}}{\frac{n!}{s!(n-s)!}} \cdot 2^d = \frac{s(s-1)\cdots(s-d+1)}{(n-s+1)(n-s+2)\cdots(n-s+d)} \cdot 2^d$$
$$\leq \frac{s^d}{(n-s+1)^d} \cdot 2^d \leq \left(\frac{2s}{n-s}\right)^d \leq \left(\frac{cs}{n}\right)^d.$$

Here we first use Exercise 9.7. The first inequality amounts to picking the larger term in the numerator and the smallest in the denominator. The last inequality holds because we can assume $s \leq cn$, for else the probability bound in Lemma 9.4 is larger than one.

The auxiliary information will give us a factor of w^d, leading to the claimed bound. Specifically, as before, recall that we only want to encode restrictions for which $C|_\rho$ requires large depth. So no function in $C|_\rho$ is 1, for else the circuit is 1 and has decision trees of depth 0. Also, you have d stars among inputs to functions that are unfixed (i.e., not even fixed to 0), for else again you can compute the function reading less than d bits. Because the functions are unfixed, there is a setting for those d stars (and possibly a few more stars – that would only help the argument) that make the corresponding functions 1. We are going to pick precisely that setting in our restriction ρ' with $s-d$ stars. This allows us to "signal" which functions had inputs with the stars we are saving (namely, those that are the constant 1). To completely recover ρ, we add extra information to indicate where the stars were. The saving here is that we only have to say where the stars are among w symbols, not n.

Specifically, we can encode the positions of the stars with an element $a \in [cw]^d$, indicating which of the w symbols is a star, and also whether the star is the last in this function. To recover the restriction, we look for the first function that's set to 1. We then read a to find out how may stars were there and their positions. Then we move to the second function that's fixed to 1. Again we look at a to know how many stars were there and what their positions were, and so on.

The general case. The idea is the same, except we have to be slightly more careful because when we set values for the stars in one function we may also affect other functions. The idea is to fix one function at the time. Specifically, starting with ρ, consider the first function f that's not made constant by ρ. So the inputs to f have some stars. As before, let us replace the stars with constants that make the function f equal to the constant 1, and append the extra information that allows us to recover where these stars were in ρ.

We'd like to repeat the argument. Note however we only have guarantees about $C|_\rho$, not $C|_\rho$ with some stars replaced with constants that make f equal to 1. We also can't just jump to the 2nd function that's not constant in $C|_\rho$, since the "signal" fixing for that might clash with the fixing for the first – this is where the overlap in inputs makes things slightly more involved. Instead, because $C|_\rho$ required decision tree depth at least d, we note there have to be some assignments to the m stars in the input to f so that the resulting, further restricted circuit still requires decision tree depth $\geq d-m$ (else $C|_\rho$ has decision trees of depth $< d$). We append this assignment to the auxiliary information and we continue the argument using the further restricted circuit.

9.3 $\mathbf{AC^0}$ vs L, $\mathbf{NC^1}$, $\mathbf{TC^0}$

In this section we prove that constant-depth alternating circuits can simulate with sub-exponential size several classes of interest.

9.3.1 L

We first utilize the checkpoint technique (section §6.3) to simulate L.

Theorem 9.3. Let $f \in \mathrm{BrL}$ and $d \in \mathbb{N}$. Then f_n has alternating circuits of depth d and size $2^{n^{cf/d}}$.

Another way of saying this is that the function is in AC^0 on inputs of power-log length (padded to length n). This is quite useful.

Exercise 9.8. Prove that for any d the Majority function on $\log^d n$ bits (padded to n bits) is in AC^0.

Proof. It suffices to prove the result for boolean f. Consider a branching program of size $S = n^a$ for f. We apply the checkpoint technique to this branching program recursively, with parameter $b := n^{ca/d}$. Each application of the technique reduces the path length by a factor b. Hence with cd applications we can reduce the path length to 1. The corresponding gate is connected to the input bit which the branching program reads.

The resulting depth of the circuit is d. Next we bound the size. In one application, we have an $\exists$ quantifier over $b-1$ nodes, corresponding to an Or gate with fan-in S^{b-1}, and then a $\forall$ quantifier over b smaller paths, corresponding to an And gate with fan-in b. This gives a tree with $S^{b-1} \cdot b \leq S^b = n^{ab}$ leaves. Iterating, the number of leaves will be n^{abd}, and the total size of the circuit $\leq cn^{abd}$. This is $\leq 2^{n^{ca/d}}$ for large enough n. **QED**

We can refine the connection between branching programs and small-depth circuits in Theorem 9.3 to take into account width.

Theorem 9.4. Let $f : [2]^n \to [2]$ be computable by a branching program with width W and time t. Then f is computable by an alternating circuit of depth-3 of size $\leq 2^{c\sqrt{t \log W}}$.

As we have seen in Corollary 9.3, parity requires depth-3 circuits of size $2^{c\sqrt{n}}$. Theorem 9.4 shows that improving this would also improve the lower bounds for small-width branching programs (cf section §6.8).

Exercise 9.9. Prove Theorem 9.4.

A more general version of Theorem 9.4. states that for any parameter b one can have a depth-3 circuit with size

$$2^{b \log W + t/b + \log t},$$

output fan-in W^b, and input fan-in t/b. Interestingly, again this trade-off essentially matches known impossibility results for depth-3 circuits! This can be shown with the same reasoning as in the proof of Corollary 9.3.

9.3.2 Linear-size log-depth

There is a non-trivial simulation of linear-size log-depth circuits by alternating circuits of depth 3.

Theorem 9.5. Any circuit $C : [2]^n \to [2]$ of size an and depth $a \log n$ has an equivalent alternating circuit of depth 3 and size $2^{c_a n/\log\log n}$.

The idea of the simulation is to identify a set of $o(n)$ wires to remove from C so that the resulting circuit becomes very disconnected: each connected component has depth $\leq 0.1 \log n$. Since the circuit has fan-in 2, the output of each component can depend on at most $n^{0.1}$ input bits, and so, given the assignment to the removed edges, the output can be computed in brute-force by a depth-2 circuit of sub-exponential size. Trying all $2^{o(n)}$ assignments to the removed edges and collapsing some gates completes the simulation. We now proceed with a formal proof.

The graph corresponding to C in Theorem 9.5 is connected, but we will also work with disconnected graphs.

For the depth reduction in the proof, it is convenient to think of depth as a function from nodes to integers. The next definition and simple claim formalize this.

Definition 9.7. Let $G = (V, E)$ be a directed acyclic graph. The *depth* of a node in G is the number of nodes in a longest directed path terminating at that node. The depth of G is the depth of a deepest node in G. A *depth function D for G* is a map $D : V \to \{1, 2, \ldots, 2^k\}$ such that if $(a, b) \in E$ then $D(a) < D(b)$.

Exercise 9.10. Prove that a directed acyclic graph $G = (V, E)$ has depth at most 2^k if and only if there is a depth function $D : V \to \{1, 2, \ldots, 2^k\}$ for G.

The following is the key lemma which allows us to reduce the depth of a graph by removing few edges.

Lemma 9.5. Let $G = (V, E)$ be a directed acyclic graph with w edges and depth 2^k. It is possible to remove $\leq w/k$ edges so that the depth of the resulting graph is $\leq 2^{k-1}$.

Proof. Let $D : V \to \{1, 2, \ldots, 2^k\}$ be a depth function for G. Consider the set of edges E_i for $1 \leq i \leq k$:

$$E_i := \{(a, b) \in E | \text{ the most significant bit position where } D(a) \text{ and } D(b) \text{ differ is the } i\text{-th}\}.$$

Note that $E_1, E_2, \ldots, E_k$ is a partition of E. And since $|E| = w$, there exists an index $i, 1 \leq i \leq k$, such that $|E_i| \leq w/k$. Fix this i and remove E_i. We need to show that the depth of the resulting graph is at most 2^{k-1}. To do so we exhibit a depth function $D' : V \to [2]^{k-1}$. Specifically, let D' be D without the i-th output bit. We claim that D' is a valid depth function for the graph $G' := (V, E \setminus E_i)$. For this, we need to show that if $(a, b) \in E \setminus E_i$ then $D'(a) < D'(b)$. Indeed, let $(a, b) \in E \setminus E_i$. Since $(a, b) \in E$, we have

$D(a) < D(b)$. Now, consider the most significant bit position j where $D(a)$ and $D(b)$ differ. There are three cases to consider:

j is more significant than i: In this case, since the j-th bit is retained, the relationship is also maintained, i.e., $D'(a) < D'(b)$;

$j = i$: This case cannot occur because it would mean that the edge $(a, b) \in E_i$;

j is less significant than i: In this case, the i-th bit of $D(a)$ and $D(b)$ is the same and so removing it maintains the relationship, i.e., $D'(a) < D'(b)$. **QED**

Now we prove the main theorem.

Proof of Theorem 9.5. For simplicity, we assume that both a and $\log n$ are powers of two. Let $2^\ell := a \cdot \log n$.

Applying the above lemma we can reduce the depth by a factor 1/2, i.e. from 2^ℓ to $2^{\ell-1}$, by removing $\leq a \cdot n/\ell$ edges. Applying the lemma again we reduce the depth to $2^{\ell-2}$ by removing $\leq a \cdot n/(\ell - 1)$ edges. If we repeatedly apply the lemma $\log(2a)$ times the depth reduces to

$$\frac{a \log n}{2^{\log(2a)}} = \frac{\log n}{2},$$

and the total number of edges removed is at most

$$a \cdot n \left(\frac{1}{\ell} + \frac{1}{\ell - 1} + \ldots + \frac{1}{\ell - \log(2a) + 1} \right) \leq a \cdot n \cdot \frac{\log(2a)}{\ell - \log(2a) + 1} = a \cdot n \cdot \frac{\log(2a)}{\log \log n}.$$

For slight convenience we also think that the circuit has an output edge e_{output}, and remove it; this way we can represent the output of the circuit in terms of the value of e_{output}. We remove at most

$$r := c_a \cdot n / \log \log n$$

edges.

We define the depth of an edge $e = g \to g'$ as the depth of g, and the value of e on an input x as the value of the gate g on x.

For every input $x \in [2]^n$ there exists a unique assignment h to the removed edges that corresponds to the computation of $C(x)$. Given an arbitrary assignment h and an input x we can check if h is the correct assignment by verifying if the value of every removed edge $e = g \to g'$ is correctly computed from (1) the values of the removed edges whose depth is less than that of e, and (2) the values of the input bits g is connected to. Since the depth of the component is $\leq (\log n)/2$ and the circuit has fan-in 2, at most $\sqrt{n}$ input bits are connected to g; we denote them by $x|_e$. Thus, for a fixed assignment h to the removed edges, the check for e can be implemented by a function $f_h^e : [2]^{\sqrt{n}} \to [2]$ (when fed the $\leq \sqrt{n}$ values of the input bits connected to g, i.e. $x|_e$).

Induction on depth shows:

$$C(x) = 1 \Leftrightarrow \exists \text{ assignment } h \text{ to removed edges such that } h(e_{\text{output}}) = 1$$
$$\text{and } \forall \text{ removed edge } e \text{ we have } f_h^e(x|_e) = 1.$$

We now claim that the above expression for the computation $C(x)$ can be implemented with the desired resources. Since we removed $r = c_a \cdot n/\log\log n$ edges, the existential quantification over all assignments to these edges can be implemented with an $\vee$ (OR) gate with fan-in 2^r. Each function $f_h^e(x|_e)$ can be implemented via brute-force by a CNF, i.e. a depth-2 $\wedge\vee$ circuit, of size $\sqrt{n} \cdot 2^{\sqrt{n}}$. For any fixed assignment h, we can combine the output $\wedge$ gates of these CNF to implement the check

$$\forall \text{ removed edge } e : f_h^e(x|_e) = 1$$

by a CNF of size at most

$$r \cdot \sqrt{n} \cdot 2^{\sqrt{n}}.$$

Finally, accounting for the existential quantification over the values of the r removed edges, we get a circuit of depth 3 and size

$$2^r \cdot r \cdot \sqrt{n} \cdot 2^{\sqrt{n}} = 2^{c_a n/\log\log n}.$$

QED

9.3.3 TC^0

The same simulation in Theorem 9.3 applies to functions in NC^1 and TC^0, just because BrL contains these classes. In the other direction, we can use Theorem 9.2 to show that AC^0 can be simulated by threshold circuits of depth 3, albeit not quite of power size.

Theorem 9.6. Let $f \in \mathrm{AC}^0\text{-}\oplus$. Then f_n has threshold circuits of depth 3 and size $2^{\log^{c_f} n}$.

Exercise 9.11. Prove Theorem 9.6 but for depth 4 instead of 3.

9.4 The power of AC^0: Gap majority

We proved in Theorem 9.1 that Majority is not in AC^0. In fact it requires exponential size, even if parity gates are allowed. Yet, AC^0 can approximate majority in a certain sense. This approximation is very useful and is the basis for Theorem 5.8 that $\mathrm{BPP} \subseteq \mathrm{PH}$.

Definition 9.8. $\text{Gap-Maj}_{\alpha,\beta}$ is the problem of deciding if an input $x \in [2]^n$ has weight $|x| \leq \alpha n$ or $|x| \geq \beta n$.

We have the following somewhat surprising result:

Lemma 9.6. $\text{Gap-Maj}_{1/3,2/3} \in \mathrm{AC}^0$.

Proof. This is a striking application of the probabilistic method. For a fixed pair of inputs (x, y) we say that a distribution C on circuits *gives* $(\leq p, \geq q)$ if $\mathbb{P}_C[C(x) = 1] \leq p$ and

$\mathbb{P}_C[C(y) = 1] \geq q$; and we similarly define *gives* with reverse inequalities. Our goal is to have a distribution that gives

$$(\leq 2^{-n}, \geq 1 - 2^{-n}) \tag{9.1}$$

for every pair $(x, y) \in [2]^n \times [2]^n$ where $|x| \leq n/3$ and $|y| \geq 2n/3$. Indeed, if we have that we can apply a union bound over the $< 2^n$ inputs to obtain a fixed circuit that solves Gap-Maj.

We construct the distribution C incrementally. Fix any pair (x, y) as above. Begin with the distribution $C_\wedge$ obtained by picking $2 \log n$ bits uniformly from the input and computing their And. This gives

$$\left((1/3)^{2 \log n}, (2/3)^{2 \log n}\right).$$

Let $p := (1/3)^{2 \log n}$ and note $(2/3)^{2 \log n} = p \cdot n^2$. So we can say that $C_\wedge$ gives

$$\left(\leq p, \geq p \cdot n^2\right).$$

Now consider the distribution $C_\vee$ obtained by complementing the circuits in $C_\wedge$. This gives

$$\left(\geq 1 - p, \leq 1 - p \cdot n^2\right).$$

Next consider the distribution $C_{\wedge\vee}$ obtained by taking the And of $m := p^{-1}/n$ independent samples of $C_\vee$. This gives

$$\left(\geq (1-p)^m, \leq (1 - p \cdot n^2)^m\right).$$

Approximations for the exponential function, Fact B.7, yield $(1-p)^m \geq e^{-2pm} = e^{-2/n} \geq 0.9$ and $(1 - p \cdot n^2)^m \leq e^{-n}$:

$$(\geq 0.9, \leq e^{-n}).$$

Next consider the distribution $C_{\vee\wedge}$ obtained by complementing the circuits in $C_{\wedge\vee}$. This gives

$$(\leq 0.1, \geq 1 - e^{-n}).$$

Finally, consider the distribution $C_{\wedge\vee\wedge}$ obtained by taking the And of n independent samples of $C_{\vee\wedge}$. This gives

$$\left(\leq 0.1^n, \geq \left(1 - e^{-n}\right)^n\right).$$

For the rightmost quantity we can use Fact B.8; this gives

$$\left(\leq 0.1^n, \geq 1 - ne^{-n}\right).$$

We have $ne^{-n} < 2^{-n}$. Thus this distribution in particular gives equation (9.1). The bounds on the number of gates and the fan-in holds by inspection. **QED**

By inspection, this circuit has depth 3 and the fan-in of the gates at level 1 is $c \log n$. We shall use these facts next.

9.4.1 Back to the PH

We now return to the result that BPP $\subseteq$ PH, Theorem 5.8. We provide a proof of the first part of Theorem 5.8 (which recall suffices for BPP $\subseteq$ PH). A good way to think of these simulations is as follows. Fix a BPTime(t) machine M and an input $x \in [2]^n$, and write y for its random bits. In time t the machine uses $t' \leq ct \log t$ random bit, so $|y| \leq t'$. The simulating alternating machine is trying to decide if for most choices of the random bits y we have $M(x, y) = 1$, or if for most choices we have $M(x, y) = 0$. This is an instance of Gap-Maj on the exponentially long input

$$(M(x, 0), M(x, 1), M(x, 2), \ldots, M(x, 2^{t'} - 1)) \in [2^{t'}].$$

To prove Theorem 5.8 we "only" need the circuits for Gap-Maj in Lemma 9.6 to be sufficiently explicit, or uniform. Let us denote these circuits on T bits by GMC_T. A simple notion of uniformity is that GMC_T is constructible in FP. This does not work here because T is exponential in the input length $|x|$ of the BPTime computation. Instead, we need a refined notion of uniformity, arguably even more natural.

Definition 9.9. A family of alternating circuits $C_n : [2]^n \to [2]$ is *gate-uniform* if given n, an index to a gate $\bigwedge_{j \in [r]} t_j$ or $\bigvee_{j \in [r]} t_j$, and j, we can compute t_j (which is either an index to another gate or a literal) in Quasi-Linear-Time.

Lemma 9.7. Suppose GMC_T is gate uniform. Then Theorem 5.8 follows.

Proof. Let M be a machine corresponding to some function in BPTime(t) that uses $t' \leq ct \log t$ random bits. Consider GMC_T for $T := 2^{t'}$. As remarked earlier, this is a depth-3 circuit, and the gates at depth 1 have fan-in $c \log T = ct'$. Let us first prove (1) in Theorem 5.8. Use two quantifiers and the fact that the circuits are gate-uniform to index an And gate at depth 1. This takes quasi-linear time in t' and hence quasi-linear in t. Then for each $i \leq ct'$ compute input literal i of that gate, which corresponds to a choice for the random bits for the machine, and evaluate the machine on that choice. Each evaluation takes time ct, for a total of time ctt'. **QED**

Exercise 9.12. Show that (2) in Theorem 5.8 also follows.

There remains to construct explicit circuits for Gap-Maj. We give a construction which has worse parameters than Lemma 9.6 but is simple and suffices for (1) in Theorem 5.8. The idea is that if the weight of x is large (where recall weight is the number of bits set to 1 in x), then we can find a few *shifts* of the ones in x that cover each of the n bits. But if the weight of x is small we can't. By "shift" by s we mean the string $x_{i \oplus s}$, obtained from x by permuting the indices by xoring them with s. (Other permutations would work just as well.)

Lemma 9.8. Let $r := \log n$. The following circuit solves $\text{GapMaj}_{1/r^2, 1-1/r^2}$ on every $x \in [2]^n$:

$$\bigvee s_1, s_2, \ldots, s_r \in [2]^r : \bigwedge i \in [2]^r : \bigvee j \in \{1, 2, \ldots, r\} : x_{i \oplus s_j}.$$

Note that the subformula rooted at $\bigwedge$ means that every bit i in $[n] = [2]^r$ is covered by some shift s_j of the input x.

Proof. If weight$(x) \leq n/r^2$. Each shift s_i contributes at most n/r^2 ones. Hence all the r shifts contribute $\leq n/r$ ones, and we do not cover every bit i.

Now assume weight$(x) \geq n(1-1/r^2)$. We show the existence of shifts s_i that cover every bit by the probabilistic method. Specifically, for a fixed x we pick the shifts uniformly at random and aim to show that the probability that we do not cover every bit is < 1. Indeed:

$$\begin{aligned}
&\mathbb{P}_{s_1,s_2,\ldots,s_r}\left[\exists i \in [2]^r : \forall j \in \{1,2,\ldots,r\} : x_{i\oplus s_j} = 0\right] \\
\leq &\sum_{i\in[2]^r} \mathbb{P}_{s_1,s_2,\ldots,s_r}\left[\forall j \in \{1,2,\ldots,r\} : x_{i\oplus s_j} = 0\right] && \text{(union bound)} \\
= &\sum_{i\in[2]^r} \mathbb{P}_s[x_{i\oplus s} = 0]^r && \text{(independence of the } s_i\text{)} \\
\leq &\sum_{i\in[2]^r} (1/r^2)^r && \text{(by assumption on weight}(x)\text{)} \\
\leq &(2/r^2)^r \\
< &1.
\end{aligned}$$

QED

Exercise 9.13. Prove (1) in Theorem 5.8.

Lemma 9.8 is not sufficient for (2) in Theorem 5.8. One can prove (2) by *derandomizing* the shifts in Lemma 9.8. This means generating their r^2 bits using a seed of only $r \log^c r$ bits (instead of the trivial r^2 in Lemma 9.8.). This is done in section 11.1.4.

9.5 Mod 6

The polynomial method (section 9.1) is effective to prove impossibility results against AC^0-Mod-m if m is a prime power. These techniques are illustrated in section 9.1 in the fundamental case $m = 2$. They can be extended to any prime power m (see Problem 9.1), but they break down when m is composite. It is consistent with our knowledge that AC^0-Mod-6 equals CktP. Even the status of simple functions is unkonwn:

Question 9.2. *Is Majority in AC^0-Mod-6?*

Nevertheless AC^0-Mod-m can be simulated by polynomials even for composite m. The simulation is incomparable to the one for $m = 2$ that we saw in section 9.1 (see Theorem 9.2). In the new simulation we work with polynomials over a larger domain which we then map to a boolean value. Equivalently, we can think of this as a depth-2 circuit whose output gate is a symmetric function. On the other hand, this new simulation works for every input as opposed to most inputs.

Lemma 9.9. Let $f \in \text{ACC}^0$. Then for every n and $d := \log^{c_f} n$

$$f_n(x_1, x_2, \ldots, x_n) = b(p(x_1, x_2, \ldots, x_n))$$

for some polynomial p of degree d over $\mathbb{Z}_{2^d}$, and some function $b : [2^d] \to [2]$.

Note this Lemma 9.9 generalizes Theorem 9.6.

The proof uses *modulus-amplifying polynomials*, which allow us to treat a Mod m gate as an integer sum and move it towards the output of the circuit.

Lemma 9.10. [Modulus-amplifying polynomials] For every integer ℓ there is a univariate polynomial F_ℓ of degree $2\ell - 1$ over $\mathbb{Z}$ such that for every $y \in \mathbb{Z}$:

$$y \mod 2 = F_\ell(y) \mod 2^\ell.$$

Exercise 9.14. Prove Lemma 9.10 with the weaker degree bound ℓ^c, which suffices for all applications in this book. Guideline:

(1) Let $a(m) := m^2(3-2m)$. Prove that for both $b \in [2]$, if $m = b \mod 2^j$ then $a(m) = b \mod 2^{2j}$, for any j.

(2) Conclude the proof.

One can now prove Lemma 9.9. We present the proof of a representative special case.

Proof of special case of Lemma 9.9.. Consider a depth-3 $\mathrm{Sym}_{n^a} \oplus \wedge_{\log^a n}$ circuit C: The output is a symmetric function of n^a polynomials p_i over $\mathbb{F}_2$ of degree $\log^a n$. (The symmetric function could be Mod-3, but the argument is more general.) View each p_i as a polynomial over $\mathbb{Z}$ and consider a modulus amplifying polynomial F of degree $2\ell - 1$ such that $2^\ell > n^a$, for which it suffices $\ell \leq c_a \log n$.

The value of C is determined by

$$\sum_{i \in [n^a]} (p_i \mod 2) = \sum_{i \in [n^a]} \left(F(p_i) \mod 2^\ell\right) = \left(\sum_{i \in [n^a]} F(p_i)\right) \mod 2^\ell,$$

where the last equality holds because $2^\ell > n^a$. Each $F(p_i)$ is a polynomial of degree $\leq \log^{c_a} n$. Hence so is the sum, and the result follows. **QED**

9.5.1 The power of ACC^0

We give an example of the power of ACC^0 which complements Problem 7.2.

Theorem 9.7. Let G be a finite solvable group. Iterated product of elements in G is in ACC^0.

Proof. Let us first illustrate the main idea with a jargon-free example. Let G be the (dihedral) group whose elements are (t, b) where $t \in \mathbb{Z}_3$ and $b \in \mathbb{Z}_2$ and $(t, b)(t', b')$ equals

$$(t + t', b') \text{ if } b = 0,$$
$$(t - t', b' + 1) \text{ if } b = 1.$$

We have to show that given

$$(t_1, b_1), (t_2, b_2), \ldots, (t_m, b_m)$$

we can compute their product in ACC^0. To do this, first "take all the b to the right," i.e., compute a tuple

$$(t'_1, 0), (t'_2, 0), \ldots, (t'_m, 0), (0, b')$$

with the same product.

Here each t'_i depends on t_i and $\prod_{j<i} b_j$. This latter product can be computed with a Mod 2 gate.

Then we can compute the product of the t' with a Mod 3 gate. Finally, we can simulate both a Mod 2 and a Mod 3 gate using a Mod 6 gate and repeating bits. This concludes the proof for this case.

To prove the result for any solvable group we proceed by induction on the length of the series in the definition of solvable group, see section §B.6.2. The base case corresponds to the group $\{1\}$ and is therefore trivial. For the inductive step, we are given

$$g_1, g_2, \ldots, g_m \in G$$

and we want to compute their product. Write each g_i as a coset representative h_i times an element n_i of the normal subgroup N. So now we want to compute

$$h_1 n_1 h_2 n_2 \cdots h_m n_m.$$

Since the quotient is cyclic we can write $h_i = a^{\epsilon_i}$ (all these conversions can be done in brute force since the group is finite). Let

$$b_i := a^{\epsilon_1} a^{\epsilon_2} \cdots a^{\epsilon_i}$$

and note we want to compute

$$(b_1 n_1 b_1^{-1})(b_2 n_2 b_2^{-1}) \cdots (b_m n_m b_m^{-1}) b_m.$$

Each b_i can be computed by summing the exponent, which can be done with a Mod c_G gate.

Also, the product in each bracketed expression belongs to N because N is normal. By induction iterated product in N is in ACC^0, and so is iterated product in G. **QED**

9.6 Impossibility results for ACC^0

Essentially the best negative result we know for ACC^0 is the following.

Theorem 9.8. $\bigcup_k \mathrm{NTime}\left(n^{\log^k n}\right) \not\subseteq \mathrm{ACC}^0$.

In particular, NExp $\not\subseteq$ ACC^0. It remains open if NP $\subseteq$ AC^0-Mod-6.

The only proof we know of this Theorem 9.8 is a brilliant combination of the simulation by polynomials (Lemma 9.9), nondeterministic diagonalization (Problem 5.2), and quasilinear completeness (Theorem 5.4). In this section we sketch it

The high-level idea in the proof is that the satisfiability of alternating circuits with modular gates can be decided faster than brute-force, and if NExp were in ACC^0 we can use this to derive a contradiction with the non-deterministic time hierarchy (Problem 5.2). First, let us present the satisfiability algorithm.

Lemma 9.11. [Satisfiability for alternating circuits with modular gates] Given an alternating circuit C on m bits of size m^d, depth d, with Mod d gates, we can decide if there is $x \in [2]^m : C(x) = 1$ in time $2^{m-m^{c_d}}$.

Note this is noticeably faster than trying all 2^m assignments.

Proof sketch. Let $\delta := c_d$ and write an m-bit input x to C as $x = yz$ where $|y| = m^\delta$ and $|z| = m' := m - m^\delta$. Consider the circuit C' on m' input bits defined as

$$C'(z) := \bigvee_{y \in [2]^{m^\delta}} C(y, z).$$

It suffices to determine the satisfiability of C'. We shall show how to do that in time close to $2^{m'}$.

Apply the transformation in Lemma 9.9 and let p and b be the corresponding functions. The lemma was only stated for power-size circuits, but it applies to C' as well, since the $\bigvee$ gate has low-degree probabilistic polynomials. Moreover, (a representation of) the functions p and b can be computed in the desired time; for the special case of Lemma 9.9 we proved, this can be verified by inspection.

Next, compute the entire truth-table of p, i.e., the evaluations of p on every possible input. This is a classic transformation that can be performed by a divide-and-conquer approach in time quasi-linear in $2^{m'}$. Once we have the truth table we apply b to every output and determine the satisfiability. **QED**

To prove impossibility results for ACC^0, let f be a function in NTime(2^n) that is not computable in, say, NTime$(2^n/n)$. The existence of such f follows from the hierarchy for non-deterministic time, Problem 5.2. Consider the function h that on input $x \in [2]^n$ and $i \leq 2^n \cdot n^c$ computes the 3CNF from Theorem 5.4 on $2^n \cdot n^c$ variables, computes its first satisfying assignment if one exists, and outputs its bit i. We shall show that $h \notin$ ACC^0.

An important feature of Theorem 5.4 that we shall use is that the 3CNF is explicit: In FP we can construct an alternating circuit I (for indexing) of constant-depth that given an index to a clause outputs the corresponding literals.

Suppose towards a contradiction that $h \in$ ACC^0. By hardwiring, for every $x \in [2]^n$ there is a corresponding circuit C_x that on input i computes that bit i. We contradict the assumption on f by showing how to compute it in NTime$(2^n/n)$.

Consider the algorithm that on input $x \in [2]^n$ guesses the above circuit C_x. Then it constructs the following circuit D:

$$D(j) = C_x(i_0) \oplus b_0 \bigvee C_x(i_1) \oplus b_1 \bigvee C_x(i_2) \oplus b_2$$

where $I(j) = i_0 b_0 i_1 b_1 i_2 b_2$. In words, D takes as input an index j to a clause in ϕ. It computes the corresponding indexes i_0, i_1, i_2 of the variables and corresponding bits b_0, b_1, b_2 using I. Then it runs C_x on each of i_0, i_1, i_2, complements the output accordingly, and takes an Or.

Running the satisfiability algorithm in Lemma 9.11 on D determines if ϕ is satisfiable and hence if $f(x) = 1$.

9.7 The power of AC^0: sampling

Recall that even though NC^0 cannot compute parity, we showed in Exercise 7.16 that they can sample (or generate) input-output pairs of the parity function. It turns out that AC^0 has even greater sampling capabilities: We can sample $(X, f(X))$ for uniform X for any symmetric function f. This is more involved and beautiful, and is only known to be possible up to a small statistical distance (see Definition B.1 for a definition of this distance). Try for a few minutes to sample $(X, \text{majority}(X))$ efficiently using alternating circuits before reading on!

The first step is sampling a uniform permutation.

Lemma 9.12. There are alternating circuits $C : [2]^{n^c} \to [n]^n$ of depth c and size n^c whose output distribution is 2^{-n^c} close (in statistical distance) to a uniform permutation π over $[n]$.

Exercise 9.15. Assume Lemma 9.12. Prove that there are alternating circuits $C : [2]^{n^c} \to [2]^n$ of depth c and size n^c whose output distribution is 2^{-n^c} close to:
(1) a uniform string of weight i, for any $i \leq n$,
(2) $(X, \text{majority}(X))$ for uniform $X \in [2]^n$.

Proof of Lemma 9.12. The high-level idea is "dart throwing:" we view the input random bits as random pointers $p_0, p_1, \ldots, p_{n-1} \in [m]$ into $m \gg n$ words. We then write $i \in [n]$ in the p_i-th word (unhit words get "$*$"). If there are no collisions, the ordering of $[n]$ in the words gives a random permutation of $[n]$. However, it is not clear how to explicitly write out this permutation using small depth, because to determine the image of i one needs to count how many words before p_i are occupied, which cannot be done in AC^0.

The key insight is to view the words as representing the permutation in a different format, known as *cycle format*, from which we can easily write out the permutation. Just like the standard format, the cycle format represents a permutation via an array $A[0..n-1]$ whose entries contain all the elements $[n]$. However, rather than thinking of $A[i]$ as the image of i, we think of the entries of A as listing the cycles of the permutation in order. Each cycle is listed starting with its smallest element, and cycles are listed in decreasing order of the

first element in the cycle. This format allows for computing the permutation efficiently: the image of i is the element to the right of i in A, unless the latter element is the beginning of a new cycle, in which case the image of i is the first element in the cycle containing i. Identifying the first element of a cycle is easy, because it is smaller than any element preceding it in A. This format works even if the array A has $m \gg n$ words, of which $m - n$ are unhit and marked by "$*$."

One can now verify that computing the image of i is in AC^0. Here in particular we use the fact that in AC^0 we can, given an array A and an index i, compute the least $j > i$ such that $A[j]$ is not "$*$". This can be accomplished by trying all j, noting that one can determine if a fixed j is the least $j > i$ such that $A[j]$ is not "$*$" using one unbounded fan-in And.

This gives a uniform permutation, unless there is a collision in the pointers, which happens with probability $\leq 1/n^c$ for $m \geq n^c$. Because we can detect if there is a collision in AC^0, the error probability can be shrunk to exponentially small as in section §3.1. Specifically, pick ℓ uniform and independent sets of pointers $p_0^i, \ldots, p_{n-1}^i$, $i \in [\ell]$, where each pointer has range $[m]$ for m the smallest power of 2 larger than $2n^2$ (thus each pointer can be specified by $\log m$ bits). If there exists i such that the pointers $p_0^i, \ldots, p_{n-1}^i$ are all distinct (i.e., there are no collisions), then run the above algorithm on the output corresponding to the first such i. This results in a random permutation.

Since the pointers are chosen independently, the probability that there is no such i is

$$\mathbb{P}\left[\forall i \in [\ell], \exists j, k \in [n] : p_j^i = p_k^i\right] = \mathbb{P}\left[\exists j, k \in [n] : p_j^0 = p_k^0\right]^{\ell} \leq (1/2)^{\ell}.$$

Choosing say $\ell := n$ proves the lemma. **QED**

9.8 Problems

Problem 9.1. Prove that Parity $\notin \mathrm{AC}^0$-Mod-3 by developing the polynomial method over $\mathbb{F}_3$.

Guideline: Follow the argument in section 9.1 and prove:

(1) AC^0-Mod-3 has low-degree probabilistic polynomials over $\mathbb{F}_3$.

(2) Any polynomial of degree d over $\mathbb{F}_3$ fails to compute parity on at least a $1/2 - cd/\sqrt{n}$ fraction of the n-bit inputs.

Problem 9.2. Let $a(n) : \mathbb{N} \to \mathbb{N}$ be a function. Show that $\text{Gap-Maj}_{1/2-1/\log^{a(n)} n, 1/2+1/\log^{a(n)} n}$ is in AC^0 iff $a(n)$ is bounded by a constant.

9.9 Notes

Impossibility results for AC are among the most famous results in complexity and were obtained in the 80's in [97, 12, 313, 231, 257, 131] via various techniques. The first two works obtained super-power results, the others exponential. Each work gives a negative result for a symmetric function and hence applies to majority as well.

The polynomial method Theorem 9.2 is from [231] (an earlier work developed the method for *monotone* circuits). The precise degree bound is Lemma 3.6 in [176]. Theorem 9.1 is from [258]. Stronger bounds for alternating circuits with parity gates matching switching-lemma parameters are obtained in [288] but for less explicit functions (think NExp), building on the proof of Theorem 9.8. For a survey on correlation bounds for AC^0-$\oplus$ and more on Question 9.1 see [302].

As we have seen switching lemmas are an instantiation of the restrict-and-simplify method. The first such lemma is Lemma 9.3, from [97, 12]. Other such lemmas were obtained in [313, 135, 133], the last one giving a refined tradeoff between size and correlation for parity. See also the book [131]. For a switching lemma primer see [41]. Lemma 9.4 is essentially in [135]. For a perspective on these lemmas see [301]. Various proofs exist; I have tried to give a slightly different exposition of a proof based on an encoding argument.

Simulations of various classes by alternating circuits go back to [212]. The main ideas behind Theorem 9.5 and its proof are from [84]. The stated version is from [282]. Our exposition is based on [293], apparently the first exposition after [282].

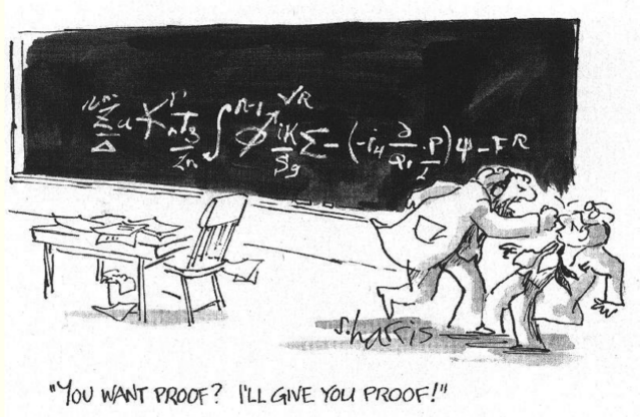
Turning to ACC^0, Lemma 9.9 (and Theorem 9.6) is from [16, 316, 47]. The result that NEexp is not a subset of ACC^0 is from [310]. One step of the original proof was somewhat indirect and was streamlined in [159], this is reflected in the exposition here. The general approach goes back to [309]. The class NExp was later improved to quasi-power non-deterministic time, yielding Theorem 9.8, in [208]. Theorem 9.7 is from [205].

The sampling capabilities of AC^0, Lemma 9.12, follow from [197, 124], though they don't mention AC^0. Our presentation follows [297].

Two lines of research appear intertwined around few main ideas. The first line is the study of complexity classes defined in terms of various quantifiers (or operators). The second is that of circuits with various gates. One basic idea is that And can be approximated by low-degree polynomials. This idea appears in [284] and [231]. Another basic idea is that of modulus amplifying polynomials. They originate from [273] and were studied further in [316, 47], the latter proving Lemma 9.10. A third basic idea is that gap majority is in AC^0. This is from [12] (where Lemma 9.6 is proved) and [255]. For more constructions, see [292] and [13].

Chapter 10

Proofs



The notion of proof is pervasive. We have seen many proofs in this book until now. But the notion extends to others realms of knowledge, including empirical science, law, and more. Complexity theory has contributed a great deal to the notion of proof, with important applications in several areas such as cryptography.

10.1 Static proofs

As remarked in Chapter 5, we can think of problems in NP as those admitting a solution that can be verified efficiently, namely in P. Let us repeat the definition of NP using the suggestive letter V for verifier.

Definition 10.1. A function $f : X \subseteq [2]^* \to [2]$ is in NP iff there is $V \in \mathrm{P}$ (called "verifier") and $d \in \mathbb{N}$ s.t.:

$$f(x) = 1 \Leftrightarrow \exists y \in [2]^{|x|^d} : V(x, y) = 1.$$

We are naturally interested in fast proof verification, and especially the complexity of V. It turns out that proofs can be encoded in a format that allows for very efficient verification. This message is already in the following.

Theorem 10.1. Definition 10.1 does not change if we insist that V_n is a 3CNF.

That is, whereas when defining NP as a proof system we considered arbitrary verifiers V in P, in fact the definition is unchanged if one selects a very restricted class of verifiers: small 3CNFs.

Proof. This is just a restatement of Theorem 5.1. **QED**

This extreme reduction in the verifier's complexity is possible because we are allowing proofs to be long, longer than the original verifier's running time. If we don't allow for that, such a reduction is not known. Such "bounded proofs" are very interesting to study, but we shall not do so now.

Instead, we pursue a different direction. The 3CNF in the above theorem still depends on the entire proof. We can ask for a verifier that only depends on few bits of the proof. Taking this to the extreme, we can ask whether V can only read a constant number of bits from y. Without randomness, this is impossible.

Exercise 10.1. Suppose V in Definition 10.1 only reads $\leq d$ bits of y, for a constant d. Show that the corresponding class would be the same as P.

Surprisingly, if we allow randomness this is possible. Moreover, the use of randomness is fairly limited – only logarithmically many bits – yielding the following central characterization, a.k.a. the PCP theorem.

Theorem 10.2. A function $f : X \subseteq [2]^* \to [2]$ is in NP iff there is $V \in \mathrm{P}$ and $d \in \mathbb{N}$ s.t.:
$f(x) = 1 \Rightarrow \exists y \in [2]^{|x|^d} : \mathbb{P}_{r \in [2]^{d \log |x|}}[V(x, y, r) = 1] = 1$,
$f(x) = 0 \Rightarrow \forall y \in [2]^{|x|^d} : \mathbb{P}_{r \in [2]^{d \log |x|}}[V(x, y, r) = 1] < 0.01$,
and moreover V reads $\leq d$ bits of y.

We give a proof of this theorem assuming Theorem 4.9.

Proof. For the "if" direction, note we can enumerate over all choices for r in power time. So we can compute in P the probability that V accepts, regardless of how many bits of the proof it reads.

For the "only if" direction, it suffices to give the proof system for 3Sat. In turn, by Theorem 4.9 it suffices to give it for $(1-c)$-Gap-3Sat. Let ϕ be an instance of $(1-c)$-Gap-3Sat. The random choice r is used to select a uniform clause in ϕ, the verifier then reads the 3 bits from that clause, and accepts accordingly. We can repeat this a constant number of times to reduce the error probability. **QED**

10.2 Zero-knowledge proofs

In Theorem 10.2 the verifier gains statistical confidence about the validity of the proof, just be inspecting a constant number of bits. Hence the verifier "learns" at most a constant number of bits of the proof. This is remarkable, but we can further ask if we can modify

the proof so that the verifier learns nothing about the proof. Such proofs are called *zero knowledge* and are extensively studied and applied.

We sketch how this is done for Gap-3Color, which is also NP-complete. (This is the problem of deciding if a graph has a 3-coloring or every coloring of the nodes with 3 colors will result in a constant fraction of edges with the same color at the nodes.) Rather than a single proof y, now the verifier will receive a random proof Y. This Y is obtained from a 3 coloring y by randomly permuting colors (so for any y the corresponding Y is uniform over 6 colorings). The verifier will pick a random edge and inspect the corresponding endpoints, and accept if they are different.

The verifier "learns nothing" because all that they see is two random different color. One can formalize "learning nothing" by noting that the verifier can produce this distribution by themselves, without looking at the proof. (So why does the verifier gain anything from y? The fact that a proof y has been written down means that colors have been picked so that every two endpoints are uniform colors, something that the verifier is not easily able to reproduce.)

This gives a zero-knowledge proof for verifiers that follow the protocol of just inspecting an edge. In a cryptographic setting one has to worry about verifiers which don't follow the protocol. Using cryptographic assumptions, one can force the verifier to follow the protocol by considering an *interactive* proof: First a proof y is committed to but not revealed, then the verifier selects an edge to inspect, and only then the corresponding colors are revealed, and only those. This protocol lends itself to a physical implementation that can astonish the right audiences.

10.3 Interactive proofs

We now consider interactive proofs. Here the verifier V engages in a protocol with a prover P. Given an input x to both V and P, the verifier asks questions, the prover replies, the verifier asks more questions, and so on. The case of NP corresponds to a passive verifier which does not ask questions, and a prover that simply sends the proof y to the verifier.

It turns out that it suffices for the verifier to send uniformly random strings $Q_1, Q_2, \ldots$ bits to the prover. This leads to a simple definition.

Definition 10.2. We denote by IP (for *interactive proof systems)* the class of functions $f : X \subseteq [2]^* \to [2]$ for which there is $V \in \mathrm{P}$ and $d \in \mathbb{N}$ such that for every $x \in [2]^n$, letting $b := n^d$:

- If $f(x) = 1$ then $\exists P : [2]^* \to [2]^b$ such that

$$V\left(x, P(Q_1), P(Q_1, Q_2), \ldots, P(Q_1, Q_2, \ldots, Q_b)\right) = 1$$

  for every $Q_1, Q_2, \ldots, Q_b \in [2]^b$.

- If $f(x) = 0$ then $\forall P : [2]^* \to [2]^b$ we have

$$\mathbb{P}_{Q_1, Q_2, \ldots, Q_b \in [2]^b}\left[V\left(x, P(Q_1), P(Q_1, Q_2), \ldots, P(Q_1, Q_2, \ldots, Q_b)\right) = 1\right] \leq 1/3.$$

The following amazing result shows the power of interactive proofs, compared to non-interactive proofs.

Theorem 10.3. IP = PSpace.

Note that like for NP, it is not clear if IP is closed under complement from its definition. Yet it obviously follows from Theorem 10.3.

As a first step towards the proof of Theorem 10.3 we show that IP contains problems not known to be in NP. We introduce a problem that is an extension of ZIC (Definition 3.4).

Definition 10.3. An *arithmetic circuit* of size s in n variables is a sequence of s instructions (or gates) $g_0, g_1, \ldots$ of the following types:

- $g_i := b$, with $b \in [2]$
- $g_i := t \circ t'$, where $\circ$ is either $+$ or $\times$, and each of the terms t, t' is either a gate g_j with $j < i$, or a variable x_k.

All operations are over $\mathbb{N}$. The circuit represents the natural computed by the last instruction.

We are interested in computing the sum of the outputs of the circuit over every $x \in [2]^n$, modulo p, i.e., over the field $\mathbb{F}_p$. The same theory works over any field; we focus on the prime case for simplicity.

Definition 10.4. The SAC (sum arithmetic circuit) problem: Given a prime p, an arithmetic circuit $C(x_1, x_2, \ldots, x_v)$ computing a polynomial of degree $d \leq |C|$ with $(1 - d/p)^v \geq 2/3$, and $s \in [p]$, decide if

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} C(x_1, x_2, \ldots, x_v) = s \mod p. \tag{10.1}$$

Theorem 10.4. SAC is in IP.

The protocol in this result is known as the *sum-check protocol*.

Proof. If $v = 1$ then V can decide this question by itself, by evaluating the circuit. For larger v we give a way to reduce v by 1.

As the first prover answer, V expects a polynomial a of degree d in the variable x, which is meant to be

$$s'(x) := \sum_{x_2, x_3, \ldots, x_v \in [2]} C(x, x_2, x_3 \ldots, x_v).$$

V checks if $a(0) + a(1) = s$, and if not rejects. Otherwise, it recursively runs the protocol to verify that

$$\sum_{x_2, x_3, \ldots, x_v \in [2]} C(Q_1, x_2, x_3, \ldots, x_v) = a(Q_1), \tag{10.2}$$

where Q_1 is uniform in $[p]$.

This concludes the description of the protocol. We now verify its correctness.

In case equation (10.1) is true, P can send polynomials that cause V to accept.

Otherwise, suppose equation (10.1) is false. Consider the first iteration of the protocol. We have $s'(0) + s'(1) \neq s$. Hence, unless V rejects right away because $a(0) + a(1) \neq s$, we also have $a(x) \neq s'(x)$. The polynomials a and s' have degree $\leq d$. Hence $a - s'$ is a non-zero polynomial of degree $\leq d$, and therefore it has $\leq d$ roots (Fact B.28). So we get:

$$\mathbb{P}_{Q_1}[a(Q_1) \neq s'(Q_1)] \geq 1 - d/p.$$

When this event occurs, equation (10.2) is again false, and we can repeat the argument. Overall, the probability that we maintain a false statement throughout the protocol is $\geq (1 - d/p)^v \geq 2/3$. **QED**

Exercise 10.2. The protocol exchanges cv messages (equivalently, has v alternations between prover and verifier, or cv rounds). Modify it so that it exchanges only $v/100$ messages.

To apply the sum-check protocol to *boolean* rather than *algebraic* circuits we use a far-reaching technique: *arithmetization.* We construct an arithmetic circuit C_ϕ over a field $\mathbb{F}$ which agrees with ϕ on *boolean* inputs, but that can then be evaluated over other elements of the field. This is done in the following way:

$$\begin{aligned} x &\to x \\ f \wedge g &\to f \cdot g \\ f \vee g &\to f + g - f \cdot g \\ \neg f &\to 1 - f. \end{aligned}$$

Theorem 10.5. Given a 3CNF formula ϕ and $k \in \mathbb{N}$, deciding if ϕ has exactly k satisfying assignments is in IP.

Proof. Let C_ϕ be the arithmetization of ϕ, which note has degree $\leq cn$. Let p be a prime of size $\leq 2^{cn}$. It suffices to solve the SAC instance (Definition 10.4) $\sum_x C_\phi(x) = k \mod p$, since $k \leq 2^n$. The prime specifying the field can be sent from the prover (the verifier can check it using that Primes is in P). **QED**

To show PSpace $\subseteq$ IP or in fact even weaker statements like Π_2P $\subseteq$ IP one needs to consider formulas with both $\exists$ and $\forall$ quantifier. To determine the validity of such a formula it is natural to proceed as in Theorem 10.5 and make the following substitutions:

$$\begin{aligned} \exists x : f(x) &\to \sum_{x \in [2]} f(x) \\ \forall x : f(x) &\to \prod_{x \in [2]} f(x). \end{aligned}$$

This is a valid transformation: the formula is true iff the corresponding expression is > 0.

We would then be running the sum-check protocol, with the difference that when the polynomial p corresponds to a Π gate we check that $p(0) \cdot p(1) = s$ (instead of $p(0)+p(1) = s$, note here the variable s does not just correspond to a sum). We call this the *sum-prod-check* protocol.

The main problem with this approach is that the degree of the polynomials to be sent in the protocol can explode, making it unfeasible for the verifier to even receive such polynomials.

Example 10.1. Let

$$f(x, y_1, \ldots, x_k) := \exists x \forall y_1 y_2 \cdots y_k : x.$$

The arithmetization of this would be

$$f(x, y_1, \ldots, x_k) := \sum_{x \in [2]} \prod_{y_1, \ldots, y_k \in [2]} x = \sum_{x \in [2]} x^{2^k}.$$

The polynomial in x corresponding to the $\sum$ gate has too large a degree.

The solution is similar in spirit to the reduction of circuits to Sat, Theorem 5.1: we are going to add new variables.

Lemma 10.1. Let $f = Q_1 x_1 Q_2 x_2 \cdots Q_k x_k g(x_1, x_2, \ldots, x_k)$ be a formula where g is quantifier-free. All quantifiers are over [2]. Proceeding from left to right, replace an occurrence of

$$\forall x_i \exists x_{i+1} \cdots Q_k x_k : g(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \cdots, x_k)$$

with the formula

$$\forall x_i \exists x'_1, x'_2, \ldots, x'_{i-1} : \left(\left(\wedge_{j=1}^{i-1} x_j \iff x'_j \right) \wedge \exists x_{i+1} \cdots Q_k x_k : g(x'_1, x'_2, \ldots, x'_{i-1}, x_i, x_{i+1}, \cdots, x_k) \right).$$

At the end of the process we have obtained f' s.t.:
(1) $f \iff f'$,
(2) $|f'| \leq |f|^c$,
(3) If f' is true then there exists a prover that in the sum-prod-check protocol on the arithmetization of f' sends polynomials of degree at most the degree of the arithmetization of g plus a constant.

Note that $x_j \iff x'_j$ means that the variables are equal; it can be written as $(x_j \wedge x'_j) \vee (\neg x_j \wedge \neg x'_j)$.

Proof. (1) and (2) follow by inspection.

(3): Note that we are applying the sum-prod-check protocol to the arithmetization of a formula that is not in prenex form (a formula is in prenex form if all quantifiers are at the beginning). However, it is almost in prenex form, the only difference are the $\iff$ equality checks. In a generic step of the execution, we have fixed some variables to field elements, and we are considering a univariate polynomial in a variable x corresponding to a quantifier Qx. If a $\forall$ quantifier appears to the right of x, then the degree of x is $\leq$ the degree of the

equality check; the rest of the formula does not involve the variable x and thus does not affect its degree. If no $\forall$ quantifier appears then the degree is at most that of the equality check plus the degree in the arithmetization of g. **QED**

Example 10.2. Let us return to Example 10.1 and set $k = 2$ for simplicity. Thus we consider the formula

$$\exists x \forall y_1 y_2 : x.$$

First note that the arithmetization is

$$\sum_{x \in [2]} x^4,$$

which is a polynomial of degree 4. Let us now see how the transformation in Lemma 10.1 reduces the degree. Applying it to the leftmost $\forall y_1$ quantifier in f we get

$$\exists x \left[\forall y_1 \exists x' : (x' \iff x) \wedge \forall y_2 : x' \right].$$

Next the transformation would be applied again to the $\forall y_2$ quantifier. We don't write this down because the above formula only depends on x', but this wouldn't change the degree of x in the protocol. The important point is that in the arithmetization the degree in x is only the degree of the equality check, which is 2 (whereas recall previously it was 4).

Proof of IP=PSpace Theorem 10.3.. To show IP $\subseteq$ PSpace one can give a recursive algorithm that, given a verifier and an input, computes the highest probability that the verifier can be made to accept by some prover. The details of this are similar to the proof that QBF $\in$ PSpace, see Theorem 6.12, and are omitted.

For the other direction it suffices to show that QBF $\in$ IP by Theorem 6.12. Given a QBF f, the verifier applies the transformation in Lemma 10.1 to obtain f'. The verifier then computes the arithmetization h of f' where $\exists x \in [2]$ in f' becomes $\Sigma_{x \in [2]}$ in h and $\forall x \in [2]$ becomes $\prod_{x \in [2]}$. Hence h denotes a number and we note

$$f \text{ is true} \iff f' \text{ is true} \iff h > 0.$$

The prover will show that $h > 0$ by first sending a prime p and then proving $h = K$ over $\mathbb{F}_p$ using the sum-prod-check protocol. Note that g is at most doubly exponential in n. By Theorem B.3 a prime of n^c bits can be found so that $h \neq 0$ over $\mathbb{F}_p$. By Lemma 10.1 during the execution of the sum-prod-check protocol the degree remains $\leq cn$. **QED**

10.4 Interactive proofs within P

The study of interactive proofs is rich. Many interrelated aspects are of interest, including the efficiency of the verifier, the number of rounds of the protocol, the communication complexity,

and the error parameter. The efficiency of the prover is also of interest. By this we mean the efficiency of the prover in the case $f(x) = 1$. The verifier should reject with high probability in case $f(x) = 0$ even when interacting with a computationally unbounded prover. (Again, variants in which the protocol only withstands computationally bounded provers are of interest too).

In this section we discuss interactive proofs for problems in P. The goal is having the verifier in Quasi-Linear-Time and the prover in FP. Surprisingly, this can be accomplished for any function in NL or any boolean function in NC that satisfies a certain uniformity condition.

Theorem 10.6. Any function in NL has interactive proofs where the verifier runs in quasi-linear time, and moreover when a proof exists it can be computed in FP.

The appeal of this theorem is clear: One can delegate expensive computation of functions in L (for example, functions which naively take time n^{10}) and verify it in quasi-linear time; and the delegated computation is still feasible.

The proof of Theorem 10.6 displays a beautiful interplay between algebra and computation, and in fact, we will establish stronger results (applying to NL and other classes). Before this, however we give a simple example where an efficient prover exists. This serves as a warm-up for the proof of Theorem 10.6.

10.4.1 Warm-up: Counting triangles

We consider the problem of counting the number of triangles in a graph. Such a problem can be trivially solved in time n^c. Whether faster run time such as $n \log^c n$ are possible is unknown. We show that such time bounds can be achieved via interaction, and moreover the proofs are still feasible.

Theorem 10.7. Given an undirected graph G with n nodes and $m \geq n$ edges, and an integer s, there is an interactive proof for deciding if the number of triangles in G is s such that the verifier is in Quasi-Linear-Time and the prover is in FP.

Proof. We construct a suitable arithmetic circuit and then apply the sum-check protocol from Theorem 10.4. The circuit has $v := 3 \log n$ variables x_i organized in 3 blocks y_0, y_1, y_2 where each y_i consists of $\log n$ variable. Each y_i corresponds to a node in the graph. Thus, the number of triangles can be written as

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} E(y_0, y_1) \cdot E(y_0, y_2) \cdot E(y_1, y_2)$$

where E is 1 if y_i and y_j are an edge in G.

To run the sum-check protocol we need an arithmetic circuit, that is, we need to be able to make sense of evaluating the circuit over large fields. The function E is only defined over bits, so we need to view it as a polynomial that can be evaluated over larger fields. At the

same time, computing this polynomial should be easy for the verifier (so we can't just say it has some polynomial like any other function, since the polynomial could have degree $2 \log n$ and require quadratic time, which isn't in the verifier budget). The standard expansion will do. Define

$$\widehat{E}(z, z') := \sum_{\alpha, \alpha'} [z = \alpha] \cdot [z' = \alpha'] \tag{10.3}$$

where the sum is over all edges $\{\alpha, \alpha'\}$ and $z = z_0 z_1 \cdots z_{\log n - 1}$. (We should assume that the graph has no self loops.) In turn, we can write

$$[z = \alpha] \iff \prod_{i \in [\log n] : \alpha_i = 1} z_i \prod_{i \in [\log n] : \alpha_i = 0} (1 - z_i)$$

and the same for z'. Note $\widehat{E}$ is a polynomial of degree $2 \log n$.

With this notation, the verifier needs to verify that

$$\sum_{x_1, x_2, \ldots, x_v \in [2]} \hat{E}(y_0, y_1) \cdot \hat{E}(y_0, y_2) \cdot \hat{E}(y_1, y_2) = s.$$

The whole expression is a polynomial of degree 3 times the degree of $\hat{E}$, that is $6 \log n$.

Because the sum is $\leq n^c$, by the remaindering Theorem 6.6 it suffices to verify the expression modulo a prime p uniformly chosen from a set of $\geq c \log n$ primes. We shall run the sum-check protocol over such a field $\mathbb{F}_p$. For the correctness of the sum-check protocol (Theorem 10.4) it suffices that $p \geq \log^c n$. Selecting a uniform prime $\leq \log^c n$ suffices for the overall correctness (most primes will be large enough for Theorem 10.4, using Lemma 3.1).

Let us now verify the running times. The prover at each round sends a univariate polynomial which is obtained by summing over n^c values. By inspection, this polynomial has degree 2 (degree 1 for each of the factors $\hat{E}$ where the variable occurs). Hence the prover is in FP. The verifier at each round except the last one needs to evaluate this polynomial, and perform a constant number of field operations. This takes time $\log^c n$. At the last round, the verifier needs to evaluate

$$\hat{E}(y'_0, y'_1) \cdot \hat{E}(y'_0, y'_2) \cdot \hat{E}(y'_1, y'_2)$$

for some $y'_i \in \mathbb{F}_p$. Each factor $\widehat{E}(y'_i, y'_j)$ can be computed by the verifier using equation (10.3). This requires going through the m edges of the graph, and for each edge perform $c \log n$ field operations. This concludes the proof that the verifier is in Quasi-Linear-Time. **QED**

Exercise 10.3. Reduce the number of rounds to constant (i.e., a constant number of Q_i in Definition 10.2) while maintaining the complexity of the verifier and the prover. Hint: Modify the sum-check protocol.

10.4.2 Proof of Theorem 10.6

As mentioned, we actually prove stronger results for circuits. Naturally, the circuits need to satisfy a certain uniformity condition. This condition (stated in the theorem) will be algebraic, as one can expect from arithmetization.

Convetions on circuits. We shall consider functions computable by circuits of power-size n^e and depth $d(n) := \log^e n$. (The results generalize to larger depth, with the depth factoring in the verifier runtime.) It will be convenient to arrange the gates into a matrix of $d(n)$ rows and n^e columns, where row 0 corresponds to the output and row $d-1$ to the input literals, where row i takes as input row $i-1$ only, at the cost of possibly duplicating gates. We shall always index the gates in a level using $\log n^e$ bits; unused gates can be set to the constant 0. Any gate in a circuit is indexed with $\log(dn^e)$ bits.

For simplicity, we use NAnd gates only (which are easily seen to be universal).

Also, we will need to work over larger fields; at the same time, this switch to larger fields should not cause the number of descriptions of gates to become infeasible. So, we shall pick a field $\mathbb{F}$ and $\mathbb{H} \subseteq \mathbb{F}$ and index gates by strings of

$$m := \frac{\log(dn^e)}{\log |\mathbb{H}|}$$

elements from $\mathbb{H}$. In terms of bits, this is $\log(dn^e)$ like before.

When working over larger fields, we will use the same number of elements, but this time from $\mathbb{F}$. In terms of bits this will be

$$\log |\mathbb{F}| \cdot m$$

which will still be $c \log(dn^e)$ if $|\mathbb{F}| \leq |\mathbb{H}|^c$. Indeed, we set

$$|\mathbb{H}| := \log n$$
$$|\mathbb{F}| := \log^{c_e} n.$$

TBD

Definition 10.5. We denote by algebraic-uniform (boolean) NC the class of functions $f : [2]^* \to [2]$ for which there is e such that given n we can compute in FP a polynomial $\phi : (u, v, w) \in \mathbb{H}^{3m} \to [2]$ of degree $\log^e n$ which computes if gate u takes as input gates v and w in a circuit for f_n of size n^e and depth $\log^e n$.

We used the same parameter e for both the complexity of f and the degree of ϕ for simplicity and w.l.o.g., for increasing e makes the definition easier and easier to satisfy, as we can always ignore some gates.

Satisfying this definition is not typically an issue, in the sense that it's hard to find examples where it is hard to satisfy it.

Lemma 10.2. NL $\subseteq$ algebraic-uniform NC.

The proof of this is somewhat tangential to proof systems. Yet let us give it.

Proof. Let $f \in$ NL. The circuit for f_n can have the following structure (cf Theorem 7.1) for $a = c_f$. There is a layer A of n^a gates at depth 1, each depending on a single input bit. The rest of the circuit consists of $a \log n$ copies of a circuit $M : [2]^{n^a} \to [2]^{n^a}$ stacked on top of each other, with one copy taking A as input.

Here A computes the $n^{a/2} \times n^{a/2}$ adjacency matrix of the configuration graph of f on input x. Each gate corresponds to an edge $u \to v$ where $u \to v$ are configurations.

M is matrix squaring but with $+$ replaced by $\vee$. Each output bit is an $\vee$ over $n^{a/2}$ products of 2 gates.

There remains to exhibit ϕ. First we show that ϕ is computable by power-size constant-depth alternating circuits over the $3m \log |\mathbb{H}|$ bits corresponding to the field elements. Then we use a generic transformation to obtain a polynomial.

For A, there are some low-level details. ϕ is given as input a gate representing two configurations u, v, and a gate representing an input bit literal ℓ_i, or a constant b, and we need to decide if that's the right connection.

Configuration u may not read any input. In this case the gate should be the constant 1 if v is the next configuration and the constant 0 otherwise, that is, we have to make sure that it's connected to the right constant b. Looking back at Definition 1.7, we can compute v from u with the desired circuit for all the operations except multiplication and division (which are known not to be in AC^0). However, when only space complexity is concerned, these operations can be dispensed with. For example, we can simulate multiplication using many additions, and similarly for division. This concludes the case when u does not read any input.

Otherwise, configuration u reads an input bit x_i. In this case, the value of the bit is in some register in v, and ϕ checks that the gate in A is connected to the corresponding literal ℓ_i.

For M, we can arrange the $\vee$ in a binary tree, and index by $i \to 2i, i \to 2i+1$, as in the heap data structure. Then ϕ can be computed just using bit-shift and addition by 1, which is in AC^0 (Example 9.1).

This concludes the description of a circuit for ϕ. Now we transform this circuit into a polynomial. We can view the circuit as using $\wedge$ and $\neg$ gates, and we arithmetize it (replace $\wedge$ by multiplication and $\neg x$ by $1 - x$). The degree is as desired since the circuit has power size and constant depth. We could just use this polynomial if our inputs were bits, but they are in fact field elements in $\mathbb{H}$. However, we can extract each of the $\log |\mathbb{H}|$ bits using a polynomial of degree $\leq |\mathbb{H}|$ computable in FP. There are several ways to do this; since $\mathbb{H}$ has size only $\log n$, a brute-force approach (which works for every function) suffices. Namely, for any function $f : [2]^{\log |\mathbb{H}|} \to [2]$ in FP we can write down the polynomial $p(x) = \sum_{a:f(a)=1} [x = a]$. This has degree $\leq |\mathbb{H}|$ and can be computed in time $|\mathbb{H}|^c \leq \log^c n$. Composing these polynomials with the ones coming from the alternating circuit concludes the proof. **QED**

Main statement and proof Thanks to Lemma 10.2, Theorem 10.6 follows from the following statement about circuits.

Theorem 10.8. Algebraic-explicit NC has interactive proofs where the verifier is in Quasi-Linear-Time and the prover is in FP.

In the remainder of this section we present the proof of this Theorem 10.8. Let f be in algebraic-uniform NC and e as in Definition 10.5. Given input x, let α_i denote the value of

the gates at distance i from the output. So α_0 is the output and α_{d-1} is the input literals. The protocol proceeds in d stages. In stage $i-1$ a claim of the form

$$\widehat{\alpha}_{i-1}(z_{i-1}) = b_{i-1}$$

is reduced to a claim of the form

$$\widehat{\alpha}_i(z_i) = b_i.$$

Here $z_i \in \mathbb{F}^m, b_i \in \mathbb{F}$, and $\widehat{\alpha} : \mathbb{F}^m \to \mathbb{F}$ is the arithmetization of α which agrees with α over $\mathbb{H}^m$ and is defined as

$$\widehat{\alpha}(x) := \sum_{y \in \mathbb{H}^m} \mathrm{EQ}(x,y) \cdot \alpha(y).$$

where $\mathrm{EQ}(x,y)$ is a polynomial that, when evaluated over $\mathbb{H}^m$, computes equality. This is known as *low-degree extension*.

Exercise 10.4. Give such a polynomial for EQ that has degree $m(\mathbb{H}-1)$ and can be evaluated in time $\log^{c_e} n$. Hint: Use that $x^{q-1} = 1$ for $x \neq 0$ in a field of size q.

Note these polynomials are never sent to the verifier, as they are not within their budget. Various *univariate* restrictions of these polynomials (of the same degree d^c) will sent to the verifier. On the other hand, the polynomials are computable by the prover in power time. What constitutes a "claim" are the values i, z_i, b_i.

At Stage $i = 1$ we have that z_0 is the output gate and b_0 is the output of the circuit.

Exercise 10.5. Explain how the claim at Stage $i = d$, corresponding to the input level, is verified in Quasi-Linear-Time without further interaction.

The induction step: Stage $i-1$. W.l.o.g. assume the circuit consists of NAnd gates only (see the discussion after Definition 2.1). We have

$$\widehat{\alpha}_{i-1}(z) = \sum_{u,v,w \in \mathbb{H}^m} \mathrm{EQ}(z,u) \cdot \phi(u,v,w) \cdot (1 - \widehat{\alpha}_i(v) \cdot \widehat{\alpha}_i(w)).$$

Note that on the rhs we could have written α_i instead of $\widehat{\alpha}_i$. Indeed, they agree on $\mathbb{H}^m$. However, we need to apply the sum-check protocol so we need algebraic computation.

The current claim is that the rhs equals b_{i-1}. The term inside the sum on the rhs is an algebraic circuit computing a polynomial of degree

$$\leq m \cdot |\mathbb{H}| + \log^e n + 2m|\mathbb{H}| \leq \log^{c_e} n.$$

The sum-check protocol reduces the claim to

$$\mathrm{EQ}(z,\widehat{u}) \cdot \phi(\widehat{u},\widehat{v},\widehat{w}) \cdot (1 - \widehat{\alpha}_i(\widehat{v}) \cdot \widehat{\alpha}_i(\widehat{w})) = b,$$

for some $\widehat{u}, \widehat{v}, \widehat{w} \in \mathbb{F}^m$ and $b \in \mathbb{F}$. (We only stated the sum-check protocol for sums over $[2]$, but one can readily extend it to sums over larger sets such as $\mathbb{H}$.) This requires m iterations.

In each iteration the prover sends a univariate polynomial of degree $\log^{c_e} n$ and the verifier evaluates it at $|\mathbb{H}| = \log n$ points (this number of points corresponds to the extension). This takes time $\log^{c_e} n$. The error will be $\leq \log^{c_e} n/|\mathbb{F}| \leq 1/\log^{c_e} n$ in each iteration, by our choice for $|\mathbb{F}|$. Overall, the error is $\leq 1/\log^{c_e} n$.

However, the new claim appears to require two evaluations of $\widehat{\alpha_i}$, which would yield an exponential increase in complexity. To avoid it, we use another idea to reduce the evaluation at two points to a single point. The prover sends a univariate polynomial $p(t)$ of the same degree as $\widehat{\alpha_i}$, which is meant to be $p(t) := \widehat{\alpha_i}(\widehat{v} + t(\widehat{w} - \widehat{v}))$. The verifier checks that

$$\mathrm{EQ}(z, \widehat{u}) \cdot \phi(\widehat{u}, \widehat{v}, \widehat{w}) \cdot (1 - p(0) \cdot p(1)) = b$$

and it rejects if it does not hold. If it does hold, it picks a uniform value $t \in \mathbb{F}$ and the new claim is now

$$\widehat{\alpha_i}(\widehat{v} + t(\widehat{w} - \widehat{v})) = p(t).$$

Note that if $p(t) \neq \widehat{\alpha_i}(\widehat{v} + t(\widehat{w} - \widehat{v}))$ then the claim is still false, except with probability, again, $\log^{c_e} n/|\mathbb{F}| \leq 1/\log^{c_e} n$ over t. The complexity and error probability at this step are no more than those incurred in the sum-check protocol.

By a union bound over the $\log^e n$ stages, the probability of error is $\leq 1/\log^{c_e} n$.

10.5 Problems

Exercise 10.6. Show that L has interactive proofs where the verifier is in Quasi-Linear-Time. Give an alternative proof not using Theorem 10.6 but following instead the proof of Theorem 10.3 and using Problem 6.2.

10.6 Notes

Interactive proofs were put forth simultaneously in [33, 115]. The latter paper also introduced zero-knowledge. The zero-knowledge protocol for 3Color is from [110].

Theorem 10.3 is from [193, 247], with the last paper proving it as stated. For the history of this famous result see [27].

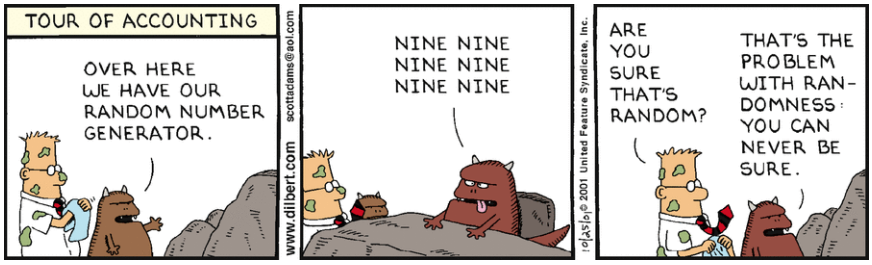
The proof of Theorem **??** follows [1].

Interactive proofs for functions in P with efficient verifier were first studied in [114], where essentially Theorem 10.8 appears. Our presentation of this result follows [108]. The latter differs from the former in the way the uniformity of circuits is handled. Our presentation also differs and uses that the circuits for simulating L or NL are sufficiently uniform. In fact, this was studied already in [158] where even constant-locality uniformity is achieved. Theorem 10.7 is from [112]. [235] give protocols that are even constant-round for TiSP. [272] gives alternative arguments (not achieving constant-round) based on matrix powering. Simpler constant-round protocols for smaller classes are in [113]. For a survey of some of these works, see [108].

For more on interactive proofs and zero knowledge see the book [272].

Chapter 11

Pseudorandomness



Suppose I say to you that I've tossed coin 40 times and got this sequence of heads (0) and tails (1):

$$0101010101010101010101010101010101010101.$$

You'd probably think this can't be true. But suppose instead I claim that I got

$$1000111110100010011110100101111101100100.$$

Maybe you would think this is possible then? But why do we feel this way? For a fair coin, the two strings have the same probability of 2^{-40}!

This example leads to a very interesting question: What is randomness? Of course, we've been using randomness all along since the first chapter. Still, let's step back and consider three possible answers:

1. **Classical:** Each string has the familiar probability. This viewpoint is useful in mathematics but the example above shows that it doesn't capture our intuitive notion of randomness.

2. **Intrinsic (or ontological):** A string is the less random the shorter description it has. The first string has the short program "Print 01 for 20 times," while the shortest program for the second seems to be "Print 1000111110100010011110100101111101100100."

3. **Behaviouristic:** Randomness is in the eyes of the beholder: A string R is random for f if f can't distinguish it from a truly random string. In other words, R fools f into thinking that R is random.

The last answer seems the most useful, and we now make it precise.

Definition 11.1. A distribution R over $[2]^n$ *ϵ-fools* (or is ϵ-pseudorandom for) a function $f : [2]^n \to [2]$ if $|\mathbb{E}[f(R)] - \mathbb{E}[f(U)]| \leq \epsilon$, where U is uniform in $[2]^n$. A function f ϵ-breaks (or tells, distinguishes) distributions D and E if $|\mathbb{E}[f(D)] - \mathbb{E}[f(E)]| \geq \epsilon$. If E is omitted it is assumed to be the uniform distribution.

We are naturally interested in distributions that are pseudorandom yet have very little entropy. As in section §1.8, counting arguments show that very little entropy is needed for non-explicit distributions, about logarithmic in the number of tests to be fooled. But our ability to explicitly construct such distributions is limited by the grand challenge:

Claim 11.1. Let R be a distribution over $[2]^n$ with support of size $< 2^n/2$. Suppose that R $1/2$-fools a set F of functions. Then the indicator function $g : [2]^n \to [2]$ of the support of R is not in F.

Proof. We have $\mathbb{E}[g(U)] < 1/2$ while $\mathbb{E}[g(R)] = 1$, so g is not $1/2$-fooled and cannot be in F. **QED**

For example, if $F = \text{CktP}$ and R can be sampled in P then $g \in \text{NP}$, and so $\text{NP} \not\subseteq \text{CktP}$.

The above claim can be strengthened. In general, constructing such distributions can be thought of as a refined impossibility result that is closely related to correlation, recall Definition 8.3.

To simplify the following discussion, we introduce the notion of a pseudorandom generator which makes it easier to talk about the entropy of the distribution and its explicitness.

Definition 11.2. An algorithm G is a *pseudorandom generator*, abbreviated generator or PRG, that ϵ-fools a class F of functions (or a generator for F with error ϵ) with seed length s (a function of both n and ϵ) if on input n and ϵ, and a uniform seed U of length $s(n, \epsilon)$, G outputs a distribution on n bits that ϵ-fools any function in F on inputs of length n. The *stretch* is $n - s(n, \epsilon)$.

Note that we use n to denote the *output length* of G, because it is the *input length* for a test that's trying to tell G from random. Also, we typically have the output length of G much longer than the input length. For some applications, it suffices if G is computable in power-time in the output length, which can be exponential in the input length. We shall simply say that G is *explicit* in this case. However many generators we present below, especially those for restricted models, are explicit in a stronger sense: Given an input and an index to an output bit, that bit can be computed in P.

Exercise 11.1. Suppose there is $a > 0$ and an explicit generator with seed length $s(n) = a \log n$ that 0.1-fools circuits of size n, for a constant a. Prove that P = BPP.

Note that to eliminate one parameter we set the size of the circuit test equal to the output length of G. Recall from Definition 2.1 that input gates are not counted towards size; the

circuit may simply ignore most of its input bits, which makes sense since very few input bits suffice, information-theoretically, to tell the output of G from uniform.

Given Claim 11.1, there are two main avenues for research, closely paralleling the development of earlier chapters. The first is proving unconditional results for restricted models, like AC. Actually, pseudorandomness being a more refined notion of impossibility, even very simple models like local functions are non-trivial, and results for them very useful. The second is proving reductions, that is linking the existence of PRGs to other conjectures. Interestingly, some of the techniques are general and apply to both settings.

11.1 Basic PRGs

In this section we present PRGs for several basic classes of tests. Besides being basic, these tests are the backbone of several other constructions, and somewhat surprisingly suffice to fool apparently stronger classes of tests.

11.1.1 Local tests

The simplest model to consider is perhaps that of local functions.

Definition 11.3. A distribution over $[2]^n$ is *k-wise uniform* if every k bits are uniform in $[2]^k$ (equivalently, any k-local function is 0-fooled). A k-wise generator is a map whose output distribution is k-wise uniform.

Exercise 11.2. Given an explicit 1-wise uniform generator with seed length $s(n) = 1$.

Theorem 11.1. There are explicit k-wise uniform generators with seed length $s = ck \log n$.

Proof. Wlog assume n is a power of 2 and let $\mathbb{F}$ be the field of size n. More generally, the range of G will be $\mathbb{F}^n$, and the distribution of any k coordinates will be uniform over $\mathbb{F}^k$. View the input as coefficients a_i $i \in [k]$ of a polynomial p of degree $k-1$. Define the i output element of G to be $p(i)$. Any k-tuple of field elements is uniform, for if two different polynomials give the same tuple then their difference is a non-zero polynomial of degree $k-1$ with $\geq k$ roots, violating Fact B.28. (We can assume $k \leq n$ for else the theorem is trivial.) **QED**

The bound on s is tight for $k \leq n^c$. For k closer to n different arguments apply, but we won't need them here.

Theorem 11.2. The minimum seed length is $\geq ck \log(2n/k)$.

Proof. Think of the support as $\{-1, 1\}^n$, and write down a $2^s \times n$ matrix where row x is $G(x)$. For even k, and for any $T \subseteq [n]$ of size $k/2$, consider the 2^s-long vector v_T obtained by multiplying together the columns indexed in T. Note that the v_T are orthogonal, hence independent (B.23), and so $2^s \geq \binom{n}{k/2}$, whence $s \geq ck \log(2n/k)$. **QED**

Powerlog-wise uniformity suffices to fool AC:

Theorem 11.3. Any $\log(m/\epsilon)^{cd}$-wise uniform distribution over $[2]^n$ ϵ-fools AC of size m and depth d.

In particular, there are explicit generators that ϵ-fool such circuits with seed length $\log(m/\epsilon)^{cd}$. We give generators with this seed length below, via a different route.

11.1.2 The power of NC^0: Local maps can fool local tests

To compute the construction in the proof of Theorem 11.1 it seems we need to read the entire input of $ck \log n$ bits. We now give a different construction where it suffices to read $c \log n$ bits. Note this is a dramatic saving when k is large. In fact we give a general tradeoff between the locality and the seed length.

Theorem 11.4. There are k-uniform generators $G : [2]^s \to [2]^n$ that are d-local, whenever

$$\left(\frac{cdk}{s}\right)^{d/2} \leq \frac{1}{n}.$$

For example, we can have $s = ck \log n$ and $d = c \log n$. At the other extreme, we can also have $s = n^{0.1}$ and $d = c$. It even suffices if the seed is dk uniform as opposed to completely uniform.

Proof. Pick a random bipartite graph with s nodes on the left and n nodes on the right. Every node on the right side has degree d and computes the XOR of its neighbors. By Fact B.29 it suffices to show that for any non-empty subset $S \subseteq [n]$ of size $\leq k$, the XOR of the corresponding bits is unbiased. For this it suffices that S has a unique neighbor. For that, in turn, it suffices that S has a neighborhood of size greater than $\frac{d|S|}{2}$ (because if every element in the neighborhood of S has two neighbors in S then S has a neighborhood of size $< d|S|/2$). We pick the graph at random and show by standard calculations that it has this property with non-zero probability. Write $N(S)$ for the set of neighbors of nodes in S. We need to bound

$$\mathbb{P}\left[\exists S \subseteq [n], 0 < |S| \leq k, \text{ s.t. } |N(S)| \leq \frac{d|S|}{2}\right].$$

We can rewrite this as the probability that there is a small T that contains $N(S)$, and then

bound the latter:

$$\begin{aligned}
&\mathbb{P}\left[\exists S \subseteq [n], 0 < |S| \leq k, \text{ and } \exists T \subseteq [s], |T| \leq \frac{d|S|}{2}, \text{ s.t. } N(S) \subseteq T\right] \\
\leq &\sum_{i=1}^{k} \binom{n}{i} \cdot \binom{s}{d \cdot i/2} \cdot \left(\frac{d \cdot i}{s}\right)^{d \cdot i} \\
\leq &\sum_{i=1}^{k} \left(\frac{e \cdot n}{i}\right)^{i} \cdot \left(\frac{e \cdot s}{d \cdot i/2}\right)^{d \cdot i/2} \cdot \left(\frac{d \cdot i}{s}\right)^{d \cdot i} \\
= &\sum_{i=1}^{k} \left(\frac{e \cdot n}{i}\right)^{i} \cdot \left(\frac{e \cdot d \cdot i/2}{s}\right)^{d \cdot i/2} \\
= &\sum_{i=1}^{k} \left[\underbrace{\frac{e \cdot n}{i} \cdot \left(\frac{e \cdot d \cdot i/2}{s}\right)^{d/2}}_{A}\right]^{i}.
\end{aligned}$$

It suffices to have $A \leq 1/2$, so that the probability is strictly less than 1, because $\sum_{i=1}^{k} 1/2^i = 1 - 2^{-k}$. The result follows. **QED**

11.1.3 Low-degree polynomials

Another natural model is that of low-degree polynomials. section §5.5 and section 9.1 give several applications, and we encounter more below in section 11.1.4.

Theorem 11.5. There are explicit generators that ϵ-fool degree-1 polynomials over $\mathbb{F}_2$ with seed length $s = c \log(n/\epsilon)$.

Such distributions care called *ϵ-bias*, or *small-bias*.

Proof. Let $q := 2^{\log n/\epsilon}$ and identify the field $\mathbb{F}_q$ with bit strings of length $\log q$. We view r as a pair $(s, t) \in (\mathbb{F}_q)^2$. Then we define

$$f((s, t), i) := \langle s^i, t \rangle$$

where s^i is exponentiation in $\mathbb{F}_q$ and $\langle ., . \rangle : (\mathbb{F}_q)^2 \to [2]$ is defined as $\langle u, v \rangle := \sum u_i \cdot v_i$ over $\mathbb{F}_2$.

Consider any non-zero $x \in [2]^n$. The critical step is to note that

$$\sum_i \langle S^i, T \rangle x_i = \sum_i \langle x_i \cdot S^i, T \rangle = \langle \sum_i x_i \cdot S^i, T \rangle.$$

Now, if $x \neq 0$, then the probability over S that $\sum_i x_i \cdot S^i = 0$ is $\leq n/q \leq \epsilon$. This is because any S that gives a zero is a root of the non-zero, univariate polynomial $q(y) := \sum_i x_i \cdot y^i$ of degree $\leq n$ over $\mathbb{F}_q$, and so the bound follows by Fact B.28.

Whenever $\sum_i x_i \cdot S^i \neq 0$, the probability over T that $\langle \sum_i x_i \cdot S^i, T\rangle = 0$ is $1/2$. **QED**

For a different construction see Problem 11.4.

To fool polynomials of degree $d > 1$, we can take the xor of d independent copies of generators for degree 1. This is known to work for $d < \log n$, and is unknown beyond that.

Theorem 11.6. The sum of d generators that ϵ-fool degree-1 polynomials over $\mathbb{F}_2$, on independent seeds, fools degree-d polynomials with error $\leq c\epsilon^{1/2^{d-1}}$.

Question 11.1. *Does this work for $d > \log n$?*

11.1.4 Expander graphs and combinatorial rectangles: Fooling AND of sets

In this subsection we construct a PRG to fool the And of (the indicator functions) of sets. We use "set" and "function" interchangeably in this section. This basic construction ties together many things we have seen and showcases techniques which allow to build even more powerful PRGs. Also, it suffices for the time-efficient simulation of BPP in PH, Item (2) in Theorem 5.8, see Problem 11.6.

Theorem 11.7. There is a PRG G that ϵ-fools the product of t subsets of $[2]^m$ with seed length $m + c(\log t)\log(mt/\epsilon)$. In other words, for any functions $f_i : [2]^m \to [2]$ we have $|\mathbb{E}[\prod_i f_i(U_i)] - \mathbb{E}[\prod_i f_i(X_i)]| \leq \epsilon$ where $(X_1, X_2, \ldots, X_t) = G(U)$.

Except for the extra $\log t$ factor, the seed length is optimal.

The fundamental case of $t = 2$ corresponds to *expander graphs.*

Exercise 11.3. [Where is the graph, and why is it expanding?] Let L and R be two disjoint sets with $M := 2^m$ nodes each, and define the graph on vertices $L \cup R$ and edges (x, y) from L to R for any output $(x, y) = G(z)$. For simplicity, further assume that x is uniform for uniform z (a condition satisfied by the construction below). Prove that any set $X \subseteq L$ of αM nodes has $\geq (1 - \epsilon/\alpha)M$ neighbors in R.

For expander graphs, explicit constructions with seed length $m + c\log 1/\epsilon$ are known. We give below a simpler construction with seed length $m + c\log(m/\epsilon)$. Expander graphs have many applications. A simple example is that the general case $t > 2$ is obtained from the $t = 2$ case via recursion.

Recursion

To fool $2t$ sets, first run the generator for 2 sets with error ϵ/c to get two seeds for generators for t sets with error ϵ/c. Then, run twice the generator for t sets on those seeds. Specifically, given $2t$ functions f_i, let $g_1 : [2]^{mt} \to [2]$ be the product of the first t, and g_2 the product of the last t. Let G_t be a generator for the product of t functions. We have:

$$|\mathbb{E}[g_1(U) \cdot g_2(U)] - \mathbb{E}[g_1(G(S_1)) \cdot g_2(G(S_2))]| \leq \epsilon/2.$$

To see this, define the "hybrid" distribution $H = g_1(G(S_1)) \cdot g_2(U)$, and note that the distance of $\mathbb{E}[H]$ from each of the expectations inside the absolute value is $\leq \epsilon/c$, and use the triangle inequality.

Now the key idea is that we can think of g_1 composed with G as another function h_1, and similarly for g_2. We can fool $h_1 \cdot h_2$ with the generator for two sets with error $\epsilon/2$, obtaining a generator for t sets with error $\epsilon/2 + \epsilon/2 = \epsilon$, as desired.

To analyze the seed length, denote it by $s(m, t, \epsilon)$ for parameters m, t, and ϵ. The definition above gives the recursion

$$s(m, 2t, \epsilon) \leq s(s(m, t, \epsilon/c), 2, \epsilon/c) \leq s(m, t, \epsilon/c) + c \log s(m, t, \epsilon/c)/\epsilon \leq s(m, t, \epsilon/c) + c \log(mt/\epsilon).$$

The second inequality is by the base $t = 2$ case, and the next is because seed mt always suffices, trivially. Iterating $\log_2 t$ times, we obtain seed length

$$\leq s(m, 2, \epsilon/t^c) + c \log(t) \log(mt/\epsilon)$$

which is as desired, using again the base case.

Expander graphs

The generator for the base case outputs $(U, U+D)$ where U is uniform and D is a distribution that ϵ-fools linear polynomials over $\mathbb{F}_2$ (!). By Theorem 11.5 the seed length is as desired. To analyze, it is natural to write the functions f_1 and f_2 in terms of polynomials. For slight convenience we think of the inputs in $\{-1, 1\}$ instead of $\{0, 1\}$, so that multiplication of input bits corresponds to xoring. In particular we will write $U \cdot D$ for $U + D$.

Exercise 11.4 (Hypercube analysis)**.** For $\alpha \subseteq [n]$, we write x^α for $\prod_{i \in \alpha} x_i$, with $x^\emptyset := 1$. Let $f : \{-1, 1\}^n \to \mathbb{R}$ be a function.

(1) Show that f can be written as

$$f(x) = \sum_\alpha \hat{f}_\alpha \cdot x^\alpha,$$

for some $\hat{f}_\alpha \in \mathbb{R}$. Guideline: First write $f(x) = \sum_{a \in \{-1,1\}} f(a) I_a(x)$, where $I_a(x) = 1$ if $x = a$ and 0 otherwise.

(2) Show that $\hat{f}_\alpha = \mathbb{E}_x[f(x)x^\alpha]$. In particular and for example, $\hat{f}_\emptyset = \mathbb{E}[f(U)]$.

(3) Show that $\sum_\alpha \hat{f}_\alpha^2 = \mathbb{E}[f^2(U)]$.

Writing $f = f_1$ and $g = f_2$ we need to bound $|\mathbb{E}[f(U)g(U \cdot D)] - \mathbb{E}[f(U)]\mathbb{E}[g(U)]|$. Note that in the first $\mathbb{E}$ the two occurrences of U denote the same sample, whereas in the second they denote independent samples. Using Exercise 11.4, the second summand is $\hat{f}_\emptyset \hat{g}_\emptyset$. The first is

$$\mathbb{E}_{x \leftarrow U, D}[\sum_{\alpha, \beta} \hat{f}_\alpha \hat{g}_\beta x^\alpha (x \cdot D)^\beta].$$

Because $(x \cdot D)^\beta = x^\beta \cdot D^\beta$, the terms with $\alpha \neq \beta$ give 0. So we can rewrite it as

$$\mathbb{E}_D \left[\sum_\alpha \hat{f}_\alpha \hat{g}_\alpha D^\alpha \right].$$

Putting this together, we remove the $\alpha = \emptyset$ term, and our goal is to bound

$$\left| \mathbb{E}_D \left[\sum_{\alpha \neq \emptyset} \hat{f}_\alpha \hat{g}_\alpha D^\alpha \right] \right|.$$

This is at most

$$\begin{aligned}
&\sum_{\alpha \neq \emptyset} |\hat{f}_\alpha| \cdot |\hat{g}_\alpha| \cdot |\mathbb{E}_D[D^\alpha]| \text{ (by the triangle inequality)} \\
\leq &\epsilon \sum_\alpha |\hat{f}_\alpha| \cdot |\hat{g}_\alpha| \text{ (by the assumption on } D) \\
\leq &\epsilon \sqrt{\sum_\alpha \hat{f}_\alpha^2} \cdot \sqrt{\sum_\alpha \hat{g}_\alpha^2} \text{ (by Fact B.11)} \\
= &\epsilon \sqrt{\mathbb{E}[f^2(U)]} \cdot \sqrt{\mathbb{E}[g^2(U)]} \text{ (by Exercise 11.4, Item (3))} \\
\leq &\epsilon \text{ (because the range of f and g is [2]).}
\end{aligned}$$

11.2 PRGs from hard functions

In this section we present a general paradigm to construct PRGs from hard functions. We begin with a general claim showing that PRGs with non-trivial seed length $s(n) = n - 1$ are in fact equivalent to correlation bounds, recall Definition 8.3.

Claim 11.2. Let $f : [2]^n \to [2]$ be a function. We have:

(1) If $C : [2]^{n+1} \to [2]$ ϵ-breaks $G(x) := xf(x)$ then there is $b \in [2]$ s.t. $C'_b : [2]^n \to [2]$ defined as $C'_b(x) := C(xb) \oplus b$ has correlation $\mathbb{E}e[C'_b(x) \oplus f(x)] \geq \epsilon$ with f.

(2) Conversely, suppose $C : [2]^n \to [2]$ has ϵ-correlation with f. Then $C' : [2]^{n+1} \to [2]$ defined as $C'(x, b) := C(x) \oplus b$ $\epsilon/2$-breaks $xf(x)$.

Proof of (1). Pick b uniformly and write

$$\mathbb{E}_{x,b} e[C'_b(x) \oplus f(x)] = \mathbb{E}_{x, b \oplus f(x)} e[C'_{b \oplus f(x)}(x) \oplus f(x)] = \mathbb{E}e[C_{b \oplus f(x)}(x(b \oplus f(x))) \oplus b] = \frac{1}{2} |\mathbb{E}_x C(xf(x)) - \mathbb{E}_x C(x\overline{f}(x$$

So there is b s.t. the LHS is at least the RHS. This establishes the first claim. **QED**

Exercise 11.5. Prove (2) in Claim 11.2.

The contrapositive of (1) is that functions with small correlation immediately imply a 1-bit of stretch generator. Naturally, we'd like to increase the stretch. A natural idea is *repetition:* From a pseudorandom distribution D over $[2]^n$, we construct $D^k := D, D, \ldots, D$ over $[2]^{k \cdot n}$.

Claim 11.3. If f ϵ-distinguishes D^k and E^k then a restriction of f ϵ/k-distinguishes D and E.

Proof. Via the "hybrid method," a.k.a. the triangle inequality, see proof of Theorem 11.7. Define $H_i := D_0 D_1 \cdots D_{i-1} E_i E_{i+1} \cdots E_{k-1}$ over nk bits for $i \in [k]$, where each factor in the RHS is over n bits. Note that H_0 is E^k and H_k is D^k. Write

$$\begin{aligned}
\epsilon &\leq |\mathbb{E}[f(H_0)] - \mathbb{E}[f(H_{k-1})]| \\
&= |\sum_{i \in [k]} \mathbb{E}[f(H_i)] - \mathbb{E}[f(H_{i+1})]| \\
&\leq \sum_{i \in [k]} |\mathbb{E}[f(H_i)] - \mathbb{E}[f(H_{i+1})]|.
\end{aligned}$$

So one of the terms on the RHS is $\geq \epsilon/k$. The corresponding distributions H_i and H_{i+1} differ in only one factor. We can fix all others and the claim follows. **QED**

Note we went from ϵ to ϵ/k. This means the claim is only applicable when ϵ is fairly small. In general, this loss cannot be avoided:

Exercise 11.6. Exhibit a distribution D that is 0.1-pseudorandom (for say CktP) but D^k is not even 0.9 pseudorandom, for suitable k. Now strengthen this to D of the form $xf(x)$, for some boolean function f.

However, repetition works for *resamplable* functions, like parity. These are functions h s.t. given any "correct" pair $(x, h(x))$ we can generate uniform correct pairs $(y, h(y))$, and similarly for "incorrect" pairs $(x, h(x) \oplus 1)$ – using the same distribution.

Definition 11.4. A function $h : [2]^n \to [2]$ is *resampled* by a distribution F on functions from $[2]^{n+1}$ to $[2]^{n+1}$ if for every $x \in [2]^n$ and $b \in [2]$, $F(x, h(x) \oplus b)$ outputs $(y, h(y) \oplus b)$ for uniform $y \in [2]^n$.

Claim 11.4. Suppose $h : [2]^n \to [2]$ is balanced (i.e., $\mathbb{P}[h(U) = 1] = 1/2$) and resampled by F. Let $D = (X, h(X))$. Suppose f ϵ-breaks D^k. Then $f(G, G, \ldots, G)$ $\epsilon/2$-breaks D, where each occurrence of G is either an occurrence of F or a fixed value.

Proof. Because h is balanced, we can sample U_{n+1} by first tossing a coin b, and then outputting $(X, h(X) \oplus b)$. Because f ϵ-breaks D, we can fix coins $b_1, \ldots, b_k$ s.t. the quantities

$$\begin{aligned}
&\mathbb{E}\left[f\left((X_1, h(X_1)), (X_2, h(X_2)), \ldots, (X_k, h(X_k))\right)\right] \\
&\mathbb{E}\left[f\left((X_1, h(X_1) \oplus b_1), (X_2, h(X_2) \oplus b_2), \ldots, (X_k, h(X_k) \oplus b_k)\right)\right]
\end{aligned}$$

have distance $\geq \epsilon$.

The coordinates where $b_i = 0$ are the same. So we can fix those and obtain a restriction f' of f s.t. for some $j \leq k$ the quantities

$$\mathbb{E}\left[f'\left((X_1, h(X_1)), (X_2, h(X_2)), \ldots, (X_j, h(X_j))\right)\right] =: (I)$$
$$\mathbb{E}\left[f'\left((X_1, \overline{h}(X_1)), (X_2, \overline{h}(X_2)), \ldots, (X_j, \overline{h}(X_j))\right)\right] =: (II)$$

have distance $\geq \epsilon$.

Now we use this to break D. As in the proof of Claim 11.2 it suffices to tell D from $\overline{D} := (X, \overline{h}(X))$. On input $z \in [2]^{n+1}$, we compute

$$f'(F(z), F(z), \ldots, F(z)).$$

If $z = D$ then this is the same as (I), and if $z = \overline{D}$ this is the same as (II). **QED**

Claim 11.5. Parity is resamplable in AC.

Exercise 11.7. Prove this.

Combining the results in this section with the correlation of ACs and parity – Corollary 9.2 – we obtain a PRG with seed length $n - n/\log^{c_d} n$ that fools ACs of size n^d and depth d on n bits. In the next section, leveraging the exponentially-small correlation bounds between ACs and parity, we will obtain a much shorter, logarithmic seed length for ACs.

However, for other classes of circuits like AC[2] such strong correlation bounds are not known. For these classes, the results in this section give the best-known explicit generator. For example, for AC[3] we can again use that parity has correlation $\leq 1/100$ with such circuits, as follows from Problem 9.1, and obtain a generator stretching $n - n/\log^{c_d} n$ bits to n. For AC[2] one can work with a different function and again obtain that stretch. Better stretch is not known.

Question 11.2. *Give an explicit generator with seed length* $0.9n$ *for AC*$[2]$ *circuits of size* n^c *and depth* c *on* n *bits.*

11.2.1 From correlation bounds to stretch: Sets with bounded intersections

The repetition PRG outputs values of a hard function h on independent inputs. We now study a powerful technique which instead outputs values from *dependent* inputs. This gives a better trade-off between seed and output length. It is a derandomized analogue of Claim 11.3. Rather than picking independent inputs as in the repetition generator, we select them based on a collection of subsets of $[u]$, where u is the seed length.

Definition 11.5. Let $S = T_1, T_2, \ldots, T_S$ be collection of subsets of $[u]$ of size ℓ. Then the *bounded-intersection generator*

$$\text{BIG}_S : [2]^u \to \left([2]^\ell\right)^S$$

is defined as $\mathrm{BIG}_S(x) := x_{T_1}, x_{T_2}, \ldots, x_{T_S}$. Here x_{T_j} are the ℓ bits of x indexed by T_j.

For a distribution H on functions from $[2]^\ell \to [2]$ and a generator $G : [2]^u \to \left([2]^\ell\right)^S$ we write $H \circ G(\sigma)$ for the result $H(x_1), H(x_2), \ldots, H(x_S)$ of applying H to the outputs of G, where $G(\sigma) = (x_1, x_2, \ldots, x_S)$ and the occurrences of H denote independent samples.

For example, if H is a uniform function then $H \circ \mathrm{BIG}_S$ is uniform over $[2]^S$. The next key result shows that BIG *preserves indistinguishability*, similar to the repetition generator, as long as the sets in S have small intersections. The intersection size governs the locality (recall Definition **??**), and hence the complexity, of the reduction.

Theorem 11.8 (BIG PI)**.** Let BIG and S be as in Definition 11.5. Furthermore, suppose $|T_i \cap T_j| \leq w$ for any $i \neq j$. Let V and W be two distributions on functions from $[2]^\ell$ to $[2]$.

Suppose f ϵ-tells the distributions $(\sigma, V \circ \mathrm{BIG}_S(\sigma))$ and $(\sigma, W \circ \mathrm{BIG}_S(\sigma))$, over $u + |S|$ bits.

Then there are w-local functions g_i s.t. $f(g_1, g_2, \ldots, g_{u+|S|})$ distinguishes $(X, V(X))$ from $(X, W(X))$ with advantage $\geq \epsilon/|S|$, where X is uniform in $[2]^\ell$.

Exercise 11.8. Derive Claim 11.3 from Theorem 11.8 for the special case $D = (X, V(X))$ and $E = (X, W(X))$.

Proof. Write $D = (\sigma, V \circ G_S(\sigma)) = D_0 D_1 \cdots D_{|\sigma|+S-1}$ and $E = (\sigma, W \circ G_S(\sigma)) = E_0 E_1 \cdots$. As in the proof of Claim 11.3, define hybrids

$$H_i := D_0 D_1 \cdots D_{i-1} E_i E_{i+1} \cdots E_{|\sigma|+S-1}$$

over $|\sigma| + S$ bits. Note that H_0 is E and $H_{|S|}$ is D. So there is $i \in [S]$ s.t. f distinguishes two adjacent hybrids $H_{|\sigma|+i}$ and $H_{|\sigma|+i+1}$ with advantage $\geq \epsilon/S$. (The first $|\sigma|$ bits in D and E are equal.)

We can fix the $u - \ell$ bits in the seed σ that are not in set T_i. Now every bit in position $j \neq i$ depends on $\leq w$ bits in T_i, and so can be computed by a distribution G_j on w-local functions.

The following distribution on circuits tells $(X, V(X))$ from $(X, W(X))$, again with advantage $\geq \epsilon/|S|$: On input (x, b) run f on

$$(G_0(x), G_1(x), \ldots, G_{i-1}(x), b, G_{i+1}(x), G_{i+2}(x), \ldots, G_{|\sigma|+|S|-1}(x)).$$

We can fix the G_i to g_i and maintain the advantage. **QED**

To apply Theorem 11.8 we need a collection S with small intersections. We'd like to have as many sets as possible (that's the output length of the generator) which are as large as possible (that's the input length to the hard function) which are subsets of as small a set as possible (that's the seed length) and such that any two have as small intersection as possible (that's the overhead in the reduction). The probabilistic method shows that collections with great parameters exist. The following is a simple explicit construction.

Lemma 11.1. [Sets with small intersections] There are explicit collections of q^d subsets of $[q^2]$ of size q such that any two sets intersect in $\leq d$ elements, for any q that is a power of 2 and $> d$.

Proof. View the universe $[u] = [q^2]$ as $\mathbb{F}_q^2$. For a parameter d, the sets correspond to the graphs of polynomials p of degree $< d$. (I.e., the set $\{(x, p(x)) : x \in \mathbb{F}\}$.) The number of sets is q^d. The size of each set is $q = \sqrt{u}$. To bound the intersection of two sets, consider the corresponding polynomials and let p be their difference, which is not zero. Any element in the intersection of the sets corresponds to a zero of p. By Fact B.28, the intersection has size $\leq d$. **QED**

To illustrate parameters, we can have $m = q^d$ subsets of size $\ell = \sqrt{q}$ from a universe of size $u = \ell^2 = q$ with intersections at most d. For example, given m we can set $d = \log m$ and $q = \log^a m$, and the intersection size is only $\ell^{1/a}$ while the universe is only quadratic in the set size, i.e., $u = \ell^2$.

Corollary 11.1. There are generators $G : [2]^{\log^{c_d} n} \to [2]^n$ that $1/n$-fool ACs of size n and depth d.

As in Exercise 11.1, in this corollary, to eliminate one parameter we set the size of the circuit equal to the output length of G. The same statement holds if the size is n^d instead of n.

Exercise 11.9. Prove Corollary 11.1. Explain how the parameters are set and which results you are combining.

The seed length in Corollary 11.1 is about the best we can do given current impossibility results, and recall once again from section §6.3 that stronger impossibility would imply major separations.

Still, one can ask if PRGs *could* be built *if* we had such stronger results. In particular, one would like to have seed length say $c \log n$ instead of $\log^c n$. This is the setting that allows for conclusions such as P = BPP, see Exercise 11.1. Lemma 11.1 doesn't give this, since the universe is always at least quadratic in the set size, but the following construction does.

Lemma 11.2. [Sets with small intersections, II] For any a and $n \geq c_a$ there is an explicit collection of n subsets of $[c_a \log n]$ of size $c_a \log n$ with pairwise intersection $\leq a \log n$.

Using this we can prove the following weaker version of Theorem 3.5.

Corollary 11.2. Suppose there is $\epsilon > 0$ and $f \in \mathrm{E}$ that on inputs of length n has correlation at most $2^{-\epsilon n}$ with circuits of size $2^{\epsilon n}$. Then P = BPP.

Exercise 11.10. Prove Corollary 11.2 assuming Lemma 11.2.

11.2.2 Turning hardness into correlation bounds

> We remark that none of the known hardness amplification results can be applied to the computational models for which we actually can establish the existence of hard functions (i.e. prove lower bounds).

We can't expect to prove that correlation bounds under uniform are equivalent to impossibility or hardness results, as one can construct pathological functions which are easy to compute on, say, .75 fraction of the inputs, but impossible to compute on a .76 fraction. So instead our approach will be to *construct* functions which have small correlation under the uniform distribution.

A natural candidate for such a function, starting from a "mildly hard" function $f : [2]^n \to [2]$ is $f' : [2]^{nk} \to [2]$ defined as

$$f'(x_1, \ldots, x_k) := \oplus_{i=1}^k f(x_i).$$

An *XOR Lemma* is a statement showing that if f has correlation $\leq \epsilon$ with a certain computational model (e.g., CktP), then the correlation of f' with a related model decays *exponentially fast* with the number k of copies (see Definition 8.3 of correlation).

There is a strong *informational* (as opposed to computational) intuition why the XOR Lemma should work. To illustrate, consider the "computational model" of constant functions 0 or 1. The claim that f has correlation at most ϵ with this model then simply means that for every constant function $g(x) = 0$ or $g(x) = 1$ we have

$$|\mathbb{E}e f(x) + g(x)| = |\mathbb{E}e f(x)| \leq \epsilon.$$

And indeed in this case the correlation decays exponentially fast:

$$|\mathbb{E}e f'(x') + g(x')| = |\mathbb{E}e f'(x')| = |\mathbb{E}e f(x)|^k \leq \epsilon^k.$$

More generally, consider that if f has correlation $\leq \epsilon$ with small circuits C, then f' indeed has correlation $\leq \epsilon^k$ with small circuits *of the special product form* $C(x_1, \ldots, x_k) := \oplus_{i=1}^k C_i(x_i)$. This is again because

$$|\mathbb{E}e f'(x') + C(x_1, \ldots, x_k)| = \left|\mathbb{E}e \oplus_{i=1}^k (f(x_i) \oplus C(x_i))\right| = |\mathbb{E}e f(x) \oplus C(x)|^k \leq \epsilon^k.$$

This generalizes the example of constant functions since they are trivially in the special product form.

Intuitively, no circuit can do better than a circuit in the special form, and the XOR Lemma is true. But is the intuition true?

Exercise 11.11. Consider circuits C made of a single majority gate. Prove that the XOR lemma is false for C. Feel free to pick n even and define the value of Majority on inputs of weight $n/2$ to be 1, and recall $\binom{n}{n/2} \cdot \frac{\sqrt{n}}{2^n} \in [c, c]$.

One can extend this result to AC with a small number of majority gates.

Question 11.3. *Does the XOR lemma hold for AC with parity gates, or even constant-degree polynomials over $\mathbb{F}_2$?*

But for more powerful models, we can indeed prove the xor lemma, and the proof follows the information-theoretic intuition above. To connect to this intuition, we consider functions which may output a uniform bit on some inputs.

Definition 11.6. We say that a distribution on functions $F : [2]^n \to [2]$ is *δ-random* if there exists a subset $H \subseteq [2]^n$ with $|H| = 2\delta 2^n$ such that $F(x) = U_1$ (i.e. a coin flip) for $x \in H$ and $F(x)$ is deterministic (i.e., a fixed value) for $x \notin H$.

Thus, a δ-random function has a set of relative size 2δ on which it is information-theoretically unpredictable. To illustrate the XOR lemma, suppose that f is δ-random. Then f' will be almost a coin flip. Specifically, the probability that the output is not a coin flip is $(1-2\delta)^k$, the probability that no input falls into H. When some input falls into H, the output is a coin flip, and no circuit, efficient or not, can have non-zero correlation.

This intuition can be formalized via the *hardcore-set* lemma, which allows us to pass from computational hardness to information-theoretic hardness. Before stating the lemma we emphasize an important point:

> *The hardcore-set lemma is only known to hold for computational models which can compute majority.* This is because the proof of correctness uses majority, as will be apparent in section §11.3. So to apply it, we have to start from an impossibility result for circuits that can compute majority. As discussed in Chapter 7, we essentially have no such result. In fact, in some restricted models, the xor lemma is false (see Exercise 11.11). So the results in this section are mostly conditional. Still, they allow us to spin a fascinating web of reductions between correlation and randomness, pointing to several challenges.

The following hardcore set lemma says that any function that has somewhat small correlation with small circuits admits a somewhat large hardcore set on which the function has very small correlation with small circuits. To illustrate parameters, note that a δ-random function has correlation $\leq 1-2\delta$ with any fixed function (or circuit) (we can extend Definition 8.3 to random functions by taking expectation over both the input and the random function). This is because when the input falls in the set H of density 2δ from Definition 11.6 then the correlation is zero. The hard-core set lemma shows that this is the only way that small correlation may arise: any function with small correlation with small circuits is in fact close to a δ-random function. We state the result in terms of distinguishing input-output pairs, as opposed to computing the function. This is equivalent by an argument similar to Claim 11.2 but is more convenient as it immediately allows us to talk about multiple inputs, as we also do in the next statement. Here we use $\cdot$ to denote concatenation.

Lemma 11.3 (Hardcore Set)**.** Let $f : [2]^n \to [2]$ have correlation $\leq 1 - 2\delta$ with circuits of size s. Then there exists a $c\delta$-random function $g : [2]^n \to [2]$ such that $X \cdot f(X)$ and $X \cdot g(X)$ are ϵ-indistinguishable by circuits of size $cs\epsilon^2\delta^2$, for any ϵ, where $X \equiv U_n$.

In particular, by Claim 11.3,

$$X_1 \cdots X_k \cdot f(X_1) \cdots f(X_k) \text{ and } X_1 \cdots X_k \cdot g(X_1) \cdots g(X_k)$$

are $k\epsilon$-indistinguishable for size $cs\epsilon^2\delta^2$, where the X_i's are uniform and independent.

We can now easily formalize the proof of the xor lemma.

Lemma 11.4. Suppose $f : [2]^n \to [2]$ has correlation $\leq (1-2\delta)$ with circuits of size s. Then $f' : [2]^{nk} \to [2]$ defined as $f'(x_1, \ldots, x_k) := \oplus_{i=1}^k f(x_i)$ has correlation $\leq (1-c\delta)^k + k/s^c$ with circuits of size $\delta^c s^c$.

For example, if $s = 2^{n^c}$ and $\delta = c$, we can take $k = cn$ and have hardness 2^{-cn}. However the function is on cn^c bits, so in terms of the input length n', f' has hardness $2^{-n'^c}$.

Proof. We use Lemma 11.3 with $\epsilon := 1/s^c$. From its conclusion it follows that

$$X_1 \cdots X_k \cdot \oplus_i f(X_i) \text{ and } X_1 \cdots X_k \cdot \oplus_i g(X_i)$$

are k/s^c-indistinguishable for size $\delta^c s^c$. Following the intuition above, the right-hand distribution is $(1-c\delta)^k$ close to $X_1 \cdots X_k \cdot U_1$. Hence the left-hand distribution is $((1-c\delta)^k + k/s^c)$-indistinguishable from $X_1 \cdots X_k \cdot U_1$ and the result follows from Claim 11.2. **QED**

11.2.3 Derandomizing the XOR lemma

A drawback of the xor lemma is that the input length of the new function is $\geq kn$. This prevents us from obtaining correlation 2^{-cn} (as opposed to $2^{-c\sqrt{n}}$) which is important for the flagship conclusion P = BPP, see Corollary 11.2. To remedy this we shall use... PRGs! Rather than independently, we will pick the k inputs to f' using a generator. We need two properties from this PRG. First, to behave like repetition, we need BIG (Theorem 11.8). Also, we need to "hit" the hard-core set, for which we need HIT. We can get both properties by xor-ing the generators together. The generator is defined as

$$\text{BIG-HIT}(\sigma_1, \sigma_2) := \text{BIG}_S(\sigma_1) \oplus \text{HIT}(\sigma_2),$$

where HIT is a hitter, given next.

Lemma 11.5. For every ϵ and δ there exists an explicit generator $\text{HIT} : [2]^{2n} \to ([2]^n)^s$ with $s = 1/\epsilon\delta$ s.t. for every set $H \subseteq [2]^n$ of size ϵ, $\mathbb{P}_\sigma[\text{HIT}(\sigma)_i \notin H \text{ for every } i] \leq \delta$.

Proof. Pairwise independence. Consider the field $\mathbb{F}_{2^n}$. The seed σ specifies $a, b \in \mathbb{F}$ and we output $b, a + b, 2a + b, \ldots$. Let X_i be the indicator variable of $\text{HIT}(\sigma)_i \in H$. The X_i are pairwise independent. Their expectation is ϵs. Hence the probability to bound is $\leq \mathbb{P}[|\sum X_i - \epsilon s| \geq \epsilon s]$. Squaring both sides of the inequalities and doing calculations gives the result. **QED**

Exercise 11.12. Do the calculations.

Using this, we can boost correlation $(1-2^{-cn})$ to correlation $\leq 2^{-cn}$. We give an example for an interesting setting of parameters.

Lemma 11.6. Suppose E has a function $f : [2]^* \to [2]$ that on inputs of length n has correlation $(1-2^{-cn})$ with circuits of size 2^{cn}. Then E has a function $f' : [2]^* \to [2]$ that has correlation $\leq 2^{-cn}$ with circuits of size 2^{cn}.

Note the conclusion implies P = BPP by Corollary 11.2.

Proof. Let $\epsilon := 2^{-cn}$ and $\delta := 2^{-cn}$. Define $f' : [2]^{cn} \to [2]$ as $f'(\sigma) := \oplus_{i=1}^{s} f(x_i)$ where BIG-HIT$(\sigma) = (x_1, x_2, \ldots, x_s)$, where $s = 1/\delta\epsilon$ and the set system for BIG is from Lemma 11.2.

We use Lemma 11.3 with $\epsilon := s^c$. Let g the corresponding δ-random function. From Theorem 11.8 it follows that

$$\sigma, f \circ G \text{ and } \sigma, g \circ G$$

are ϵ^c-indistinguishable for size $1/\epsilon^c$. In particular this holds if we take parities, so

$$\sigma, \oplus_{i=1}^{k} f(X_1) \text{ and } \sigma, \oplus_{i=1}^{k} g(X_1)$$

are no more distinguishable, where $(X_1, \ldots, X_k) = G(\sigma)$. By the hitting property of HIT, Lemma 11.5, the chance of not hitting the hardcore set is $\leq \delta$, and we conclude as in the proof of Lemma 11.4. **QED**

Exercise 11.13. Gotcha!? We can't quite use Theorem 11.8 and Lemma 11.5 as stated. Explain why and how to modify the statements and proofs of Theorem 11.8 and Lemma 11.5 so that the above proof of Lemma 11.6 does work.

11.2.4 Encoding the whole truth-table

The results in the previous section give us functions with small correlation starting from functions on n bits with with correlation $(1-2^{-cn})$, but not quite from any impossibility results, which only gives correlation $\leq (1-2/2^{-n}) < 1$.

Exercise 11.14. Explain where the previous proofs break down.

To start from worst-case hardness we need to encode the entire truth table of the function. We give a simple code that suffices for our results.

Theorem 11.9. Suppose there is $f \in$ E that on inputs of length n cannot be computed by circuits of size $s(n)$. Then there is $f' \in$ E that has correlation $(1-1/n^c)$ with circuits of size $n^c s(cn)$.

Recall in Theorem 8.4 we saw an equivalence between computing and correlating under every distribution. Had we had that result for the *uniform* distribution we could have skipped all the amplification results, including Theorem 11.9 and constructed PRGs much more directly. However in general we can't guarantee that. In fact, one can construct functions that are very easy over the uniform distribution, say because they are almost always one, but still are hard to compute, say because there is a small set of inputs that makes the function hard.

Proof. Thinks of an n-bit input to f as ℓ variables of n/ℓ bits; so each variable is over a set H of size $2^{n/\ell}$. We can write down f as a polynomial p_f of degree $(H-1)\ell$ over any field that includes H. This p_f is done as in section 10.4.2, using Exercise 10.4. That is:

$$f(x) = \sum_{a \in H^\ell} f(a) \cdot \mathrm{EQ}(x, a).$$

Now the gain is that we can think of evaluating p_f over a larger fields. Set $q := n^{10}$ and $d := n^5$ and $\ell := n/\log d$. The new function f' is constructed in two steps. First, we consider inputs over $\mathbb{F}_q^\ell$. Note the length of such inputs is $\leq c\ell \log q \leq cn$ bits, as desired. This gives a non-boolean function. To make the function boolean, we output bit i of p_f, where i is part of the new input. That is,

$$f'(x_1, \ldots, x_\ell, i) := p_f(x_1, \ldots, x_\ell)_i$$

where $x_i \in \mathbb{F}_q$ and $i \in [\log q]$.

We'd like to show that if there's a small circuit C computing f' on a $(1 - 1/n^c)$ fraction of inputs then there's another small circuit computing f everywhere. Let $C(x) := C(x, 1) \cdots C(x, \log q)$. First note that the fraction α of $x \in \mathbb{F}_q^\ell$ such that $C(x) \neq p_f(x)$ is $\leq 1/n^c \leq c/d\ell$. Because if it's larger, every such x contributes at least one input (x, i) where C disagrees with f', contradicting the assumption.

Using C we give a distribution C' on circuits which computes p_f w.h.p. on every given input y. Pick a uniform line going through y, and run C on this line for $d\ell$ points. That is, pick uniform $s \in \mathbb{F}_q^\ell$ and run $C(y + 1s), C(y + 2s), \ldots, C(y + d\ell s)$.

Because each evaluation point is uniform, and $d\ell\alpha \leq c$, with prob. $> 1/2$ the evaluations of C will be correct, and equal $p_f(y + 1s), p_f(y + 2s), \ldots, p_f(y + d\ell s)$.

Note that for fixed y and s, $p_f(y + ts)$ is a univariate polynomial q in t of degree $\leq d\ell$. We can compute the coefficients of q from its evaluations at $d\ell$ points. (It's a linear system, with a unique solution by Fact B.28 because the degree of q is $\leq \ell \cdot d < q$.)

We can then output $q(0) = p_f(y)$.

Finally, we can repeat this cn times and output the most likely value. On every input x this errs w.p. $< 2^{-n}$. Hence we can fix the random choices and obtain a fixed circuit that succeeds on every x. **QED**

Exercise 11.15. "Put it all together" and prove Theorem 3.5.

11.2.5 Monotone amplification within NP

To increase the hardness of functions in NP we cannot use XOR since NP is not known to be closed under complement. We will use a combination of many things in this chapter – including the (unconditional) generator for AC in Corollary 11.1 – to establish the following.

Theorem 11.10. If NP has a balanced function that has correlation $\leq 1/10$ with circuits of size 2^{n^c}, then NP also has a balanced function with correlation $\leq 2^{-n^c}$ with circuits of size 2^{n^c}.

Several optimizations have been devised, see the Notes. Still, we don't enjoy the same range as for E:

Question 11.4. *Prove Lemma 11.6 for NP instead of E, even starting from hardness $\delta \geq c$.*

11.2.6 Proof of Theorem 11.10

Rather than XOR, to amplify we use the Tribes function, a monotone read-once DNF.

Definition 11.7. The Tribes function on k bits is:

$$\text{Tribes}(x_1, \ldots, x_k) := (x_1 \wedge \ldots \wedge x_b) \vee (x_{b+1} \wedge \ldots \wedge x_{2b}) \vee \ldots \vee (x_{k-b+1} \wedge \ldots \wedge x_k)$$

where there are k/b clauses each of size b, and b is the largest integer such that $(1-2^{-b})^{k/b} \geq 1/2$. Note that this makes $b \leq c \log k$.

The property of xor that we used is that if one bit is uniform, then the output is uniform. We use an analogous property for tribes, that if several bits are uniform, then the output is close to uniform.

Lemma 11.7. Let N_p be a noise vector where each is 1 independently with probability p. Then $\mathbb{E}_{x,N_p} e[\text{Tribes}(x) \oplus \text{Tribes}(x \oplus N_p)] \leq 1/k^{c_p}$.

We shall take k exponentially large. The resulting function is still in NP as we can use non-determinism to pick a clause. We use the generator BIG-AC$(\sigma) = (x_1, x_2, \ldots, x_s)$, which is like BIG-HIT except that HIT is replaced with the generator in Corollary 11.1, for circuits of size $(k2^n)^c$. Note its seed length is n^c for $k \leq 2^{n^c}$.

Define $f' : [2]^{2n} \to [2]$ as $f'(\sigma) :=$ Tribes $\circ (f(x_1), \ldots, f(x_s))$ where BIG-AC$(\sigma) = (x_1, x_2, \ldots, x_s)$, and the set system for BIG is from Theorem 16.7. Following the proof of Lemma 11.6, use Lemma 11.3 with $\epsilon := s^c$. Let g the corresponding δ-random function. From Theorem 11.8 it follows that

$$\sigma, f \circ \text{BIG-AC} \text{ and } \sigma, g \circ \text{BIG-AC}$$

are ϵ^c-indistinguishable for size $1/\epsilon^c$. In particular this holds if we take Tribes of the output, i.e.What remains to show is that

$$\sigma, \text{Tribes} \circ g \circ \text{BIG-AC}$$

is close to uniform. That is, we have to show that with high probability over σ, just over the choice of g, the value Tribes$\circ g \circ$ BIG-AC is close to a uniform bit.

It suffices to bound

$$\mathbb{E}_\sigma |\mathbb{E}_g e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma)]|.$$

Here the inner expectation is over the random choices in all the s evaluations of g. Up to a power, this is

$$\leq \mathbb{E}_\sigma \mathbb{E}_g^2 e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma)] = \mathbb{E}_{\sigma,g,g'} e[\text{Tribes}\circ g \circ \text{BIG-AC}(\sigma) \oplus \text{Tribes}\circ g' \circ \text{BIG-AC}(\sigma)].$$

Now the critical step is that Tribes $\circ\, g$ is computable by a distribution on AC of size $(s2^n)^c$ and depth c. Note that the circuit computes g in brute-force, but the dependence on s is good. Because BIG-AC fools such circuits with error 2^{-n^c}, the latter expectation equals

$$\mathbb{E}_{(x_1,\ldots,x_s),g,g'} e[\text{Tribes}\circ g\circ(x_1,\ldots,x_s)\oplus\text{Tribes}\circ g'\circ(x_1,\ldots,x_s)] = \mathbb{E}_{x,N_p} e[\text{Tribes}(x)\oplus\text{Tribes}(x\oplus N_p)],$$

for $p = c$. We conclude by Lemma 11.7.

11.3 Proof of the hardcore-set Lemma 11.3

At the high-level, this is just min-max and concentration of measure, just like the equivalence between computation and correlation in Theorem 8.4. However, the proof is slightly more involved than one might anticipate. We break it up in two claims. First we obtain hardness w.r.t. a "smooth" distribution D: $D(x) \leq d/N$ for every x. For example, D could be "flat," i.e. uniform over a set of size N/d (then $D(x)$ is either 0 or d/N). In the proof, D will be a combination of such flat distributions. Second we obtain a set from a smooth distribution. The straightforward combination of the claims yields the lemma. The parameter d corresponds to $1/\delta$ where δ is the hardness parameter in Definition 8.3; note that $d \geq 2$ so $\log d \geq 1$.

Claim 11.6. Suppose $f : [2]^n \to [2]$ is $1/d$-hard for circuits of size s (see Definition 8.3). Then there is a distribution D on $[2]^n$ s.t. $D(x) \leq d/N$ for every x, and every circuit C of size $s' := s \cdot (\epsilon/\log d)^c$ has $\mathbb{E}_{x\leftarrow D} e[C(x) + f(x)] \leq \epsilon$.

Proof. We use the duality Theorem 8.5 where one set consists of sets S of N/d inputs, and the other consists of circuits C of size $\leq s'$, and $p(S, C) := \mathbb{E}_{x\in S} e[C(x) + f(x)]$. (In the proof of Theorem 8.4 p was just a "single point," here it's an average.)

Suppose there is a distribution over sets S of size N/d such that for every circuit we have $\mathbb{E}_S[p(S, C)] = \mathbb{E}_S \mathbb{E}_{x\in S} e[C(x) + f(x)] \leq \epsilon$. Let D be the induced distribution over inputs x, and note that $D(y) \leq d/N$ for every y, and we're done.

Otherwise by the min-max Theorem 8.5 there is a distribution C on circuits s.t. for any set T of size N/d we have

$$\mathbb{E}_C p(T, C) \geq \epsilon. \tag{11.1}$$

Call an input x *hard* if C does not do well on it: $\mathbb{E}_C e[C(x) + f(x)] \leq \epsilon/2$. Now, there can't be N/d hard inputs, for else we contradict equation (11.1) for a set T of N/d hard inputs. In fact, we can't even have $(1 - \epsilon/2)N/d$ hard inputs. Otherwise the set T consisting of the N/d elements where $\mathbb{E}_C e[C(x) + f(x)]$ is smallest would have only $(N/d)\epsilon/2$ easy inputs, yielding $\mathbb{E}_C p(T, C) < \epsilon/2 + \epsilon/2 = \epsilon$, again contradicting equation (11.1).

We conclude by observing that for every easy x, picking $\log(d)/\epsilon^c$ samples from C and taking majority gives error prob. $\leq \epsilon/2d$, using Lemma B.1 as in the proof of Theorem 3.1. The prob. of not computing correctly a uniform $x \in [N]$ is then at most the prob. that x is hard plus the prob. that the samples of C give the wrong value: $(1 - \epsilon/2)/d + \epsilon/2d = 1/d$. This contradicts the hardness of f. **QED**

When using the following claim for a hard function h, we can let F be the set of functions of the type $e(h(x)+C(x))$ where C is a small circuit. In this way $|\mathbb{E}|_D[f(D)]|$ is the correlation of h and C w.r.t. D.

Claim 11.7. Let D be a distribution over $[N]$ s.t. $D(x) \leq d/N$. Let F be a set of $\leq ce^{c\epsilon^2 N/d^2}$ functions $f : [N] \to \{-1, 1\}$. Suppose for every $f \in F$ we have $\mathbb{E}_D[f(D)] \leq \epsilon$.

Then there is a set $S \subseteq [N]$ of size $|S| \geq cN/d$ s.t. for every $f \in F$ we have $\mathbb{E}_{x \in S}[f(x)] \leq c\epsilon$.

Even this second step is not immediate, due to the fact that the set S is constructed probabilistically and so its size – which is the normalization in the correlation – is not fixed. So we'll first prove concentration around a quantity related to D only, then connect it to $|S|$.

Proof. Construct S by placing each $x \in [N]$ in S independently with prob. $D(x)N/d \in [0, 1]$. Consider $X := \sum_{x \in [N]} S(x)f(x)$, where S is the indicator of set S. The variables $S(x)f(x)$ are independent and have range $[-1, 1]$. Also, $\mathbb{E}[X] = (N/d)\mathbb{E}_D[f(x)]$, and so $|\mathbb{E}[X]| \leq c\epsilon N/d$. By tail bounds, Exercise B.3:

$$\mathbb{P}_S \left[\left| \sum_{x \in [N]} S(x)f(x) \right| \geq c\epsilon N/d \right] \leq 2e^{-c\epsilon^2 N/d^2}.$$

Also, $\mathbb{E}[\sum_{x \in [N]} S(x)] = N/d]$. And so again by tail bounds the probability that $|S| \leq cN/d$ is, say, $\leq e^{-cN/d^2}$.

By a union bound, there exists S of size $\geq cN/d$ s.t. for every $f \in F$ we have

$$\left| \sum_{x \in [N]} S(x)f(x) \right| \leq c\epsilon N/d.$$

Now it's the moment to connect to $|S|$. Dividing both sides by $|S|$ we have

$$|\mathbb{E}_{x \in S}[f(x)]| \leq c\epsilon(N/d)/|S| \leq c\epsilon,$$

as desired. **QED**

11.4 PH is a random low-degree polynomial

In this section we use small-bias generators (Theorem 11.5) to prove a uniform, "scaled up version" of Theorem 9.2. First, let us define the relevant class. We will only be concerned with power times so we can afford a definition that is a bit simpler than those we saw in Chapter 5.

Definition 11.8. We denote by BP $\oplus$ P the class of functions $f : X \subseteq [2]^* \to [2]$ for which there is $g \in$ P a constant d such that for every $x \in X$:

- $f(x) = 1 \Rightarrow \mathbb{P}_{y_1 \in [2]^{n^d}} \left[\oplus y_2 \in [2]^{n^d} : g(x, y_1, y_2) = 1 \right] \geq 2/3$, and
- $f(x) = 0 \Rightarrow \mathbb{P}_{y_1 \in [2]^{n^d}} \left[\oplus y_2 \in [2]^{n^d} : g(x, y_1, y_2) = 1 \right] \leq 1/3$.

Theorem 11.11. PH $\subseteq$ BP $\oplus$ P.

This is saying that any constant number of $\exists$ and $\forall$ quantifier can be replaced by a BP quantifier followed by a $\oplus$ quantifier.

Let's see what Theorem 11.11 has to do with the title of this section. Where is the polynomial? The polynomial corresponding to a PH computation will have an exponential number of terms, so we can't write it down. The big sum over all its monomials corresponds to the $\oplus$ in Theorem 11.11. The polynomial will be sufficiently explicit: we will be able to compute each of its monomials in P. Finally, there won't be just one polynomial, but we will have a distribution on polynomials, and that's the BP part.

As in Lemma 9.7, let us first identify how explicit the polynomial needs to be to yield Theorem 11.11. We need to be able to compute monomials efficiently given its index.

Lemma 11.8. Suppose that for every $d \in \mathbb{N}$ we have:

Let C be the alternating formula on 2^{dn^d} bits which has depth d and each gate has fan-in 2^{n^d}. Suppose there is a distribution P on polynomials as in Theorem 9.2 but that moreover:

(1) Can be sampled from n^{c_d} random bits r,

(2) Given r and an index i of n^{c_d} bits we can compute the (coefficient of) monomial i of P_r in time n^{c_d}.

Then Theorem 11.11 follows.

Proof. Let $L \in \Sigma_d$P. Consider the corresponding alternating circuit C. Similarly to section §9.4.1, the input consists of the bits

$$M(x, y_1, y_2, \ldots, y_d)$$

over all values of the quantified variables y_i. We use the BP quantifier to select r, and then the $\oplus$ quantifier to pick a monomial. This monomial will correspond to n^{c_d} bits as above. We evaluate each of them and return the result. **QED**

There remains to construct explicit polynomials. Again, this is similar to the way we proceeded in section §9.4.1: After a non-explicit construction (Lemma 9.6) we then obtained an explicit construction (Lemma 9.8). Though note here we still aim for a distribution.

Let us go back to the simplest case of Or. Recall that the basic building block in the proof of Lemma 9.1 was the construction of a distribution p_A on linear polynomials which are zero on the all-zero input (which just means that they do not have constant terms), and are often non-zero on any non-zero input. This is equivalent to constructing pseudorandom generators for degree-1 polynomials over $\mathbb{F}_2$, so we can use Theorem 11.5. Inspection of the proof reveals that the monomials are sufficiently explicit. With this in hand, we can now reduce the error in the same way we reduced it in the proof of Lemma 9.1.

Lemma 11.9. Given n, a seed $r \in [2]^{c \log(1/\epsilon) \log n}$, and $m \leq n^{c \log 1/\epsilon}$ we can compute in P a monomial $X_{r,m}$ of degree $c \log 1/\epsilon$ such that the distribution

$$\sum_m X_{R,m}$$

for uniform R computes Or on n variables with error ϵ.

Proof. We use the construction

$$p_{A_1,A_2,\ldots,A_t}(x) := p_{A_1}(x) \vee p_{A_2}(x) \vee \cdots \vee p_{A_t}(x)$$

from the proof of Lemma 9.1, except that each A_i is now generated via Lemma 9.1, and that $t = c \log 1/\epsilon$ (as opposed to $t = \log 1/\epsilon$ before). The bound on the degree is the same as before, as is the proof that it computes Or: The error will be $(1/3)^{c \log 1/\epsilon} \leq \epsilon$.

There remains to show that the monomials can be computed in P. For this we can go back to the polynomial for Or in Example 9.2. Plugging that gives

$$p_{A_1,A_2,\ldots,A_t}(x) = \sum_{a \in [2]^t : a \neq 0} \prod_{i \leq t} (p_{A_i}(x) + a_i + 1).$$

We can use m to index a choice of a and then a choice for a monomial in each of the t linear factors $p_{A_i}(x) + a_i + 1$. For each factor, the explicitness of the monomials in the small-bias generator (Theorem 11.5) allows us to conclude the proof. **QED**

We can now compose these polynomials at each gate to obtain an explicit version of Theorem 9.2 which suffices to prove Lemma 11.8 and hence Theorem 11.11.

Claim 11.8. The assumption of Lemma 11.8 is true.

Proof. Replace each gate with the distribution on polynomials given by Lemma 11.9. (Lemma 11.9 only covers Or gates, but And gates are similar, see Exercise **??**.) The desired polynomial is obtained by composing all these polynomials.

Note each of these polynomials is on $2^{n^{c_d}}$ variables and we can set the parameter so that it has degree n^{c_d} and error $2^{-n^{c_d}}$, less than c times the total number of gates in the circuit. The seed length necessary to sample it will also be $\leq n^{c_d}$.

The seed used to sample the polynomials is re-used across all gates. We can afford this because we use a union bound in the analysis. Hence the seed length for the final composed

polynomial is again just n^{c_d}, giving (1). Degrees multiply as in Theorem 9.2; so the final degree is also n^{c_d}.

It remains to show (2), that is, that we can evaluate the polynomial. We are going to show how we can compute the monomials of the composed polynomial in the same way as we computed monomials in Lemma 11.9. It amounts to parsing the index to the monomial in the natrual way. Some details: Start at the output gate. We use n^{c_d} bits in the given index to choose a monomial in the corresponding polynomial. We write down this monomial, using Lemma 11.9. This monomial is over the $2^{n^{c_d}}$ variables $z_1, z_2, \ldots$ corresponding to the children of the output gate, and as remarked has degree $\leq n^{c_d}$. To each z_i there corresponds another polynomial p_i. Choose a monomial from p_i and replace z_i with that monomial; do this for every i. The choice of the monomials is done again using bits from the index. Because each monomial is over n^{c_d} variables, and the depth is constant, the total number of bits which are needed to choose monomials is n^{c_d}.

We continue in this way until we have monomials just in the bits $M(x, y_1, y_2, \ldots, y_d)$. Those bits can then be computed from x in P running M. **QED**

11.4.1 BP $\oplus$ P $\subseteq$ SymP

Using the ideas in Theorem 9.6 one can show that BP$\oplus$P (and hence PH) is in SymP:

Definition 11.9. We denote by SymP the class of functions $f : X \subseteq [2]^* \to [2]$ for which there are $h, g \in \mathrm{P}$ and a constant d such that for every $x \in X$ we have $f(x) = h(z)$ where z is the number of $y \in [2]^{n^d}$ s.t. $g(x, y) = 1$.

In other words, f is efficiently computable from the number of y s.t. $g(x, y) = 1$. Note that SymP includes Σ_1P, Π_1P, $\oplus$P, MajP. On the other hand, SymP $\subseteq$ MajMajP. This latter inclusion can be proved along the lines of Exercise 8.3. (A relatively simple proof gives three Maj quantifiers; we won't need the fine result that you can use 2.)

The following simulation is a uniform, scaled-up analogue of the proof of Lemma 9.9.

Theorem 11.12. BP $\oplus$ P $\subseteq$ SymP.

Proof. Let f be a function in the LHS. We set $\ell = n^{c_f}$. For an input x, $f(x)$ is determined by the sum

$$\sum_{i \in [2^\ell]} (p_i \mod 2) \tag{11.2}$$

where the p_i are linear polynomials over the integers, over $N = 2^{n^{c_f}}$ input bits.

Lemma 9.10 allows us to write that sum as

$$\left(\sum_{i \in [2^\ell]} F_\ell(p_i) \right) \mod 2^\ell.$$

Indeed, in the last expression we can move the mod inside, and then apply Lemma 9.10. Now write F_ℓ as a sum of monomials: $F_\ell = \sum_j r_j$. The latter sum can be merged with the one over i, sum over all q_i, and so we obtain

$$\left(\sum_{i \in [2^\ell]} \sum_{j \in [2^\ell]} r_j(p_i) \right) \mod 2^\ell.$$

Now r_j has degree $\leq n^{c_L}$. We can expand $r_j(p_i)$ into a sum of $\leq 2^{n^{c_L}}$ monomials of power degree. We use that the polynomials in Lemma 9.10 satisfy this property: Given an index to a monomial its coefficient in $[2^\ell]$ can be computed in time ℓ^c. The result follows. **QED**

11.5 Problems

Problem 11.1. TBD Let $\text{ACSize}(d, s)$ be the functions computable by explicit alternating ACs of size s and depth d. Prove that $\text{BP} \cdot \text{ACSize}(d, s) \subseteq \text{ACSize}(d + c, 2^{\log^{c_d} s})$. That is, we can de-randomize small-depth circuits in time much smaller than exponential.

Problem 11.2. Let f be the And function on n bits.

(1) Letting $f = \sum_\alpha \hat{f}_\alpha x^\alpha$ as in Exercise 11.4, give an expression for the $\hat{f}_\alpha$.

(2) Use (1) to show that a pseudorandom generator for degree-1 polynomials fools any And function (on any subset of the bits).

Problem 11.3. Give a "direct" construction of PRGs sufficient to prove P = BPP from a δ-hard function h in E. Guideline: Use BIG-HIT to generate an $n \times n$ matrix of inputs, evaluate h on every input, and then XOR the rows. Start with $\delta = c$. How small can you make δ and still have this construction?

Problem 11.4. [Or property vs. error-correcting codes] Actually, small-bias generators are *equivalent* to error-correcting codes. (The main novelty in theoretical computer science seems the level of explicitness that's typically required in applications.) In this problem you will explore this connection.

A set $C \subseteq [2]^n$ of size 2^k is called *linear* if there exists an $n \times k$ matrix M over $\mathbb{F}_2$ such that $C = \{Mx : x \in [2]^k\}$. Recall error-correcting codes from Exercise **??**.

1. Prove that a linear set C is an error-correcting code iff the weight of any non-zero string in C is at least $n/3$.

2. Prove the existence of linear error-correcting codes matching the parameters in Exercise **??**.

3. Let S be a subset of $[2]^k$ s.t. the uniform distribution over S has the Or property. Define $|S| \times k$ matrix M_S where the rows are the elements of S. Prove that $\{M_S x : x \in [2]^k\}$ is an error-correcting code.

4. Give explicit error-correcting codes over ck^2 bits of size 2^k.

5. This motivates improving the parameters of distributions with the Or property. Improve the seed length in Lemma **??** to $\log n + c \log\log n$. Hint: What property you need from T?

6. Give explicit error-correcting codes over $k \log^c k$ bits of size 2^k.

Problem 11.5. TBD Give an alternative construction of small-bias generators as follows:

(1) Fool inputs with weight 1.

(2) For any j, fool every input x with weight between 2^j and 2^{j+1}, with a seed of length $c \log n$. Use Lemma 5.4.

(3) Combine (2) with various j to fool satisfy the Or property for every input.

(4) State the seed length for your distribution and compare it to that of Lemma **??**.

Problem 11.6. Prove Item (2) in Theorem 5.8 using Theorem 11.7.

11.6 Notes

For more on unconditional pseurandom generators see [136]. For a broader view of pseudorandomness, with an emphasis on connections between various objects, see [280]. For hypercube analysis, see [219].

For expander graphs see [145]. They have many equivalent presentations, for example in terms of eigenvalues. My presentation is in terms of the mixing lemma, see Section 2.4 in [145]. In my definition I allow for repeated edges. Different notions of explicitness are also natural. In my definition one can output an edge given an index. More stringently, one can ask, given a node and an index to an incident edge, to compute the corresponding neighbor. The construction I presented immediately gives the more stringent explicitness as well.

k-wise uniform distributions were studied before complexity theory, see [228]. The complexity viewpoint is from [68, 21].

Generators for degree-1 polynomials originate in [209], with alternative constructions in [22]. The idea of xoring generators for degree-1 polynomials to fool higher-degree polynomials is from [51]. It was studied further in [191, 294, 81]. [294] proves Theorem 11.6.

Regarding 11.2: A construction computable in time n^c first appeared in [216], where 11.2 also appears. Alternative constructions computable with small space or with alternations appeared respectively in [172] and [289]. Still, all these constructions use resources at least exponential in the seed length, while for several applications such as Lemma 11.6 one needs power in the seed length. This stronger explicitness is obtained in [123] building on an idea presented in [128] of using error-correcting codes. Specifically, one can use the polynomial code from **??** in combination with 11.2 for logarithmic-scale collections (for which the former notion of explicitness is now acceptable).

The XOR lemma was reportedly announced in talks associated with the work [312], see [111]. Hardness amplification within NP was first studied in [218] which established correlation about $1/\sqrt{n}$. Exponentially small correlation was achieved in [139], with optimizations in [192, 116]. Our exposition follows [139].

Corollary 8.1 and the connection to min-max is from [314]. Hardcore sets were introduced in [150]. They were optimized and shown to be connected to boosting techniques in machine learning in [171] and subsequent works.

Theorem 11.9 is from [30], building on results in the same spirit from [45, 189, 29, 102].

Other proofs of Theorem 3.5 don't use the derandomized xor lemma, or only use it from constant hardness, and instead rely on results in [150] (see e.g. the original proof [154]) or [268] (see e.g. [280] or [25]).

Problem 11.3 is similar to a construction in [268], except I use XOR instead of extractors, see Remark 15 in [268].

For more on hitters, see Appendix C in [106].

The quote at the beginning of section 11.2.2 is from my PhD thesis [290].

Several proofs of Theorem 11.11 have appeared [163, 89]. Our presentation based on Theorem 9.2 seems a little different ([163, 89] don't cite [231]).

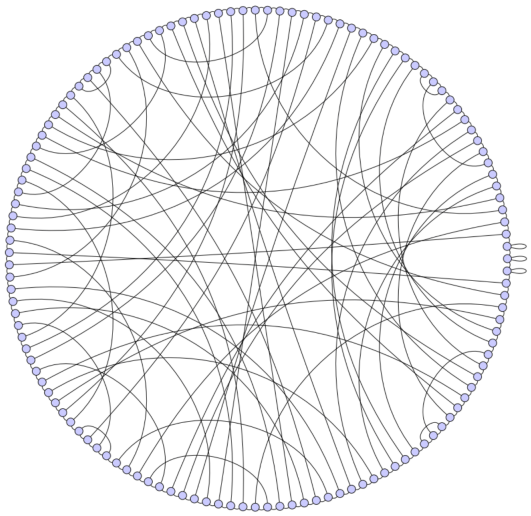
Theorem 11.11 and Theorem **??** are from [273].

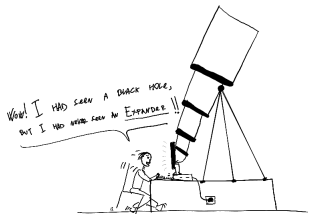
Parity P was defined in [221].

Maj P was defined in [148].

Chapter 12

Expanders





12.1 Expanders without eigenvalues

In this section we give a simple and self-contained proof of the existence of constant-degree *expander graphs*. Informally, an expander graph is a graph that is sparse yet "highly con-

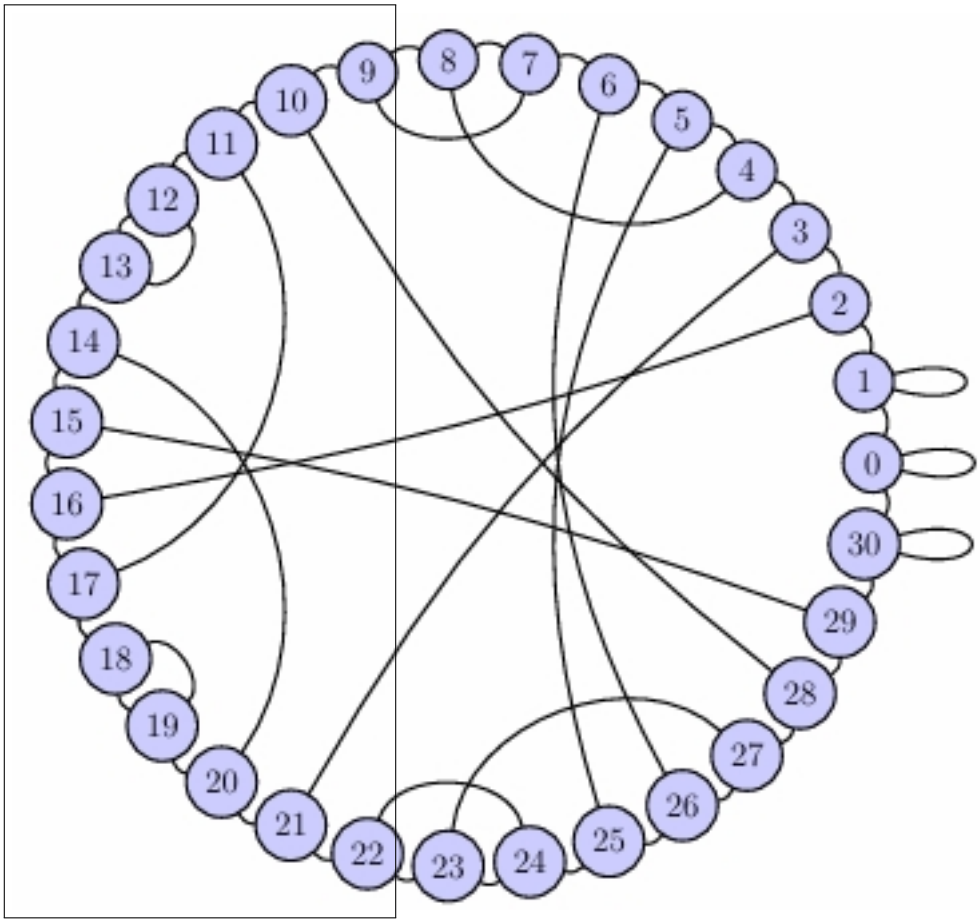


Figure 12.1: A 3-regular expander.

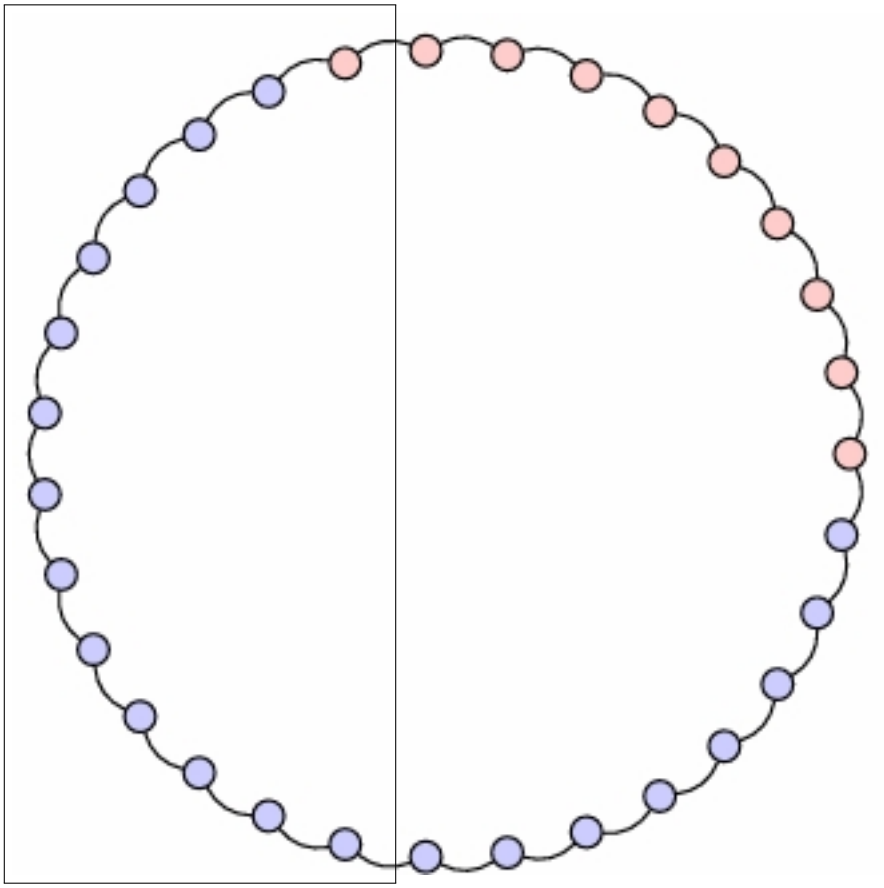


Figure 12.2: The circle graph, on 31 nodes. A set of 10 nodes, shown in red, only has 2 edges leaving it.

nected." Several related notions of expansion exist: vertex, edge, and spectral expansion. In this section we work solely with edge expansion:

Definition 12.1. A d-regular graph $G = (V, E)$ is a *combinatorial edge δ-expander* if for every set $S \subseteq V$ of size $\leq V/2$ its *crossing probability* $\nmid(S)$ that from a uniform $s \in S$ moving to a uniform neighbor takes us out of S is $\geq \delta$. Note

$$\nmid(S) = \frac{E(S, \overline{S})}{dS},$$

where $E(S, T)$ is the set of edges with an endpoint in S and the other in T.

Exercise 12.1. Whereas Definition 12.1 is stated for sets of density $\leq 1/2$, it implies expansion for other sets as well: Let $G = (V, E)$ be a δ-expander and let $S \subseteq V$ of density $\leq (1 - \epsilon)$. Prove $\nmid(S) \geq \delta\epsilon/(1 - \epsilon)$.

Any connected graph $G = (V, E)$ is a δ-expander for noticeable $\delta \geq 1/E$, since at least one edge leaves S. *Expander graphs* are graphs where δ is bounded independent of the size of the graph.

Example 12.1. The circle graph over nodes $\mathbb{Z}_N$ with edges $\{x, x+1\}$, depicted in figure 12.2, is not an expander. For example, $\nmid ([N/2]) \to 0$ with $N \to \infty$. On the other hand, the 3-regular expander in figure 12.1, which can be obtained from the circle by adding "chords," can be shown to be an expander. This expander has $\mathbb{F}_p$ as nodes, and the neighbors of x are $x \pm 2$ and $1/x$. For $x = 0$ we replace the undefined $1/x$ with a self-loop, making the graph 3-regular.

Here's another simple expander. TBD the vertex set of a graph G on N nodes is $Z_{\sqrt{N}} \times Z_{\sqrt{N}}$, where $Z_{\sqrt{N}}$ is the ring of the integers modulo $\sqrt{N}$. Each vertex v is a pair $v = (x, y)$ where $x, y \in Z_{\sqrt{N}}$. For matrices T_1, T_2 and vectors b_1, b_2 defined below, each vertex $v \in G_N$ is connected to $T_1 v, T_1 v + b_1, T_2 v, T_2 v + b_2$ and the four inverses of these operations. It can be shown that for the choices $T_1 := \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $T_2 := \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $b_1 := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $b_2 := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ the resulting graph (which is 8-regular) is an expander.

Pictures are sexy, but we need to compute with expanders. Typically the graph is huge (think Internet) and we have to compute neighbors efficiently given a node name (or index) and an edge name. The expansion properties don't depend on the naming, but the computation does. Several conventions about indexing are possible. *Consistency* is convenient and natural. It asks that an edge name is the same from either endpoint.

Definition 12.2. A d-regular graph $G = (V, E)$ has *neighbor* function $f : V \times [d] \to V$ if $f(u, i)$ for $i \in [d]$ are the d neighbors of u. We say f is *consistent* if whenever $f(u, i) = v$ then also $f(v, i) = u$.

A consistent neighbor function is thus equivalent to an edge coloring of the graph with d colors. While not every d-regular graph admits an edge coloring with d-colors all the graphs in this section have this stronger property.

The main result in this section is that there are expander graphs that are explicit: They have a consistent neighbor function computable in P.

Theorem 12.1. There are c-expanders on 2^n nodes with degree c and a consistent neighbor function in P.

The expander is obtained by starting with expanders with logarithmic degree, and combining them using an operation on graphs called *replacement product* which reduces the degree without sacrificing the expansion. It suffices to apply this product three times to reduce the problem to constructing expander graphs on very few nodes – which we can just brute force.

We now give each of these component. First, we give a non-explicit construction of expanders.

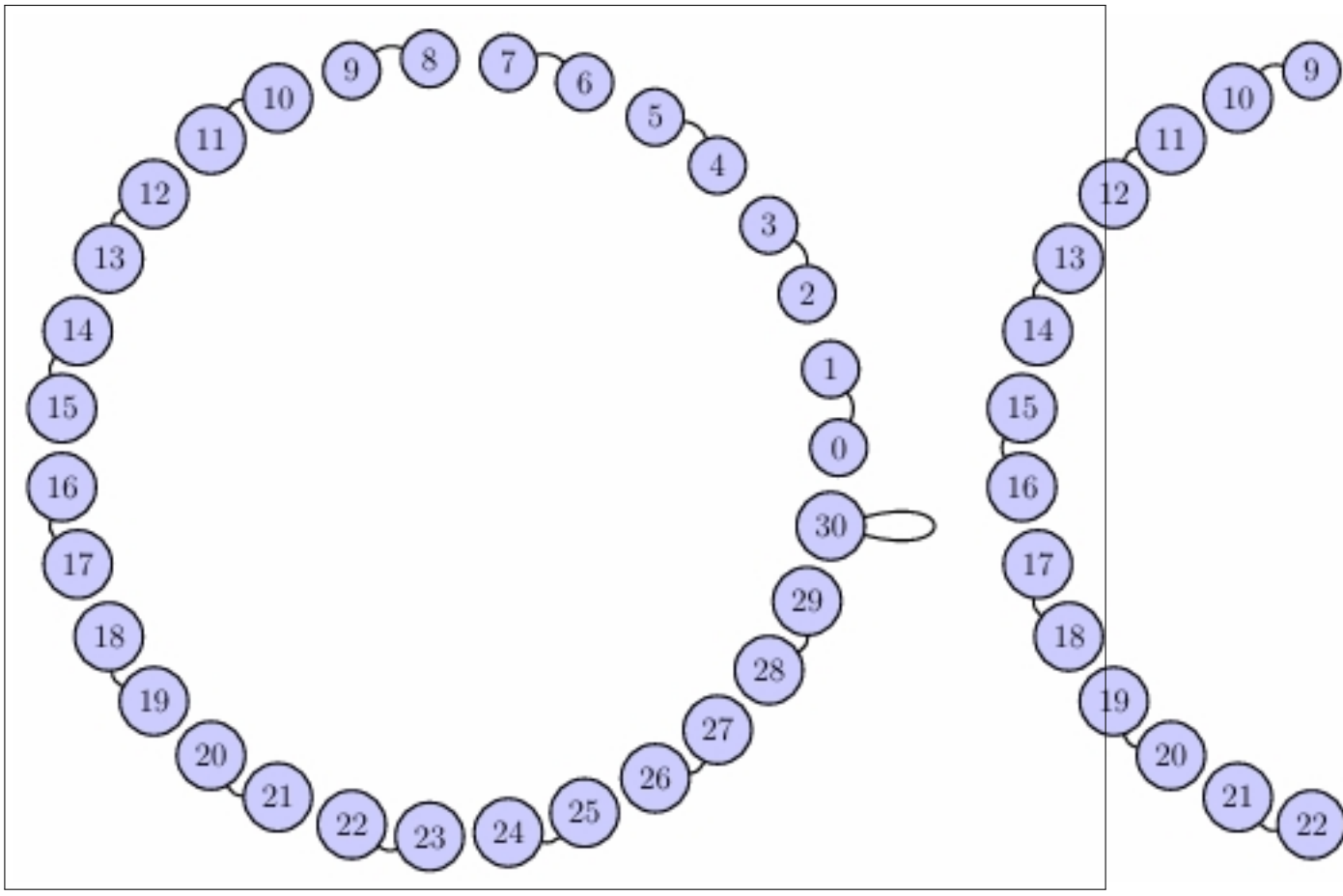


Figure 12.3: The expander in figure 12.1 is the union of these three matchings.

Theorem 12.2. There are c-regular c-expanders on N nodes with a consistent neighbor function computable in time 2^{N^c}.

Proof. We use the probabilistic method. We pick a random d-regular graph by picking d independent, uniform *matchings*. A matching is a maximal collection of disjoint edges. For simplicity, we assume that N is even, and we exclude self-loops from matchings, so a matching has size $N/2$. figure 12.3 illustrates a decomposition in matchings for the expander in figure 12.1; because the number of nodes is odd there, each matching also includes a self-loop. We will soon need to analyze how the distribution of a uniform matching looks locally. For this purpose, we note that for any node v we can sample a uniform matching by first sampling a uniform edge $\{v, w\}$ for $w \neq v$ and then sampling a uniform matching on the remaining $N - 2$.

Now fix a set S. We pick the matching iteratively, at every iteration matching the first unmatched node (in an arbitrary ordering of S). This is possible because of the way we can sample a uniform matching we just described. The iterative process is repeated over $\geq \lceil S/2 \rceil =: T$ elements of S (accounting for the fact that one edge matches 2 or 1 nodes). For each of the first T such iterations, we have matched $\leq S/2 \leq N/4$ elements outside of S. Hence the probability of matching within S at that iteration is $\leq S/(N - N/4) = S/0.75N =: p \leq 2/3$. These events are not independent, but we can still use the deviation bound Lemma B.1 via Exercise B.3. So the prob. that this matching is *bad* in the sense that more than q fraction of these T iterations is matched within S satisfies

$$\mathbb{P}[\text{bad}]^{1/T} \leq \left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q} \leq \left(\frac{p}{q}\right)^q \left(\frac{1}{1-q}\right)^{1-q} \leq p^{q/2}$$

for $q \in [1 - c, 1]$. The latter inequality holds because using $p \leq 2/3$ it is implied by

$$(2/3)^{q/2} \leq q^q (1-q)^{1-q}$$

which holds for $q \to 1$ as the lhs goes to $\sqrt{2/3} < 1$, and the right-had side goes to 1.

If we pick d matchings the probability that they are all bad is $\leq p^{cdT}$. (This loose bound suffices for our claim.) When that does not happen, the crossing prob. $\nmid (S)$ is at least the prob. of selecting a good matching ($\geq 1/d$), times the probability of picking one of the $\geq (1-q)T \geq (1-q)\lceil S/2 \rceil \geq cS$ nodes matched outside of S. Overall, $\nmid (S) \geq c/d$, which is as desired for constant d.

The prob. that there exists a bad set S of size k with lower $\nmid (S)$ is by a union bound and Fact B.6

$$\leq \binom{N}{k} p^{cdT} \leq \left(\frac{eN}{k}\right)^k \left(\frac{k}{0.75N}\right)^{cd\lceil k/2 \rceil} \leq \frac{1}{2^k}$$

for $d \geq c$.

Hence the prob. there is any bad set of size $\leq N/2$ is $\leq \sum_{k=1}^{N/2} 2^{-k} < 1$.

For the explicitness, we enumerate over all graphs. Each graph can be described using $Nc \log N$ bits. Its expansion can be checked by again enumerating over all $\leq 2^N$ subsets of

nodes. Because our graph is obtained as the union of matchings, it has a consistent neighbor function (each matching corresponds to an edge index, or equivalently a color). **QED**

Exercise 12.2. Repeat this proof with the goal of finding the smallest value d for the degree that works (to obtain expansion dependent on d only).

Next we give explicit expanders with logarithmic degree. We essentially already saw this construction in section 11.1.4.

Theorem 12.3. There are c-expanders on 2^n nodes with degree n^c with a consistent neighbor function computable in P.

Proof. Let Y be an ϵ-biased distribution on $\mathbb{F}_2^n$ (see Theorem 11.5); the neighbors of $x \in \mathbb{F}_2^n$ are $x + Y$. Theorem 11.5 implies that the graph is explicit. The neighbor function is consistent because $(x + Y) + Y = x$.

To analyze, let S be a set as in Definition 12.1 and let f be the $0 - 1$ characteristic function of f, and $p := S/N$ the density of S. We have

$$\nmid (S) = p^{-1}\mathbb{E}[f(X)(1 - f(X + Y))] = 1 - p^{-1}\mathbb{E}[f(X)f(X + Y)].$$

We write $f = \sum_\alpha \widehat{f}_\alpha \chi_\alpha$ in the basis of parity functions (see Exercise 11.4). Note that

$$\widehat{f}_0 = \mathbb{E}[f] = \mathbb{E}[f^2] = \sum_\alpha \widehat{f}_\alpha^2 = p.$$

Hence $\mathbb{E}[f(X)f(X+Y)] = \sum_\alpha \widehat{f}_\alpha^2 \epsilon_\alpha = p^2 + \sum_{\alpha \neq 0} \widehat{f}_\alpha^2 \epsilon_\alpha \leq p^2 + \epsilon(p - p^2)$. Plugging this above we get

$$\nmid (S) \geq 1 - p - \epsilon(1 - p) = (1 - p)(1 - \epsilon).$$

The latter is $\geq c$ for p and ϵ both $\leq c$. **QED**

We now seek to reduce the degree to constant by means of the following operation.

Definition 12.3. [Replacement product] Let G be a D-regular graph on N vertices and H a d-regular graph on D vertices. The replacement product $G \boxtimes H$ is the following $2d$-regular graph on $N \cdot D$ vertices (x, i). For every vertex $x \in G$ there is a copy H_x of H, i.e., we connect the nodes (x, i) for $i \in [D]$ according to H. In addition, we connect (x, i) to (y, i) if $x[i] = y$, with d repeated edges.

Equivalently, we can write edges as $[bj]$ where $b \in [2]$ and $j \in [d]$; then $(v, i)[0j]$ is connected to $(v, i[j])$ and $(v, i)[1j]$ is connected to $(v[i], i)$. (The square brackets correspond to 3 different neighbor functions.)

Note that if G and H have consistent neighbor function, then so does $G \boxtimes H$. Repeating edges makes it equally likely that a random neighbor makes a step inside a copy of H or outside, corresponding to an edge in G.

Theorem 12.4. Suppose E_1 is a D-regular δ_1-expander on N nodes, and E_2 is a d-regular δ_2-expander on D nodes. Then $E_3 := E_1 \boxtimes E_2$ is $2d$-regular $c\delta_1^2\delta_2$-expander on ND nodes.

Proof. Let X be a set of nodes in E_2 of size $\leq ND/2$. We view the vertex set of E_3 as composed of N clusters of vertices C_i, each of size $D = C_i$. Let $X_i := X \cap C_i$ and consider two cases. Either many X_i are *underfull* (in C_i), in which case many edges are leaving X within the clusters due to the expansion of E_2; or many X_i are almost full, in which case there are many edges leaving X between the clusters, due to the expansion of E_1. Details follow.

Let I' be the indices of the X_i which we call *underfull* and have size $\leq (1-\delta_1/4)C_i$. and let I'' be the others; let $X' := \cup_{i\in I'}X_i$ and $X'' := \cup_{i\in I''}X_i$.

If $X'/X \geq \delta_1/10$ then we can think of the experiment in $\nmid(X)$ as sampling a uniform node from X' with prob.$\geq \delta_1/10$. For any $X_i \subseteq X'$ we have $\nmid(X_i) \geq 0.5\cdot\delta_2(\delta_1/4)$ by Exercise 12.1, where the 0.5 is for taking a step withing C_i. Hence $\nmid(X) \geq (\delta_1/10)\cdot 0.5\cdot\delta_2\delta_1/4$ and we are done in this case.

Otherwise $X''/X \geq (1-\delta_1/10)$. Let F (for "full") be the union $\cup_{i\in I''}C_i$ of the almost full clusters we have

$$E(X,\overline{X}) \geq E(F,\overline{F}) - E(\cup_{i\in I''}\overline{X_i},\overline{F}) - E(X',\overline{X'}). \tag{12.1}$$

To bound $E(F,\overline{F})$, recall $X_i \geq C_i(1-\delta_1/4)$ for $i \in I''$. Summing over such i we get $F \leq X''/(1-\delta_1/4) \leq 4X''/3 \leq 4X/3 \leq 2ND/3$. By Exercise 12.1, $E(F) \geq 0.5dF\delta_1$. Next, the term $E(\cup_{i\in I''}\overline{X_i},\overline{F})$ is $\leq (I''\cdot D\cdot\delta_1/4)d = (F\delta_1/4)\cdot d$. (Note each node in $\cup_{i\in I''}\overline{X_i}$ has $\leq d$ neighbors outside of F.) So the first difference in the rhs in equation (12.1) above is $\geq (F\delta_1/4)\cdot d$. Because $F \geq (1-\delta_1/4)X'' \geq (1-\delta_1/4)(1-\delta_1/10)X \geq (3/4)(9/10)X \geq 0.5X$, this difference is $\geq Xd\delta_1/8$. Finally, $E(X',\overline{X'})$ is trivially $\leq dX' \leq dX\delta_1/10$. Hence $E(X) \geq Xdc\delta_1$. **QED**

We can now mix these three ingredients to construct explicit expanders.

Proof of Theorem 12.1.. Start with E_1 the expander from Theorem 12.3 with 2^n nodes and degree n^c, and E_2 the same expander but on n^c nodes and degree $\log^c n$.

Then $E_1 \boxtimes E_2$ is an expander with degree $\log^c n$. That is, one replacement product allows us to reduce the degree logarithmically. Doing this again, we can reduce the degree to $\log^c\log n$. Finally, we take a replacement product with the non-explicit expander from Theorem 12.2 to obtain constant degree. The neighbor function in the latter is computable in time $2^{\log^c\log n} = n^{o(1)}$. Because each graph is an expander, and we only apply replacement product three times, the final graph is an expander by Theorem 12.4. **QED**

Exercise 12.3. Prove that the neighbor function is computable in L.

12.2 Eigen stuff

We now turn to eigenvalues, a.k.a. spectral, analysis. (I suspect one can also present the main results in the next sections without eigenvalues, but not clear there is much gain and

eigenvalues are important for many things, so we won't pursue that direction now.) Let us view a vector as a *probability distribution* over the vertices of a graph G. For example, $u = (1/n, \ldots, 1/n)$ is the uniform distribution over n vertices. For now, we also assume that the graph G is *regular*, i.e. each vertex has degree d, and also that each vertex has a self-loop. We justify this assumption later for our applications. For each graph G, we have a normalized adjacency matrix A, where "normalized" means each entry is divided by the degree d. The following matrix is the adjacency matrix of the graph in Figure 12.4.

$$A = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \end{bmatrix}.$$

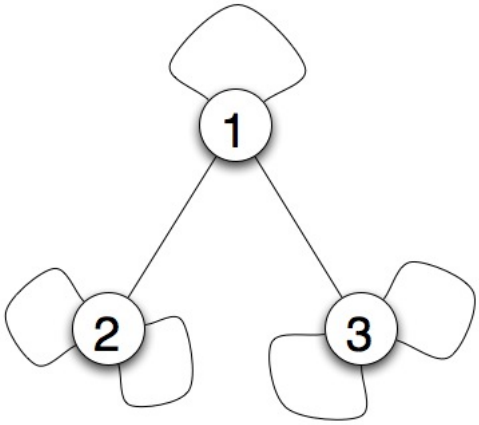


Figure 12.4: Graph example.

For a starting random vector v, Av would be the probability distribution after doing one random step starting at v. For example, for $v = (1, 0, 0)$ we have $Av = (1/3, 1/3, 1/3)$ in the graph above. It would be convenient if the act of multiplying by A corresponded to some simple behavior, like $Av = \lambda \cdot v$ for some scalar λ. Such λ and v are called *eigenvalue* and *eigenvector* respectively. Although every matrix has them, in general they need to be complex, as for example is needed if A corresponds to a rotation of the space. However, the matrices that arise from graphs have a special structure, in particular they are symmetric. One can show that in this case there is a basis of n *real* eigenvectors, called an *eigenbasis*. Moreover, we can choose them to be *orthonormal*, i.e. length-1 vectors that have zero inner product.

Theorem 12.5 (Eigenbasis)**.** Let A be an $n \times n$ real symmetric matrix. Then there exists an orthonormal basis of real eigenvectors $v_1, v_2 \ldots, v_n \in \mathbb{R}^n$.

Some thoughts on the proof are in section B.8.1. This theorem allows us to write any vector v in the eigenbasis and see the act of multiplying the vector by the matrix A as simply multiplying each coordinate of v by the corresponding eigenvalues.

We list several basic properties.

Lemma 12.1. Let A be the normalized random-walk matrix of a d-regular graph G. We have:

(1) The eigenvalues of A are in $[-1, 1]$. This holds more generally for any matrix that is real, symmetric, where each entry is ≥ 0, and each row sums to ≤ 1 (a.k.a. *row sub-stochastic).*

(2) If moreover G has a self-loop on each node and is d-regular then the eigenvalues are in $[-1 + 1/d, 1]$.

(3) If G is regular then the uniform distribution is an eigenvector with eigenvalue 1. Its multiplicity (that is, the dimension of the span of the eigenvectors) equals the number of connected components of G.

(4) G is bipartite iff -1 is an eigenvalue.

Proof. (1) Let v be an eigenvector and wlog let $v_1 = \max_i |v_i|$. As $Av = \lambda v$, in particular $|(Av)_1| = |\lambda v_1|$. We have $|(Av)_1| = |\sum_i A_{1i} v_i| \leq \sum_i A_{1i}|v_i| \leq v_1 \sum_i A_{1i} \leq v_1$. Hence $|\lambda v_1| \leq |v_1|$ and the result follows. **QED**

Exercise 12.4. Prove (2)-(4). Hint: For (2), consider $A - I/d$, then use (1). For (3), first give orthogonal eigenvectors, then prove every other eigenvector with eigenvalue 1 is in the span.

The next useful lemma gives a useful characterization and property of the second largest eigenvalue, in absolute value order. It shows that it gives a bound on how closer we get to uniform after taking a random step. We simply write $|v|$ for the 2-norm $|v|_2 = \sqrt{\sum_i v_i^2}$.

Lemma 12.2. Let A be the normalized adjacency matrix of a connected graph G. Sort the eigenvalues of A by absolute value: $1 = |\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n| \geq 0$ then

$$|\lambda_2| = \max_{v \perp u} \frac{|Av|}{|v|}$$

where u is the uniform distribution.

In particular, for any probability distribution v we have

$$|Av - u| \leq |\lambda_2||v - u|.$$

Proof. Take any vector $v \in \mathbb{R}^n$. Write $v = a_1 v_1 + \cdots + a_n v_n$ where the v_i are the eigenvectors of the matrix A, from Theorem 12.5, and $v_1 = u$. Because $v \perp u$ and G is connected we have $a_1 = 0$ by Lemma 12.1. So $v = a_2 v_2 + \cdots + a_n v_n$. We now have, by Fact B.21 and orthonormality of the v_i:

$$|Av|^2 = |\lambda_2 a_2 v_2 + \cdots + \lambda_n a_n v_n|^2 = |\lambda_2 a_2|^2 + \cdots + |\lambda_n a_n|^2 \leq |\lambda_2|^2 |v|^2.$$

This bound is met by $v = v_2$.

The "in particular" part is because $|Av - u| = |A(v - u)|$ and $(v - u) \perp u$ by direct verification. So by the previous claim $|A(v - u)| \leq |\lambda_2||v - u|$. **QED**

Exercise 12.5. Let A and G be as in Lemma 12.2. Sort the eigenvalues (without taking absolute values) as $1 = \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n \geq -1$. Prove

$$\lambda_2 = \max_{v \perp u} \frac{\langle Av, v\rangle}{\langle v, v\rangle},$$
$$1 - \lambda_2 = \min_{v \perp u} \frac{\sum_{\{i,j\} \in E} (v_i - v_j)^2}{d \cdot \langle v, v\rangle}. \tag{12.2}$$

The rhs of Exercise 12.5 gives a useful way to think of the *spectral gap* $1 - \lambda$ as of minimizing a certain "energy" function on the edges over the choice of a uniform edge.

Notation 12.1. We write λ_i for the eigenvalues in standard order, and λ'_i for those in absolute value order. We write $\lambda_2(G)$ and $\lambda'_2(G)$ for the maximum non-trivial eigenvalue in each order, and omit G when is clear from the context.

Spectral vs. edge expansion We now show that a graph is an edge expander in the sense of Definition 12.1 iff λ'_2 is small. We first illustrate this via the following "gem" that directly relates the bias of the construction in Theorem 12.3 to the second largest eigenvalue.

Theorem 12.6. Let D be an ϵ-biased distribution on $[2]^n$, and let A be the $2^n \times 2^n$ adjacency matrix of the graph on $[2]^n$ where $A_{a,b} := \mathbb{P}[D = a + b]$ (i.e., the weight of edge $a \to b$ is the prob. of going from a to b when xor-ing a with a sample from X.

Then $\lambda'_2 \leq \epsilon$.

In particular, explicit graphs with λ'_2 exist with degree $(\epsilon^{-1} \log N)^c$.

Proof. By Lemma 12.2 we need to bound $|Av|/|v|$ for any vector $v \perp u$. Let's pick a natural and convenient basis. The 2^n vectors $\chi_S(x) := (-1)^{\langle S,x\rangle}$ of length 2^n, for $S \in [2]^n$, are orthogonal, hence independent (Fact B.23). Let us write v in this basis: $v = \sum a_S \chi_S$. Because $v \perp u$, $a_0 = 0$.

We now have

$$Av = \lambda v = \sum \lambda a_S \chi_S = \sum_{S \neq 0} a_S A \chi_S = \sum a_S \epsilon_S \chi_S,$$

where $\epsilon_S = \mathbb{E}[\chi_S(D)]$ is the bias of χ_S wrt D. is To check the last equation:

$$(A\chi_S)(x) = \sum_y A_{x,y} \chi_S(y) = \sum_y \mathbb{P}[D = x + y]\chi_S(y) = \sum_y \mathbb{P}[D = y]\chi_S(x + y) = \chi_S(x)\mathbb{E}[\chi_S(D)],$$

where we replace y with $x + y$ and then use $\chi(x + y) = \chi(x) \cdot \chi(y)$

Hence

$$\frac{|Av|}{|v|} = \frac{\sqrt{\sum_{S \neq 0} a_S^2 \epsilon_S^2}}{\sqrt{\sum_{S \neq 0} a_S^2}} \leq \epsilon.$$

The "in particular" follows from Theorem 11.5. **QED**

More generally, we have the following connections between λ and δ in Definition 12.1. The bottom line is that each of δ and λ is bounded iff the other is.

Theorem 12.7. [Edge vs. spectral expansion] Let G be a graph, and let δ be the maximum s.t. G is a δ-expander. Then:

(1) [Spectral expansion $\Rightarrow$ edge expansion] $\delta \geq (1-\lambda_2)/2$,

(2) [Edge expansion $\Rightarrow$ spectral bound] $\lambda_2 \leq 1 - \delta^2/2$,and

(3) [Edge expansion + self-loops $\Rightarrow$ absolute spectral expansion] if G has self-loops on each node then

$$\lambda_2' \leq 1 - \min\{\delta^2/2, c/d\}.$$

Proof. 2(1) We use the characterization in Exercise 12.5. Let $x_i := \overline{S}$ if $i \in S$ and $-S$ if $i \in \overline{S}$. Note that x is orthogonal to uniform. By Exercise 12.5, $1 - \lambda_2(G) \geq \sum_{\{i,j\}\in E}(x_i - x_j)^2/d\langle x, x\rangle$.

The numerator of the rhs is $2E(S,T)V^2$ since each edge leaving S contributes V^2. Also note that $\langle x, x\rangle = S\overline{S}^2 + \overline{S}S^2 = S\overline{S}V$. Hence

$$1 - \lambda_2(G) \leq \frac{2E(S,\overline{S})V^2}{dS\overline{S}V} = \frac{2E(S,\overline{S})V}{dS\overline{S}} = \nmid(S)\frac{2V}{\overline{S}} \leq 2\nmid(S).$$

(2) Let $Q := I - A$. Note if λ is an eigenvalue of A then $1-\lambda$ is an eigenvalue of Q. So it suffices to prove that the eigenvalues of Q are $\geq \delta^2/2$. Let z be an eigenvector orthogonal to uniform. This orthogonality implies that z has both coordinates > 0 and also < 0. Sort the coordinates as $z_1 \geq z_2 \geq z_3 \geq ... \geq z_n$. We assume that exactly $m \leq n/2$ of its coordinates are ≥ 0. (Otherwise, one can consider $-z$.) Note that $(Qz)_i = \lambda z_i$ for all i. In particular,

$$\lambda = \frac{\sum_{i\leq m}(Qz)_i z_i}{\sum_{i\leq m} z_i^2}.$$

Consider the numerator, multiplied by d. Plugging the definition of Q and expanding out, and writing E for the multi-set of edges, e.g. $E = \{\{1,2\},\{2\},\{2\},\{1,2\},\{2,3\}\}$ (two self-loops, and two parallel edges), we can write it as

$$\begin{aligned}\sum_{i\leq m}(dz_i^2 - \sum_{j:\{i,j\}\in E} z_i z_j) &= \sum_{\{i,j\}\in E: i<j\leq m}(z_i - z_j)^2 + \sum_{\{i,j\}\in E: i\leq m<j} z_i(z_i - z_j) \\ &\geq \sum_{\{i,j\}\in E: i<j\leq m}(z_i - z_j)^2 + \sum_{\{i,j\}\in E: i\leq m<j} z_i^2. \qquad (12.3)\end{aligned}$$

Let x be equal to z except negative entries are zero. We claim that we can switch from z to x, and in fact sum over all edges: The rhs above is

$$\geq \sum_{\{i,j\}\in E}(x_i - x_j)^2.$$

To verify this, note that edges with $i < j \leq m$ contribute the same in the expression with the z and with the x. The same holds for edges with $i \leq m < j$ since $z_j = 0$. And edges with both i and j bigger than 0 contribute nothing.

The key thing is that this sum is now over all edges, so we can rely on the expansion of the graph. For intuition, suppose that x was "flat:" equal to 1 over $[m]$, and 0 otherwise. In this case, the contribution of $x_i - x_j$ is 0 unless $\{i,j\} \in E([1..m],[m+1..n])$, in which case it is c. By edge expansion, the number of crossing edges is $\geq \rho m d$. Hence the numerator is $\geq c\rho m$. The denominator is m. Hence

$$\lambda \geq \frac{c\rho m}{m} \geq c\rho$$

and we are done.

The rest of the proof is for the general case where x may not be flat; it is based on the same idea but it has a few algebraic manipulations. Let x be equal to z except negative entries are zero. We can replace z_i with x_i for any $i \leq m$ by definition, and replace $\sum x_i^2$ with $|x|^2$. Hence by above we have

$$\lambda \geq \frac{\sum_{\{i,j\}\in E}(x_i - x_j)^2/d}{|x|^2}. \tag{12.4}$$

Now our goal is to somehow "turn" $(x_i - x_j)^2$ into $x_i^2 - x_j^2$, because it allows us count more easily the contribution from each term. For this purpose, consider the quantity $\sum_{\{i,j\}\in E}(x_i + x_j)^2/d$. Note that this equals $|x|^2$ up to constants. Indeed, it is larger than $\sum_{\{i,j\}\in E}(x_i^2 + x_j^2)/d = |x|^2$, the equality holding because the degree of each node is d. Also, it is $\leq \sum_{i,j} A_{i,j}(x_i + x_j)^2 \leq c|x|^2 + c\sum_{i,j} A_{i,j}x_i x_j \leq c|x|^2$ by Fact B.11.

Hence multiplying equation (12.4) by $d|x|^4$ we obtain

$$d|x|^4\lambda \geq c\sum(x_i - x_j)^2 \cdot \sum(x_i + x_j)^2 \geq c\left(\sum(x_i - x_j)(x_i + x_j)\right)^2 = c\left(\sum_{\{i,j\}\in E}(x_i^2 - x_j^2)\right)^2 \tag{12.5}$$

where we use Fact B.11.

Finally, we can rewrite the inner sum telescopically as

$$\sum_{\{i,j\}\in E, i<j}\sum_{k=i}^{j-1} x_k^2 - x_{k+1}^2.$$

Note now that each term $x_k^2 - x_{k+1}^2$ appears as many times as the number of edges $\{i,j\}$ with $i \leq k < j$. Hence the double sum equals

$$\sum_{k=1}^{m} E([1..k],[k+1..n])(x_k^2 - x_{k+1}^2);$$

where we use that $x_k = 0$ for $k \geq m$. By expansion and the fact that $m \leq n/2$, the E term is $\geq dk\rho$. So the sum is

$$\geq d\rho \sum_{k=1}^{n/2} k(x_k^2 - x_{k+1}^2) = d\rho \left(\sum kx_k^2 - (k-1)x_k^2 \right) = d\rho |x|^2.$$

Plugging this bound inside equation (12.5) we obtain $\lambda \geq c\rho^2$. **QED**

Exercise 12.6. Prove equation (12.3) in the proof. Prove (3) in Theorem 12.7.

Exercise 12.7. Prove that a connected graph on n nodes with a self-loop on each node has $\lambda_2' \leq 1 - 1/n^c$. Hint: Use Theorem 12.7.

Analysis of the random-walk algorithm Using eigenvalues we can analyze a simple randomized log-space algorithm for UConn (see Definition 6.5). This *random walk algorithm*, on input a graph on n nodes and vertices s and t decides if s and t are connected as follows. Starting with $v := s$, it moves to a uniformly selected neighbor of v for n^c times. If it ever encounters t, it reports *connected*, otherwise it reports *not-connected*.

Theorem 12.8. The random walk algorithm runs in logarithmic space and has error prob. $\leq 1/2$.

Exercise 12.8. Prove this. Guideline: Use Lemma 12.2 and Exercise 12.7, Divide the walk of length ℓ in sub-walks of length $\sqrt{\ell}$. First prove that each subwalk has noticeable prob. of reaching t if it is connected to s.

12.3 Robust UConn

To introduce the approach, note that we can solve UConn (see Definition 6.5) in *deterministic* log-space on graphs with $\lambda_2' \leq 1/n^c$. This is because by Lemma 12.2 the distance between Av and u is $\leq 1/n^c$. But if Av puts no mass on t the distance would be larger. Hence simply trying all neighbors of s we can determine if s and t are connected. Our input graph doesn't have to satisfy this strong requirement, nevertheless just the fact that it's connected (which is the interesting case of the analysis) implies a noticeable spectral gap (Exercise 12.7).

Thus we would be done if we could somehow turn the graph into a strong expander where every non-trivial eigenvalue is $\leq 1/n^c$ in absolute value. While this can be done, it is slightly more convenient to work with a "robust" version of the problem which will "only" require a constant spectral bound. The following claim also justifies our assumption that graphs are regular, which we used in previous sections.

Claim 12.1. UConn is log-space reducible to the following robust-UConn problem: given a 4-regular graph with a self-loop on each node, and given two connected sets of nodes S and T of density $\geq 1/3$, decide whether S and T are in the same connected component.

Proof. Given an instance G, s, t of UConn, where G has n nodes, we construct G' in the following way. Add n copies of s and n copies of t. Set S consists of all copies of s, and same for T. Put $n/2$ copies of a cycle on S, and similarly on T, as shown in Figure 12.5, so that the extra copies of s and t have degree n. Observe that the degree of each node is $\leq n$

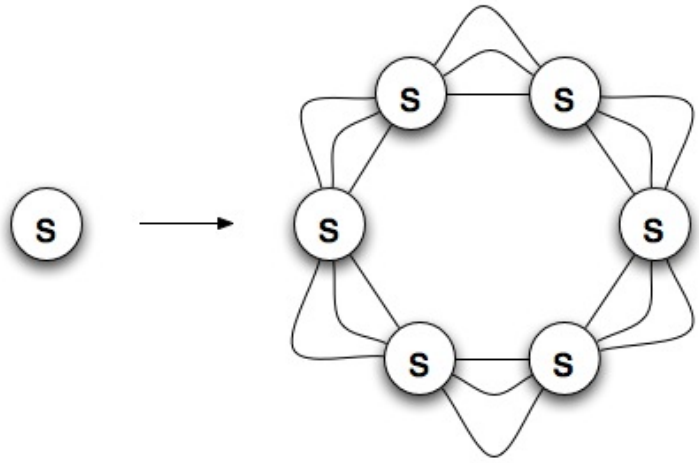


Figure 12.5: Duplicate s and add cycles.

except for s and t which may have degree as large as $2n$. To reduce the degree, replace each node of degree d with a 4-regular graph on d nodes, as shown in Figure 12.6. And call this final graph G'.

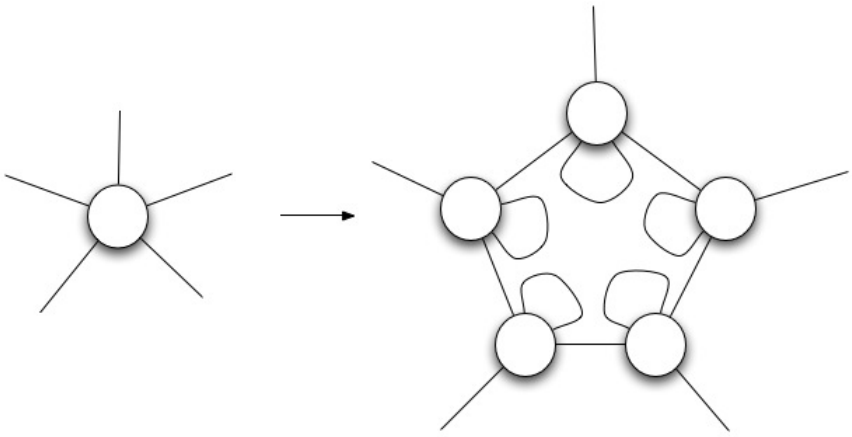


Figure 12.6: Replace each node with a 4-regular graph.

By construction, the number of nodes in G' is n^c, G' is 4-regular, and $|S| \geq |G'|/3$, $|T| \geq |G'|/3$. The bound on $|S|, |T|$ follows because (1) this bound holds before reducing the degree, (2) the degree reduction blows up a node by its degree, and (3) every node in $S \cup T$ has degree at least as large as that of any other node in the graph. **QED**

We now show that this more robust version is "easy" when the second eigenvalue is small. This is similar to the previous observation that we can solve UConn easily on graphs with very small eigenvalue bound. However, we will now work with constant eigenvalue bound. We do not restrict the degree: we will apply the following claim to graphs of power degree. Jumping ahead, these will arise by modifying the 4-regular graphs given by Claim 12.1 using appropriate operations that reduce the eigenvalue bound.

Claim 12.2. Let G be a graph with $\lambda_2' \leq 1/10$. Let S and T be connected sets of nodes of density $\geq 1/3$. If some node in S is *connected* to some node in T, then some node in S is *adjacent* to some node in T.

Note the reverse implication is trivial. Therefore, if the neighbors of G are computable in logspace then we can also decide in log-space whether given dense sets S and T are in the same connected component, by cycling over all nodes in S and their neighbors.

The basic idea is that since S is large and λ_2' is small, S has many neighbors ($> 2n/3$), and one of them must be in T.

Exercise 12.9. Prove Claim 12.2. Guideline: Let u represent the uniform distribution (i.e. $u = (1/n, 1/n, \ldots, 1/n)$), v represent the uniform distribution on S (i.e. $v = (3/n, \ldots, 3/n, 0, \ldots, 0)$ where the coordinates with mass $3/n$ are exactly those in S). Our goal is to show Av has non-zero weight on some coordinate in T. Bound $|Av - u|$ from above using 12.2 and from below in case Av has no mass on T.

We are now left with the task of reducing the eigenvalue bound of our graph.

12.3.1 An attempt to reduce the eigenvalue bound

One attempt to reduce the eigenvalue of a graph is by squaring. The squared graph is the graph in which edges correspond to paths of length 2 in the original graph; Figure 12.7 shows an example.

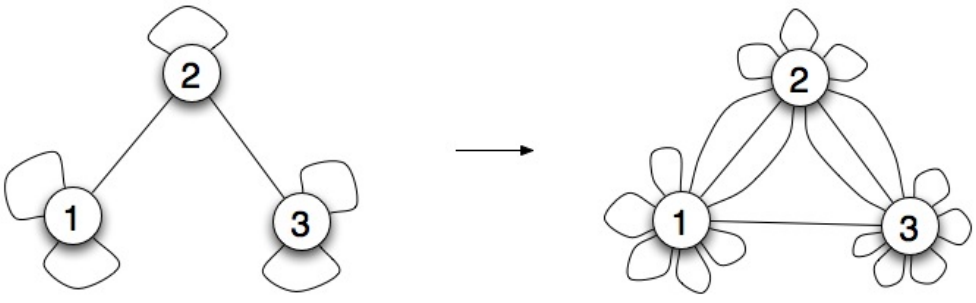


Figure 12.7: Squaring a graph.

Note that the eigenvectors of (the normalized adjacency matrix A of) G^2 are the same as those of G, and the corresponding eigenvalues square, because

$$A^2 v = A(Av) = \lambda Av = \lambda^2 v. \tag{12.6}$$

So if we start with a connected graph with a self-loop at each node – which has $\lambda_2' \leq 1 - 1/n^c$ by Exercise 12.7– and we square it $\ell = c \log n$ we obtain the bound

$$\lambda_2'\left(G^{2^\ell}\right) \leq \left(1 - \frac{1}{n^c}\right)^{2^\ell} = \frac{1}{10}.$$

Although we obtain the desired eigenvalue bound, the degree of the graph G^{2^ℓ} is $D^{2^\ell} \geq D^n$, which is exponential in n. This means we cannot apply Claim 12.2 to determine connectivity in logarithmic space, since the idea there was to cycle over all neighbors of nodes in S.

In order to apply Claim 12.2, we need another graph operation that can decrease λ_2' while at the same time keeping the degree small. Note that an edge in G^{2^ℓ} corresponds to a path of length n^c in G. With the new operation we will still have that an edge in the final graph corresponds to a path of that length in G. But the crucial difference is this: whereas in G^{2^ℓ} we consider all, exponentially many paths, in the new graph we only consider a sparse, polynomial-size collection of paths. We will prove that this sparse collection has the same hitting properties of the collection of all paths, as measured by the eigenvalue bound.

12.3.2 Reducing the eigenvalue via derandomized graph squaring

Derandomized graph squaring is similar to the replacement product (see Definition 12.3). One advantage is that it does not increase the number of nodes.

Definition 12.4. [Derandomized graph squaring] Let X be a k-regular graph on n nodes, and G be a d-regular graph on k nodes. $X \ominus G$ is a graph on n nodes with degree $k \cdot d$. The neighbors of v in $X \ominus G$ are $v[a][b]$ where b is a neighbor of a in graph G, i.e. $v[a][a[e]]$ where $a \in [k]$ and $e \in [d]$.

Note that in the above definition we see a as both an edge index for X and a node in G. Whereas in graph squaring the neighbors of v are $v[a][b]$ for *every* a, b, in derandomized graph squaring the neighbors are $v[a][b]$ for *some* a, b. The following main result shows that applying derandomized graph squaring we somewhat decrease λ_2:

Theorem 12.9 (Analysis of derandomized graph squaring)**.** If $\lambda_2'(X) = \lambda$ and $\lambda_2'(G) = \mu$, then $\lambda_2'(X \ominus G) \leq (1 - \mu)\,\lambda^2 + \mu$.

To illustrate parameters, note when μ is sufficiently small, derandomized squaring essentially squares λ. Numerically, one can verify that if $\lambda_2'(G) = \frac{1}{100}$ and $\lambda_2'(X) = 1 - \gamma \geq 1/10$, then $\lambda_2'(X \ominus G)| \leq 1 - \frac{12}{11}\gamma$. So if we start with $\lambda_2' \leq 1 - 1/n^c$ and repeat this operation $c \log n$ times we will get $\lambda_2' \leq 1/10$, qualitatively the same as graph squaring.

To prove Theorem 12.9 we start with a useful lemma that shows that a random step in a graph G with $\lambda_2'(G) = \lambda$ can be seen as going to the uniform distribution with probability $(1 - \lambda)$, and not doing harm otherwise. Denote by J_n the $n \times n$ matrix with $1/n$ everywhere. Multiplying any probability distribution v by J we obtain the uniform distribution u.

Lemma 12.3. Let G be a graph on n nodes with $n \times n$ normalized adjacency matrix A satisfying $\lambda_2'(A) = \lambda$. Then $A = (1 - \lambda)J_n + \lambda C$ where $\forall v : |Cv| \leq |v|$.

Proof. Let $C := (A - (1-\lambda)J_n)/\lambda$. Let v be a vector and write $v = a \cdot u + w$ where a is a constant, u represents uniform distribution and $u \perp w$. Then

$$|Cv|^2 = |au + Aw/\lambda|^2 = |au|^2 + |Aw/\lambda|^2 \leq |au|^2 + |w|^2 = |v|^2.$$

The first equality is obtained by definition and using $Au = u$ and $Jw = 0$. The rest is Fact B.21, Lemma 12.2, and Fact B.21 again. **QED**

Exercise 12.10. Prove the first equality.

To analyze the adjacency matrices arising from derandomized graph squaring we write random-walk matrices in a specific way, also using tensor products (Definition B.2). Let us illustrate in a simple case. Taking a random step in a graph G can be thought as as 3-step process:

$$v \to (v, a) \to (v[a], a) \to v[a],$$

where a is a random edge. This is overkill when dealing with a single random-walk matrix A, but it will be useful when analyzing derandomized graph squaring as we will be able to "remember" the edge label. Each of the three steps above can be implemented by a different matrix, to obtain

$$A = P\tilde{A}L$$

Let G be d-regular on n nodes. The first step is given by the "lift" matrix $L := I_n \otimes (1/d, \ldots, 1/d)^T$ which is $n \cdot d \times n$.

The second step is given by $\tilde{A}$ which is a $n \cdot d \times n \cdot d$ matrix, where $\tilde{A}_{(u,a),(u',a')} = 1$ if and only if $a = a'$ and $u' = u[a]$. This matrix corresponds to taking a step in G *after the choice for the step has been made*. No entropy is added by $\tilde{A}$, which is a permutation matrix.

Finally, the last step is given by the "projection" matrix $P := I_n \otimes (1, \ldots, 1)$ which is $n \times n \cdot d$. Starting with a distribution vector v concentrated on a single node, L spreads the mass onto the d edges, $\tilde{A}$ permutes the node accordingly, keeping the edge label intact, and finally P collects the mass.

Proof of Theorem 12.9. Let A be the normalized adjacency matrix of X, and B be the normalized adjacency matrix of G. We can view a random step in the derandomized-squaring graph $X \ominus G$ as

$$v \to (v, a) \to (v[a], a) \to (v[a], b) \to (v[a][b], b) \to v[a][b]$$

where a is a random node in G and b is a random neighbor of a in G.

We now define matrices that implement each of the above steps.

The first step is given by the "lift" matrix $L := I_n \otimes (1/k, \ldots, 1/k)^T$ which is $n \cdot k \times n$.

The second step is given by $\tilde{A}$ which is a $n \cdot k \times n \cdot k$ matrix, where $\tilde{A}_{(u,a),(u',a')} = 1$ if and only if $a = a'$ and $u' = u[a]$. This matrix corresponds to taking a step in X *after the choice for the step has been made*. No entropy is added by $\tilde{A}$, which is a permutation matrix.

The third step is given by $\tilde{B} = I_n \otimes B$.

The fourth step is $\tilde{A}$ again.

Finally, the fifth step is given by the "projection" matrix $P := I_n \otimes (1, \ldots, 1)$.

The adjacency matrix M of $X \ominus G$ satisfies

$$M = P\tilde{A}\tilde{B}\tilde{A}L.$$

By Lemma 12.3, $B = (1-\mu)J_k + \mu C$ where $|Cv| \leq |v|$ for all v. It follows that

$$\tilde{B} = I_n \otimes B = (1-\mu)I_n \otimes J_k + \mu I_n \otimes C.$$

Plugging this into the expression for M one gets

$$M = (1-\mu)P\tilde{A}\left(I_n \otimes J_k\right)\tilde{A}L + \mu P\tilde{A}\left(I_n \otimes C\right)\tilde{A}L.$$

One can now observe the following:

(1) $I_n \otimes J_k = L \cdot P$;

(2) $P \cdot \tilde{A} \cdot L = A$, as remarked before the proof; and

(3) $D := P\tilde{A}\left(I_n \otimes C\right)\tilde{A}L$ satisfies $|Dv| \leq |v|$ for every v. This can be shown also using the fact that C satisfies this property as we saw before.

Plugging (1) and (2) in the above expression for M, and the definition of D we get

$$M = (1-\mu)P\tilde{A}LP\tilde{A}L + \mu D = (1-\mu)A^2 + \mu D.$$

Then, by Lemma 12.2 and the triangle inequality for $|.|$:

$$\lambda_2'(X \ominus G) = \max_{v \perp u} \frac{|Mv|}{|v|} \leq \frac{|(1-\mu)A^2 v|}{|v|} + \frac{|\mu D v|}{|v|} \leq (1-\mu)\lambda^2 + \mu.$$

QED

Exercise 12.11. Prove (3). Guideline. For a matrix A define $||A|| := \max_v |Av|/|v|$. This is like λ_2' (see Lemma 12.2), except v does not need to be perpendicular to u. Our goal is to show $||D|| \leq 1$. Prove:

(I) $||A \cdot B|| \leq ||A|| \cdot ||B||$,

(II) $||A \otimes B|| \leq ||A|| \cdot ||B||$. Hint: Write $|A \otimes Bv|^2 = \sum_{iA,iB} \left(\sum_{jA,jB} A_{iA,jA} B_{iB,jB} v_{jA,jB}\right)^2$,

(III) For a permutation matrix Π such as $\tilde{A}$, $||\Pi|| = 1$,

(IV) $||L|| = 1/\sqrt{k}$, $||P|| = \sqrt{k}$.

Combine this to prove $||D|| \leq 1$.

12.4 Proof of Theorem 6.8 that Uconn is in L

We can assume that we are given a graph G and nodes s and t which are connected, because if s and t are not connected, the algorithm we are about to present will never declare them

connected. By Claim 12.1, we can focus on 4-regular graph with sets S and T. Call this 4-regular graph X. To start-up our sequence, let $X_1 := X^t$ for a suitable constant t.

Define the sequence of graphs

$$X_{i+1} := X_i \ominus G_i$$

where each G_i is an expander graph with degree c, $\lambda_2' \leq 1/100$, on a number of nodes equal to the degree of X_i. Such expanders are obtained from Theorem 12.1. The latter gives edge expansion, and we infer spectral expansion by Theorem 12.7. The specific bound $\lambda_2' \leq 1/100$ can be obtained by squaring the graph a few times, using equation (12.6). (To show that UConn is in space $c \log n \log \log n$ the log-degree expanders in Theorem 12.3 suffice.)

The neighbor function of G_i is computable in L (see Exercise 12.3).

Let $\ell := c \log n$. By Theorem 12.9 (see the observation right after its proof) we have $\lambda_2'(X_\ell) \leq 1/10$.

Then we can apply Claim 12.2 and solve UConn by going through all $s \in S$ and checking if one of its neighbors in X_ℓ lies in T.

It remains to verify that we can compute neighbors in X_ℓ in L.

The intuition is: if $v \in X_1$, then the neighbors are $v[a_1]$ where $a_1 \in [d]$; if $v \in X_2$, then the neighbors are $v[(a_1, a_2)] = v[a_1][a_1[a_2]]$ where $a_1, a_2 \in [d]$; if $v \in X_3$, then the neighbors are $v[(a_1, a_2, a_3)] = v[(a_1, a_2)][(a_1, a_2)[a_3]] = v[a_1][[a_1[a_2]][(a_1', a_2')]] = v[a_1][a_1[a_2][a_1'][a_1'[a_2']]]$ where $a_1, a_2, a_3 \in [d]$; etc.

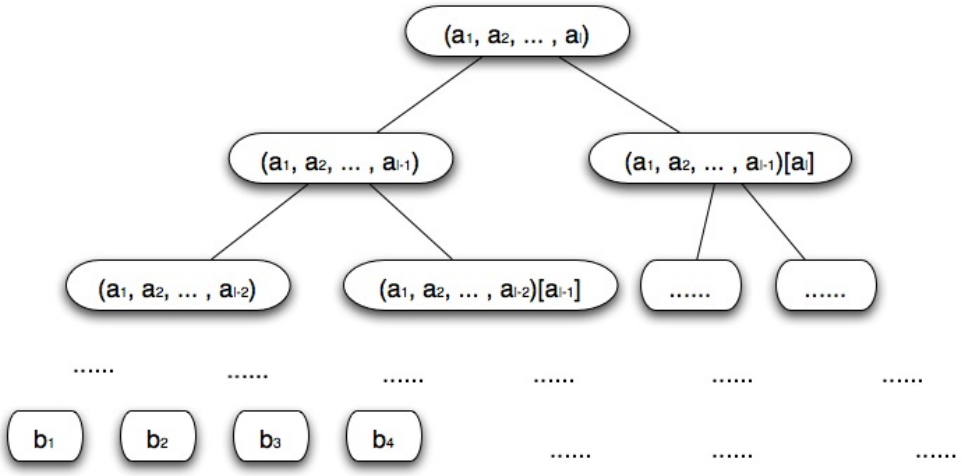


Figure 12.8: Compute b_i's from a_i's.

An edge in X_ℓ is specified by $(a_1, a_2, \ldots, a_\ell)$ where $a_i \in [d]$. To this edge there corresponds a path of length $2^{\ell-1}$ in the graph X_1 with labels $(b_1, b_2, \ldots, b_{2^{\ell-1}})$ where $b_i \in [d]$. The associated neighbor of v in X_ℓ is $v[b_1][b_2] \cdots [b_{2^{\ell-1}}]$. So to compute $v[(a_1, \ldots, a_\ell)]$, we proceed in 2 phases:

1. compute $(b_1, \ldots, b_{2^{\ell-1}})$, and

2. compute $v[b_1][b_2]\cdots[b_{2^{\ell-1}}]$.

We must do this in space about $\log n$, and so we cannot afford to write down the output of the first phase. Instead, the following shows given $(a_1, \ldots, a_\ell)$ and an index $i \leq 2^{\ell-1}$ how to compute $b_i \in [d]$ in space $c \log n$. From this, one can perform Phase 2 one step at the time.

Observe that the the indices b_i are obtained from the indices a_i as in figure 12.8.

So to compute b_i, we just need to go from the root to the leaf b_i in the tree. The space needed for this is just the name of the node in the tree, plus the space needed to compute neighbors in the expander graphs, which is comparable. This completes the proof that UConn is in L.

12.5 Notes

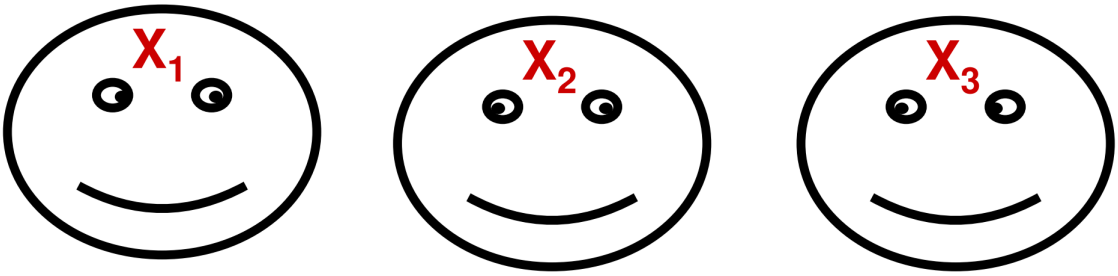
TBD expanders.

The proof of Theorem 6.8 in [234] is similar to the one we presented, but uses different graph operations to reduce the eigenvalue bound. The proof using derandomized graph squaring is from [239]. We follow the presentation in [291]

12.6 Historical vignette: TBD

Chapter 13

Communication



Communication complexity is the study of *rectangles.*

(... alright, let me try to make this sound more fun to the uninitiated; though the ultimate goal is to convince the uninitiated that rectangles are fun.)

Communication complexity is the study of the amount of information that needs to be exchanged among two or more parties (or players) which are interested in reaching a *common* computational goal. The critical difference with interactive proofs is that in communication complexity the parties *cooperate.* By contrast, the setting of cryptographic proofs is *adversarial* or *cryptographic:* The verifier may be interactive with a malicious party. Other than that, the same parameters are studied in both settings, like the number of rounds, the length of the communication (which is a lower bound on the efficiency of the party), etc. But, in communication complexity we can be more basic by disregarding the computational model and instead bestowing unlimited computational power on the parties, and only paying attention to the amount of communication.

13.1 Two parties

We start with the model in which there are only 2 parties, A and B. Their task is to compute a function of two inputs

$$f : X \times Y \to Z$$

where A only knows $x \in X$, and B only knows $y \in Y$. The parties A and B engage in a communication protocol and exchange bits. We can generally assume that the parties

alternate sending a message, and that the last message is the output of the protocol. We say that the protocol uses communication d bits if for every input, A and B exchange $\leq d$ bits. The bits exchanged are called *transcript*.

We can visualize a protocol via a binary tree (13.1). Each node is labeled with a function mapping that party's input to the possible messages.

TBD

Figure 13.1: Protocol tree

13.1.1 The communication complexity of equality

Consider the function Equality : $[2]^n \times [2]^n \to [2]$, Equality$(x, y) = 1 \Leftrightarrow x = y$. Trivially, Equality can be computed with communication $n + 1$: A sends her input to B; B then communicates the value of Equality. The same trivial upper bound holds for any function $f : [2]^n \times [2]^n \to [2]$. We now prove the following lower bound.

Theorem 13.1. Any protocol for equality must exchange at least n bits.

Before proving this theorem, we cover some properties of protocols.

Definition 13.1. A rectangle in $X \times Y$ is a subset $R \subseteq X \times Y$ such that $R = A \times B$ for some $A \subseteq X$ and $B \subseteq Y$. Equivalently, $R \subseteq X \times Y$ is a rectangle if whenever $\{(x, y), (x', y')\} \subseteq R$ then we also have $\{(x, y'), (x', y)\} \subseteq R$.

Exercise 13.1. Prove the equivalence.

The connection between rectangles and protocols is the following.

Lemma 13.1. Let P be a protocol that uses d bits, let $t \in [2]^d$ be a transcript. The set of inputs that induce transcript t is a rectangle.

Proof. Let $A \subseteq X \times Y$ be the set of inputs that induce communication t. Suppose that $(x, y), (x', y') \in A$, we want to show that $(x, y') \in A$ (similarly for (x', y)). We prove by induction on i that the i-th bit exchanged by P on input (x, y') is t_i. Of course this means that the protocol exchanges t on input (x, y') and so $(x, y') \in A$ as desired.

For $i = 1$, the bit sent by A only depends on x, but we know $P(x, y)$ exchanges t_1, so we are done.

For general i, suppose it is A's turn to speak. The bit she sends is a function of x and the communication so far. By induction hypothesis the communication so far is $t_1, \ldots, t_{i-1}$. So A cannot distinguish between (x, y) and (x, y') and will send t_i as next bit.

If it is B's turn to speak, we reason in the same way replacing (x, y') with (x', y'). **QED**

Corollary 13.1. Suppose $f : X \times Y \to [2]$ is computable by a d-bit protocol, then there is a partition of $X \times Y$ in 2^d rectangles, where each rectangle is f-monochromatic: all inputs in the rectangle give the same value of f.

Proof. For each transcript t, consider $R_t :=$ the set of inputs that induce t. R_t is a rectangle by the previous lemma. It is obviously a partition and f-monochromatic. **QED**

Example 13.1. The equality function, seen as a matrix, is the identity matrix. figure 13.2 shows two ways to partition it in monochromatic rectangles. Intuitively, because the ones are only on the diagonal, we need many rectangles in any monochromatic partition.

Figure 13.2: Two ways to partition equality in monochromatic rectangles.

We can now prove the lower bound for equality.

Proof of Theorem 13.1. Assume we can partition $X \times Y$ in equality-monochromatic rectangles. Consider the 2^n inputs (e, e) where $e \in \{0,1\}^n$. Observe that no equality-monochromatic rectangle can contain both (e, e) and (b, b) if $e \neq b$, for else (e, b) is in the rectangle, but since $e \neq b$ this cannot be equality-monochromatic.

Since the rectangles must cover all of the 2^n inputs (e, e), we need $\geq 2^n$ rectangles which implies that any protocol must use at least n bits of communication. **QED**

13.1.2 The power of randomness

We define randomized protocols as a distribution on protocols, similarly to the randomized polynomials in Definition 9.3.

Definition 13.2. A *randomized protocol* P with communication d is a distribution on protocols with A function $f : X \times Y \to Z$ has randomized communication d with error ϵ if there is a randomized protocol P that on every input x computes it correctly w.p. $\geq 1 - \epsilon$, that is, $\mathbb{P}_P[P(x) \neq f(x)] \leq \epsilon$.

The equality function demonstrates the power of randomness in communication:

Theorem 13.2. Equality has randomized protocols with error ϵ and communication $c \log 1/\epsilon$, for any $\epsilon \leq 1/2$.

Exercise 13.2. Prove this.

13.1.3 Public vs. private coins

Our definition of randomized protocols is *public-coin*: the parties share randomness. One can also consider *private-coin* protocols. These are defined like (deterministic) protocols, except that each party's message is a *distribution* on messages.

Exercise 13.3. Suppose that $f : [2]^n \times [2]^n \to [2]$ has a (public-coin) randomized protocol with communication d and error ϵ. Show that f has a private-coin protocol with communication $d + c \log(n/\epsilon)$ and error 2ϵ. Guideline: Use tail bounds and the union bound to show that the public coin protocol needs only be supported on few protocols.

13.1.4 Disjointness

The *disjointness function* $\text{Disj} : [2]^n \times [2]^n \to [2]$ is defined as $\text{Disj}(x, y) = \vee_{i \in [n]} x_i \wedge y_i$. It asks to determine if x and y, viewed as subsets of $[n]$, (do not) intersect. This function is of central importance pretty much for the same reason that 3Sat is: Its simple structure makes it excellent for reductions, as we shall see in section 13.1.7.

Theorem 13.3. The randomized communication complexity of Disj with error ϵ is $\geq c_\epsilon n$.

The hard distribution D is defined as follows for $n = 4m - 1$. First pick a uniform partition of $[n]$ into $(P, Q, \{i\})$ where P (and Q) is a uniform set of size $2m - 1$. Now let X (resp., Y) be a uniform subset of $P \bigcup \{i\}$ (resp. $Q \bigcup \{i\}$) of size m. In particular, the intersection is either empty or a singleton. Note that the distribution is not product; it is known that the communication is $\leq c\sqrt{n}$ on product distributions.

13.1.5 Greater than

Another well-studied function is Greater-Than, where the parties wish to determine if $x > y$ as integers.

Theorem 13.4. The randomized communication complexity of greater-than is $\leq c \log n$.

Proof. We sketch the clever protocol. We perform binary search to find the most significant bit where x and y differ. Each comparison during this binary search corresponds to an equality problem, which as we saw has small randomized communication complexity (Theorem 13.2).

The naive way to implement this search is to set the error to $\leq c/\log n$ in Theorem 13.2. But this won't give overall communication $c \log n$.

Instead, we set the error to constant, and perform *binary search with noisy comparisons*. A random-walk-with-backtrack algorithm shows that $c \log n$ comparisons suffice, leading to the result. The idea is to start each recursive call with a check that the target element is contained in the current interval, and if not backtrack. **QED**

The above bound is tight.

Theorem 13.5. The randomized communication complexity of greater-than is $\geq c \log n$.

13.1.6 Application to TMs

One-tape TMs have efficient randomized communication protocols. This is essentially the same as the crossing-sequence argument we saw in section §16.5.

Theorem 13.6. For a function $f : [2]^n \times [2]^n \to [2]$ consider the padded function $p_f : [2]^{3n} \to [2]$ defined as $p_f(x0^n y) = f(x, y)$. If p_f is computable by an s-state TM in time t then f has randomized protocols with communication $c(\log s)t/n$ and error $\leq 1/2$.

Proof. For $i \in [n]$, define the protocol P_i as follows: A is in charge of the first $n + i$ cells (which include x); B is in charge of last $n + (n - i)$ cells (which include y). They simulate the TM in turn, communicating $\log s + c$ bits whenever the TM crosses the boundary of the $(n + i)$-th cell. These bits represent the state of the machine or a special symbol denoting that the computation is over with final state s, from which the value of the function can be determined. The parties carry this simulation for up to $(t/n)/(c \log s)$ crossings. If the TM hasn't stopped they stop and output, say, 0.

The distribution on protocols is P_I where I is uniform in $[n]$. **QED**

We will soon exhibit functions which require linear randomized communication, recovering the quadratic impossibility results for TMs from section §16.5. In fact, we will show stronger results.

13.1.7 Application to streaming

tbd

13.2 Number-on-forehead

There are various ways in which we can generalize the 2-party model of communication complexity to $k > 2$ parties. The obvious generalization is to let k players compute a k-argument function $f(x_1, \ldots, x_k)$ where the i-th party only knows the i-th argument x_i. This model is known as "number-in-hand" and useful in some scenarios, but we will focus on a different, fascinating model which has an unexpected variety of applications: the "Number on the Forehead" model . Here, again $f(x_1, \ldots, x_k)$ is a Boolean function whose input is k arguments, and there are k parties. The twist is that the i-th party knows all inputs except x_i, which we can imagine being placed on his forehead. Communication is broadcast.

The grand challenge here is to give an explicit function $f : \overbrace{[2]^n \times ... \times [2]^n}^{k} \to [2]$ that cannot be computed with $k := 2 \log n$ parties exchanging k bits. This would have many applications, one of which is described next.

13.2.1 An application to ACC

Functions computable by small ACC (recall section §9.5) have low communication complexity:

Lemma 13.2. AC[d] on n bits of size n^d and depth d have equivalent protocols with $\log^{c_d} n$ parties communicating $\log^{c_d} n$ bits, for any partition of the input bits.

Proof. By Lemma 9.9 it suffices to prove it for depth-2 circuits consisting of a symmetric gate on s And gates of fan-in t, where s and t are $\leq \log^{c_d} n$. Fix an arbitrary partition of the input in $t+1$ sets $x_1, \ldots, x_{t+1}$. All that the players need to compute is the number of And gates that evaluate to 1. Consider any And gate. Since it depends on at most t variables, it does not depend on the bits in one of the sets, say x_j. Then the j-th party can compute this And without communication. So let us partition the And gates among the parties so that each party can compute the gates assigned to them without communication. Each party evaluates all the And gates assigned to them privately and broadcasts the number $\leq s$ of these gates that evaluate to 1. This takes a total of $ct \log s$ bits. **QED**

13.2.2 Generalized inner product is hard

In this section we prove an impossibility result for computing the generalized inner product function $GIP : ([2]^n)^k \to [2]$:

$$\mathrm{GIP}(x_1, \ldots, x_k) := \sum_{i=1}^{n} \bigwedge_{j=1}^{k} (x_j)_i \mod 2.$$

In fact, we shall bound even the correlation between GIP and k-party protocols exchanging d bits, denoted $\mathrm{Cor}(\mathrm{GIP}, d\text{-bit } k\text{-party})$. Recall from section §8.4 that this is defined as the maximum of $|\mathbb{E}_x e[\mathrm{GIP}(x) + f(x)]|$ for any protocol f with corresponding parameters, where x is uniform and $e(z) := (-1)^z$. By (the easy direction of) Corollary 8.1, this implies that the randomized communication complexity of GIP is large. Let us spell out again this implication: Having randomized communication complexity $\leq d$ with small error means that there is a distribution of protocols with communication d s.t. on every input x, a randomly selected protocol achieves error $\leq \epsilon$. From this, we can average over x, and then fix a protocol to obtain small correlation. Because we prove next that small correlation is impossible, it follows that the randomized communication complexity is large too.

Theorem 13.7. $\mathrm{Cor}(\mathrm{GIP}, d\text{-bit } k\text{-party}) \leq 2^d \cdot 2^{-cn/4^k}$.

To prove the theorem we associate to any function a quantity $R(f) \in \mathbb{R}$ enjoying the following two lemmas:

Lemma 13.3. $\mathrm{Cor}(f, d\text{-bit } k\text{-party}) \leq 2^d \cdot R(f)^{1/2^k}$, for any $f : X_1 \times ... \times X_k \to [2]$.

Lemma 13.4. $R(\mathrm{GIP}) \leq 2^{-cn/2^k}$.

The combination of these two facts proves Theorem 13.7.

Intuition for $R(f)$: Think of $k = 2$; we saw that any 2-party d-bit protocol partitions the inputs in 2^d f-monochromatic rectangles. How about we check how well f can be so partitioned? Instead of picking an arbitrary rectangle, let us pick one in which each side has length 2, and see how balanced the function is there. If a "good" partition exists, with somewhat high probability our little rectangle should fall in a monochromatic rectangle, and we should always get the same values of f. Otherwise, we should get mixed values of f.

Specifically, for $k = 2$,

$$R(f) := \mathbb{E}_{\substack{x_1^0, x_2^0 \\ x_1^1, x_2^1}} e\left[f(x_1^0, x_2^0) + f(x_1^0, x_2^1) + f(x_1^1, x_2^0) + f(x_1^1, x_2^1)\right] \in \mathbb{R}.$$

In general, for any k:

$$R(f) := \mathop{\mathbb{E}e}_{\substack{x_1^0, \ldots, x_k^0 \\ x_1^1, \ldots, x_k^1}} \left[\sum_{\varepsilon_1, \ldots, \varepsilon_k \in [2]} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k})\right] \in \mathbb{R}.$$

Exercise 13.4. Prove:

$R(f) \geq 0$ for every f.

$R(f) = 1$ for constant f.

$\mathbb{E}_F R(F) = 1 - (1 - 2^{-n})^k \leq k/2^n$ for uniform $F : ([2]^n)^k \to [2]$. Hint: The inequality is Fact B.8.

13.2.3 Proof of Lemma 13.3

We prove this theorem via a sequence of claims.

Definition 13.3. A function $g_i : X_1 \times \ldots \times X_k \to [2]$ is a *cylinder* in the i-th dimension if $\forall (x_1, \ldots, x_k)$ and x_i' we have $g_i(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_k) = g_i(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k)$. A set $S \subseteq X_1 \times \ldots \times X_k$ is a *cylinder intersection* if $\exists$ cylinders $g_1, \ldots, g_k$ such that $S = \{x : \prod g_i(x) = 1\}$.

Recall we saw that a 2-party protocol partitions the input in monochromatic rectangles. The following extension of this fact to k parties is via cylinder intersections.

Claim 13.1. Any d-bit k-party protocol for $f : \overbrace{[2]^n \times \ldots \times [2]^n}^{k} \to [2]$ partitions the inputs in 2^d f-monochromatic cylinder intersections.

Proof. Fix a transcript t, and consider the set A_t of inputs yielding that transcript. We claim that A_t is a cylinder intersection. To see this, consider the cylinder functions $g_i(x) = 1 \Leftrightarrow$ "From the point of view of the i-th party, x could yield transcript t" $\Leftrightarrow \exists x_i'$ such that $P(x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_k)$ yields transcript t.

Obviously if x is in A_t then $g_i(x) = 1$ for all i.

To see the converse, take some input $x = (x_1, \ldots, x_k)$ such that $g_i(x) = 1$ for all i. This means that $\exists (x'_1, \ldots, x'_k)$ such that
$(x'_1, x_2, \ldots, x_k)$ yields t;
$(x_1, x'_2, \ldots, x_k)$ yields t;
... ...
$(x_1, x_2, \ldots, x'_k)$ yields t.
We must show that x yields t as well, i.e. $x \in A_t$. This is argued by induction on the bits in t, using the same "copy and paste" argument that was used for $k = 2$. **QED**

Using the notion of cylinder intersections we can now relate an arbitrary protocol to a special class of protocols p^*. Each protocol p^* can be written as $p^*(x) = \sum g_i(x) \ \ mod\ 2$, where g_i is a cylinder in i-th dimension. This corresponds to each party sending just one bit independently of the others, and the output of the protocol being the XOR of the bits. Note the communication parameter is not present anymore. We write Cor^* for the corresponding correlation, where k is given by the context.

Claim 13.2. $\mathrm{Cor}(f, d - \mathrm{bit}) \leq 2^d \cdot \mathrm{Cor}^*(f)$.

Proof. We use a general trick to turn products $\overbrace{\prod_i g_i(x) = 1}^{cylinder\ \ intersection}$ into sums $\overbrace{\sum g_i(x) \ \ mod\ 2}^{p^*}$.

Fix any d-bit protocol, let $\{x : \prod_i g_i^1(x) = 1\}, \ldots, \{x : \prod_i g_i^D(x) = 1\}$ be the corresponding $D := 2^d$ f-monochromatic cylinder intersections (by the previous claim). Observe that for a fixed x,

$$\mathop{\mathbb{E}}_{y_1,\ldots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1(x)} \cdot (y_2)^{1+g_2(x)} \cdot \ldots \cdot (y_k)^{1+g_k(x)} \right] = \begin{cases} 1 & \text{if } \exists i : g_i(x) = 0 \\ 0 & \text{if } \forall i : g_i(x) = 1. \end{cases}$$

Therefore,

$$e(p(x)) = \sum_{i=1}^{D} r(i) \mathop{\mathbb{E}}_{y_1,\ldots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \ldots \cdot (y_k)^{1+g_k^i(x)} \right]$$

where $r(i) \in \{-1, 1\}$ is the value of the protocol on the i-th cylinder intersection. Note that for any x exactly one expectation will be 1, the one corresponding to the cylinder intersection where x lands. So we have:

$$\begin{aligned}
&\mathbb{E} e[f(x) + p(x)] \\
&= \mathbb{E}_x [e(f(x)) \cdot e(p(x))] \\
&= \mathbb{E}_x \left[e(f(x)) \cdot \textstyle\sum_{i=1}^{D} r(i) \mathop{\mathbb{E}}_{y_1,\ldots,y_k \in \{-1,1\}} \left[(y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \ldots \cdot (y_k)^{1+g_k^i(x)} \right] \right] \\
&= \textstyle\sum_{i=1}^{D} \mathbb{E}_{x, y_1,\ldots,y_k \in \{-1,1\}} \left[e(f(x)) \cdot r(i) \cdot (y_1)^{1+g_1^i(x)} \cdot (y_2)^{1+g_2^i(x)} \cdot \ldots \cdot (y_k)^{1+g_k^i(x)} \right] \\
&\leq D \cdot \mathbb{E}_{x, y_1,\ldots,y_k \in \{-1,1\}} \left[e(f(x)) \cdot r(i) \cdot (y_1)^{1+g_1^{i*}(x)} \cdot (y_2)^{1+g_2^{i*}(x)} \cdot \ldots \cdot (y_k)^{1+g_k^{i*}(x)} \right],
\end{aligned}$$

where $i*$ is the value of i that gives the largest summand. Now fix $y_1, \ldots, y_k$ to maximize the expectation, and let $J \subseteq \{1, \ldots, k\}$ be the indices corresponding to $y_j = -1$, i.e., $j \in J \Rightarrow y_j = -1$. The last expression above is

$$\begin{aligned}
&D \cdot \mathbb{E}_x \Big[e(f(x)) \cdot \prod_{j \in J} (-1)^{1+g_j^{i*}(x)} \Big] \\
&= D \cdot \mathbb{E}_x e \left[f(x) + \sum_{j \in J} \big(1 + g_j^{i*}(x)\big) \right] \\
&\leq D \cdot \mathrm{Cor}^*(f).
\end{aligned}$$

QED

Claim 13.3. $\mathbb{E} e_x[g(x)] \leq R(g)^{1/2^k}$ for every function $g := X_1 \times \ldots \times X_k \to [2]$.

Proof. Recall that for every random variable X: $\mathbb{E}[X^2] \geq \mathbb{E}[X]^2$, Fact B.10. Also recall that if X, X' are independent then $\mathbb{E}[X \cdot X'] = \mathbb{E}[X] \cdot \mathbb{E}[X']$.

Applying the "squaring trick:"

$$\mathbb{E}_{x_1,\ldots,x_k} e[g(x_1, \ldots, x_k)]^2 = \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k} e[g(x_1, \ldots, x_k)]]^2 \leq \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k} e[g(x_1, \ldots, x_k)]^2]$$

$$= \mathbb{E}_{x_1,\ldots,x_{k-1}} [\mathbb{E}_{x_k^0, x_k^1} e[g(x_1, \ldots, x_{k-1}, x_k^0) + g(x_1, \ldots, x_{k-1}, x_k^1)]].$$

The lemma follows by repeating this k times. **QED**

Claim 13.4. For every function $f : X_1 \times \ldots \times X_k \to [2]$, and every protocol* p^*,

$$R(f \oplus p^*) = R(f),$$

where $f \oplus p^*$ simply is the function whose output is the XOR of f and p^*.

Proof. Suppose $p^*(x) = g_1(x) + \ldots + g_k(x)$, where g_i is a cylinder in the i-th dimension. We show $\forall f, R(f \oplus g_k) = R(f)$; the same reasoning works for the other coordinates. Note for every x,

$$\begin{aligned}
&\sum_{\varepsilon_1,\ldots,\varepsilon_k \in [2]} (f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + g_k(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k})) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + \sum_{\varepsilon_1,\ldots,\varepsilon_k} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + \sum_{\varepsilon_1,\ldots,\varepsilon_k} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{0}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) + 2 \sum_{\varepsilon_1,\ldots,\varepsilon_{k-1}} g_k(x_1^{\varepsilon_1}, \ldots, x_k^{0}) \\
&= \sum_{\varepsilon_1,\ldots,\varepsilon_k} f(x_1^{\varepsilon_1}, \ldots, x_k^{\varepsilon_k}) \mod 2,
\end{aligned}$$

where the second equality holds because g_k does not depend on x_k. **QED**

The straightforward combination of the claims in this section proves Lemma 13.3.

13.2.4 Proof of Lemma 13.4

We have:

$$R(\text{GIP}) = \mathop{\mathbb{E}e}_{\substack{x_1^0,\ldots,x_k^0\\ x_1^1,\ldots,x_k^1}} \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k\in\{0,1\}} \sum_i \prod_j (x_j^{\varepsilon_j})_i \right] = \mathbb{E} \prod_i e \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k} \prod_j (x_j^{\varepsilon_j})_i \right]$$

$$= \mathbb{E}e \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k} \prod_j (x_j^{\varepsilon_j})_1 \right]^n = R\left(\bigwedge_k\right)^n,$$

using in the last equality the fact that any two independent random variables X, Y satisfy $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$, and where $\bigwedge_k$ is the AND function on k bits.

To save in notation let us replace $(x_1^0)_1, \ldots, (x_k^0)_1$ with $(y_1^0, \ldots, y_k^0)$, where $(y_i^0) \in [2]$; and similarly for $(x_1^1)_1, \ldots, (x_k^1)_1$. So we have:

$$R(\text{GIP}) = \mathop{\mathbb{E}e}_{\substack{y_1^0,\ldots,y_k^0\\ y_1^1,\ldots,y_k^1}} \left[\sum_{\varepsilon_1,\ldots,\varepsilon_k\in\{0,1\}} \prod_j y_j^{\varepsilon_j} \right]^n.$$

Suppose that $y_1^0 \neq y_1^1, \ldots, y_k^0 \neq y_k^1$; then there exists exactly one choice of $\varepsilon_1, \ldots, \varepsilon_k$ making $\prod_j y_j^{\varepsilon_j} = 1$ (recall that $y_j^{\epsilon} \in [2]$; if any one of them is 0 the whole product is zero), and consequently

$$e\left(\sum_{\varepsilon_1,\ldots,\varepsilon_k} \prod_j y_j^{\varepsilon_j} \right) = e(1) = -1.$$

We have $y_1^0 \neq y_1^1, \ldots, y_k^0 \neq y_k^1$ with probability 2^{-k}. Therefore:

$$R(\text{GIP}) = \mathbb{E}e \left[\sum \prod_j y_j^{\varepsilon_j} \right]^n \leq (-1 \cdot 2^{-k} + 1 \cdot (1 - 2^{-k}))^n = (1 - 2^{-k+1})^n \leq e^{-cn/2^k}.$$

Exercise 13.5. (1) Rewrite the proof of the GIP correlation bound Theorem 13.7 in the case $k = 2$ of inner product $\text{IP}(x, y) := \sum_i x_i y_i \mod 2$, simplifying the notation.

(2) Optimize the constant in the exponent.

(3) Derive the following basic result. Let D be the uniform distribution over a rectangle $X \times Y \subseteq [2]^n \times [2]^n$ where $X = Y = 2^{\alpha n}$. Show $\mathbb{E}e[\text{IP}(D)] \leq c \cdot 2^{(2(1-\alpha)-c)n}$. Show for $\alpha = 1/2$ there is a rectangle for which the expectation is 1.

Exercise 13.6. Formally state and prove, using Theorem 13.7, that TMs running in subquadratic time do not correlate with IP.

13.3 Any partition

The results in the previous sections, in particular the correlation bound for GIP in Theorem 13.7 apply to a specific partition of the input in foreheads.

Exercise 13.7. Give a different partition of the input in foreheads s.t. GIP has constant communication.

To generalize these impossibility results to any partition is natural and has a number of applications. It turns out that the same parameters for GIP can be achieved under any partition.

Theorem 13.8. There is an explicit function $h : [2]^n \to [2]$ s.t. for any partition of the input into k sets of equal size, $\mathrm{Cor}(h, d\text{-bit } k\text{-party}) \leq 2^d \cdot 2^{-n/c^k}$.

The construction is a hyper-edge analogues of GIP, and the analysis shows that in every partition we can find a large instance of GIP. In more detail, the construction of h is based on *expander graphs*. The expansion property that suffices is that any 2 sets of size $\geq t$ are connected, closely related to edge expansion Definition 12.1. From any graph, one constructs the hypergraph where the edges are the subsets of size k where one node is adjacent to all other $k-1$. Using the expansion property one can iteratively find

$$\geq \frac{n-kt}{d}$$

disjoint hyperedges, each intersecting each part. We'd like t and d to be as small as possible, and suitable expander graphs of degree d are $cn/\sqrt{d}$-interconnected. Hence setting $d := ck^2$ we find $\geq n/k^c$ disjoint hyperedges. Defining h to be the parity over hyperedges of the Ands over the bits in the hyperedge, we conclude by the GIP bound Theorem 13.7.

13.4 The power of logarithmic players

The impossibility results in the previous sections are effective when the number of players is $k \leq c \log n$, but useless when $k \geq \log n$. We now show that this is for a good reason: there are efficient protocols for large k. For generalized inner product this is unsurprising, since the function is almost always 0 for large k. But this is not clear if we replace, say, And with Majority. In fact, surprisingly there is a general protocol that works for many such composed functions. For functions $f : [2]^n \to [2]$ and $g : [2]^k \to [2]$ we consider computing $f \circ g^{(k)}$ whose input is a $k \times n$ matrix M and the output is obtained by computing g on each of the n columns, and then evaluating f on that. Here player j has row j of M on the forehead.

We first show the seminal result that there are efficient protocols whenever f and g are both symmetric.

Theorem 13.9. Let $k \geq \log n + 2$. There is a simultaneous k-party protocol with communication $c \log^3 n$ s.t. given a $k \times n$ matrix M (player j sees all M except row j) computes $(y_0, y_1, \ldots, y_k)$ where y_i is the number of columns in M with weight i.

In particular, $f \circ g^{(k)}$ has simultaneous protocols with the same efficiency in case both f and g are symmetric.

Proof. We prove this for $k = \lceil \log n + 2 \rceil$ (see Exercise 13.8). Each player j communicates the number $a_j(i)$ of columns that they see having weight i, for every i. This takes communication $ck \log n = c \log^2 n$ per player.

We claim that the $a_j(i)$ uniquely specify the y_i, which concludes the proof.

To show this, note

$$b_i := \sum_j a_j(i) = (k-i)y_i + (i+1)y_{i+1}$$

for $i < k$. This is because a column with weight i will be seen as a column of the same weight i by $k - i$ players (those missing a 0) while a column of weight $i + 1$ will seen as having weight just i but $i + 1$ players (those missing a 1).

We in fact claim that even the b_i uniquely specify the y_i. To verify this, assume towards a contradiction that there are y_i and y'_i for $i \leq k$ that satisfy these equations, are non-negative, and have the same sum, n, and for some i we have $y_i \neq y'_i$. We derive a contradiction as follows. Let $d_i := y_i - y'_i$. Since both the y_i and the y'_i satisfy the equations above, we get for $i < k$

$$(k-i)d_i + (i+1)d_{i+1} = 0.$$

Hence

$$d_i = -\frac{k-(i-1)}{i} d_{i-1} = (-1)^i \binom{k}{i} d_0.$$

We know that $d_0 \neq 0$ (for else by above d_i would be 0 too, but we assumed it is not) and in fact $d_0 \geq 1$ since it is an integer. Also note that $y_i + y'_i \geq |y_i - y'_i| = |d_i|$.

Hence we obtain the following contradiction

$$2n = \sum_{i=0}^{k} y_i + y'_i \geq |d_i| \geq \sum_{i=0}^{k} \binom{k}{i} = 2^k > 2n.$$

QED

Exercise 13.8. Explain why it indeed suffices to consider $k = \lceil \log n + c \rceil$.

In the next result, g can be arbitrary. Compared to Theorem 13.9, we obtain a worse communication bound and require interaction. However, both shortcomings can be addressed by an extension of the proof (see Problem 13.1 and Exercise 13.9). I have chosen the simplest exposition giving the main takeaway, which is that when $k = \log^c n$ the communication is $\log^c n$.

Theorem 13.10. Let $k \geq \log n + 2$. For any symmetric $f : [2]^n \to [2]$ and $g : [2]^k \to [2]$ there is a k-party protocol with communication k^c for $f \circ g^{(k)}$.

Proof. Let M be the $k \times n$ input matrix. For $v \in [2]^k$ denote by n_v the number of columns equals to v. It suffices to compute

$$\sum_{v : g(v)=1} n_v \tag{13.1}$$

because f is symmetric.

Let us assume that the players know a "base" vector $u \in [2]^k$ with $n_u = 0$. From this, they can compute any n_v as follows. Consider a path from v to the base vector u: $w_0 := v \to w_1 \to w_2 \to \ldots \to w_s := u$ where w_i and w_{i+1} differ in exactly one bit and $1 \leq s \leq k$. Note these paths only depend on u.

Then, telescopically:

$$n_v = \sum_{i \in [s]} (-1)^i \left(n_{w_i} + n_{w_{i+1}} \right), \tag{13.2}$$

using that $n_{w_s} = n_u = 0$. Note there is player that can compute $n_{w_i} + n_{w_{i+1}}$: since w_i and w_{i+1} differ in exactly one position h, player h can communicate the number of columns which agree in all other positions.

To compute equation (13.1) each player will communicate the sum over $v : g(v) = 1$ of their terms in equation (13.2). The total sum is $\leq 2^k kn$, so $k + \log kn$ bits suffice.

There remains to compute u. Player 1 can simply communicate a string $u' \in [2]^{k-1}$ that does not occur as! a column in matrix M with the first row removed. This takes $\leq k$ bits. Such a u' exists because $2^{k-1} > n$. In particular, $u := 0u' \in [2]^k$ does not occur and so $n_u = 0$. **QED**

The protocol in Theorem 13.10 requires interaction. We now explain how we can make it non-interactive, a.k.a. simultaneous. The main tool is the following.

Exercise 13.9. Give a simultaneous version of Theorem 13.10, with communication $c \log^3 n$.

13.4.1 Pointer chasing

We consider the basic problem of following a path in a directed graph. We have k layers, with edges going from nodes in layer i to nodes in layer $i + 1$ only. The last layer does not have edges but labels in [2] for each node. The goal is to compute the label at the node reached from a start node in Layer 1. (Equivalently, instead of labels we can allow for $k + 1$ layers with the last layer consisting of two nodes only, and the task is outputting the node reached.)

It is convenient to work with a version of this problem where we output m labels, and where the size of each layer grows by a factor b. Note for $m = 1$ this is a boolean function. We shall show that the communication is at least b.

Formally, the input to the pointer-chasing function $G_k^{m,b}$ is a layered graph as above, where Layer i has mb^i nodes, and each node has outdegree 1. In other words, the input are functions g_i for $i \in [k]$ where $g_i : [mb^i] \to [mb^{i+1}]$ for $i \in [k]$, and $g_{k-1} : [mb^{k-1}] \to [2]$.

Exercise 13.10. Let $m = 1$. Give a simultaneous protocol with communication cb.

The next theorem implies that the communication is $\geq c_k mb$.

Theorem 13.11. Let P be a k-party one-way protocol using communication $\leq c_k mb$. Then $\mathbb{P}_x[P(x) \neq G_k^{m,b}] \geq c_k^m$.

For example, for $m = 1$ and fixed k we obtain a tight bound of b up to constant factors.

The total input length is $n \leq kb^{k-1}$. Thus for constant k one needs communication $\geq c_k n^{1/(k-1)}$. One can work out the dependence on k and show that the bound remains non-trivial for any $k \leq \log^c n$ where n is the total input length.

Proof. We proceed by induction on k. For every k we prove the statement for any setting of m, b. The base case $k = 1$ is clear: $P(x)$ is a fixed string, while G is a uniform string in $[2]^m$. The error probability is $\geq 2^{-m}$.

For the induction step, let P use communication t and $p := \mathbb{P}_x[P(x) \neq G_k^{m,b}]$. Write $x = (x_1, y)$ where x_1 is on the forehead of the first party. We have

$$\mathbb{P}_y \left[\mathbb{P}_{x_1}[P(x) \geq G_k^{m,b}] \geq p/2 \right] \geq p/2.$$

Let P_a be the protocol P where the first party always communicates string a, regardless of y. We claim there exists a s.t.

$$\mathbb{P}_y \left[\mathbb{P}_{x_1}[P_a(x) = G_k^{m,b}] \geq p/2 \right] \geq 2^{-t} p/2.$$

Note that the probability over x_1 is not reduced because the first party's message does not depend on x_1.

Now let $m' := mb$ and define protocol P' for $G_{k-1}^{m',b}$. On input y, P' runs P_a for r times on inputs (x^i, y) for $i \in [r]$, where the x^i are independent choices for the first party's input. For any of the m' bits that are pointed to by some x^i, P' outputs the corresponding bit. In case different runs give different values, the answer can be arbitrary. For any bit that is not pointed by any x^i, P' guesses at random. This gives a randomized protocol; one can fix the randomness and preserve the success probability.

The communication of P' is $\leq rt$.

To analyze the success probability. Fix any y for which $\mathbb{P}_{x_1}[P_a(x) = G_k^{m,b}] \geq p/2$. The probability that all the r runs are correct is $\geq (p/2)^r$. The probability that there are $\geq cm'$ bits that are not pointed to by some x^i is at most the probability that there is a set of size cm' s.t. mr pointers fall there, which is at most

$$\leq \binom{m'}{cm'} (1-c)^{mr} \leq c^{m'} c^{-mr} \leq c^{m'},$$

for $r \geq cb$.

When that does not happen, the random guesses will be correct w.p. $\geq 2^{-cm'}$.

Overall, the success probability over uniform y is

$$\geq 2^{-t} p/2 \cdot ((p/2)^r - c^{m'}) \cdot 2^{-cm'}.$$

For $p \geq 2^{-cm}$ and $t \leq cm'$, the overall success probability is $\geq c^{m'}$. **QED**

In the case $k = 3$ according to Theorem 13.11 we need communication $\geq c\sqrt{n}$. As mentioned above, this is tight for G, because of the way the layers are constructed. But one

can consider the natural question of chasing pointers where each layer (except the first) has n nodes. It is a tantalizing open question whether there is a protocol with communication $\leq c\sqrt{n}$. The trivial protocol takes communication n, and one might wonder if that's tight. But a clever protocol achieves sublinear communication.

13.4.2 Sublinear communication for 3 player

For $k = 3$ we consider pointer chasing on layers of sizes $1, n, n$. Define G as

$$G(i, g, h) := h(g(i))$$

where $i \in [n]$,$g : [n] \to [n]$, and $h : [n] \to [2]$.

We consider the even more restricted *simultaneous* communication model where the players speak once, non-interactively. Naive intuition suggests that linear communication might be needed. In fact, such bounds were claimed several times, but each time the proof only applied to special cases. Indeed, we have:

Theorem 13.12. G above has simultaneous communication $o(n)$.

Proof. TBD Add details

We sketch the ideas in case g is a permutation π. Let H be a bipartite graph H between the n nodes in the middle layer and the n nodes in the last. For any permutation π, let $G_{H,\pi}$ denote the graph on the n last nodes where $\{x, y\}$ is an edge iff $\pi^{-1}(x)$ has an edge to y in H.

The main claim is that there is H of degree $d = (1 + \epsilon)pn$ s.t. for any π $G_{H,\pi}$ has a partition in $r = o(n)$ sets s.t. any two nodes in the same set are connected in $G_{H,\pi}$. We call the sets *cliques.* Note any graph has a trivial partition consisting of n singletons.

The protocol is as follows. H is known to all.

Player 1, for each of the r cliques, announces the parity of the bits $h(x)$ for x in the clique.

Player 2 announces $h(x)$ for all the d neighbors x of i in H. This is d bits.

Player 3 knows $k := \pi(i)$. It considers the clique of $G_{H,\pi}$ containing k. It knows the parity of the $h(x)$ for x in this clique. Also, for any x in the clique, x and k are connected, hence $\pi^{-1}(k) = i$ is adjacent to x. So from the message of Player 2 we know $h(x)$. We can subtract off all these bits to get $h(k)$.

As stated, this protocol is not simultaneous. To make it simultaneous, let Player 3 announce which of the cliques k is in, and also which of the d neighbors of i are connected via H to nodes in that clique that are not k. Then the referee has a bit per clique, knows which bit to look at, and knows which bits of Player 2 to consider.

Player 3 message takes $\log r + d$.

The existence of H with suitable parameters can be established by the probabilistic method. Specifically, let H be distributed as $G(n, p)$, a random graph where each edge is present independently with prob. p. We observe that for any permutation $G_{H,\pi}$ is random

from $G(n, p^2)$. Thus its complement $\overline{H}$ is random from $G(n, 1-p^2)$. One can show that w.h.p. $\overline{H}$ has chromatic number at $\leq r = o(n)$. This implies that H has a covering is equal to the minimum number of independent sets that you need to cover the nodes of the graph. From this the result follows. **QED**

This protocol can be generalized to more players.

Exercise 13.11. TBD Prove that if $\overline{H}$ has chromatic number r then we can partition the nodes of H in r sets so that

13.5 Problems

Problem 13.1. Extend the proof of Theorem 13.10 to obtain:

(1) The same bound for the more general case of $f(g_1, g_2, \ldots, g_k)$ where the g_i may be different.

(2) An improved bound of $\log^c n$ for any k.

13.6 Notes

"Because it is *basic*."

Communication complexity was initiated in [315] (to whose author the quote is credited, according to [305]). Exercise 13.3 is from [214].

Several proofs of Theorem 13.3 exist [162, 233, 38], see the books [179, 229] or the survey [63] for two different expositions. For more on disjointness see also the survey [250].

Theorem 13.4 is from [215]. For the random-walk with backtrack, see [86]. The matching lower bound, Theorem 13.5, is from [298].

The number-on-forehead model is from [61]. Computing degree-k polynomials with $k+1$ parties, used in the proof of Lemma 13.2, is from [134].

Theorem 13.7 is from [34]. Several presentations of the proof exist: [69, 230, 303]. We followed the latter.

The any-partition result, section §13.3, is from [137].

Theorem 13.10 is from [7] but our presentation has some minor differences. The result can be extended in various ways, see [7] and [127]. Theorem 13.9 is from [31], and is the first amazing protocol in this line of works.

The result can be extended in various ways, such as making the protocol simultaneous, applying different functions g_i to each column, allowing for more complicated matrix structures, and so on. The first amazing protocol in this line of works is from [31] and applies when g is symmetric too.

A proof of Theorem 13.11 in the case $k = 3$ appeared in [32] but did not readily extend to larger k. The proof we presented is a streamlined version of the argument in [304]. The latter paper works with trees instead of graphs to obtain slightly better parameters at the cost of a slightly more involved analysis of the number of bits hit by pointers.

The main idea in Theorem 13.12 for permutations, which we sketched, is from [227]. The extension to general functions is from [55]. These works don't quite give simultaneous protocols though.

Regarding randomness vs. determinism in number-on-forehead protocols, a non-explicit linear separation is in [43]. This work only contains a weaker explicit separation. An explicit, power separation is in [168]. A candidate for an explicit linear separation the problem of deciding if $xyz = 1_G$ for a group G. With randomness, this can be solved with constant communication, for any group. (Show this!) Without randomness, there are groups where this requires $c \log \log |G| \geq c \log n$ communication, see [300], quantitatively the same explicit separation in [43].

Chapter 14

Algebraic complexity

$$\sum_{S\subseteq[n],|S|=d}\prod_{i\in S}x_i = (-1)^n \sum_{j_1,j_2,\ldots,j_d\geq 0:\sum_{k=1}^d kj_k=d}\prod_{k=1}^n \frac{(-1)^{j_k}}{j_k!k^{j_k}}\left(\sum_{i\in[n]}x_i^k\right)^{j_k}.$$

Stepping back, previous chapters have investigated the complexity of computing strings of length 2^n, corresponding to the truth-table of functions from $[2]^n$ to $[2]$ starting from basic strings (or functions) and changing them via simple operations. For example for boolean circuits the basic functions are the constants $0, 1$ and the variables x_i, and we combine them via And/Or/Not gates.

It is natural to consider other objects and to allow for different operations. In fact, we have already encountered other models which are more algebraic, like polynomials and matrices. In this chapter we explore more algebraic models, and in particular we consider computing other objects, namely polynomials.

Perhaps unexpectedly, the development of this theory closely parallels that of its boolean counterpart. We will encounter again many of the main themes and results seen so far, including depth reduction, impossibility results for small-depth models that are "just short" of proving major separations, the grand challenge, reductions, completeness, the surprising power of restricted models, and an algebraic analogue of NP.

14.1 Linear transformations

A very simple algebraic model consists of *Xor circuits*, made only of Xor gates on 2 bits. Obviously such circuits only compute *linear functions* $M : [2]^n \to [2]^m$. Note that any such linear function can be computed using $\leq mn$ wires. Similarly to Theorem 2.3, most linear functions require about that many wires. The challenge is to come up with an explicit linear function requiring many wires. Again similarly to the boolean setting (see section §9.3.2) we do not know of explicit linear functions requiring a super-linear number of wires, even for log-depth circuits (for example for $m = cn$). In this setting, the techniques developed earlier (Theorem 9.5) relate this quest to that of *rigid matrices*.

Theorem 14.1. Let $C : [2]^n \to [2]^{an}$ be a circuit with an wires and depth $a \log n$. Then the matrix M_C corresponding to the linear transformation computed by C can be written as

$$M_C = L + E$$

where L has rank $\leq c_a n / \log \log n$, and E has $\leq n^{1.01}$ non-zero entries.

In other words, C computes a linear function that is close to a low-rank function L. The matrix E gives the errors.

Exercise 14.1. Prove Theorem 14.1 by going through the proof of Theorem 9.5 and explaining what changes are needed.

As before, we can consider small-depth circuits, with unbounded fan-in Xor gates. The trivial upper bound of cnm wires mentioned above can be implemented in depth 1.

A natural candidate hard linear transformation is given by good $(n, an, bn)_2$ codes $C : [2]^{an} \to [2]^n$. Recall this means that a and b are constants independent of n, see Exercise **??**.(1). The fact that such codes can be linear is in Problem 11.4.

Exercise 14.2. Prove that any depth-1 Xor circuits computing an $(n, an, bn)_2$-good code of length n has $\geq c_{a,b} n^2$ wires.

You might suspect that the quadratic bound holds even for depth 2 and higher. In fact, even depth 2 suffices for quasi-linear size.

Theorem 14.2. There are good linear codes $(n, cn, cn)_2$ that are computable by depth-2 Xor-circuits with $cn \log^2 n$ wires.

Proof. It suffices to give a construction that for any non-zero input outputs a string with weight $\geq cn$, see Problem 11.4. Divide the gates in the middle layer in $c \log n$ *blocks.* We will show that for every input there is a block that is nearly balanced, that is, the fraction of gates in the block that evaluates to 1 is in $[c, c]$. From this, we can obtain the output layer probabilistically by summing together one uniformly chosen gate from each block, and conclude by the tail bound Lemma B.1. The output layer has $cn \log n$ wires.

It remains to construct the middle layer. The construction is again probabilistic. Let $X_i \subseteq [2]^k$ be the set of inputs of, say, weight $\in [2^i, 2^{i+1}]$ for $i \in [\log k]$, assuming $k = cn$ is a power of two. For each gate in the block, first select $k/2^i$ uniformly chosen input bits, then pick a random subset of them, and output the sum. For every fixed input $x \in X_i$, each gate has a constant probability of selecting a 1 bit in the first step, in which case the sum is 1 with probability 1/2 in the second step. We want to fix the block so that it works for every input in X_i. By the probabilistic method and tail bounds (Lemma B.1) it suffices to pick $c \log |X_i|$ gates in that block.

Now the intuition is that X_i is about $\binom{k}{2^i}$, which is $\leq (ck/2^i)^{2^i}$ by Fact B.6. Hence, $\log |X_i| \leq c 2^i \log k$. Hence overall the number of wires in the block is

$$c \frac{k}{2^i} \cdot 2^i \log k \leq ck \log k.$$

Summing over all blocks, the total number of wires in the middle layer is $ck \log^2 k$. **QED**

Exercise 14.3. Prove $\log |X_i| \leq c2^i \log k$.

Exercise 14.4. Come up with an idea to improve the number of edges to $o(n \log^2 n)$ by modifying this construction slightly. This is good practice of balancing parameters. The execution of the idea is slightly technical and deferred to Problem 14.1.

14.2 Computing integers

Another basic algebraic question is that of computing n-bit integers using arithmetic circuits over the integers, with no variables and no constants except 1 and -1. Usual counting argument like that in the proof of Theorem 2.3 show that most n-bit integers require circuits of size $n/\log^c n$. And this is again nearly tight since any integer $t \in [2]^n$ can be computed with cn operations by writing $t = 2^0 t_0 + 2^1 t_1 + \cdots + 2^{n-1} t_{n-1}$ (computing all the 2^i takes cn operations).

As usual, the grand challenge is to exhibit "explicit" integers that are hard to compute. In particular, integers that cannot be computed with $\log^c n$ operations. A prominent integer in this context is the factorial. Recall that $n!$ is a number with $\Theta(n \log n)$ bits. The log factor won't play a role in these connections, so one can informally think of $n!$ as an n-bit number.

Similarly to Theorem 1.2, we can show that if computing factorials is easy then factoring is also easy. The first conclusion of the following theorem gives the simplest setting that conveys the main idea. The second conclusion refutes a popular conjecture in cryptography that is the basis of several cryptosystems.

Theorem 14.3. Suppose $n!$ has algebraic circuits of size $\log^a n$. Then

(1) there are boolean circuits of size n^{c_a} that factor the product of any two n-bit primes p and q s.t. $p \leq k < q$, for any k dependent only on n.

(2) let P and Q be i.i.d. n-bit primes, arbitrarily distributed. There is a boolean circuit of power size that given $P \cdot Q$ computes P with prob. $\geq c$.

Proof. (1) By assumption, there are algebraic circuits for $k!$ of size $\log^{c_a} k$. Note $k < q \leq 2^n$, so the size is $\leq n^{c_a}$. We use this circuit to compute $r := k! \mod x$, as a boolean integer. To do so, we simply run the circuit and compute $\mod x$ at each gate to keep the bit-length feasible. Finally, we output $\gcd(r, x)$ as one of the factors. **QED**

Exercise 14.5. Explain why this proof of (1) works. Hint: Fact B.2.

Prove (2).

Thus we have shown that if factorial is easy, then factoring is also easy. And we will see below in section §14.4 that if it factorial is hard then another long-sought separation follows. Hence the complexity of computing factorials appears pivotal.

14.3 Univariate polynomials

A next natural question is computing univariate polynomials. We make two remarks on the model.

First, here the goal is to understand *monomials*, not coefficients, so we allow gates that compute any field element (unlike in section §14.2).

Second, an important distinction must be made. We can consider computing polynomials *formally*, which we can think of as a sequence of coefficients, or *informally*, as functions. This distinction disappears when the field is larger than the degree by Fact B.28, but otherwise leads to different theory. For example over $\mathbb{F}_2$ we have $x^2 = x$ informally (i.e. the identity holds for every field element) but obviously not formally. Obviously formal identities are also informal, so informal impossibility results are harder to establish than formal. Given this, the cleanest setting may be when the underlying field is infinite.

Regarding impossibility results, the situation is similar to the previous section §14.2. A specific polynomial of interest is the approximation to the exponential function: $\sum_{i=0}^{n} X^i/i!$.

14.4 Multivariate polynomials

Arguably the most studied setting is the one of polynomials in n variables, because it is closely related to other classes (as we shall see).

Exercise 14.6. Let $B : [2]^n \to [2]$ be a (boolean) circuit of size s. Show that over any field there is an algebraic circuit A of size s s.t. $A(x) = B(x)$ for all $x \in [2]^n$.

The same remarks in section §14.3 apply to this section.

Again, the challenge is to exhibit "explicit" polynomials that are hard to compute. For larger-depth there is a superlinear informal result that does not have a formal counterpart. For several explicit degree-d polynomials in n variables it can prove bounds of the form $cn \log d$. We state one example:

Theorem 14.4. Computing $\sum_{i \in [n]} x_i^d$ requires size $cn \log d$.

The beautiful proof is the combination of these two facts.

Lemma 14.1. Computing a polynomial p and simultaneously all its n partial derivatives w.r.t. the n variables only costs a constant factor more than computing p.

Lemma 14.2. Computing the polynomials $p_i(x_1, x_2, \ldots, x_n)$ for $i \in [n]$ where x are n variables requires size $\geq cn \log d$.

We now turn to constant-depth circuits.

Algebraic impossibility from boolean impossibility

Note that over the field $\mathbb{F}_2$ the informal imp. results obtained in section 9.1 (see especially Theorem 9.1) are algebraic (since And is like multiplication) – and nothing better is known even if one is informal, for small depth. The techniques in section 9.1 can be extended to slightly larger fields. The idea is similar to that in section 9.1: we show that such circuits are approximated by low-degree polynomials. We sketch this idea in the case of $\Sigma\Pi\Sigma$ circuits, highlighting where the field size plays a role. It suffices to approximate $\Pi\Sigma$ circuits well. Consider one such circuits, and let r be the rank of the linear forms input to the Π gate (excluding their constants, if any). If the rank is large, then over a uniform input it's likely that at least one linear form will be zero and so the whole circuit is zero. If the rank r is small, then we can write each linear form as a linear combination of $\leq r$ linear forms. Now if we expand the Π gate we will have a sum of products of these r linear forms. Now we can use the fact that over a field of size q we have $X^q = X$, so we reduce the degree of each form in any product to at most $q-1$. Overall, the degree will be $\leq (q-1)r$.

Using these ideas one can obtain algebraic impossibility results over small fields. But for larger, or infinite fields a different set of techniques appears necessary, and only formal results are known.

14.4.1 VNP

Similarly to NP, an important class of polynomials can be defined by summing over all boolean values of a set of variables.

Definition 14.1. The Σ-algebraic circuits $S(X_1, \ldots, X_n)$ of size $\leq s$ are those that can be written as $\Sigma_{y_1,\ldots,y_s \in [2]} C(X_1, \ldots, X_n, y_1, \ldots, y_s)$ where C is an algebraic circuit of size $\leq s$.

Several polynomials of interest that are not known to have small algebraic circuits can be shown to have small Σ-algebraic circuits.

Example 14.1. We show that the *permanent* polynomial in n^2 variables x,

$$p(x) = \sum_{\pi} \prod_{i \in [n]} x_{i,\pi(i)}$$

where π ranges over all permutations of $[n]$, has Σ-algebraic circuits of size n^c. It is an open problem whether it has (plain) arithmetic circuits of power size.

We will encode π using n^2 bits M specifying an $n \times n$ permutation matrix also written M. Suppose we have a polynomial g s.t. $g(M) = 1$ if M is a permutation and 0 otherwise. Then we have:

$$p = \sum_{M \in [2]^{n^2}} g(M) \prod_{i \in [n], j \in [n]} x_{i,j} \cdot M_{i,j}.$$

Thus it only remains to show that g has small algebraic circuits.

Exercise 14.7. Finish the example.

Similar to the P vs. NP question, the prominent question here is whether Σ-algebraic and algebraic circuits have similar power (known as the VNP vs. VP question). The next two results prove several consequences of algebraic equivalences.

First we have the following consequence in the boolean world. For simplicity we work on suitable fields.

Exercise 14.8. [Σ-algebraic circuits are easy$\Rightarrow$ $\mathrm{Maj} \cdot \mathrm{CktP} \subseteq \mathrm{CktP}$] Suppose there is a constant a s.t. over any field any Σ-algebraic circuit of size s has an equivalent algebraic circuit of size s^a. Then $\mathrm{Maj} \cdot \mathrm{CktP} \subseteq \mathrm{CktP}$. (See section §**??** for the definition of the operator.)

The following result connects the power of Σ-algebraic circuits to the complexity of computing integers (section §14.2). One can get similar results for other integers or even univariate polynomials (including those mentioned in 14.3). For simplicity we state this connection for circuits over the integers, and only using fixed constants.

Theorem 14.5. [If Σ-algebraic circuits are easy then so is factorial] Suppose there is a constant a s.t. over the integers, every Σ-algebraic circuit of size s using constants 0 and 1 only has an equivalent algebraic circuit of size n^a using constants 0 and 1 only.

Then $n!$ has algebraic circuits of size $\log^{c_a} n$.

The proof is an excellent display of "scaling up and down" and connecting disparate complexity results.

Proof. Rather than giving circuits for $n!$, a number of $\leq cn \log n$ bits, we will give circuits for $2^n/n^c!$, a number of 2^n bits. This makes it slightly easier to connect to other results, and to index bits.

First we claim bit i of this 2^n-bit factorial given $i \in [2]^n$ is computable in

$$\mathrm{Maj} \cdot \mathrm{Maj} \cdot \cdots \cdot \mathrm{Maj} \cdot \mathrm{CktP},$$

where the number of applications of the Maj operator is c. This follows from the fact that iterated multiplication of integers is in TC (Theorem 8.2). Similarly to Exercise 14.8, the hypothesis implies that $\mathrm{Maj} \cdot \mathrm{CktP} \subseteq \mathrm{CktP}$. Repeating this c times we obtain a boolean circuit C of size n^c s.t. $C(i)$ is bit i of the factorial.

We can view C as an algebraic circuit over the integers and consider

$$S'(X_{n-1}, \ldots, X_0) := \sum_{j_0, j_1, \ldots, j_{n-1} \in [2]} C(j) X_0^{j_0} X_1^{j_1} \cdot X_{n-1}^{j_{n-1}}$$

in the variables X_i. Note that $S'(2^{n-1}, \ldots, 8, 4, 2, 1)$ equals the desired factorial. We can't immediately apply the hypothesis to S', until we note

$$X_0^{j_0} = (X_0 j_0 + 1 - j_0)$$

which allows to write S' as a Σ-algebraic circuit. Applying the hypothesis again yields an equivalent n^c-size algebraic circuit C', and then again the desired factorial is $C'(2^{n-1}, \ldots, 8, 4, 2, 1)$. The powers of 2 take cn operations. **QED**

14.5 Depth reduction in algebraic complexity

To set the stage, we note that reducing the depth of general algebraic circuits is impossible:

Exercise 14.9. Give an algebraic circuit of size s that does not have an equivalent algebraic circuit of depth $< s$, for all s.

However, interestingly it is possible to reduce the depth under the additional assumption that the circuit computes a polynomial of low degree:

Theorem 14.6. Any algebraic circuit of size s computing a polynomial degree d has an equivalent circuit of size s^c and depth $c(\log s)(\log d)$.

In the unbounded fan-in setting, the following is known and reminiscent of Theorem 9.5.

Theorem 14.7. Any n-variate polynomial of degree d computable by a size-n^a arithmetic circuit can be computed by a depth-3 $\Sigma\Pi\Sigma$ of size $n^{c_a\sqrt{d}}$.

We shall see in section §14.8 that impossibility results for circuits of size $n^{c\sqrt{d}}$ computing degree-d polynomials are known. Thus, as mentioned at the beginning of the chapter, the situation in the algebraic world is strikingly analogous to that in the boolean world discussed in section §6.3. We have impossibility results for small-depth circuits that are "just short" of having major consequences for larger-depth models.

14.6 Completeness

The results in section §7.5, extended to other fields, show that iterated product of 3×3 matrices is complete for algebraic circuits of small depth. As in that section, the reduction is as simple as it gets: For any power-size circuit one can write down a power-size product where the matrix entries are either constants or variables that computes the same polynomial. Using the depth reduction in section §14.5, one can extend this completeness to arbitrary circuits *computing low-degree polynomials.*

TBD: Determinant, permanent

14.7 The power of AAC: algebraic AC

Consider the elementary symmetric polynomial of degree d in n variables:

$$e_{n,d}(x_1, x_2, \ldots, x_n) := \sum_{S \subseteq [n], |S|=d} \prod_{i \in S} x_i. \tag{14.1}$$

These polynomials, and efficient ways for computing them, are of central importance, as we also see below in section §14.8. The RHS is an expression for a circuit of size $\geq n^d$. But in fact one can do better.

Theorem 14.8. For any d, $e_{n,d}$ has depth-3 circuits of size cn^2.

Proof. *Linear algebra magic.* Note that

$$p(t,x) := \prod_{i=1}^{n}(1+tx_i) = \sum_{i=0}^{n} t^i e_{n,i},$$

that is, $e_{n,i}$ is the coefficient of t^d in $p(t,x)$, where $x = (x_1, x_2, \ldots, x_n)$. We can compute $p(t,x)$ efficiently, and so we should be able to get its coefficients via interpolation. Specifically, for a field element α denote by v the "moment" vector

$$v := (\alpha^0, \alpha^1, \ldots, \alpha^n).$$

Also denote by e the vector of polynomials in x

$$e := (e_0, e_1, \ldots, e_n).$$

Note that $p(\alpha, x) = \langle v, e\rangle$. Suppose we have field elements α_i for $i \in [n+1]$ whose corresponding vectors v_i are linearly independent. Then we can find a linear combination

$$\sum_i a_i v_i$$

that equals the vector w_d with 1 in the coordinate with index d, and zero elsewhere. We could then compute e_d as

$$\sum_i a_i p(\alpha, x) = \sum_i a_i \langle v_i, e\rangle = \langle \sum_i a_i v_i, e\rangle = \langle w_d, e\rangle = e_d.$$

The LHS is an algebraic circuit of size cn^2.

There remains to exhibit such field elements. We can simply pick $\alpha_i = i$,$i \in [n+1]$, and the vectors v_i are linearly independent (Fact B.22). Over the complex numbers we can also let $\alpha_i := \omega^i$ where ω is the $(n+1)$-th primitive root of unity $e^{\sqrt{-1}2\pi/(n+1)}$. The vectors v_i are then orthogonal because

$$\langle v_i, v_j\rangle = \sum_{k\in[n+1]} \omega^{k(i-j)}$$

which is 0 if $i \neq j$ and otherwise is $n+1$. Hence they are independent (Fact B.23). **QED**

We shall see in Claim 14.1 a different approach where the circuit is somewhat larger but has additional structure.

14.8 Impossibility results for small-depth circuits

In this section we prove impossibility results for small-depth algebraic circuits. For simplicity, we only prove such results for circuits of depth 3, and over fields $\mathbb{F}$ of characteristic zero such as $\mathbb{R}$. The argument contains most of the ideas that go into extending the result to higher depth and other fields. The main result we prove is:

Theorem 14.9. There are explicit polynomials over $\mathbb{R}$ of degree $d := c \log n$ in n variables that require depth-3 arithmetic circuits of size $\geq n^{c\sqrt{d}}$.

The proof has the following steps.

1. We define set-multilinear circuits and prove that any circuit can be converted into a set-multilinear circuit efficiently.

2. We introduce a complexity measure, and give an explicit set-multilinear polynomial for which it is large.

3. We show that the measure is small for efficient set-multilinear circuits.

We now develop each step in turn.

For concreteness, we allow multiplication by arbitrary constants along wires. So for example, if two gates compute polynomials p and q, a Σ gate can compute $ap + bq$ for any $a, b \in \mathbb{F}$, and a Π gate can compute apq for any $a \in \mathbb{F}$.

14.8.1 Step 1

We partition the n variables into d sets X_i, $i \in [d]$. This partition is fixed throughout the argument. Jumping ahead, the sizes of the X_i will not be equal, but we will worry about this later.

Definition 14.2. A polynomial p is *set-multilinear*, abbreviated sm, if there is $D \subseteq [d]$ s.t. every monomial in p has exactly one variable in X_i for every $i \in D$, and no variable in X_i for $i \notin D$. A circuit is sm if every gate computes an sm polynomial

In particular, p has degree D and is multilinear. Note that D needs not equal $[d]$; in particular, p does not have to have degree d, and this will be useful later. On the other hand, if p does have degree d then necessarily $D = [d]$.

Lemma 14.3. Any $\Sigma\Pi\Sigma$ circuit of size s computing a degree-d sm polynomial has an equivalent sm $\Sigma\Pi\Sigma\Pi\Sigma$ circuit of size $d^{cd}s^c$.

In turn we break the proof of this lemma in three claims. The second and third claim are technically easy, but provide useful breaking points for the proof. Some of the "magic" happens right in the first claim. A simple consequence of it, stated in the second claim, is making the fan-in of multiplication gates $\leq d$. This then makes the size blow-up in the third claim tolerable. The lemma and the first claim generalize without new ideas to circuits of any depth, we focus on $\Sigma\Pi\Sigma$ for simplicity. The other claims we state in full generality.

The first claim makes polynomials *homogeneous*, i.e., each gate computes a polynomial where each term has the same degree (but the polynomial is not necessarily sm or even multilinear). The second reduces the fan-in of the multiplication gates. The third gets sm.

For a polynomial p we define $p^{(k)}$ as the degree-k homogeneous part, consisting of the monomials of degree exactly k. We say p is homogeneous if $p = p^{(k)}$ for some k

Claim 14.1. Suppose a $\Sigma\Pi\Sigma$ circuit of size s compute polynomial p. Then there is a $\Sigma\Pi\Sigma\Pi\Sigma$ homogeneous circuit of size $d^d s^c$ which computes the homogeneous parts $(p^{(0)}, p^{(1)}, \ldots, p^{(d)})$ of p.

The size bound can be improved to $c^{\sqrt{d}} s^c$ by a less crude analysis of the repeated applications of Fact B.27 below.

Proof. Consider one product gate q. We show how to compute $q^{(k)}$ with the desired resources. From this we compute $p^{(k)}$ as follows. Either $p^{(k)} = 0$, in which case there is nothing to do, or else it is the sum of $q_i^{(k)}$ where $p = \sum q_i$.

Write the input Σ gates of q as $\ell_i + b_j$ where each ℓ_i is a sum $\sum_j a_{i,j} x_j$ and b_j is a constant term. Assuming $b_j = 1$, we have

$$q = \prod_{i \le s} (\ell_i + 1) = \sum_{S \subseteq [s]} \prod_{i \in S} \ell_i.$$

The k-homogeneous part of q is

$$q^{(k)} := \sum_{S \subseteq [s], |S| = k} \prod_{i \in S} \ell_i.$$

This is the elementary symmetric polynomial $e_{s,k}$, equation (14.1), evaluated at the ℓ_i. It is a homogeneous polynomial, but as mentioned in section §14.7, computing it directly as in the RHS above would give size $\ge n^d$, which we cannot afford. The construction in the proof Theorem 14.8 has smaller size, but it not homogeneous.

We seek an alternative efficient, homogeneous, $\Sigma\Pi\Sigma\Pi\Sigma$ circuit. Towards this, consider the power sum polynomials

$$p_{s,k} := \sum_{i \le s} x_i^k.$$

We have (suppressing the subscript s for simplicity)

$$\begin{aligned} e_1 &= p_1 \\ e_2 &= p_1^2/2 - p_2/2 \\ e_3 &= p_1^3 - 3p_1 p_2/2 + p_3 \\ &\ldots \end{aligned}$$

and so on, which convinces us that we can express the e_k via the p_k, and as it turns out this expression is more efficient.

More in detail, we have by B.27 that

$$k \cdot e_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} \cdot p_i,$$

for all k. Applying this repeatedly, we write e_k as a sum of $\le k^k$ products of $\le k$ polynomials p_i. Since each p_i is naturally a homogeneous $\sum\sum\prod$ circuit, we obtain a homogeneous

$\sum\prod\sum\prod$ circuit for e_k. Instantiating the variables with the ℓ_i we obtain a $\sum\prod\sum\prod\sum$ circuit for $q^{(k)}$. The size of this circuit is $\leq k^k \cdot k \cdot s \cdot k \cdot n$, where each factor corresponds to the fan-in at a level.

Because of homogeneity, we only need to consider $k \leq d$. Hence the total size is $\leq d^d s^c$. **QED**

Exercise 14.10. Show that we can assume that $b_j = 1$.

Exercise 14.11. Explain why the construction in the proof of Theorem 14.8 is not homogeneous.

Claim 14.2. A homogeneous circuit computing a polynomial of degree d has an equivalent homogeneous circuit of no larger size in which the fan-in of each Π gate is $\leq d$.

Proof. Replace all gates computing polynomials of degree $> d$ with the constant 0. The correctness of this step can be argued inductively. For Σ gate computing a polynomial of degree d, it is safe to set to 0 any summand with degree $> d$. This is because by homogeneity, the summands do not have monomials of degree $\leq d$, so all their monomials must cancel in the output. For a Π gate the same holds, because multiplication increases degree (Fact B.26), so if it is computing a polynomial p of degree d and a term has degree $> d$, then in fact $p = 0$.

After this, a non-zero product gate can have $\leq d$ non-constant terms. The constant terms multiply to a constant that can be placed on a wire following our convention. **QED**

Finally, we get sm.

Claim 14.3. Suppose sm degree-d polynomial p is computable by a circuit of size s and depth t where the fan-in of each multiplication gate is $\leq d$. Then p is also computable by a sm circuit of depth t and size $d^{cd}s$.

Proof. We inductively replace each gate g in the circuit with 2^d gates g_D where for $D \subseteq [d]$ gate g_D computes the monomials in g that are sm w.r.t. D; and add circuitry accordingly.

If g is an input variable $x \in X_i$ we have $g_D = x$ if $D = \{i\}$ and $g_D = 0$ otherwise.

If g is a constant we have $g_\emptyset = g$ and $g_D = 0$ otherwise. (Alternatively, we can disallow constant but define more general addition and multiplication gates with constants in them.)

If

$$g = g_1 + g_2 + \cdots + g_i$$

then simply

$$g_D = g_{1,D} + g_{2,D} + \cdots + g_{i,D}$$

for any D.

Finally, if

$$g = g_1 \cdot g_2 \cdot \cdots \cdot g_i$$

note that

$$g_D = \sum g_{1,D_1} g_{2,D_2} \cdots g_{i,D_i}$$

where the sum is over all partitions $(D_1, \ldots, D_i)$ of D. Now we use the assumption that the fan-in i of this multiplication is $\leq d$. Hence the number of such partitions is at most the number of partitions of $[d]$ into d elements, which is $\leq d^d$.

Applying these transformations, the depth of the circuit does not increase as we can merge adjacent Σ and Π gates. The size multiplies by d^{cd}. **QED**

Example 14.2. Let $n = d = 2$, $X_0 = \{0\}$ and $X_1 = \{1\}$ and $x_i \in X_i$. Consider the circuit $C := (x_0 + x_1)^2 - x_0^2 - x_1^2$. This circuit is homogeneous but not sm or even multilinear, yet it computes the sm polynomial $2x_0x_1$.

Let us illustrate how we transform C into an sm circuit.

Consider the Π gate g computing x_0^2. Then $g_{\{0\}} = x_{0,\{0\}} \cdot x_{0,\emptyset} + x_{0,\emptyset} \cdot x_{0,\{0\}} = x_0 \cdot 0 + 0 \cdot x_0 = 0$.

Consider now $g = (x_0 + x_1)^2$. Then $g_{\{0,1\}} = x_0x_1 + x_0x_1 = 2x_0x_1$. And the other g_D are $\emptyset$.

Note how non-multilinear terms such as x_0^2 are removed during the process.

14.8.2 Step 2

For this step we further partition the blocks $[d]$ in two. We consider blocks with index $i < t$, denoted T_1 and those with $i \geq t$ denoted T_2, for a threshold t that will be set later.

Definition 14.3. Let p be an sm polynomial w.r.t. $D \subseteq [d]$. We define the matrix M_p where the rows are indexed by the monomials with variables X_i for $i \in D \cap T_1$ and the columns by monomials with variables X_i for $i \in D \cap T_2$. The m_1, m_2 entry of the matrix is the coefficient of the monomial m_1m_2 in p.

If either $D \cap T_1$ or $D \cap T_2$ is empty the matrix is a row or column matrix, and we can think of either m_1 or m_2 as being the constant 1.

The complexity measure $\mu(p)$ is the rank of M_p normalized by the geometric mean of the two sides:

$$\mu(p) := \frac{\text{rank}(M_p)}{\sqrt{\prod_{i \in D \cap T_1} |X_i| \cdot \prod_{i \in D \cap T_2} |X_i|}}.$$

Note that the denominator can be written simply as $\sqrt{\prod_{i \in D} |X_i|}$. However thinking of it as a mean of the two sides may help following the argument.

Example 14.3. Consider a sm polynomial ℓ of degree 1. All the variables belong to one set X_i, and $D = \{i\}$. Then M_ℓ is a vector, of rank 1. So $\mu(\ell) = 1/\sqrt{|X_i|}$.

The hard polynomials will have large μ. To construct such polynomials p, we can set the dimensions to be equal:

$$\prod_{i \in T_1} |X_i| = \prod_{i \in T_2} |X_i| \tag{14.2}$$

so that M_p is square; and then pick p so that M_p has full rank, for example it is diagonal or permutation. For such a p we obtain

$$\mu(p) = \frac{\prod_{i \in T_1} |X_i|}{\sqrt{\prod_{i \in T_1} |X_i| \cdot \prod_{i \in T_2} |X_i|}} = \frac{\prod_{i \in T_1} |X_i|}{\prod_{i \in T_1} |X_i|} = 1.$$

This value of μ is maximum as also given by the next lemma which will be used in the next step.

Lemma 14.4. The measure μ enjoys:

1. $\mu(p) \leq 1$
2. [Sub-additivity] If p and q are sm w.r.t. the same D, $\mu(p+q) \leq \mu(p) + \mu(q)$.
3. [Multiplicativity] If p_1 and p_2 are sm w.r.t. D_1 and D_2, where $D_1 \cap D_2 = \emptyset$, then $p_1 p_2$ is sm w.r.t. $D_1 \cup D_2$ and $\mu(p_1 p_2) = \mu(p_1) \cdot \mu(p_2)$.

Exercise 14.12. Prove this. For 3., use Fact B.24.

14.8.3 Step 3

We now need to pick the sizes of the X_i so that μ will be small for the efficient circuit, while at the same time keeping equation (14.2). We let X_i with $i \in T_1$ have size m and the others have size $m^{1-\delta}$ where $\delta := 1/(2\sqrt{d})$. The total number of variables is $n = t \cdot m + (d-t) \cdot m^{1-\delta}$, and so $m \geq n^c$.

The parameters m and t are picked so that equation (14.2) is true. This requires

$$m^t = m^{(1-\delta)(d-t)} \iff t = (1-\delta)d/(2-\delta) \iff t = d - d/(2-\delta).$$

We now proceed to show that for such parameters, μ is small.

Consider a $\Sigma\Pi\Sigma\Pi\Sigma$ circuit. Let q be a Π gate closest to the output and write

$$q := f_1 \cdot f_2 \cdot \cdots \cdot f_k$$

where the sum of the degrees of the f_i is $\leq d$ and each f_i is a $\Sigma\Pi\Sigma$ circuit.

We will show that

$$\mu(q) \leq 1/m^{c\sqrt{d}} \tag{14.3}$$

from which we conclude the proof of Theorem 14.9 below. To prove equation (14.3) we consider two cases.

Some f_i has degree $\geq c\sqrt{d}$.

Exercise 14.13. Prove this case. Guideline: Let $C_{i,j}$ be a $\prod \sum$ sub-circuit of f_i; prove:
(1) Wlog $C_{i,j}$ has the same degree as f_i (which is $\geq c\sqrt{d}$).
(2) $\mu(C_{i,j}) \leq 1/m^{c\sqrt{d}}$. Use Lemma 14.4 and Example 14.3.
(3) $\mu(f_i) \leq s/m^{c\sqrt{d}} \leq 1/m^{c\sqrt{d}}$.
(4) $\mu(q) \leq 1/m^{c\sqrt{d}}$ (equation (14.3)).

Every f_i has degree $\leq c\sqrt{d}$. This is where we use the unbalance. At the high-level, the idea is simple. Recall the target matrix is square, and has measure 1, which is the maximum. In the case we are considering, the matrix is "made up" of many small pieces. The size of the pieces are chosen so that few pieces don't make a square. For perhaps the simplest example of the phenomenon, suppose you have many domino pieces of size p and many of size q, where p and q are primes. By picking q of the former and p of the latter, you can make a $pq \times pq$ square. But, if you use *few* pieces, say $< q$ pieces of length p, you'll never get a square. Returning to the proof, the difference in the side lengths translates into a bound on μ for each f_i, showing that μ is significantly smaller than 1. By multiplicativity of μ, the bound for q will be the product of these bounds, giving something substantially smaller than 1.

Let f_i be sm w.r.t. D_i and let $a_i := |D_i \cap T_1|$ and $b_i := |D_i \cap T_2|$. The matrix $M(f_i)$ has sides m^{a_i} and $m^{(1-\delta)b_i}$. Because the rank of $M(f_i)$ is at most the minimum of the sides, we have that $\mu(f_i)$ is at most the minimum of $\sqrt{\frac{m^{a_i}}{m^{(1-\delta)b_i}}}$ and $\sqrt{\frac{m^{(1-\delta)b_i}}{m^{a_i}}}$, which is

$$\frac{1}{m^{0.5|a_i - b_i(1-\delta)|}}.$$

We now argue that this is small, using that a_i and b_i are integers summing to $\leq \sqrt{d}$. The quantity of interest is the distance

$$|a_i - (1-\delta)b_i|,$$

which we need to bound below.

Generally, we claim that if a, b are integers in $[c/\epsilon]$ then $|a - (1-\epsilon)b| \geq c\epsilon(a+b)$.

Exercise 14.14. Prove this.

Using this claim we get

$$\mu(f_i) \leq \frac{1}{m^{c(a_i+b_i)/\sqrt{d}}}.$$

By the multiplicativity property from Lemma 14.4 we have

$$\mu(q) \leq \prod_i \frac{1}{m^{c(a_i+b_i)/\sqrt{d}}} \leq \frac{1}{m^{cd/\sqrt{d}}} \leq \frac{1}{m^{c\sqrt{d}}},$$

as desired.

14.8.4 Putting the steps together, proof of Theorem 14.9

By Lemma 14.3, equation (14.3), and the sub-additivity property in Lemma 14.4, the measure of the circuit is

$$\leq d^{cd} s^c / m^{c\sqrt{d}}.$$

As remarked earlier, $m \geq n^c$ and so the denominator is $\geq n^{c\sqrt{\log n}}$. This is bigger than the numerator and so the ratio will be < 1.

Picking a target polynomial as in Step 2 for which the measure is 1 completes the proof.

14.9 Algebraic TMs

TBD

14.10 Problems

Problem 14.1. Execute the idea in Exercise 14.4.

Problem 14.2. Prove that Theorem 14.8 is false over $\mathbb{F}_2$.

14.11 Notes

The complexity of linear transformations was studied independently in [118, 282].

Theorem 14.2 is from [99]. For depth 2 they prove a slightly stronger, tight bound; they also show that depth $\log^* n$ suffices for a linear number of wires.

For the connection to factoring see [246, 190].

Impossibility results for univariate polynomials were first studied in [266]. That paper establishes negative results for polynomials with very large coefficients, but the results are non-trivial since arbitrary constants can be used by the circuit. For a survey see the book [58].

For an introduction to multivariate algebraic complexity see [58] and the survey [253]. 14.1 is from [40]. 14.2 is from [265]. For section 14.4 see [119].

Theorem 14.5 is from [175, 57].

For Theorem 14.8 and related constructions, also involving some of the ideas that go into Claim 14.1, see [252]. For more on the power of algebraic AC see [23].

Theorem 14.9 in section §14.8 is from [187]. It builds on a long line of works, see [188] for discussion. The complexity measure originates in [217]. The transformation to sm in the proof of Lemma 14.3 only works for large characteristic, due to the factors arising in Fact B.27 and related steps. [87] proves the lemma over any field, although whether circuits can be made homogeneous over any field remains open, see [87] for more discussion.

Theorem 14.7 is the culmination of a line of works about depth reduction that was rekindled by [11]. See discussion of subsequent work in [120] for this statement for depth 3. A formulation for any depth is stated in [188].

A general depth reduction theorem was first established in [147], but the size bounds were not controlled. Theorem 14.6, where the size is simultaneously controlled, is from [283] (see discussion in [283] for more on the history).

Chapter 15

Data structures

Data structures aim to maintain data in memory so as to be able to support various operations, such as answering queries about the data, and updating the data. The study of data structures is fundamental and extensive. We distinguish and study in turn two types of problems: *static* and *dynamic*. In the former the input is given once and is not modified by the queries. In the latter queries can modify the input; this includes classical problems such as supporting insert, search, and delete of keys.

15.1 Static data structures

Here we have n bits of input data about which we would like to answer m queries. The data structure aims to accomplish this by storing the input into s words of memory, where each word is w bits. This is accomplished via an arbitrary map g, with no bound on resources. But after that, the queries can be answered by a very efficient map h: each query only depends on t words. In general, these words can be read adaptively. But for simplicity we focus on the case in which the locations are fixed by the data structure and the same for every input $x \in [2]^n$. Refer to figure 15.1.

Definition 15.1. A static data-structure problem is simply a function $f : [2]^n \to [q]^m$. A data structure for f with word-space s, word size w and time t is a decomposition

$$f(x) = h(g(x))$$

where $g : [2]^n \to [2^w]^s$, $h : [2^w]^s \to [q]^m$, and each output bit coordinate of h depends on $\leq t$ input words.

We say *word-space* to emphasize s refers to the number of words, not bits. The *bit-space* of the structure is sw. Sometimes the queries and or the word size are boolean. Another typical setting is $q = 2^w$.

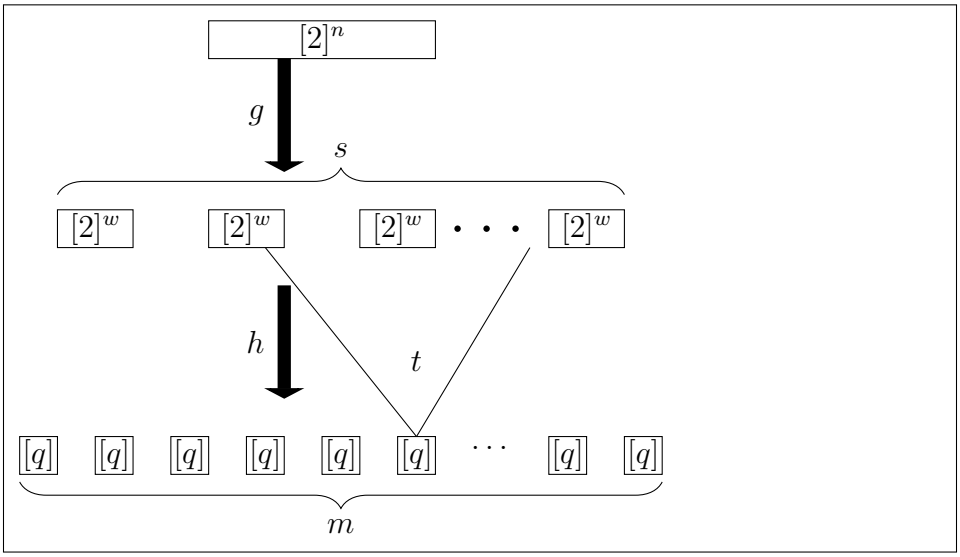


Figure 15.1: The static data-structure problem.

Exercise 15.1. Consider the data structure problem $f : [2]^n \to [2]^m$ where $m = n^2$ and query $(i, j) \in \{1, 2, \ldots, n\}^2$ is the parity of the input bits from i to j.

Give a data structure for this problem with $s = n$, $w = 1$, and $t = 2$.

Exercise 15.2. Show that any data-structure problem $f : [2]^n \to [2]^m$ has a data structure with $w = 1$ and the following parameters:

(1) $s = m$ and $t = 1$, and

(2) $s = n$ and $t = n$.

Exercise 15.3. Prove that for every n and $m \leq 2^{n/2}$ there exist functions $f : [2]^n \to [2]^m$ s.t. any data structure with space $s = m/2$ and $w = 1$ requires time $t \geq n - c$.

By contrast, next we present the the best known impossibility result. To set the stage, we first show how to establish slightly weaker impossibility results via a calculation-free reduction to communication protocols (see Chapter 13).

Exercise 15.4. Let f be a data-structure problem as in Definition 15.1. Show that if a f has a data structure with parameter names as in Definition 15.1 then the function $g : [2]^n \times [2]^{\log m} \to [q]$ defined as $g(x, i) := f(x)_i$ has communication complexity $\leq ct(\log s + w)$.

Focusing on the natural setting where $\log m \leq n$, we have that if g requires maximum communication complexity $\geq c \log m$ then

$$t \geq c \frac{\log m}{\log s + w}.$$

This bound is meaningful when m is large; but in typical settings where $n \leq s \leq m \leq n^c$ it gives nothing.

The next result gives a refined bound where the term $\log s$ in the denominator is replaced with $\log s/n$. This makes for a meaningful bound as long as s is close to n. Settings such as $s = n^2$ are *terra incognita*. The refined bound is established for bounded-uniformity, see section 11.1.1 and Definition 11.3.

Definition 15.2. A function $f : [2]^n \to [q]^m$ is d-wise uniform, or simply d-uniform, if any d output coordinates are uniform when the input to f is uniform.

Theorem 15.1. Let $f : [2]^n \to [q]^q$ be d-wise uniform. Let q be a power of 2 and $c \log q \leq d$. Then any data structure with $w = \log q$ using word-space s and time t has:

$$t \geq c\frac{\log q}{\log(s/d)}.$$

Let's make sense of the many parameters. Note that such d-wise uniform functions are computable with $n = d \log q = dw$ input bits, interpreting the input as coefficients of a degree $d-1$ univariate polynomial over $\mathbb{F}_q$ and outputting its evaluations, as in the proof of Theorem 11.1. Moreover, this is trivially the smallest possible n. Also, let us set $m = q$ so that $m = q = 2^w$. Plugging these parameters, we obtain for any n and m (powers of 2) an explicit function $f : [2]^n \to [m]^m$ with the data-structure bound

$$t \geq c\frac{\log m}{\log(sw/n)},$$

valid as long as $c \log q \leq d \iff cw \leq n/w$. A typical setting is $w = 10 \log n$ giving $m = q = n^{10}$. Recall that sw is the bit-space. It follows that if the bit-space is linear in n then $t \geq c \log m$. This result remains non-trivial for s slightly super-linear. But remarkably, if $sw = n^{1+c}$ then nothing is known (for m power in n one only gets $t \geq c$).

Proof. The idea in the proof, known as *cell sampling,* is to find a subset S of less than d memory cells that still allows us to answer $\geq d$ queries. This is impossible, since we can't generate d uniform outputs from less than d memory cells.

Let $p := 1/q^{1/4t}$. Include each memory cell in S with probability p, independently. By Lemma B.1, $\mathbb{P}[|S| \geq cps] \leq 2^{-cps}$.

We are still able to answer a query if all of its queries fall in S. The probability that this happens is $\epsilon := p^t = 1/q^{1/4}$. We now claim that with probability $\geq 0.5\epsilon$, we can still answer $0.5\epsilon q \geq \sqrt{q}$ queries. Indeed, let B be the number of queries we cannot answer. We have $\mathbb{E}[|B|] \leq q(1-\epsilon)$. And so

$$\mathbb{P}[|B| \geq q(1-0.5\epsilon)] \leq \frac{1-\epsilon}{1-0.5\epsilon} \leq 1 - 0.5\epsilon.$$

Thus, if the inequality $2^{-cps} < 0.5\epsilon = 1/q^c$ holds then there exists a set S of cps cells with which we can answer $\geq \sqrt{q} > d$ queries. Since the function is d-uniform, answering the latter queries requires $\geq dw$ input bits, and so $Sw \geq dw \iff S \geq d \iff cps \geq d$.

Therefore we reach a contradiction if the following chain of inequalities is true:

$$c \log q \leq cps < d.$$

Because $d > c \log q$ by assumption, we can say that the first inequality is automatically satisfied. Specifically, we can set s to be the largest s.t. $cps < d$, and the first inequality is satisfied. We reach a contradiction if $cps < d$, and the result follows. **QED**

Surprisingly, this result is nearly tight. There are many parameters floating around, so "tight" should be qualified. But basically, there are bounded-uniform functions f, with nearly optimal parameters, which have data structures with a tradeoff matching Theorem 15.1. One can in fact achieve results in the same spirit for f with optimal parameters (i.e., polynomial evaluation), thus matching the function in Theorem 15.1, but this requires a different construction, and I don't know if constant time is known.

Theorem 15.2. Let $q = n^{10} = 2^w$. There is $f : [2]^n \to [q]^q$ which is d-wise uniform and has a data structure with w-bit words, time t, word-space $s = cdw \cdot m^{1/t}$, and $n = ctdw$.

Let us illustrate parameters. Both n and sw (regardless of time) are $\geq dw$, for else you can't generate d uniform coordinates. So there is a d-wise uniform f with nearly optimal n that has a data structure achieving space optimal up to a factor $cm^{1/t}$, and time t. For example, we can have $n = cdw$, word space $dw \cdot q^{0.01}$, and $t = c$. Whereas you could have thought that word space close to q would be necessary. Qualitatively, the space approaches optimal *exponentially* fast with the time t, the same behavior exhibited in the Theorem 15.1. For a proof see Problem 15.1.

15.1.1 Succinct data structures

Succinct data structures are those where the space is close to the minimum, n. Specifically, we let $s = n + r$ for some $r = o(n)$ called *redundancy.* Unsurprisingly, stronger bounds can be proved in this setting. But, surprisingly, again these stronger bounds were shown to be tight. Moreover, it was shown that improving the bounds would imply stronger circuit lower bounds.

To illustrate, consider error-correcting codes, see Exercise **??**.

Theorem 15.3. Any data-structure for an a-good code $f : [2]^n \to [2]^m$ with $w = 1$ using space $n + r$ requires time $\geq c_a n/r$.

This is nearly matched by the following result. In fact, later we will prove the stronger result that *dynamic* data structures exist for error-correcting codes.

Theorem 15.4. There is a good code $f : [2]^n \to [2]^m$ that for any r has a data structure with $w = 1$, space $n + r$, and time $c(n/r) \log^3 n$.

Exercise 15.5. Prove this using the code in Theorem 14.2. Hint: If a circuit has ℓ wires, there are $\leq r$ gates with fan-in $> \ell/r$. That's your redundancy.

The techniques in the hint apply generically. They imply that proving a time lower bound of $(n/r)\log^c n$ would imply new circuit lower bounds, for circuits with XOR gates only, or for circuits with arbitrary gates. For the boolean model, the following connection is also known.

Theorem 15.5. Let $f : [2]^n \to [2]^{am}$ be a function computable by bounded fan-in circuits with bm wires and depth $b \log m$, for constants a, b. Then f has a data structure with space $n + o(n)$ and time $n^{o(1)}$.

Hence, proving n^ϵ time lower bounds for succinct data structures would give functions that cannot be computed by linear-size log-depth circuits, see 9.3.2.

15.1.2 Succincter: The trits problem

In this section we present a cute and fundamental data-structure problem with a shocking and counterintuitive solution. The trits problem is to compute $f : [3]^n \to ([2]^2)^n$ where on input n "trits" (i.e., ternary elements) $(t_1, t_2, \ldots, t_n) \in [3]^n$ f outputs their representations using two bits per trit.

Example 15.1. For $n = 1$, we have $f(0) = 00, f(1) = 01, f(2) = 10$.

Note that the input ranges over 3^n elements, and so the minimum space of the data structure is $s = \lceil \log_2 3^n \rceil = \lceil n \log_2 3 \rceil \approx n \cdot 1.584\ldots$ This will be our benchmark for space. One can encode the input to f as before using bits without loss of generality, but the current choice simplifies the exposition.

Simple solutions

- The simplest solution is to use 2 bits per t_i. With such an encoding we can retrieve each $t_i \in [3]$ by reading just 2 bits (which is optimal). The space used is $s = 2n$ and we have linear redundancy.

- Another solution, which we basically already mentioned, is what is called *arithmetic coding*: we think of the concatenated elements as forming a ternary number between 0 and $3^n - 1$, and we write down its binary representation. To retrieve t_i it seems we need to read all the input bits, but the space needed is optimal.

- For this and other problems, we can trade between these two extreme as follows. Group the t_i's into blocks of t. Encode each block with arithmetic coding. The retrieval time will be ct bits and the needed space will be

$$(n/t)\lceil t \log_2 3 \rceil \leq n \log_2 3 + n/t$$

  (assuming t divides n). In other words, *block-wise arithmetic coding*. This provides a *power* trade-off between time and redundancy, but no more (see the notes).

The shocking solution: An exponential trade-off

We now present an *exponential* trade-off: retrieval time ct bits and redundancy $n/2^t + c$. In particular, if we set $t = c \log n$, we get retrieval time $c \log n$ and redundancy c. Moreover, the bits read are all consecutive, so with word size $w = \log n$ this can be implemented in constant time. To repeat, *we can encode the trits with constant redundancy and retrieve each in constant time.* This solution can also be made dynamic.

Theorem 15.6. The trits problem has a data structure with space $n \log_2 3 + n/2^t + c$ (i.e., redundancy $n/2^t + c$) and time ct, for any t and with word size $w = 1$. For word size $w = \log n$ the time is constant.

Next we present the proof.

Definition 15.3. [Encoding and redundancy] An encoding of a set A into a set B is a one-to-one (a.k.a. injective) map $f : A \to B$. The *redundancy* of the encoding f is $\log_2 |B| - \log_2 |A|$.

The following lemma gives the building-block encoding we will use.

Lemma 15.1. For all sets X and Y, there is an integer b, a set K and an encoding

$$f : X \times Y \to [2]^b \times K$$

with redundancy $\leq c/\sqrt{|Y|}$ and s.t. $x \in X$ can be recovered just by reading the b bits in $f(x, y)$.

Exercise 15.6. Prove $K \leq cY$.

The basic idea for proving the lemma is to break Y into $C \times K$ and then encode $X \times C$ by using b bits:

$$X \times Y \to X \times C \times K \to [2]^b \times K.$$

There is however a subtle point. If we insist on always having $|C|$ equal to, say, $\sqrt{|Y|}$ or some other quantity, then one can cook up sets that make us waste a lot (i.e., almost one bit) of space. The same of course happens in the more basic approach that just sets $Y = K$ and encodes all of X with b bits. The main idea will be to "reason backwards," i.e., we will first pick b and then try to stuff as much as possible inside $[2]^b$. Still, our choice of b will make $|C|$ about $\sqrt{|Y|}$.

Proof. Assume $Y > 1$ without loss of generality. Define $b := \left\lceil \log_2 \left(X \cdot \sqrt{Y} \right) \right\rceil$, and let $B := [2]^b$. To simplify notation, define $d := 2^b / X$. Note $c\sqrt{Y} \leq d \leq c\sqrt{Y}$.

How much can we stuff into B? For a set C of size $|C| = \lfloor B/X \rfloor$, we can encode elements from $X \times C$ in B. The redundancy of such an encoding can be bounded as follows:

$$\begin{aligned}
&\log B - \log X - \log C = \\
=&\log \frac{2^b}{X} - \log \left\lfloor \frac{2^b}{X} \right\rfloor \\
=&\log d - \log \lfloor d \rfloor \\
\leq&\log d - \log(d-1) \\
=&\log \left(1 + \frac{1}{d-1}\right) \\
\leq&\frac{c}{d-1} \\
\leq&\frac{c}{\sqrt{Y}-1} \\
\leq&\frac{c}{\sqrt{Y}}.
\end{aligned}$$

To calculate the total redundancy, we still need to examine the encoding from Y to $C \times K$. Choose K of size $|K| = \lceil Y/C \rceil$, so that this encoding is possible. With a calculation similar to the previous one, we see that the redundancy is:

$$\begin{aligned}
&\log C + \log K - \log Y \\
=&\log \left\lceil \frac{Y}{C} \right\rceil - \log \frac{Y}{C} \\
\leq&\log \left(1 + \frac{C}{Y}\right) \\
\leq&c\frac{C}{Y} \\
\leq&c\frac{\left\lfloor \frac{2^b}{X} \right\rfloor}{Y} \\
\leq&c\frac{2^b}{X \cdot Y} \\
\leq&\frac{c2^{\log\left(X \cdot \sqrt{Y}\right)}}{X \cdot Y} \\
\leq&c\frac{\sqrt{Y}}{Y} \\
=&c\frac{1}{\sqrt{Y}}.
\end{aligned}$$

The total redundancy is then $c/\sqrt{|Y|}$. By construction, $x \in X$ can be recovered from the element of B only. **QED**

Proof of Theorem 15.6. Break the ternary elements into blocks of size t: $(t'_1, t'_2, \ldots, t'_{n/t}) \in T_1 \times T_2 \times \ldots \times T_{n/t}$, where $T_i = [3]^t$ for all i. The encoding, illustrated in figure 15.2, is constructed as follows, where we use f_L to refer to the encoding guaranteed by Lemma 15.1.

Compute $f_L(t'_1, t'_2) = (b_1, k_1) \in B_1 \times K_1$.

For $i = 2, \ldots, n/t - 1$ compute $f_L(k_{i-1}, t'_{i+1}) := (b_i, k_i) \in B_i \times K_i$.

Encode $k_{n/t-1}$ in binary as $b_{n/t}$ using arithmetic coding.

The final encoding is $(b_1, b_2, \ldots, b_{n/t})$. We now compute the redundancy and retrieval time. One can visualize this as a "hybrid argument" transforming a product of blocks of ternary elements into a product of blocks of binary elements, one block at the time.

Redundancy: From (1) in Lemma 15.1, the first $n/t - 1$ encodings have redundancy $c3^{-t/2} \leq 1/2^{ct}$. For the last (arithmetic) encoding, the redundancy is at most 1. So the total redundancy is at most $\left(\frac{n}{t} - 1\right) \cdot \frac{1}{2^{ct}} + 1 = \frac{n}{2^{ct}} + c$.

Retrieval Time: Say that we want to recover some t_j which is in block t'_i. To recover block t'_i, Lemma 15.1 guarantees that we only need to read at b_{i-1} and b_i. This is because k_{i-1} can be recovered by reading only b_i, and t'_i can be recovered by reading k_{i-1} and b_{i-1}. Thus to complete the proof it suffices to show that each b_i has length ct.

This is not completely obvious because one might have thought that the K_i become larger and larger, and so we apply the lemma to larger and larger inputs and the B_i get large too. However, recall that each $K_i \leq cT_i = c3^t$ from Exercise 15.6. Hence, every time we apply the lemma on an input of size at most $s \leq 3^{ct}$. Since the encoding in Lemma 15.1 has small redundancy, none of its outputs can be much larger than its input, and so $B_i = 2^{ct}$. **QED**

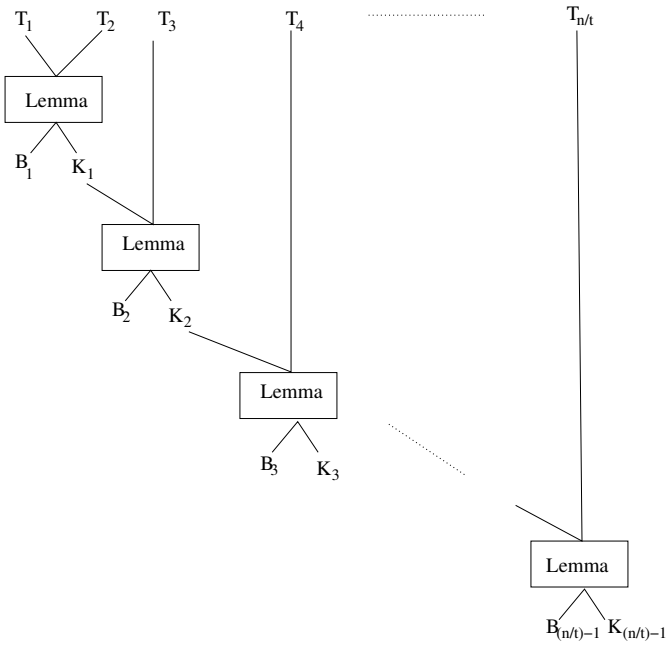


Figure 15.2: Succinct Encoding

15.2 Dynamic data structures

We now study dynamic data structures. As we mentioned, here the input is not fixed but can be modified by the queries. For concreteness, we focus on the specific problem of maintaining a codeword.

Definition 15.4. A dynamic data-structure for a code $f : [2]^n \to [2]^m$ supports two types of operations, starting with the all-zero message $x \in [2]^n$:

$M(i, b)$ for $i \in \{1, 2, \ldots, n\}$ and $b \in [2]$ which sets bit i of the message to b, and

$C(i)$ for $i \in \{1, 2, \ldots, m\}$ which returns bit i of the codeword corresponding to the current message.

The time of a dynamic data structure is the maximum number of operations in memory cells required to support an operation. This is similar to word programs where we ignore details and solely focus on information transfer among cells. Recall for word programs we do not know how to prove impossibility results. This can be considered as an *offline* mode of computation. By contrast the setting of data structure is *online* in that we have to answer queries as they arrive. In this setting it is natural to ask for small running times, like constant, and we can prove non-trivial impossibility results.

Theorem 15.7. Dynamic data-structures for any good code (see Exercise **??**) take time $t \geq c \log_w n \geq (c \log n)/\log\log n$ for cell size $w := \log n$.

One might wonder if stronger bounds can be shown. But in fact there exist good codes for which the bounds are nearly tight.

Theorem 15.8. There are dynamic data structures for good codes running in time $c \log^2 n$ with cell size $w = 1$.

Exercise 15.7. Prove Theorem 15.8 using the code in Theorem 14.2. Hint: The data structure is the middle layer. Bound the fan-out of the input gates in the proof of Theorem 14.2.

Proof of Theorem 15.7. Pick $x \in [2]^n$ uniformly and $i \in \{1, 2, \ldots, m\}$ uniformly, and consider the sequence of operations

$$M(1, x_1), M(2, x_2), \ldots, M(n, x_n), C(i).$$

That is, we set the message to a uniform x one bit at a time, and then ask for a uniformly selected bit of the associated codeword which we denote by $C_x := (C_x(1), C_x(2), \ldots, C_x(n)) \in [2]^m$.

We divide the n operations $M(i, x_i)$ into consecutive blocks, called *epochs*. Epoch e consists of n/w^{3e} operations. Hence we can have at least $E := c \log_w n$ epochs, and we can assume that we have exactly this many epochs (by discarding some bits of n if necessary).

The geometrically decaying size of epochs is chosen so that the number of message bits set during an epoch e is much more than all the cells written by the data structure in future epochs.

A key idea of the proof is to see what happens when the cells written during a certain epoch are ignored, or reverted to their contents right before the epoch. Specifically, after the execution of the M operations, we can associate to each memory cell the last epoch during which this cell was written. Let $D^e(x)$ denote the memory cells of the data structure after the first n operations M, but with the change that the cells that were written last during epoch e are replaced with their contents right before epoch e. Define $C^e_x(i)$ to be the result of the data structure algorithm for $C(i)$ on $D^e(x)$, and $C^e_x = C^e_x(1), C^e_x(2), \ldots, C^e_x(n)$.

Let $t(x, i, e)$ equal 1 if $C(i)$, executed after the first n operations M, reads a cell that was last written in epoch e, and 0 otherwise. We have

$$t \geq \max_{x,i} \sum_e t(x, i, e) \geq \mathbb{E}_{x,i} \sum_e t(x, i, e) = \sum_e \mathbb{E}_{x,i} t(x, i, e) \geq \sum_e \mathbb{E}_x \Delta(C_x, C^e_x), \qquad (15.1)$$

where the last inequality holds because $C^e_x(i) \neq C_x(i)$ implies $t(x, i, e) \geq 1$.

We now claim that if $t \leq w$ then $\mathbb{E}_x \Delta(C_x, C^e_x) \geq c$ for every e. This concludes the proof.

In the remainder we justify the claim. Fix arbitrarily the bits of x set before Epoch e. For a uniform setting of the remaining bits of x, note that the message ranges over at least

$$2^{n/w^{3e}}$$

codewords. On the other hand, we claim that C^e_x ranges over much fewer strings. Indeed, the total number of cells written in all epochs after e is at most

$$t \sum_{i \geq e+1} n/w^{3i} \leq ctn/w^{3(e+1)}.$$

We can describe all these cells by writing down their indices and contents using $B := ctn/w^{3e+2}$ bits. Note that this information can depend on the operations performed during Epoch e, but the point is that it takes few possible values overall. Since the cells last changed during Epoch e are reverted to their contents before Epoch e, this information suffices to describe $D^e(x)$, and hence C^e_x. Therefore, C^e_x ranges over $\leq 2^B$ strings.

For each string in the range of C^e_x at most two codewords can have relative distance $\leq c$, for else you'd have two codewords at distance $\leq 2c$, violating the distance of the code.

Hence except with probability $2 \cdot 2^B / 2^{n/w^{3e}}$ over x, we have $\Delta(C_x, C^e_x) \geq c$. If $t_M \leq w$ then the first probability is ≤ 0.1, and so $\mathbb{E}_x \Delta(C_x, C^e_x) \geq c$, proving the claim. **QED**

Exercise 15.8. Explain how to conclude the proof given the claim.

15.3 Problems

Problem 15.1. Prove Theorem 15.2. Hint: Use the construction Theorem 11.4. How uniform needs the input be?

Problem 15.2. In this problem you will show that proving impossibility results for dynamic data structures is easier (or no harder) than proving them for static data structures.

Let $f : [2]^n \to [2]^m$ be a static data structure problem. Give a set of $n + m$ queries s.t. if a dynamic data structure can support them in time t then f has a data structure with space $s \leq tn$ and time t. Feel free to assume word size $w = 1$ and non-adaptive query algorithms throughout (though these cases are not really satisfying, since the power of dynamic data structure relies on larger word size and adaptivity – but one can prove a suitable extension for more general cases). (A non-adaptive query algorithm means that the memory locations accessed depend only on the query, and not on the values of previous queries.)

15.4 Notes

The connection between data structures and communication complexity is from [202] and was studied more in [203]. Theorem 15.1 is from [254]. It was rediscovered in [182]. Efficient data structure for polynomial evaluation (complementing Theorem 15.2) are in [167].

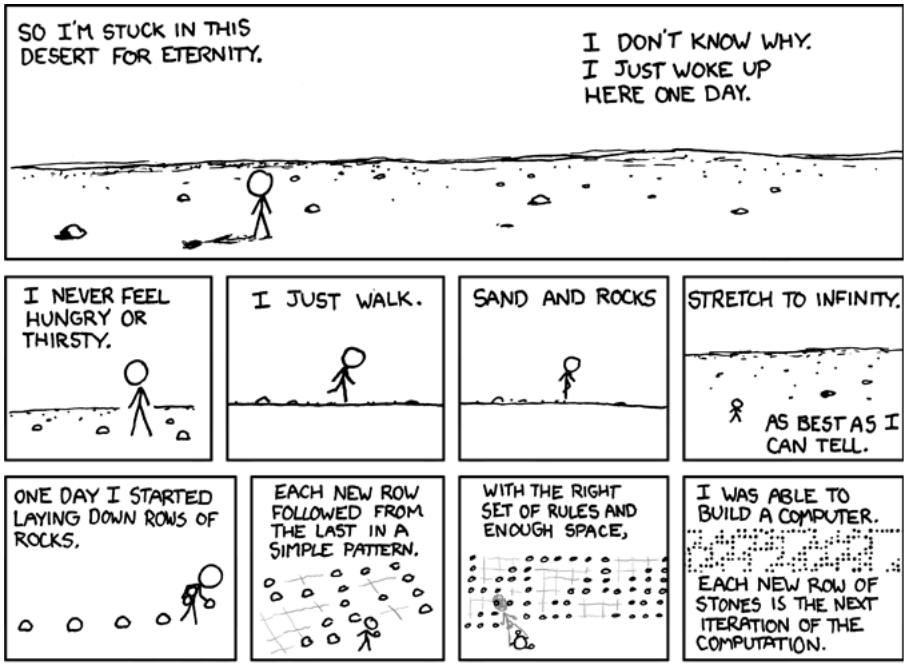
Theorem 15.3: [100]. Efficient data structures for ECC, both static in Theorem 15.4 and dynamic in Theorem 15.8, as well as the connection to circuits, are from [299].

The breakthrough result on the trits problem is from [223]. After that a negative result was proved in [296], whose parameters were then matched by Theorem 15.6, proved in [223, 82]. Our exposition is from [291]. Using number-theoretic results on logarithmic forms, it is shown in [296] that block arithmetic coding does not do better than a power tradeoff.

For dynamic data structures, the technique in the proof of Theorem 15.7 is from [93] and has been applied to many other natural problems. It is not far from the state-of-the art in this area, which is $\log^{1+c} n$ [183].

Chapter 16

Tapes



https://xkcd.com/505/ (selection)

Note we

In this chapter we discuss a uniform computational model which is more restricted than word programs but nonetheless is far reaching and fundamental to our understanding of computation. *Tape machines* are equipped with an infinite tape (or memory) of cells with symbols from the *tape alphabet* A, and a *tape head* pointing to one cell. The tape alphabet A consists of the symbols $0, 1$ which are used to encode input and output, and the blank symbol $\_$ which is used to delimit them. Later we shall also consider larger alphabets. If this sounds boring, check out section §16.1 and Question 16.1: We still do not understand if larger alphabets give more power!

The machine can read and write the cell under the head. We think of the machine as starting with an input x on the tape, and the head on the first symbol of x. After some

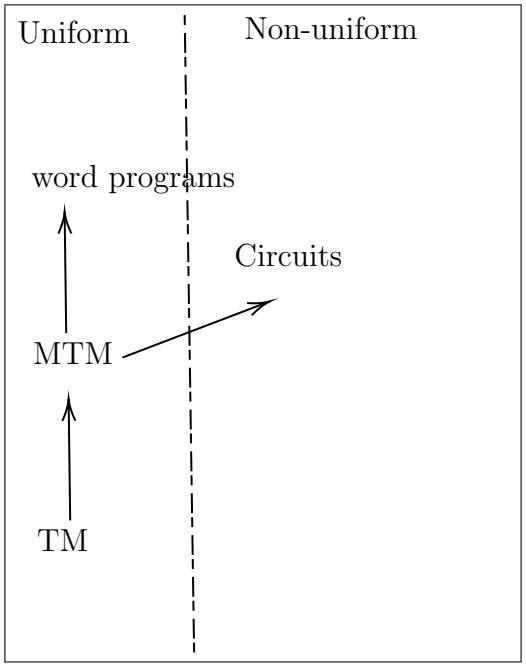


Figure 16.1: Computational models for Time. An arrow from A to B means that B can simulate A efficiently (from time t to $t \log^c t$).

computation, the output would be y if the machine stops with the head on the first symbol of y on the tape (and blank tape elsewhere).

The main differences with the word programs from Chapter 1 is that a tape machine keeps one pointer to the memory, and can only increase or decrease this pointer by 1 in one step; also, the memory cells contain symbols from a constant-size alphabet, as opposed to words. A beautiful aspect of tape machines is that we can give a shorter definition of the model, ignoring the complicated semantics of programming languages:

Definition 16.1. *A tape machine* (TM) with is a map

$$t : S \times A \to S \times A \times \{\text{Left}, \text{Right}, \text{Still}\},$$

called the *transition* map, where S is a finite set of *states,* including two special states *start* and *stop*, and $A := \{0, 1, \_\}$ is the *tape alphabet*. The symbol $\_ \in A$ is called *blank*.

A *configuration* of a TM encodes its state, the tape content, and the position of the head on the tape. It can be written as a triple (s, m, h) where $s \in S$ is the state, $m : \mathbb{Z} \to A$ specifies the tape contents, and $h \in \mathbb{Z}$ indicates the position of the head on the tape.

We write $m[h \leftarrow x]$ for the map m' obtained by m by setting the value at h equal to x, i.e., $m'[h] = x$ and $m'[i] = m[i]$ for $i \neq h$. We adopt the convention that the machine first

writes, then moves the head. So, a configuration

$$(s, m, h) \text{ with } s \neq \text{stop } \textit{yields} \begin{cases} (s', m[h \leftarrow y], h+1) \text{ if } t(s, m[h]) = (s', y, \text{Right}), \\ (s', m[h \leftarrow y], h-1) \text{ if } t(s, m[h]) = (s', y, \text{Left}), \\ (s', m[h \leftarrow y], h) \text{ if } t(s, m[h]) = (s', y, \text{Still}). \end{cases}$$

We say that a TM computes $y \in [2]^*$ on input $x \in [2]^*$ in *time* t (or in t steps) if, starting in configuration $(\text{start}, m, 0)$ where $x = m[0]m[1] \cdots m[|x| - 1]$ and m is blank elsewhere, it yields a sequence of $t + 1$ configurations where the last one is (stop, m', h) and has $y = m[h]m[h + 1] \cdots m[h + |y| - 1]$ and m is blank elsewhere.

Many details are somewhat arbitrary. For example, we allowed the head to remain still. This is redundant, as one can simulate a still operation by a sequence of Right and Left, but makes some programs such as the one in the next example more intuitive (as the machine can write the output and then stop while keeping the head on the output).

Example 16.1. The *parity* of a bit string is the sum of the bits modulo 2. This simple function receives a lot of attention in complexity theory! Here we show that it can be computed by a TM in time $n + 1$ on inputs of length n. This is n steps to read the input, and one to write the output and stop.

A machine with 3 states which scans the input once suffices. The start state also corresponds to a parity of 0 in the input so far. Another state, called *one* corresponds to a parity 1 so far. This highlights an important fact: We can use states to store small amount of information.

We can represent the TM via a diagram such as this

TBD [compile error]

Here, an arrow with label x, y, z from a state s to a state s' indicates that if the machine is in state s and read symbol x from the tape then it writes y on the tape, moves the head according to z, and changes the state to s'. For example, the label from start to one with label 1,_,Right means that if the machine is in state start and reads a 1 then it overwrites it with blank, moves the head right, and changes state to 1.

Note the machine overwrites the input with the blank symbol _ while doing a pass, and flips between states start and one whenever a 1 is read. When the machine reads a blank, it writes the output and stops.

Writing down the transition function quickly becomes uninformative. Instead, it suffices to give a high-level description of the behavior.

Example 16.2. On input an integer $x \in [2]^*$ given in binary, we wish to compute $x + 1$ (i.e., we think of x as an integer in binary, and increment by one). If x is the empty string, we can define $x + 1$ as 0.

This can be accomplished by a TM as follows. Move the head to the least significant bit of x (note the TM starts on the most significant digit of x). If you read a 0, write a 1, move the head to the beginning, and stop. If instead you read a 1, write a 0, move the head left, and repeat.

On an input x of length n, the TM does a constant number of passes over the input, so the running time (the number of steps) is $c|x| + c$. The "$+c$" terms is to take into account the empty string. (Without it, the running time would be 0 on the empty string, but that is not correct, as the machine needs at least one transition to stop.)

Example 16.3. We now describe a TM that on input $x \in [2]^*$ computes the length $n := |x|$ in binary, in time $cn \log n + c$. This illustrates the important technique of keeping a counter next to the head. The TM operates by repeating a basic routine. When starting the routine, the memory is organized as $\ldots \_\_s\_x\_\_ \ldots$ where s and x are bit strings, and the head is on the single blank symbol between s and x. String s is interpreted as an integer. The routine consists of incrementing s by 1, and then moving the symbols to the left of x right by 1 position (thus overwriting the first symbol in x). To increment s we can use the TM in 16.2. The routine is repeated until x is empty, in which case the counter contains the output. To get things started, on input x the machine moves the head left by two positions, writes 0, and then moves the head right.

For example, starting the routine in

$$\_\_11\_000010\_\_$$

would result in

$$\_\_100\_00010\_\_$$

note how the counter increased, and the string on the right got shorter.

How long does the routine take? If the counter has ℓ bits, it takes $c\ell + c$ steps to increment it by 1, as in 16.2. The same number of steps suffice to move it to the right. Since we count up to n, we know $\ell \leq \log n + c$. So each routine takes $\leq c \log n + c$ steps. This routine is repeated n times, so the total time is $\leq cn \log n + cn + c$, where the "$+c$" takes into account the initialization phase that gets things started.

Exercise 16.1. Describe a TM that decides if a string $x \in [2]^*$ is palindrome, and bound its running time.

16.0.1 One tape is all you need

How powerful are tape machines? Perhaps surprisingly, they are all-powerful. The power-time computability thesis from section 1.3 from can be formulated for tape machines.

Power-time computability thesis. For any "realistic" computational model C there is $d > 0$ such that: Anything that can be computed on C in time t can also be computed on a TM in time t^d.

We shall prove below (section 16.2.1) that TMs are power-equivalents to word programs, thus formally linking the computability theses for word programs and TMs. However, the quasi-linear computability thesis (section 1.3) for word programs does *not* hold for TMs. Both TMs and word programs shape our understanding of computation, but in different ways. We will see several examples of this.

We can define complexity classes similarly to Definition 16.2.

Remark 16.1. Some of the results in this chapter involve "fine" distinctions among time and space bounds for which it makes a difference if we work with total functions or partial. Hence we define corresponding total classes. Also, we not consider relations.

Definition 16.2. [Time complexity classes] Let $t : \mathbb{N} \to \mathbb{N}$ be a function. TM-FTime(t) is the set of functions $f : X \subseteq [2]^* \to [2]^*$for which there exists a TM M that computes $f(x)$ within $t(|x|)$ steps on any input $x \in X$ of length $\geq |M|$.

The restriction to boolean functions is denoted TM-Time(t). The restriction to total boolean functions is denoted TM-TotalTime(t).

16.1 TMs with large alphabet

As our first example of robustness, we show that TMs using arbitrarily large tape alphabet can be simulated by TMs with fixed tape alphabet, as in 16.1. This might seem like a detail, but in fact we are going to shortly present a cute problem in the area which will come up again and is, as far as I know, open. We define TMs *with alphabet* B as in Definition 16.1 but using B instead of A; we will only be interested in $B \geq A$ so we can think of adding symbols to A in Definition 16.1. We define TM-Time($t(n)$) similarly.

Theorem 16.1. TM-Time($t(n)$) with alphabet $B \subseteq$ TM-Time($c_B t(n) + c_B n^2$).

Proof. Given a machine N with alphabet B as in the LHS, we construct machine M as in the RHS as follows. We think of the tape of M as divided in *blocks* of $b := \lceil \log_A B \rceil$ symbols in A, where one block encodes one tape symbol of N.

To simulate one step of N, the machine reads the corresponding block and stores the symbol in B in its state. Then, based on the transition function of N, it determines which state to go to, which symbol in B to write on the block, and where to move the head. All these operations can be performed in c_B steps.

This would conclude the argument if the input were given in encoded format, giving a running time of $c_B t(n)$. However, the input is instead given as a string of n symbols in A. I see no other way to proceed except to re-encode the input x. Specifically, we shall re-encode x by repeating its bits. We replace 0 with b 0s, and and 1 with b 1s

To illustrate, suppose given an input

$$x_1x_2x_3x_4x_5 \in A^5$$

we seek to compute the encoded input

$$x_1x_1x_1x_1x_2x_2x_2x_2x_3x_3x_3x_3x_4x_4x_4x_4x_5x_5x_5x_5 \in A^{5b}$$

where for illustration $b = 4$.

It suffices to prove this for $b = 2$, as then we can just repeat, which is a benefit of this particular encoding. This is accomplished as follows. Read and erase an input bit. Move

the head to the second blank to the right, write two copies of this bit, move head back to the second non-blank symbol to the left. Repeat until the input is over.

It takes $c_B n$ steps to duplicate one bit, and we repeat this n times. Overall, the number of steps is $\leq c_B n^2$ steps.

To illustrate, this would be the memory after each bit is copied:

To continue our example, at the end of each stage the tape looks as follows:

$$x_1x_2x_3x_4x_5\_$$
$$x_2x_3x_4x_5\_x_1x_1\_$$
$$x_3x_4x_5\_x_1x_1x_2x_2\_.$$
$$x_4x_5\_x_1x_1x_2x_2x_3x_3\_$$
$$x_5\_x_1x_1x_2x_2x_3x_3x_4x_4\_$$
$$\_x_1x_1x_2x_2x_3x_3x_4x_4x_5x_5\_$$

Note we **QED**

This shows that the definition of P is robust w.r.t. different alphabet sizes. Yet the simulation is unsatisfactory due to the need of re-encode the input which gives a quadratic time blow-up.

Question 16.1. *Is the n^2 term in Theorem 16.1 necessary?*

16.1.1 The universal TM

We give a universal TM, similar to Lemma 1.1. Recall that to establish the time hiearchy Theorem 1.4 we needed a "clocked" version to ensure that we do not simulate for too many steps. Implementing such clocks is more subtle for TMs than it was for word programs, so we state and prove this clocked version. Whereas the clock in Theorem 1.4 only gives a constant factor slow down, here it will give a logarithmic slow down.

Lemma 16.1. There is a TM U that on input (M, t, x) where M is a TM, t is an integer, and x is a string:

-Stops in time $|M|^c \cdot t \cdot |t|$,

-Outputs $M(x)$ if the latter stops within t steps on input x.

Proof. To achieve this running time, the basic idea is to extend the idea in Example 16.3 where recall we computed the length of the input by keeping a counter close to the head. Here, we maintain the invariant that after the simulation of one step of M, the tape of U will contain

$$(x, M, s, t', y)$$

which means that M is in state s, the tape of M contains xy and the head of M is on the left-most symbol of y. Moreover, the head of U will also be on the left-most symbol of y.

The integer t' is the counter decreased at every step. Computing the transition of M takes time $|M|^c$. For example, we can write the transition function of M as a table, and computing this transition then amounts to finding a match in this table with the state with s and the left-most symbol of y. Decreasing the counter takes time $c|t|$, similarly to Example 16.2. To move M and t next to the tape head takes $c|M||t|$ time. **QED**

16.2 Multi-tape machines (MTMs)

A natural and significant extension of TMs is obtained by allowing for more tapes. Each tape has its own head. In one step, the machine can read the symbols under all the heads, and based on that and the state update the cell symbols and the head positions.

Definition 16.3. *A k-TM* with s states is a map

$$t : S \times A^k \to S \times A^k \times \{\text{Left}, \text{Right}, \text{Still}\}^k,$$

called the *transition* map, where S and A are as in 16.1.

A *configuration* of a TM encodes the state, the contents of the k tapes, and the position of the k heads on the tape. It can be written as a tuple $(s, m_1, m_2, \ldots, m_k, h_1, h_2, \ldots, h_k)$ where $s \in S$ is the state, $m_i : \mathbb{Z} \to A$ specifies the contents of Tape i, and $h_i \in \mathbb{Z}$ indicates the position of the head on Tape i.

The yield configuration can be defined similarly to Definition 16.1.

We say that a k-TM (with $k \geq 2$) computes $y \in [2]^*$ on input $x \in [2]^*$ in *time* t (or in t steps) if, starting in configuration $(m_1, m_2, \ldots, m_k, 0, 0, \ldots, 0, \text{start})$ where $x = m_1[0]m_1[1] \cdots m_1[|x|-1]$ and m_1 is blank elsewhere, and all the other m_i are blank, it yields a sequence of t configurations where the last one is $(m'_1, m'_2, \ldots, m'_k, h_1, h_2, \ldots, h_k, \text{stop})$ and the output y is written on the second tape m_2: $y = m_2[h_2]m_2[h_2+1] \cdots m_2[h_2+|y|-1]$ and m_2 is blank elsewhere; we do not require the other tapes to be blank. We call Tape 1 the *input tape* and Tape 2 the *output tape*.

We define k-TM-Time similarly to TM-Time (Definition 16.2).

Exercise 16.2. Define *yield* for 2-TMs by extending the corresponding Definition 16.1 for TMs.

The convention that the input and output tape are distinct will be useful (see Definition 6.2).

Having multiple tapes allows us to solve some problems faster (we will prove in Theorem 16.7 that Palindromes requires quadratic time on TMs).

Exercise 16.3. Prove that Palindromes is in 2-TM-Time(cn). Compare this to the run-time from the the TM in Exercise 16.1.

However, the following result implies that P is unchanged if we define it in terms of TMs or k-TMs.

Theorem 16.2. k-TM-Time$(t(n)) \subseteq$ TM-Time$(c_k t^2(n))$ for any $t(n) \geq n$ and k.

Proof. Let M be a kTM as in the lhs. We give a simulation by a TM using a large alphabet B, which we then simulate with a TM with the fixed alphabet A as in 16.1 thanks to 16.1. The alphabet B is set to $A^k \times \{\hat{}, \square\}$, and thus consists of k symbols from A plus k bits. A symbol in B represents the contents of the corresponding symbols on each of the k tapes of M, and moreover it indicates which of the k heads is at that position: Symbol$\hat{}$ indicates the head is there, $\square$ indicates it is not.

For example, suppose $k = 2$ and the tape contents and head positions (indicated with the symbol $\hat{}$ are as follows):

$$\begin{array}{cccccc} \_ & 0 & \hat{1} & 0 & 1 & \_ \\ \_ & 1 & 1 & 0 & \hat{0} & 1 \end{array}$$

then the corresponding TM tape with alphabet B would be

$$(\_, \_, \square, \square) \quad (0, 1, \square, \square) \quad (1, 1, \hat{}, \square) \quad (0, 0, \square, \square) \quad (1, 0, \square, \hat{}) \quad (\_, 1, \square, \square).$$

To simulate 1 step of the kTM, the TM does a one-way pass on the tape, storing in its state the symbols read under the k tapes. After that, it does one more pass on the tape and updates it based on the transition map of the kTM. For each of the k tapes, this pass involves finding the symbol with the corresponding $\hat{}$,and updating that symbol and its neighbors. (So this pass is not one-way, as the TM may go back to a neighboring symbol.) Since the kTM runs in time t, it does not use more than t cells. So each pass takes time $c_k t$ for the TM, and we can simulate one step in time $c_k t$.

Hence to simulate t steps the time would be $c_k t^2$. By Theorem 16.1 we can simulate this TM over alphabet B using the fixed alphabet A in time $c_B t^2 + c_B n^2$. Because we assumed $t \geq n$, the simulation runs in time $c_B t^2$ as desired. **QED**

Still, working with MTMs rather than TMs makes it slightly easier to prove that problems are in P. We give a basic example next:

Example 16.4. Let us show that computing the addition $x + y$ and multiplication $x \cdot y$ of two input integers x and y is in TM-Time(n^c). By Theorem 16.2 it suffices to give an MTM.

For addition, the standard column addition with carry over can be implemented. Specifically, we can first copy y on a separate tape (neither input or output). Then we can move the head of the input tape to the least significant bit of x, and another head to the least significant bit of y. Starting with a carry of 0 (stored in the state), the TMs writes the lower-order bit of the sum of these two bits of x and y and the carry in the output and stores any carry in the state. It then moves all the heads (on x, y, and on the output tape) to the left, and continues until the end of the input. (If x and y do not have the same length, the TM can fill missing positions with 0, and continue until both x and y are over.)

To compute $x \cdot y$ we can write $y = \sum_{i=0}^{|y|-1} y_i \cdot 2^i$. Hence

$$x \cdot y = \sum_{i=0}^{|y|-1} x \cdot y_i \cdot 2^i.$$

Now, each term $x \cdot y_i \cdot 2^i$ can be computed by first reading the bit y_i. If $y_i = 0$ the term is 0. If $y_i = 1$ we shift x to the left by i positions. To sum these terms, we can use the MTM for addition (on separate tapes).

Finally, we have the following fundamental result about MTMs. It shows how to reduce the number of tapes to *two*, at little cost in time. Moreover, the head movements of the simulator are restricted in a sense that at first sight appears too strong.

Theorem 16.3. k-TM-Time$(t(n)) \subseteq$ 2-TM-Time$(c_k t(n) \log t(n))$, for every function $t(n) \geq n$. Moreover, the 2-TM is *oblivious:* the movement of each tape head depends only on the length of the input.

Using this results one can prove the existence of universal MTMs similar to the universal TMs in Lemma 16.1. However, we won't need this result so we omit the proof. (See the notes for the proof.)

16.2.1 Time vs. TM-Time

We showed earlier than TMs can simulate MTMs with a power slow down. Perhaps more surprisingly, TMs can simulate word programs too, with a comparable slow-down.

Theorem 16.4. Time$(t(n)) \subseteq$ TM-Time$(t^c(n))$, for any $t(n) \geq n$.

Proof. We give a simulation by MTMs, from which one can obtain TMs by Theorem 16.2. Given a word program P, we construct an MTM which dedicates one tape to each register r_i, one tape for the word size, one tape for the memory bound, and one more tape for the memory of the program (represented in a certain format explained below). The tape corresponding to register r_0 is initialized to the input length n using Example 16.3.

The program and program counter (indicating which instruction is to be read next) can be stored in the transition function and state of the TM, as they have constant size. Instructions involving assignments, arithmetic operations, and conditional jumps, as well as MALLOC and FREE, can be computed in power-time. For example, for addition and multiplication we can use Example 16.4. But in fact for this simulation something weaker suffices, as these operations are on registers on w bits, and w is logarithmic in our desired time bound. All these operations are done on registers whose bit length is the current word length w.

The part that is less obvious is how to handle memory. The memory of the word program is represented on the MTM tape as pairs (i, v) meaning $m[i] = v$. In an initial phase, the MTM inserts pairs corresponding to the input. To simulate one Write instruction $M[r_i] := r_j$ the MTM appends the pair (r_i, r_j), where each register has a number of bits corresponding to the word length. To simulated a Read instruction $r_i := M[r_j]$, the MTM goes through all these pairs in order, and for any pair of the type (r_j, v) it finds, it copies v on r_j. This way of doing things allows us to correctly simulate Reads and Writes interspersed with MALLOC/FREE instructions (note the memory words of the program can hold larger and larger integers, if MALLOC instructions are executed). If no pair is found, r_i is set to 0.

After each operation, the program counter is increased by 1, except when a conditional jump is executed, in which case it is set accordingly.

Since the program runs in time $t(n)$ it can only access $t(n)$ memory locations, and so the number of pairs is $t(n)$. Moreover, the word length is always $\leq \log t(n)$ (since increasing it after that costs more than the time budget). **QED**

In the other direction, we remark that word programs can simulate TMs as well. This is less surprising, but not completely obvious as the TM handles the memory in a different way (as a tape unbounded on both sides).

Theorem 16.5. TM-Time$(t(n)) \subseteq$ Time$(t^c(n))$, for any $t(n) \geq n$.

Proof. Given a TM, we construct a word program as follows. The program contains the transition function of the TM and its state. For any input (s, a) to the transition function t of the TM we have a conditional jump taking the program to a line corresponding to the state in $t(s, a)$ and performing the corresponding write and head movement.

Memory cells $0..n-1$ contain the input. Memory cells $n, n+2, n+4, \ldots$ are used for the tape symbols to the right of the input, and memory cells $n+1, n+3, n+5, \ldots$ for the symbols to the left. We dedicate a register for the position of the tape head of the TM. If the tape head moves past the cells that we can index, we execute two MALLOC instructions (one for each side). **QED**

16.3 TMs vs circuits

By Theorem 16.5 that word programs can simulate TMs, and hence by Theorem 2.5 circuits can simulate TMs. However, as noted before in Chapter 5, this simulation would incur a quadratic loss. Interestingly, for MTMs we can give a direct simulation which achieves quasilinear time.

Theorem 16.6. k-TM-Time$(t(n)) \subseteq \bigcup_a$ CktSize$(a \cdot t(n) \log t(n))$, for any $t(n) \geq n$.

One can prove this from Theorem 16.3 (see Problem 16.10). However, next we give a direct proof that doesn't need Theorem 16.3.

Proof. We prove this for $k = 1$, the extension to larger k does not need new ideas and is omitted.

For this proof it is convenient to represent a configuration of a TM in a slightly different way. Wlog we let the states of the machines be $[s]$, and we consider an extra symbol $\square$ indicating the absence of the head. We represent a configuration as a string of symbols over the alphabet $A \times ([s] \cup \{\square\})$. String

$$(a_1, \square)(a_2, \square) \ldots (a_{i-1}, \square)(a_i, j)(a_{i+1}, \square) \ldots (a_m, \square)$$

with $j \in [s]$ indicates that (1) the tape content is $a_1 a_2 \ldots a_m$ with blanks on either side, (2) the machine is in state $j \in [s]$, and (3) the head of the m. (Yes, only one tape.) achine is on

the i tape symbol a_i in the string. Locality of computation here means that one symbol in a string only depends on the symbols corresponding to the same tape cell i in the previous step and its two neighbors – three symbols total – because the head only moves in one step.

Given a TM M, we construct a circuit S_m that on input a configuration of M with m tape symbols where the head position is within $m/4$ symbols from the center, it computes the configuration reached by M after $m/4$ steps of the computation.

We shall give an inductive construction of S_m satisfying

$$\text{Size}(S_m) \leq 2 \cdot \text{Size}(S_{m/2}) + cm$$

with base case $\text{Size}(S_c) \leq c_M$. This implies $\text{Size}(S_t) \leq c_M t \log t$. From the output of S_t we can compute the output of the function as in the proof of Theorem **??**. This concludes the proof.

To construct S_m we think of the m symbols as divided into c blocks, and we rely on a couple of auxiliary circuits. Circuit H_m, given an m-symbol configuration, computes in which block the head is; circuit R_m given an m-symbol configuration and $i \leq c$, rotates the blocks by i positions.

To illustrate, consider a configuration of 15 symbols from tape symbol -7 to tape symbol 7, where we put $\hat{}$ on top of the symbol where the head lies, and do not show the state of the TM. Divide the configuration in 5 blocks, of size 3, numbered $-2, -1, \ldots, 2$:

$01\hat{0}$	000	001	110	1__
-2	-1	0	1	2

The value of H_{15} would be -2, as the head is on position -5 in block -2.

After rotation via R_{15} we would obtain:

110	1__	$01\hat{0}$	000	001
1	2	-2	-1	0

.

Note how the head is now in position 1, much closer to the center.

We can now program S_m as follows. First run H_m and let the head be in block i. Use R_m to rotate the blocks by i positions so that the head is in a block closest to the middle. By our choice for the number of blocks, the head is guaranteed to lie within $m/8$ of the middle. Hence we can run $S_{m/2}$ to compute the configurations obtained after $m/4$ steps. Now repeat the trick. Run again H_m to get j, and then R_m to move block j closest to the middle, and then $S_{m/2}$. Finally, use R_m to restore the blocks by rotating them back by $i + j$ positions.

This circuit simulates $(m/2)/4 + (m/2)/4 = m/4$ steps, as desired. The circuits R and H can be implemented using cm gates. **QED**

16.4 The grand challenge for TMs

The grand challenge for TMs can be stated as follows:

It is consistent with our knowledge that any problem in a standard algorithm textbook can be solved

1. in Time cn^2 on a TM, and
2. in Time cn on a 2-TM, and

Recall that we can simulate 2-TMs by a TM with a quadratic loss, so the two problems are related.

16.5 Information bottleneck: Palindromes requires quadratic time on TMs

Intuitively, the weakness of TMs is the bottleneck of passing information from one end of the tape to the other. We now show how to formalize this and use it show that deciding if a string is a palindrome requires *quadratic* time on TMs, which is tight and likely matches the time in Exercise 16.1. The same bound can be shown for other functions; palindromes just happen to be convenient to obtain matching bounds.

Theorem 16.7. Palindromes $\notin$ TM-Time($t(n)$) for any $t(n) = o(n^2)$.

More precisely, for every n and s, an s-state TM that decides if an n-bit input is a palindrome requires time $\geq cn^2/\log s$.

The main concept that allows us to formalize the information bottleneck mentioned above is the following.

Definition 16.4. A *crossing sequence* of a TM M on input x and boundary i, abbreviated i-CS, is the sequence of states that M is transitioning to when crossing cell boundary i (i.e., going from Cell i to $i+1$ or vice versa) during the computation on x.

Example 16.5. We think of a step of a TM as first changing state and then moving the head. We write $u\overset{i}{\hat{v}}w$ if the tape content is uvw (with infinite blanks on each side) and the TM is in state i with the head on v, where $u, w \in A^*$ and $v \in A$, see Definition 16.1. The computation

_	Start $\hat{0}$	0	0	_
_	_	$\overset{5}{\hat{0}}$	0	_
_	_	0	$\overset{2}{\hat{0}}$	_
_	_	0	1	$\overset{2}{\hat{\ }}$
_	_	0	$\overset{3}{\hat{1}}$	_
_	_	$\overset{1}{\hat{0}}$	1	_
_	_	_	Stop $\hat{1}$	_

on input 000 has the 2-cs (marked with double vertical line; first column is cell 1) equal to $2, 1, \text{Stop}$.

The idea in the proof is very interesting. If M accepts inputs x and y and those two inputs have the same i-CS for some i, then we can "stitch together" the computation of M on x and y at boundary i to create a new input z that is still accepted by M. The input z is formed by picking bits from x to the left of cell boundary i and bits from y to the right of i:

$$z := x_1 x_2 \cdots x_i y_{i+1} y_{i+2} \cdots y_n.$$

The proof that z is still accepted is left as an exercise.

Example 16.6. The following computation on input 011 has the same 2-cs as the previous example

_	Start $\hat{0}$	1	1	_
_	_	$\overset{4}{\hat{1}}$	1	_
_	_	0	$\overset{2}{\hat{1}}$	_
_	_	$\overset{1}{\hat{0}}$	1	_
_	_	_	Stop $\hat{1}$	_

.

Both computations output 1.

Now the point is that the TM would also accept the "stitched" input

001

because on that input the TM has the following "stitched" computation:

_	Start $\hat{0}$	0	1	_
_	_	$\overset{5}{\hat{0}}$	1	_
_	_	0	$\overset{2}{\hat{1}}$	_
_	_	$\overset{1}{\hat{0}}$	1	_
_	_	_	Stop $\hat{1}$	_

.

Note that the number of steps of the stitched computations needs not be the same.

Now, for many problems, stitched input z should *not* be accepted by M, and this gives a contradiction. In particular this will be be the case for palindromes. We are going to find two palindromes x and y that have the same i-CS for some i, but the corresponding z is not a palindrome, yet it is still accepted by M. We can find these two palindromes if M takes too little time. The basic idea is that if M runs in time t, because i-CSs for different i correspond to different steps of the computation, for every input there is a value of i such that the i-CS is short, namely has length at most $t(|x|)/n$. If $t(n)$ is much less than n^2, the length of this CS is much less than n, from which we can conclude that the number of CSs is much less than the number of inputs, and so we can find two inputs with the same CS.

Proof of Theorem 16.7. Let n be divisible by four, without loss of generality, and consider palindromes of the form

$$p(x) := x0^{n/2}x^R$$

where $x \in [2]^{n/4}$ and x^R is the reverse of x.

Assume there are $x \neq y$ in $[2]^{n/4}$ and i in the middle part, i.e., $n/4 \leq i \leq 3n/4 - 1$, so that the i-CS of M on $p(x)$ and $p(y)$ is the same. Then we can define $z := x0^{n/2}y^R$ which is not a palindrome but is still accepted by M, concluding the proof.

There remains to prove that the assumption of Theorem 16.7 implies the assumption in the previous paragraph. Suppose M runs in time t. Since crossing sequences at different boundaries correspond to different steps of the computation, for every $x \in [2]^{n/4}$ there is a value of i in the middle part such that the i-CS of M on $p(x)$ has length $\leq ct/n$. This implies that there is an i in the middle s.t. there are $\geq c2^{n/4}/n$ inputs x for which the i-CS of M on x has length $\leq ct/n$.

For fixed i, the number of i-CS of length $\leq \ell$ is $\leq (s+1)^\ell$.

Hence there are $x \neq y$ for which $p(x)$ and $p(y)$ have the same i-CS whenever $c2^{n/4}/n \geq (s+1)^{ct/n}$. Taking logs one gets $ct\log(s)/n \leq cn$. **QED**

Exercise 16.4. For every s and n describe an s-state TM deciding palindromes in time $cn^2/\log s$ (matching Theorem 16.7).

Exercise 16.5. Let $L := \{xx : x \in [2]^*\}$. Show $L \in$ TM-Time(cn^2), and prove this is tight up to constants.

One may be tempted to think that it is not hard to prove stronger bounds for similar functions. In fact as mentioned above this has resisted all attempts!

16.5.1 1.5TM

As discussed earlier, we don't know how to prove that, say, 3Sat cannot be computed in linear time on a 2TM. For single-tape machines, we can prove quadratic bounds, for palindromes (Theorem 16.7) and 3Sat (Problem 16.6). Next we consider an interesting model which is between 1TM and 2TM and is a good indication of the state of our knowledge.

Definition 16.5. A 1.5TM is like a 2TM except that the input tape is read-only.

Theorem 16.8. 3Sat requires time n^{1+c} on a 1.5TM.

The proof is an adaptation of the ideas in Theorem 6.21.

Exercise 16.6. Prove Theorem 16.8 following this guideline:

1. Let M be a 1.5TM running in time $t(n)$. Divide the read-write tape of M into consecutive blocks of b cells, shifted by an offset $i < b$. (So the the first cells of the blocks include $i, i+b, i+2b, \ldots$.) Prove that for every input $x \in [2]^n$ there is i such that the sum of the lengths of the crossing sequences between any adjacent blocks of the computation M on x is at most $t(n)/b$. Here a crossing sequence also encodes the position of the head on the input tape, and the time at which each crossing occurs.

2. Prove that 1.5TM-Time$(n^{1.1}) \subseteq \exists y \in [2]^{n^{1-c}}$TiSp$(n^c, n^{1-c})$. (The right-hand side is the class of functions $f : [2]^* \to [2]$ for which there is a word program P that on input (x, y), where $|x| = n$, runs in time n^c and uses memory cells $0..n^{1-c}$ and s.t. $f(x) = 1 \Leftrightarrow \exists y \in [2]^{n^{1-c}} P(x, y) = 1$.)

3. Conclude the proof.

16.6 TM time hierarchy

Using the universal TM (Lemma 16.1) and reasoning as the time hierarchy theorem (Theorem 1.4) one can prove that TM-Time$(t(n) \log t(n)) \subsetneq$ TM-Time$(o(t(n))$, for all functions t which are "TM-time constructible," a notion similar to time-constructibility (Definition 1.10) but for TMs. Again, such functions t include most functions of interest. I do not state this time hierarchy result precisely. Instead, let us not that there is a gap of about $\log t$ in the bounds, compared to the constant gap in Definition 1.10. This gap arises from the corresponding gap in the universal simulation, see section §16.1.1.

Is there a tighter hiearchy result for TM? For example, we can we separate time $n \log^{0.9} n$ from $n \log^{0.8} n$ on a TM? Surprisingly, the answer is no: All running times below $n \log n$ compute the same functions! This is true for total functions, recall Definition 16.2.

Theorem 16.9. TM-TotalTime$(t(n))$ = TM-TotalTime$(n+1)$ for any $t(n) = o(n \log n)$.

Note that time $n+1$ is barely enough to scan the input. Indeed, the corresponding machines in Theorem 16.9 will only move the head in one direction. The "+1" only reflects that we charge one time step to write the output and stop in 16.1. Moreover, such machines have no use of writing to the tape (except to write down the output). These constrained machines are well-studied and are known as *regular* or *finite-state-automata*.

Definition 16.6. A function $f : [2]^* \to [2]$ is *regular* if it can be computed in time $n+1$ by a TM that does one pass over the input, only moving the head right.

The rest of this section is devoted to proving the above theorem. Let M be a machine for f witnessing the assumption of the theorem. We can assume that M stops on every input (even though our definition of time only applies to large enough inputs), possibly by adding $\leq n$ to the time, which does not change the assumption on $t(n)$. The theorem now follows from the combination of the next two lemmas.

Lemma 16.2. Let M be a TM running in time $t(n) \leq o(n \log n)$, on every $x \in [2]^*$. Then on every input $x \in [2]^*$ every i-CS with $i \leq |x|$ has length $\leq c_M$.

Proof. Assume towards a contradiction that for every $b \in \mathbb{N}$ there are inputs which have crossing sequences of length $\geq b$. Specifically let $x(b)$ be a shortest input of length $n(b) := |x(b)|$ such that there exists $j \in \{0, 1, \ldots, n(b)\}$ for which the j-CS in the computation of M on $x(b)$ has length $\geq b$.

We have that $n(b) \to \infty$ for $b \to \infty$ (see exercise below).

There are $n(b)+1 \geq n(b)$ tape boundaries within or bordering $x(b)$. If we pick a boundary uniformly at random, the average length of a CS on $x(b)$ is $\leq t(n(b))/n(b)$. Hence there are $\geq n(b)/2$ choices for i s.t. the i-CS on $x(b)$ has length $\leq 2t(n(b))/n(b)$.

The number of such crossing sequences is

$$\leq (s+1)^{2t(n(b))/n(b)} = (s+1)^{o(n(b)\log(n(b))/n(b)} = n(b)^{o(\log s)}.$$

Hence, the same crossing sequence occurs at $\geq (n(b)/2)/n(b)^{o(\log s)} \geq 4$ positions i, using that $n(b)$ is large enough.

Of these four, one could be the CS of length $\geq b$ from the definition of $x(b)$. Of the other three, two are on the same side of j. We can remove the corresponding interval of the input without removing the CS of length $\geq b$. Hence we obtained a shorter input with a CS of length $\geq b$, contradicting our definition of $x(b)$ and so our initial assumption. **QED**

Exercise 16.7. Prove $n(b) \to \infty$ for $b \to \infty$.

Lemma 16.3. Suppose $f : [2]^* \to [2]$ is computable by a TM such that on every input x, every i-CS with $i \leq |x|$ has length $\leq b$. Then f is computable in time n by a TM with c_b states that only moves the head in one direction.

Exercise 16.8. Prove this.

16.7 Sub-logarithmic space

TMs are convenient for defining space. Indeed, for space bounds $\geq \log n$, space complexity on TMs and on word programs is the same, up to constant factors. More importantly, TMs allow us to investigate sub-logarithmic space bounds that we cannot make sense of over word programs. We define TM-Space as a 2TM whose input tape is read-only. Recall this is consistent with Definition 16.3 of 2-TM according to which the output is written on the second tape.

Definition 16.7. A function $f : X \subseteq [2]^* \to [2]$ is computable in TM-Space($s(n)$) if there is a 2TM M which on input x on the first tape, where $x \in X$, $|x| \geq |M|$ computes $f(x)$ and M never writes on the first tape and moreover the heads are limited as:

(1) the head on the second tape is always in $[s]$,

(2) the head on the first (input) tape is always in $[-1..n]$ on inputs of length n (corresponding to the input bits plus a blank symbol on each side as delimiter).

The restriction to total functions $f : [2]^* \to [2]$ is denoted by TM-TotalSpace($s(n)$).

Condition (2) that the input cell does not wander off into blanks after the end of the input makes sense since the tape is read-only; it is also convenient technical as it allows us to bound configurations precisely.

Definition 16.8. A configuration of a space-s machine M contains cs bits for the contents of the 2nd tape, $\log n + c$ bits for the position of the head on the first tape, and finally c_M bits for the program, including the program counter.

Lemma 16.4. Configurations of a space-s machine M on inputs of length n take (where $s = s(n)$) $cs + \log n + c_M$ bits. In particular, their number is $\leq c^s \cdot n \cdot c_M$.

Proof. To record the head on the input tape it suffices to use $\log n + c$ bits. This relies on the convention in Definition 16.7 of space-bounded computation that the input head does not wander more than one blank symbol past the input. The other terms are immediate from the definition. **QED**

First we prove a result analogue to Theorem 16.9 that space $\leq c \log \log n$ equals regular. Then we prove two results showcasing the surprising power of small-space algorithms.

Theorem 16.10. TM-TotalSpace($c \log \log n$) = Regular.

Proof. By Theorem 6.1 we can simulate a machine M running in Space($c \log \log n$) in time

$$c_M \cdot c^{c \log \log n} \cdot n \leq c_M \cdot \sqrt{2}^{\log \log n} \cdot n = c_M \cdot n \cdot \sqrt{\log n}.$$

The result now follows from Theorem 16.9. **QED**

Now is a good time to pause and ask yourself if you think TM-TotalSpace($o(\log n)$) = Regular as well. After all, it is hard to imagine what you can do in space $o(\log n)$, since one can't even compute the length of the input. In fact, we have the following surprising result showing that increasing space $c \log \log n$ by a constant factor more gives more power.

Theorem 16.11. TM-TotalSpace($c \log \log n$) $\neq$ Regular.

Proof. The language witnessing the separation is that of deciding if a given input string is of the form

$$0\#1\#10\#11\#100\#101\#...$$

where # is a separator, and note adjacent numbers are consecutive.is in the language.

This problem can be shown to be not regular similarly to Problem 16.1.

To show that it can be solved in space $c \log \log n$, first check if the first number if 0. Then check that for every pair of consecutive numbers $x\#y$ in the sequence, $y = x+1$. The critical point is that this check can be done in space $c \log |x|$, regardless of the length of y, and if the check passes then the length of y can obviously be bounded as well, so the numbers stay $\leq \log n$. **QED**

Finally, consider computing Majority. It is not regular (see Problem 16.1). In particular by Theorem 16.10 it cannot be solved in constant space. Yet, surprisingly, it can be solved in constant space using randomness. A *randomized* TMs has two transition functions σ_0 and σ_1, where each is as in Definition 16.1. At each step, the TM uses σ_0 or σ_1 with probability $1/2$ each, corresponding to tossing a coin.

Theorem 16.12. Majority can be solved in constant space by a randomized TM

The corresponding time bound will be exponential.

Proof. On input $x \in [2]^n$, the algorithm performs many one-way scans of the input. In each scan the machine checks if $x = N_{3/4}$ and if $x = N_{1/4}$. Here, N_β is a string of iid bits with prob. p of being 1. Let E_β be the event that $x = N_\beta$ and p_β be $\mathbb{P}[E_\beta]$, all depending on x.

The algorithm keeps performing scans until exactly one of $E_{3/4}$ and $E_{1/4}$ happens, in which case it outputs 1 if it was $E_{3/4}$ and 0 otherwise.

Let us analyze this algorithm. The prob. of outputting 1 is

$$\frac{p_{3/4}(1-p_{1/4})}{p_{3/4}(1-p_{1/4}) + p_{1/4}(1-p_{3/4})}.$$

We claim that if x has weight more than $n/2$ this probability is $> 2/3$. This is equivalent to

$$p_{3/4}(1-p_{1/4}) \geq 2p_{1/4}(1-p_{3/4})$$

which is implied by $p_{3/4}(1-o(1)) \geq 2p_{1/4}$. This is true because every pair of 1 and 0 in x gives the same factor $3/4 \cdot 1/4$ in $p_{3/4}$ and $p_{1/4}$. Since x has more zeroes than ones, we are left with at least one factor $3/4$ in $p_{3/4}$ with a corresponding factor $1/4$ in $p_{1/4}$.

A similar analysis holds if x has weight less than $n/2$. **QED**

16.8 Problems

Problem 16.1. Prove that Majority is in TM-Time($cn \log n$)but is not regular. For the negative result, use crossing sequences but not pumping lemmas or other characterization results not covered in this book.

By Theorem 16.9 Majority provides a separation between TM-Time($o(n \log n)$) and TM-Time($cn \log n$).

Problem 16.2. Show that Palindromes can be solved in time $n \log^c n$ on a randomized TM with a single tape. Hint: View the input as coefficients of polynomials.

16.9 Notes

See the notes to Chapter 1. Crossing sequences and the quadratic bound for palindromes are from [141]. The time hierarchy for tape machines originates in [143] and was later optimized in [144]. Theorem 16.9 follows by combining results in [142, 174].

Theorem 16.12 is from [96]. The result, "a bit unexpected," is proved there for strings 0^n1^n, but a similar proof applies to majority.

Theorem 16.11 is from [261].

Problem 16.2 is from [95].

16.10 Problems

Problem 16.3. Show Space(n) is not contained in 1.5TM-Time($n^{1.99}$). You can use the Space Hierarchy Theorem that, say, Space($n^{0.9}$) $\neq$ Space(n).

Problem 16.4. [Indexing] Prove that Indexing (defined in Exercise 2.1) is in TM-Time($cn \log n$).

Problem 16.5. In this problem you will explore an alternative proof of the results in section §5.3.

(1) Show Time(t) $\subseteq \exists \cdot c$-TM-Time($t \log^c t$). Hint: Follow the proof structure in section §5.3. Use the MergeSort algorithm.

(2) Prove Theorem 5.4 from (1) using a simulation from Chapter 16.

Problem 16.6. Prove that 3Sat is not TM-Time($n^{1.99}$). (Hint: Consider a variant of the palindromes problem where the input bits are suitably spaced out with zeroes. Prove a time lower bound for this variant by explaining what modifications are needed to the proof of Theorem 16.7. Conclude by giving a suitable reduction.)

Problem 16.7. Show that computing division x/y of two input integers x and y, defined as the largest z s.t. $yz \leq x$, is in P.

Problem 16.8. A TM is b-block-respecting if on any input x it crosses cells boundaries that are multiples of b only during computation steps that are multiple of b. In this problem you will show that any MTM can be transformed into a b-block-respecting MTM, with only a constant overhead in time. To isolate the essence of this problem, we shall consider TMs with an extra feature. A TM with a b-clock is a TM where in addition the transition function can depend on whether or not the current computation step is divisible by b.

Let M be a k-TM running in time $t(n)$, and let $b(n)$ be a function. Give an equivalent c_k-TM with a $b(n)$-clock that is $b(n)$-block-respecting and runs in time $ct(n)$.

Hint: Triplicate each tape, and for each block have both the preceding and following block, *reversed*. Explain how the simulation is carried out and where the clock is used.

Problem 16.9. Let $f : [2]^n \to [2]$ be computable by an s-state k-TM in time t. Think of the input as m cells of w bits, so that $n = mw$. Consider circuits made or arbitrary functions which take as input $c_{s,k}$ cells and output one cell. (Each wire in this circuit carries one cell – the bits cannot be "broken up," but the result would be non-trivial even if they could.) Show that f can be computed by such circuits of size $(t/w)^c$. For example, if $t = 100n$ and $w = 0.01n$ we have circuits of a constant number of gates.

Problem 16.10. Prove Theorem 16.6 assuming Theorem 16.3.

16.11 Notes

The oblivious simulation of MTMs by 2TMs, Theorem 16.3, is from [144, 226].

Theorem 16.6 is towards the end of [226].

The proof of Theorem 16.8 but for computing a function in $\Sigma_1\text{Time}(cn)$ is from [195]. The proof was apparently rediscovered in [287] and extended to 3Sat.

For multi-tape machines, a separation between deterministic and non-deterministic linear time is in [225, 240].

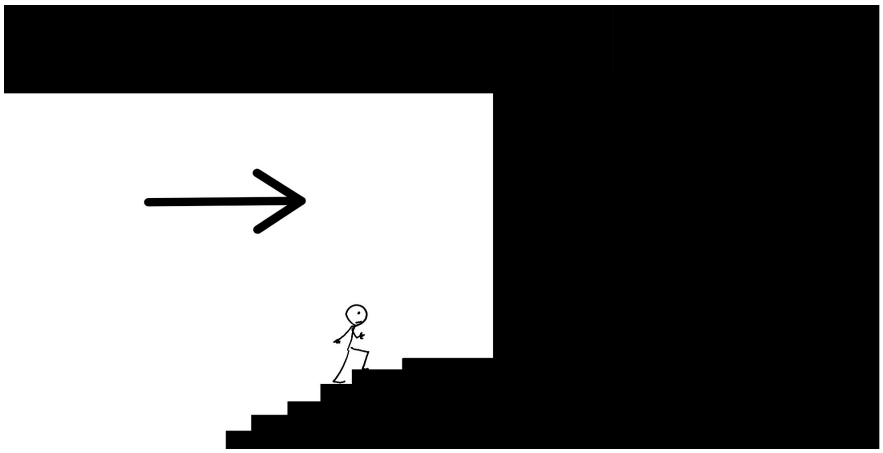
Chapter 17

Quantum

TBD

Chapter 18

Barriers



In an attempt to understand the Grand Challenge (section §1.8), one can identify several proof techniques and show or speculate that they cannot solve it. Such arguments are known as "barriers." Several barriers have been put forth, and in fact there are even barriers to barriers, i.e., arguments indicating that proving a barrier is difficult, making complexity theory a rather philosophical and introspective field.

The two main barriers are the *black-box* (a.k.a. *oracle* or *relativization* barrier) and the *natural proofs* barrier.

18.1 Black-box

As hinted, many of the results we have shown don't really exploit the specifics of the model we are working with, but work in greater generality. How to make this more precis e? When programming, we can think of having access to a powerful *library*, a.k.a. *subroutine*, *oracle*, *black box*, etc. If our argument also applies when given access to *any* black box we say it is black-box, or that it *relativizes*. The black-box barrier helps us understand the limits of basic simulation arguments, including diagonalization, which tend to relativize.

The precise model is the same we encountered in Chapter 4, see Definition **??**. We equip our models, such as TMs, with access to a function $f : [2]^* \to [2]$, known as *oracle*. At any point, the model can ask for the value of the function at an input that has been computed. In the case of TMs, we can think of a special oracle tape and a special oracle state. Upon entering the state, the contents $x \in [2]^*$ on the oracle tape are replaced in one step with $f(x) \in [2]$. We can then define corresponding complexity classes, denoted P^f, and so on.

To illustrate, consider the separation $\mathrm{P} \neq \mathrm{Exp}$, which follows from the Time Hierarchy Theorem **??**. The result relativizes:

Theorem 18.1. For every oracle $f : [2]^* \to [2]$, $\mathrm{P}^f \neq \mathrm{Exp}^f$ (i.e., $\mathrm{P} \neq \mathrm{Exp}$ relativizes).

Proof. Diagonalization and the time hierarchy work just as well for oracle machines. Specifically, when simulating a machine M running in time t with a machine M' running time $t' > t$, the simulation proceeds as before, and if M queries the oracle then M' does that as well. **QED**

Exercise 18.1. Prove $\mathrm{PSpace}^f \subseteq \mathrm{Exp}^f$ for every oracle (i.e., $\mathrm{PSpace} \subseteq \mathrm{Exp}$ relativizes).

Next we argue that relativizing techniques cannot resolve other major questions. Perhaps the simplest example is for P vs. PSpace, because the way the oracle is accessed is clear. We show that there are oracles w.r.t. which the P vs. PSpace question can be resolved either way, so one cannot resolve in a way that extends to all oracles, as in Theorem 18.1.

Theorem 18.2. There are oracles f, g s.t.:
$\mathrm{P}^f = \mathrm{PSpace}^f$, and
$\mathrm{P}^g \neq \mathrm{PSpace}^g$.

Proof. The oracle f can be any PSpace-complete function. For example, let f take as input $(M, 1^s, x)$, simulate M using space s on input x for $\leq |M|^s$ steps, and return its answer. If M exceeds space s, the oracle returns 0. We claim that $\mathrm{PSpace}^f = \mathrm{PSpace}$. This is just because the oracle queries can be answered by direct simulation using power space. Further, $\mathrm{PSpace} \subseteq \mathrm{P}^f$, because an algorithm on the rhs can query f on the input corresponding to an algorithm on the lhs. The proof is completed by combining the two claims.

The construction of g is more involved. To illustrate the main idea, let us first assume that oracle machines, on inputs of length n, only query the oracle at inputs of length n as well. Then we can define the oracle g as follows. Let $M_1, M_2, \ldots$ be an enumeration of all oracle machines. On an input of length n, run M_n for $2^n - 1$ steps on input 1^n, returning zero for all oracle queries (if M doesn't stop, force stop and output, say, 0). If M outputs 1 then set g to be zero on all $[2]^n$. Otherwise, set g to be zero on all $[2]^n$ except for one input y that M didn't query, where $g(y) = 1$. This concludes the definition of the oracle. Now consider the problem H^g of determining, on input 1^n, if there exists $y \in [2]^n : g(y) = 1$. This problem is in PSpace^g, by going through all y. But by construction it isn't in P^g. To show the latter, assume there is a s.t. M_a solves the problem in time n^a. Pick b large enough so

that M_b is equivalent to M_a and $b^a < 2^b$, and consider $M_b(1^b) = M_a(1^b)$. By construction, g returns 0 on all oracle queries, and queries $< 2^b$ of the inputs of length b. By construction, the machine returns 0 iff there is $y \in [2]^b : g(y) = 1$. Contradiction.

That's the main idea. Now, for completeness, we drop the assumption that on inputs of length n only oracle queries of length n are made. The idea is just to pick sufficiently spaced-out inputs so that the above strategy can be executed again. We set values of the oracle one machine M_i at the time (whereas previously one could have processed all machines simultaneously). For each machine we set the values of the oracle at one more input length, so that on that input length the power-time oracle machine makes a mistake. In the generic iteration, we start with having an oracle s.t. H^g cannot be solved by machine M_j running in time n^j, for any $j < i$, and only the first c_i input lengths of the oracle are set. (The latter condition is w.l.o.g..) Again, our goal is to extend the oracle so that H^g cannot be solved even by M_i running in time n^i. To do this, consider an input length m s.t. (1) $M_i = M_m$, (2) $m^i < 2^m$, and (3) $g(y)$ was not set for any $y \in [2]^m$. We set the oracle as before. That is, run M_m on input 1^m for $2^m - 1$ steps, answering all oracle queries of length m or bigger with zero, and all oracle queries of length $< m$ following the definition of g (which we can assume to be set on all lengths $< m$, and note may involve both 0 and 1 outputs). Then we define g as before: If M_m outputs 1 we set g to be zero on inputs of length m, otherwise we set it to 1 on one of the queries of length m that wasn't queried by M_m on input 1^m. **QED**

Exercise 18.2. (1) Write down the definition of NP^f. (2) show that P vs. NP cannot be solved via black-box techniques, by following the proof of Theorem 18.2.

18.2 Natural proofs

The natural proofs barrier aims to explain the limit of *combinatorial* proof techniques. The idea is simple:

(1) Most combinatorial proof techniques against a class of functions F (for example, F are the functions on n bits computable by circuits of size n^{10} and depth $10 \log n$) do more than providing a separation: They yield an *efficient* algorithm that given the truth table of length 2^n of a function can distinguish tables coming from F from those coming from uniformly random functions.

(2) The classes F for which we would like to prove impossibility are believed to be powerful enough to compute *pseudorandom functions*, i.e., truth tables that *cannot* be efficiently distinguished from uniformly random functions.

Note that (2) is not known unconditionally, but just believed to be the case. This is where complexity theory gets quite philosophical. We can't really *prove* (2) without solving the Grand Challenge, in which case these barriers are not actual barriers. On the other hand one can have a *belief* that (2) is indeed true even though we can't prove it, and if that's the case indeed to solve the Grand Challenge one needs to somehow bypass (1) and find alternative techniques. We currently don't seem to have such techniques.

That's not all, however. In some cases we can bypass (2) and claim unconditionally that efficient techniques won't work. The idea is that if lower bounds are not true, we can't prove them; but if lower bounds are true, then we can use them to construct pseudorandom functions.

18.2.1 TMs

We illustrate the natural-proofs barrier for 1TMs. Let us revisit the information bottleneck technique from section §16.5 to show that from it we can extract an efficient algorithm to distinguish truth-tables computed by fast 1TMs from uniform functions. One is tempted to consider a test checking if there is a large set where the function is constant, and moreover X is a product set $X = Y \times Z$. However, it is not clear that this test would be efficient. It is more convenient to use the simulation of 1TMs by low-communication protocols, Theorem 13.6, and use the quantity R from section 13.2.2.

18.2.1.1 Telling subquadratic-time 1TMs from random

Given the truth-table of a function $f : [2]^n \to [2]$, our test D will consider the function $f_0 : [2]^{n/3} \times [2]^{n/3} \to [2]$ defined as $f_0(x, y) := f(x0^{n/3}y)$, and check if $R(f_0) \geq 2^{-cn}$.

First, let us verify that fast 1TMs indeed pass D. Let M be an s-state 1TM running in time t computing $f : [2]^n \to [2]$. By Theorem 13.6, f_0 has 2-party protocols with communication $d := c(\log s)t/n$ and error $\leq 1/2$. By Lemma 13.3, $R(f_0) \geq 2^{d/c} \geq s^{ct/n}$.

Second, let's verify that D is efficiently computable. Indeed, following the definition we can compute D in time 2^{cn}, which is power in the input length 2^n.

Third, and finally, we show that random functions $U : [2]^n \to [2]$ don't pass the test. Indeed, we have

$$\mathbb{E}_U[R(U)] = \mathbb{E}_U \mathbb{E}_{\substack{x_1^0, x_2^0 \\ x_1^1, x_2^1}} [U(x_1^0, x_2^0) + U(x_1^0, x_2^1) + U(x_1^1, x_2^0) + U(x_1^1, x_2^1)].$$

When the x_1 are distinct and the x_2 are distinct, the expectation is 0. In the other case the expectation is 1, and this happens with prob. $\leq 2^{-cn}$. Therefore $\mathbb{E}_U[R(U)] \leq 2^{-cn}$. Moreover, R is never negative, for $R(f) = \mathbb{E}_{\substack{x_1^0 \\ x_1^1}} \left(\mathbb{E}_y e[f(x_1^0, y) + f(x_2^0, y)]\right)^2$. Hence

$$\mathbb{P}_U[R(U) \geq 2^{-cn}] \leq 2^{-cn}.$$

The upshot of all of the above is that we have devised an efficient test that can distinguish truth tables of functions computed by fast 1TMs from truth tables of uniformly random functions.

18.2.1.2 Quadratic-time 1TMs can compute pseudorandom functions

We now sketch a candidate pseudorandom function computable in quadratic time by 1TMs with cn^2 states. The candidate is an asymptotic generalization of a well-documented and

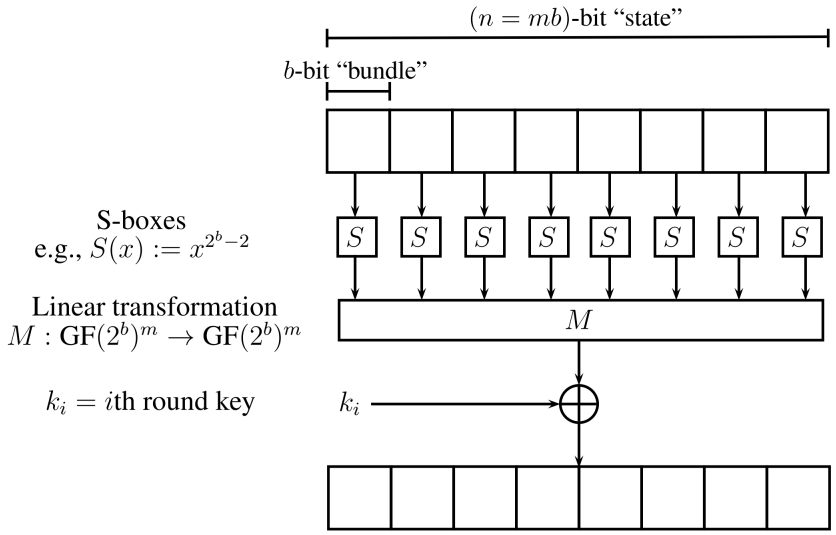


Figure 18.1: One round of an SPN

widely used block cipher: the *Advanced Encryption Standard*, *AES*. AES is based on the substitution-permutation network (SPN) structure, and will actually compute a function from $[2]^n \to [2]^n$ (whereas range [2] would suffice for our goals). On input $x \in [2]^n$, an SPN is computed over a number r of rounds, where each round "confuses" the input by dividing it into m/b bundles of b bits and applying a substitution function (S-box) to each bundle, and then "diffuses" the bundles by applying a matrix M with certain "branching" properties. At the end of each round i, the n bits are xor-ed with an n-bit round seed k_i, refer to figure 18.1.

The candidate follows the design considerations behind the AES block cipher, and particularly its S-box. For any n that is a multiple of 32, we break the input into $m := n/8$ bundles of $b = 8$ bits each, viewed as elements in the field $\mathbb{F}_{2^8}$, and perform $r = n$ rounds. We use the S-box $S(x) := x^{2^b-2}$. M is computed in two (linear) steps. In the first step, a permutation $\pi : [m] \to [m]$ is used to shuffle the b-bit bundles of the state; namely, bundle i moves to position $\pi(i)$. The permutation π is computed as follows. The m bundles are arranged into a $4 \times m/4$ matrix. Then row i of the matrix $(0 \leq i < 4)$ is shifted circularly to the left by i places. In the second step, a maximal-branch-number matrix $\phi \in \mathbb{F}^{4\times 4}$ is applied to each column of 4 bundles.

Let us now illustrate how one round can be computed in time cn with cn states. The bundles are written on the tape in column-major order: First the 4 bundles of the 1st column, then the 4 bundles of the 2nd column, and so on. The cn instances of S and φ can be computed in time cn. To see that π can also be computed in time cn, note that due to the representation, we can compute π with one pass, using that all but c bundles need to move $\leq c$ positions away. Finally, encoding the n-bit seed in the TM's state transitions, the addition of each round key also takes time cn.

Therefore, the $r = n$ rounds can be computed in time cn^2 with cn^2 states.

By the simulation of TMs by circuits, this candidate is also computable by power-size circuits. A naive implementation gives fairly large depth, so next we consider smaller-depth circuits.

18.2.2 Small-depth circuits

In Chapter 9 we saw several impossibility results for AC^0. In the next exercise you are asked that at least one of the proof techniques we saw is natural.

Exercise 18.3. Give an efficient algorithm to distinguish truth tables of functions in AC from uniform. Hint: Use Exercise 9.5.

There are candidate pseudorandom functions computable in TC. Some of them are based on popular conjectures, such as the hardness of factoring, see Theorem 14.3. The critical feature of TC that enables computing such candidates is iterated multiplication, see Theorem 8.2.

18.3 Notes

In an effort to make progress and understand the reach of current techniques, essentially every technique in complexity theory has been analyzed and, with few exceptions, filed under "black-box" or "natural-proofs." Especially for "black-box" this involved a myriad different oracle constructions. Often, these constructions are related to, and have provided some motivation for the study of, basic complexity classes. For example, PH^f basically corresponds to AC functions of the truth table of f, and one can use results about AC to give various oracle separations for PH and related classes. Indeed, a major motivation for impossibility results for AC was showing that the PH does not collapse w.r.t. some oracles [313]. After this first prolific phase of oracle constructions, starting about half a century ago, a second phase has followed during which it was realized that oracles provide limited information and they were relegated as curiosities. In a more recent third phase they have made a comeback in cryptography and quantum computing, often under the terminological disguise of *black-box*.

Relativization originated in the seminal work [37] and led to countless works on oracles. A variant of relativization where the oracles have additional algebraic structure is sharper for certain proof techniques and is studied in [5].

Natural proofs is from [232]. AES is described in [78]. The SPN structure of alternating "confusion" and "diffusion" steps was put forth already in [248]. The candidate 1TM pseudorandom function in section 18.2.1.2 is from [201].

The PRF in TC is from [210]. It gives TC of size $\geq n^c$. The work [19] considers TC of size $n^{1+\epsilon}$. [201] present a candidate, also based on AES, with these resources.

Chapter 19

P=NP?

"[...] Now it seems to me, however, to be completely within the realm of possibility that $\phi(n)$ grows that slowly. Since it seems that $\phi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and after all $\phi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to $\log N$ (or $(\log N)^2$). However, such strong reductions appear in other finite problems [...]." [104]

The only things that matter in a theoretical study are those that you can prove, but it's always fun to speculate. After worrying about P vs. NP for half my life, and having carefully reviewed the available "evidence" I have decided I believe that P = NP.

A main justification for my belief is history:

1. In the 1950's Kolmogorov conjectured that multiplication of n-bit integers requires time $\geq cn^2$. That's the time it takes to multiply using the method that mankind has used for at least six millennia. Presumably, if a better method existed it would have been found already. Kolmogorov subsequently started a seminar where he presented again this conjecture. Within one week of the start of the seminar, Karatsuba discovered his famous algorithm running in time $cn^{\log_2 3} \approx n^{1.58}$. He told Kolmogorov about it, who became agitated and terminated the seminar. Karatsuba's algorithm unleashed a new age of fast algorithms, including the next one. I recommend Karatsuba's own account [164] of this compelling story.

2. In 1968 Strassen started working on proving that the standard cn^3 algorithm for multiplying two $n \times n$ matrices is optimal. Next year his landmark $cn^{\log_2 7} \approx n^{2.81}$ algorithm appeared in his paper "Gaussian elimination is not optimal" [264].

3. In the 1970s Valiant showed that the graphs of circuits computing certain linear transformations must be a *super-concentrator*, a graph which certain strong connectivity properties. He conjectured that super-concentrators must have a super-linear number of wires, from which super-linear circuit lower bounds follow [281]. However, he later disproved the conjectured [282]: building on a result of Pinsker he constructed super-concentrators using a linear number of edges.

4. At the same time Valiant also defined *rigid* matrices and showed that an explicit construction of such matrices yields new circuit lower bounds. A specific matrix that was conjectured to be sufficiently rigid is the Hadamard matrix. Alman and Williams showed that, in fact, the Hadamard matrix is not rigid [20].

5. Constructing rigid matrices is one of *three* ways to get circuit lower bounds from a graph decomposition in [282]. Another way is via communication lower bounds. Here a specific candidate was the *sum-index* function, but then Sun [269] gave an efficient protocol for sum-index.

6. The LBA problems (Is L = NL? Is NL closed under complement?) A solution to the second problem was found more than 20 years after the formulation [149, 270, 271]. The general belief was that NL is not closed under complement, just like today the general belief seems to be that NP is not. But in fact, as we saw in Theorem 6.19, NL *is* closed under complement! Note a negative solution to the second problem would have implied a negative solution to the first, which remains open.

7. After finite automata, a natural step in lower bounds was to study sightly more general programs with constant memory. Consider a program that only maintains c bits of memory, and reads the input bits in a fixed order, where bits may be read several times. It seems quite obvious that such a program could not compute the majority function in polynomial time (see Chapter 0). This was explicitly conjectured by several people, including [53]. Barrington [205] famously disproved the conjecture by showing that in fact those seemingly very restricted constant-memory programs are in fact equivalent to log-depth circuits, which can compute majority (and many other things) (see Theorem 7.6).

8. Mansour, Nisan, and Tiwari conjectured [196] in 1990 that computing hash functions on n bits requires circuit size $\geq cn \log n$. Their conjecture was disproved in 2008 [155] where a circuit of size cn was given.

9. In [2] Aaronson made a conjecture which he called "generalized Linial-Nisan." The conjecture is that not only powerlog-wise uniform distributions fool AC (Theorem 11.3) but even a broader class of distributions which are only close to powerlog-wise uniform in a suitable sense. Aaronson himself later disproved the conjecture for depth-3 circuits [3]. For depth-2 circuits see [15].

10. For 30+ years the fastest run-time for graph isomorphism was exponential. A great deal was written on efficient proof systems for graph non-isomorphism. In 2015 Babai shocked the world with an almost power-time algorithm for graph isomorphism [28, 140].

11. Maxflow is a central problem studied since the dawn of computer science. All solutions had running time $\geq n^{1+c}$, until a stunning quasi-linear algorithm obtained in 2022 [65].

12. In number-on-forehead communication complexity, the function Majority-of-Majorities was raised as a candidate for being hard for $k \geq \log^{1+c} n$ players. This was disproved in [31] and subsequent works, where many other counter-intuitive protocols are presented, see section §13.4. For pointer chasing, a similar bound was first claimed and then retracted [79], then it was made again more recently (personal communication), only to be found in contradiction with the protocol in [227] (Theorem 13.12).

13. Chattopadhyay, Hatami, Hosseini, Lovett, and Zuckerman [64] introduced a novel technique with which they established an xor lemma for low-degree polynomials and consequently new correlation bounds. In particular, they proved that the correlation of the xor of two majority functions with *constant-degree* polynomials is $\log^c n/n$, a result for which previous xor lemmas do not seem sufficient. Note that the bound is indeed about the square of the bound in Exercise **??**. The key ingredient in the approach in [64] is a restriction result about low-degree polynomials. They prove it for degree up to $c \log n$ and they conjecture it holds even for much larger degrees. Their conjecture was disproved in [156]. In fact, [156] rules out even weaker parameters and shows that what is proved in [64] is essentially the best possible.

14. In [224] Patrascu made a conjecture in communication complexity, and showed that if true then many data-structure lower bounds would follow. He stated a general conjecture, which is overkill for the data-structure application, but has an appealing intuition. This strong form was refuted in [62].

15. Is it a knot or not? TBD

TBD: The list goes on and on. In data structures, would you think it possible to switch between binary and ternary representation of a number using constant time per digit and *zero* space overhead? Turns out it is [223, 82] (see section 15.1.2). Do you believe factoring is hard? Then you also believe there are pseudorandom generators where each output bit depends only on c input bits [24], see section §7.7. Known algorithms for directed connectivity use either super-polynomial time or polynomial memory. But if you are given access to polynomial memory full of junk that you can't delete, then you can solve directed connectivity using only logarithmic (clean) memory and polynomial time [56], section §6.9. And I haven't even touched on the many broken conjectures in cryptography, most recently related to obfuscation.

On the other hand, arguably the main thing that's surprising in the lower bounds we have is that they can be proved at all. The bounds themselves are hardly surprising. Of course, the issue may be that we can prove so few lower bounds that we shouldn't expect surprises. Some of the undecidability results I do consider surprising, for example Hilbert's 10th problem. But what is actually surprising in those results are the *algorithms*, showing that even very restricted models can simulate more complicated ones (same for the theory of NP completeness). In terms of lower bounds they all build on diagonalization, that is, go through every program and flip the answer, which is boring.

The evidence is clear: we have grossly underestimated the reach of efficient computation, in a variety of contexts. All signs indicate that we will continue to see bigger and bigger surprises in upper bounds, and P=NP. Do I really believe the formal inclusion P=NP? Maybe, let me not pick parameters. What I believe is that the idea that lower bounds are obviously true and we just can't prove them is not only baseless but even clashes with historical evidence. It's the upper bounds that are missing.

The "thousands of problems" argument for P $\neq$ NP.

- "Among the NP-complete problems are many [...] for which serious effort has been expended on finding polynomial-time algorithms. Since either all or none of the NP-complete problems are in P, and so far none have been found to be in P, it is natural to conjecture that none are in P." [146], Page 341.

- "The class NP [...] contains thousands of different problems for which no efficient solving procedure is known."[107]

- To my mind, however, the strongest argument for P $\neq$ NP involves the thousands of problems that have been shown to be NP-complete, and the thousands of other problems that have been shown to be in P. If just one of these problems had turned out to be both NP-complete and in P, that would've immediately implied P = NP. Thus, we could argue, the P $\neq$ NP hypothesis has had thousands of chances to be 'falsified by observation.'" [4]

I find these claims strange. In fact, the theory of NP completeness leads me to an opposite conclusion. As we saw in Chapters 4 and 5, see especially Problem 4.4, the problems can all be translated one into the other with extremely simple procedures, essentially doing *nothing*. In what sense are the problems different? I think a good definition of different is not reducible to each other in a simple manner. Resolving the grand challenge probably requires fantastic insight. I doubt that the standard reductions between problems in the classical theory of NP-completeness can be regarded as providing genuinely new perspectives.

The "lots of people tried" argument for P $\neq$ NP

The conjectures above were made by n top scientists in the area. On the other hand, $N \gg n$ people outside of the area attempted and failed to solve NP-hard problems. The fact that they are outsiders can be a strength or a weakness for the argument. It can be a strength, because of the sheer number, and because unshackled by the trends of the community, and without much interaction, the N people have been free to explore radically new ideas:

"Many of these problems have arisen in vastly different disciplines, and were the subject of extensive research by numerous different communities of scientists and engineers. These essentially independent studies have all failed to provide efficient algorithms for solving these problems, a failure that is extremely hard to attribute to sheer coincidence or a stroke of bad luck."[107]

But it can also be a weakness, because unaware of the well-studied pitfalls, and with little communication, these N people are likely to all have followed the same route. Indeed, most of the countless bogus proofs claiming to resolve major open problem in complexity fail in one of only a handful of different ways. So it is likely that those N people don't quite count for N distinct attempts, but in fact a much smaller number, quite possibly less than n.

The "catastrophe" argument for P $\neq$ NP

It's easy to consider scenarios in which P = NP would not cause a catastrophe. A trivial scenario is if the algorithms take time n^d for exceedingly large d. A less obvious scenario is that the algorithms use complicated component X (think the classification of simple groups, or the 4-color theorem, etc.). And then we would enter a phase in which for a problem you ask if it can be solved without using X.

A typical instantiation of the catastrophe is that most cryptography collapses. Again, one can imagine a scenario where it doesn't collapse. For example, the attacks are complicated or impractical. People continue to publish papers and use the protocols regardless. The new result just gives a more nuanced view of security. This would not be too different, perhaps, from the fact that the simplex algorithm is commonly used, even though there's a proof that it takes exponential time in some cases.

My "stop right before major results" argument for P = NP

Why do available techniques for impossibility results stop "right before" proving major results? This phenomenon appears to permeate complexity theory: we saw examples in section §6.3, Chapter 7, and Chapter 14. The most reasonable conclusion, it seems to me, is that this happens because the major results are actually false.

My "get stuck at the same point" argument for P = NP

An issue related to the "stop right before major results" issue is why the same impossibility results that we have are sometimes obtained via seemingly very different proofs. One of several examples: the polynomial method and the switching lemma give two different proofs that AC^0 can't compute parity, see Chapter 9. The proofs appear genuinely different, I would argue more different than the various NP-complete problems (see the "thousand different problems" argument above). Why should different approaches stop at the same point, except because there is nothing else to prove?

Throughout history, science has often proved wrong those who wouldn't take things at face value.

Complexity theory is perhaps unique in science. It appears that math is not ready for its problems. It is a bulwark against the business approach to science, the frenzy of the illusion of progress. For *ultimately* it doesn't matter how much you rake in or even who is writing

bombastic recommendation letters for you, etc. These problems remain untouched. And progress may be more likely to come when you are alone, staring at blank paper:

> "You do not need to leave your room. Remain sitting at your table and listen. Do not even listen, simply wait. Do not even wait, be quiet still and solitary. The world will freely offer itself to you to be unmasked, it has no choice, it will roll in ecstasy at your feet." [161]

References

[1] Scott Aaronson. Oracles are subtle but not malicious. In *CCC*, pages 340–354. IEEE Computer Society, 2006.

[2] Scott Aaronson. BQP and the polynomial hierarchy. In *42nd ACM Symp. on the Theory of Computing (STOC)*, pages 141–150. ACM, 2010.

[3] Scott Aaronson. A counterexample to the generalized linial-nisan conjecture. *Electron. Colloquium Comput. Complex.*, TR10-109, 2010.

[4] Scott Aaronson. P = np? In John Forbes Nash Jr. and Michael Th. Rassias, editors, *Open Problems in Mathematics*, pages 1–122. Springer, 2016.

[5] Scott Aaronson and Avi Wigderson. Algebrization: a new barrier in complexity theory. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 731–740, 2008.

[6] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for LCS and other sequence similarity measures. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 59–78. IEEE Computer Society, 2015.

[7] Anil Ada, Arkadev Chattopadhyay, Omar Fawzi, and Phuong Nguyen. The NOF multiparty communication complexity of composed functions. *Comput. Complex.*, 24(3):645–694, 2015.

[8] Leonard Adleman. Two theorems on random polynomial time. In *19th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 75–83. 1978.

[9] Peyman Afshani, Casper Benjamin Freksen, Lior Kamma, and Kasper Green Larsen. Lower bounds for multiplication via network coding. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum fÃŒr Informatik, 2019.

[10] M. Agrawal, N. Kayal, and N. Saxena. Primes in p. *Annals of Pure Mathematics*, 160(2):781–793, 2004.

[11] Manindra Agrawal and V. Vinay. Arithmetic circuits: A chasm at depth four. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 67–75. IEEE Computer Society, 2008.

[12] Miklós Ajtai. Σ_1^1-formulae on finite structures. *Annals of Pure and Applied Logic*, 24(1):1–48, 1983.

[13] Miklós Ajtai. Approximate counting with uniform constant-depth circuits. In *Advances in computational complexity theory*, pages 1–20. Amer. Math. Soc., Providence, RI, 1993.

[14] Miklós Ajtai. A non-linear time lower bound for boolean branching programs. *Theory of Computing*, 1(1):149–176, 2005.

[15] Yaroslav Alekseev, Mika Göös, Ziyi Guan, Gilbert Maystre, Artur Riazanov, Dmitry Sokolov, and Weiqiang Yuan. Generalised Linial-Nisan conjecture is False for DNFs. Technical Report TR25-058, Electronic Colloquium on Computational Complexity (ECCC), May 2025.

[16] Eric Allender. A note on the power of threshold circuits. In *30th Symposium on Foundations of Computer Science*, pages 580–584, Research Triangle Park, North Carolina, 30 October–1 November 1989. IEEE.

[17] Eric Allender. The division breakthroughs. *Bulletin of the EATCS*, 74:61–77, 2001.

[18] Eric Allender. Arithmetic circuits and counting complexity classes. *Complexity of Computations and Proofs, Quaderni di Matematica Vol. 13, Seconda Universita di Napoli*, 2004.

[19] Eric Allender and Michal Koucký. Amplifying lower bounds by means of self-reducibility. *J. of the ACM*, 57(3), 2010.

[20] Josh Alman and R. Ryan Williams. Probabilistic rank and matrix rigidity. In *ACM Symp. on the Theory of Computing (STOC)*, pages 641–652, 2017.

[21] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:567–583, 1986.

[22] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple constructions of almost k-wise independent random variables. *Random Structures & Algorithms*, 3(3):289–304, 1992.

[23] Robert Andrews and Avi Wigderson. Constant-depth arithmetic circuits for linear algebra problems. *Electronic Colloquium on Computational Complexity (ECCC)*, 80, 2024.

[24] Benny Applebaum, Yuval Ishai, and Eyal Kushilevitz. Cryptography in NC^0. *SIAM J. on Computing*, 36(4):845–888, 2006.

[25] Sanjeev Arora and Boaz Barak. *Computational Complexity*. Cambridge University Press, 2009. A modern approach.

[26] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. of the ACM*, 45(3):501–555, May 1998.

[27] László Babai. E-mail and the unexpected power of interaction. In *SCT*, pages 30–44. IEEE Computer Society, 1990.

[28] László Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

[29] László Babai, Lance Fortnow, and Carsten Lund. Nondeterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1(1):3–40, 1991.

[30] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3(4):307–318, 1993.

[31] László Babai, Anna Gál, Peter G. Kimmel, and Satyanarayana V. Lokam. Communication complexity of simultaneous messages. *SIAM J. on Computing*, 33(1):137–166, 2003.

[32] László Babai, Thomas P. Hayes, and Peter G. Kimmel. The cost of the missing bit: communication complexity with help. *Combinatorica. An Journal on Combinatorics and the Theory of Computing*, 21(4):455–488, 2001.

[33] László Babai and Shlomo Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comput. Syst. Sci.*, 36(2):254–276, 1988.

[34] László Babai, Noam Nisan, and Márió Szegedy. Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs. *J. of Computer and System Sciences*, 45(2):204–232, 1992.

[35] Arturs Backurs. *Below P vs NP: fine-grained hardness for big data problems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, 2018.

[36] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM J. Comput.*, 47(3):1087–1097, 2018.

[37] Theodore Baker, John Gill, and Robert Solovay. Relativizations of the P=?NP question. *SIAM J. on Computing*, 4(4):431–442, 1975.

[38] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. of Computer and System Sciences*, 68(4):702–732, 2004.

[39] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[40] Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoret. Comput. Sci.*, 22(3):317–330, 1983.

[41] Paul Beame. A switching lemma primer. Technical Report UW-CSE-95-07-01, Department of Computer Science and Engineering, University of Washington, November 1994. Available from http://www.cs.washington.edu/homes/beame/.

[42] Paul Beame, Stephen A. Cook, and H. James Hoover. Log depth circuits for division and related problems. *SIAM J. Comput.*, 15(4):994–1003, 1986.

[43] Paul Beame, Matei David, Toniann Pitassi, and Philipp Woelfel. Separating deterministic from nondeterministic nof multiparty communication complexity. In *34th Coll. on Automata, Languages and Programming (ICALP)*, pages 134–145. Springer, 2007.

[44] Paul Beame, Michael Saks, Xiaodong Sun, and Erik Vee. Time-space trade-off lower bounds for randomized computation of decision problems. *J. of the ACM*, 50(2):154–195, 2003.

[45] Donald Beaver and Joan Feigenbaum. Hiding instances in multioracle queries. In *Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 37–48, 1990.

[46] Richard Beigel, Nick Reingold, and Daniel Spielman. The perceptron strikes back. In *Structure in Complexity Theory Conference*, pages 286–291, 1991.

[47] Richard Beigel and Jun Tarui. On ACC. *Computational Complexity*, 4(4):350–366, 1994.

[48] Michael Ben-Or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. on Computing*, 21(1):54–58, 1992.

[49] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Inform. Process. Lett.*, 18(3):147–150, 1984.

[50] Anurag Bishnoi, Pete L. Clark, Aditya Potukuchi, and John R. Schmitt. On zeros of a polynomial in a finite grid. *Comb. Probab. Comput.*, 27(3):310–333, 2018.

[51] Andrej Bogdanov and Emanuele Viola. Pseudorandom bits for polynomials. *SIAM J. on Computing*, 39(6):2464–2486, 2010.

[52] Allan Borodin. Computational complexity and the existence of complexity gaps. *Journal of the ACM*, 19(1):158–174, 1972.

[53] Allan Borodin, Danny Dolev, Faith E. Fich, and Wolfgang J. Paul. Bounds for width two branching programs. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 87–93, 1983.

[54] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[55] Joshua Brody and Amit Chakrabarti. Sublinear communication protocols for multi-party pointer jumping and a related lower bound. In *25th Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 145–156, 2008.

[56] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *ACM Symp. on the Theory of Computing (STOC)*, pages 857–866, 2014.

[57] Peter Bürgisser. On defining integers and proving arithmetic circuit lower bounds. *Comput. Complex.*, 18(1):81–103, 2009.

[58] Peter Bürgisser, Michael Clausen, and Mohammad Amin Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer, 1997.

[59] Samuel R. Buss and Ryan Williams. Limits on alternation trading proofs for time-space lower bounds. *Comput. Complex.*, 24(3):533–600, 2015.

[60] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *J. of Computer and System Sciences*, 18(2):143–154, 1979.

[61] Ashok K. Chandra, Merrick L. Furst, and Richard J. Lipton. Multi-party protocols. In *15th ACM Symp. on the Theory of Computing (STOC)*, pages 94–99, 1983.

[62] Arkadev Chattopadhyay, Jeff Edmonds, Faith Ellen, and Toniann Pitassi. A little advice can be very helpful. In *SODA*, pages 615–625. SIAM, 2012.

[63] Arkadev Chattopadhyay and Toniann Pitassi. The story of set disjointness. *SIGACT News*, 41(3):59–85, 2010.

[64] Eshan Chattopadhyay, Pooya Hatami, Kaave Hosseini, Shachar Lovett, and David Zuckerman. XOR lemmas for resilient functions against polynomials. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia

Chuzhoy, editors, *ACM Symp. on the Theory of Computing (STOC)*, pages 234–246. ACM, 2020.

[65] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear-time algorithms for maximum flow and minimum-cost flow. *Communications of the ACM*, 66(12):85–92, 2023.

[66] Lijie Chen and Roei Tell. Bootstrapping results for threshold circuits "just beyond" known lower bounds. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 34–41. ACM, 2019.

[67] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Statistics*, 23:493–507, 1952.

[68] Benny Chor, Oded Goldreich, Johan Håstad, Joel Friedman, Steven Rudich, and Roman Smolensky. The bit extraction problem or t-resilient functions (preliminary version). In *26th Symposium on Foundations of Computer Science*, pages 396–407, Portland, Oregon, 21–23 October 1985. IEEE.

[69] Fan R. K. Chung and Prasad Tetali. Communication complexity and quasi randomness. *SIAM Journal on Discrete Mathematics*, 6(1):110–123, 1993.

[70] Richard Cleve. Towards optimal simulations of formulas by bounded-width programs. *Computational Complexity*, 1:91–105, 1991.

[71] James Cook and Ian Mertz. Tree evaluation is in space o(log n · log log n). In *STOC*, pages 1268–1278. ACM, 2024.

[72] James Cook and Edward Pyne. Efficient catalytic graph algorithms. *CoRR*, abs/2509.06209, 2025.

[73] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[74] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 73–80. ACM, 1972.

[75] Harald Cramér. Sur un nouveau théorème-limite de la théorie des probabilités. In *Actualités Scientifiques et Industrielles*, volume 736, pages 5–23, Paris, 1938. Hermann et Cie. Colloque consacré à la théorie des probabilités.

[76] Harald Cramér and Hugo Touchette. On a new limit theorem in probability theory (translation of 'sur un nouveau théorème-limite de la théorie des probabilités'), 2018. Translation by Hugo Touchette of the original 1938 French paper.

[77] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5(4):618–623, 1976.

[78] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer Verlag, 2002.

[79] Carsten Damm, Stasys Jukna, and Jiří Sgall. Some bounds on multiparty communication complexity of pointer jumping. *Computational Complexity*, 7(2):109–127, 1998.

[80] K. De Leeuw, Edward F. Moore, Claude E. Shannon, and Norman Shapiro. Computability by probabilistic machines. In *Automata Studies*, volume 34 of *Annals of Mathematics Studies*, pages 183–198. Princeton University Press, Princeton, N.J., 1956.

[81] Harm Derksen and Emanuele Viola. Fooling polynomials using invariant theory. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, 2022.

[82] Yevgeniy Dodis, Mihai Pătraşcu, and Mikkel Thorup. Changing base without losing space. In *42nd ACM Symp. on the Theory of Computing (STOC)*, pages 593–602. ACM, 2010.

[83] Devdatt Dubhashi and Alessandro Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.

[84] Pál Erdős, Ronald L. Graham, and Endre Szemerédi. On sparse graphs with dense long paths. *Comp. and Maths. with Appls.*, 1:365–369, 1975.

[85] Euclid. *The Thirteen Books of Euclid's Elements, Vol. 2*. Dover Publications, New York, 1956. Originally published in 300 B.C.

[86] Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, 1994.

[87] Michael A. Forbes. Low-depth algebraic circuit lower bounds over any field. In *CCC*, volume 300 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[88] Lance Fortnow. Time-space tradeoffs for satisfiability. *J. Comput. Syst. Sci.*, 60(2):337–353, 2000.

[89] Lance Fortnow. A simple proof of toda's theorem. *Theory Comput.*, 5(1):135–140, 2009.

[90] Lance Fortnow, Richard Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *J. of the ACM*, 52(6):835–865, 2005.

[91] Caxton C. Foster and Fred D. Stockton. Counting responders in an associative memory. *IEEE Trans. Computers*, 20(12):1580–1583, 1971.

[92] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[93] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *ACM Symp. on the Theory of Computing (STOC)*, pages 345–354, 1989.

[94] Michael L. Fredman and Dan Dominic Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[95] R. Freivalds. Probabilistic machines can use less running time. In *IFIP Congress*, pages 839–842, 1977.

[96] Rusins Freivalds. Probabilistic two-way machines. In *MFCS*, volume 118 of *Lecture Notes in Computer Science*, pages 33–45. Springer, 1981.

[97] Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

[98] Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.

[99] Anna Gál, Kristoffer Arnsfelt Hansen, Michal Koucký, Pavel Pudlák, and Emanuele Viola. Tight bounds on computing error-correcting codes by bounded-depth circuits with arbitrary gates. *IEEE Transactions on Information Theory*, 59(10):6611–6627, 2013.

[100] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. *Theoretical Computer Science*, 379(3):405–417, 2007.

[101] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[102] Peter Gemmell, Richard Lipton, Ronitt Rubinfeld, Madhu Sudan, and Avi Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Twenty Third ACM Symposium on Theory of Computing*, pages 32–42, New Orleans, Louisiana, 6–8 May 1991.

[103] Kurt Godel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, i. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[104] Kurt Godel, 1956. Letter to John von Neumann. https://www.anilada.com/notes/godel-letter.pdf.

[105] Mikael Goldmann, Johan Håstad, and Alexander A. Razborov. Majority gates vs. general weighted threshold gates. *Computational Complexity*, 2:277–300, 1992.

[106] Oded Goldreich. A sample of samplers - a computational perspective on sampling (survey). *Electronic Coll. on Computational Complexity (ECCC)*, 4(020), 1997.

[107] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[108] Oded Goldreich. On doubly-efficient interactive proof systems. *Found. Trends Theor. Comput. Sci.*, 13(3):158–246, 2018.

[109] Oded Goldreich. On the cook-mertz tree evaluation procedure. In Oded Goldreich, editor, *Computational Complexity and Local Algorithms - On the Interplay Between Randomness and Computation*, volume 15700 of *Lecture Notes in Computer Science*, pages 102–112. Springer, 2025.

[110] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[111] Oded Goldreich, Noam Nisan, and Avi Wigderson. On Yao's XOR lemma. Technical Report TR95–050, *Electronic Colloquium on Computational Complexity*, March 1995. www.eccc.uni-trier.de/.

[112] Oded Goldreich and Guy N. Rothblum. Simple doubly-efficient interactive proof systems for locally-characterizable sets. In *ITCS*, volume 94 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[113] Oded Goldreich and Guy N. Rothblum. Constant-round interactive proof systems for AC0[2] and NC1. In *Computational Complexity and Property Testing*, volume 12050 of *Lecture Notes in Computer Science*, pages 326–351. Springer, 2020.

[114] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *40th ACM Symp. on the Theory of Computing*

(STOC), pages 113–122, 2008.

[115] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[116] Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil Vadhan. Better pseudorandom generators from milder pseudorandom restrictions. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, 2012.

[117] Raymond Greenlaw, H. James Hoover, and Walter Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. 02 2001.

[118] D. Yu Grigoriev. Using the notions of seperability and independence for proving the lower bounds on the circuit complexity. *Notes of the Leningrad branch of the Steklov Mathematical Institute, Nauka*, 1976.

[119] Dima Grigoriev and Alexander A. Razborov. Exponential lower bounds for depth 3 arithmetic circuits in algebras of functions over finite fields. *Appl. Algebra Eng. Commun. Comput.*, 10(6):465–487, 2000.

[120] Ankit Gupta, Pritish Kamath, Neeraj Kayal, and Ramprasad Saptharishi. Arithmetic circuits: A chasm at depth 3. *SIAM J. Comput.*, 45(3):1064–1079, 2016.

[121] Yuri Gurevich. Unconstrained church-turing thesis cannot possibly be true. *Bull. EATCS*, 127, 2019.

[122] Yuri Gurevich and Saharon Shelah. Nearly linear time. In *Logic at Botik, Symposium on Logical Foundations of Computer Science*, pages 108–118, 1989.

[123] Dan Gutfreund and Emanuele Viola. Fooling parity tests with parity gates. In *8thWorkshop on Randomization and Computation (RANDOM)*, pages 381–392. Springer, 2004.

[124] Torben Hagerup. Fast parallel generation of random permutations. In *18th Coll. on Automata, Languages and Programming (ICALP)*, pages 405–416. Springer, 1991.

[125] András Hajnal, Wolfgang Maass, Pavel Pudlák, Mário Szegedy, and György Turán. Threshold circuits of bounded depth. *J. of Computer and System Sciences*, 46(2):129–154, 1993.

[126] Joseph Y. Halpern, Michael C. Loui, Albert R. Meyer, and Daniel Weise. On time versus space III. *Math. Syst. Theory*, 19(1):13–28, 1986.

[127] Yassine Hamoudi. Simultaneous multiparty communication protocols for composed functions. In *MFCS*, volume 117 of *LIPIcs*, pages 14:1–14:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[128] T. Hartman and R. Raz. On the distribution of the number of roots of polynomials and explicit weak designs. *Random Structures & Algorithms*, 23(3):235–263, 2003.

[129] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[130] David Harvey and Joris van der Hoeven. Integer multiplication in time $O(n\log n)$. *Annals of Mathematics*, 193(2):563 – 617, 2021.

[131] Johan Håstad. *Computational limitations of small-depth circuits*. MIT Press, 1987.

[132] Johan Håstad. The shrinkage exponent of de morgan formulas is 2. *SIAM J. Comput.*, 27(1):48–64, 1998.

[133] Johan Håstad. On the correlation of parity and small-depth circuits. *SIAM J. on Computing*, 43(5):1699–1708, 2014.

[134] Johan Håstad and Mikael Goldmann. On the power of small-depth threshold circuits. *Computational Complexity*, 1(2):113–129, 1991.

[135] John Hastad. Almost optimal lower bounds for small depth circuits. *Adv. Comput. Res.*, 5:143–170, 1989.

[136] Pooya Hatami and William Hoza. Theory of unconditional pseudorandom generators. *Electron. Colloquium Comput. Complex.*, TR23-019, 2023.

[137] Thomas P. Hayes. Separating the k-party communication complexity hierarchy: an application of the zarankiewicz problem. *Discret. Math. Theor. Comput. Sci.*, 13(4):15–22, 2011.

[138] Songhua He. A note on a hierarchy theorem for promise-bptime. *Electron. Colloquium Comput. Complex.*, TR25-004, 2025.

[139] Alexander Healy, Salil P. Vadhan, and Emanuele Viola. Using nondeterminism to amplify hardness. *SIAM J. on Computing*, 35(4):903–931, 2006.

[140] Harald Andrés Helfgott. Graph isomorphism, quasipolynomial time. *Astérisque*, 407:135–182, 2019.

[141] F. C. Hennie. Crossing sequences and off-line turing machine computations. In *Symposium on Switching Circuit Theory and Logical Design (SWCT) (FOCS)*, pages 168–172, 1965.

[142] F. C. Hennie. One-tape, off-line turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[143] Fred Hennie and Richard Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[144] Fred Hennie and Richard Stearns. Two-tape simulation of multitape turing machines. *J. of the ACM*, 13:533–546, October 1966.

[145] Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bull. Amer. Math. Soc. (N.S.)*, 43(4):439–561 (electronic), 2006.

[146] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[147] Laurent Hyafil. On the parallel evaluation of multivariate polynomials. *SIAM J. Comput.*, 8(2):120–123, 1979.

[148] John T. Gill III. Computational complexity of probabilistic turing machines. In *STOC*, pages 91–95. ACM, 1974.

[149] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comput.*, 17(5):935–938, 1988.

[150] Russell Impagliazzo. Hard-core distributions for somewhat hard problems. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 538–545, 1995.

[151] Russell Impagliazzo and Ramamohan Paturi. The complexity of k-sat. In *IEEE Conf. on Computational Complexity (CCC)*, pages 237–, 1999.

[152] Russell Impagliazzo, Ramamohan Paturi, and Michael E. Saks. Size-depth tradeoffs for threshold circuits. *SIAM J. Comput.*, 26(3):693–707, 1997.

[153] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *J. Computer & Systems Sciences*, 63(4):512–530, Dec 2001.

[154] Russell Impagliazzo and Avi Wigderson. $P = BPP$ if E requires exponential circuits: Derandomizing the XOR lemma. In *29th ACM Symp. on the Theory of Computing (STOC)*, pages 220–229. ACM, 1997.

[155] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography with constant computational overhead. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 433–442, 2008.

[156] Peter Ivanov, Liam Pavlovic, and Emanuele Viola. On correlation bounds against polynomials. In *Conf. on Computational Complexity (CCC)*, 2023.

[157] Kazuo Iwama and Hiroki Morizumi. An explicit lower bound of $5n - o(n)$ for boolean circuits. In *Symp. on Math. Foundations of Computer Science (MFCS)*, pages 353–364, 2002.

[158] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Succinct and explicit circuits for sorting and connectivity. Available at http://www.ccs.neu.edu/home/viola/, 2014.

[159] Hamid Jahanjou, Eric Miles, and Emanuele Viola. Local reductions. *Information and Computation*, 261(2), 2018. Available at http://www.ccs.neu.edu/home/viola/.

[160] Hermann Jung. Depth efficient transformations of arithmetic into boolean circuits. In Lothar Budach, editor, *Fundamentals of Computation Theory, FCT '85, Cottbus, GDR, September 9-13, 1985*, volume 199 of *Lecture Notes in Computer Science*, pages 167–174. Springer, 1985.

[161] Franz Kafka. Aphorisms.

[162] Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discrete Math.*, 5(4):545–557, 1992.

[163] Ravi Kannan, H. Venkateswaran, V. Vinay, and Andrew Chi-Chih Yao. A circuit-based proof of toda's theorem. *Inf. Comput.*, 104(2):271–276, 1993.

[164] A. A. Karatsuba. The complexity of computations. *Trudy Mat. Inst. Steklov.*, 211(Optim. Upr. i Differ. Uravn.):186–202, 1995.

[165] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[166] Richard M. Karp and Richard J. Lipton. Turing machines that take advice. *L'Enseignement Mathématique. Revue Internationale. IIe Série*, 28(3-4):191–209, 1982.

[167] Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM J. on Computing*, 40(6):1767–1802, 2011.

[168] Zander Kelley, Shachar Lovett, and Raghu Meka. Explicit separations between randomized and deterministic number-on-forehead communication. *CoRR*, abs/2308.12451, 2023.

[169] Leonid G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[170] V. M. Khrapchenko. A method of obtaining lower bounds for the complexity of π-schemes. *Mathematical Notes of the Academy of Sciences of the USSR*, 10:474–479, 1972.

[171] Adam Klivans and Rocco A. Servedio. Boosting and hard-core sets. *Machine Learning*, 53(3):217–238, 2003.

[172] Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. In *ACM Symposium on Theory of Computing (Atlanta, GA, 1999)*, pages 659–667. ACM, New York, 1999.

[173] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1998.

[174] Kojiro Kobayashi. On the structure of one-tape nondeterministic turing machine time hierarchy. *Theor. Comput. Sci.*, 40:175–193, 1985.

[175] Pascal Koiran. Valiant's model and the cost of computing integers. *Comput. Complex.*, 13(3-4):131–146, 2005.

[176] Swastik Kopparty and Srikanth Srinivasan. Certifying polynomials for $\mathrm{AC}^0[\oplus]$ circuits, with applications to lower bounds and circuit compression. *Theory of Computing*, 14(1):1–24, 2018.

[177] Kenneth Krohn, W. D. Maurer, and John Rhodes. Realizing complex Boolean functions with simple groups. *Information and Control*, 9:190–195, 1966.

[178] S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Inf. Control.*, 7(2):207–223, 1964.

[179] Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.

[180] Gabriel Lam. Note sur la limite du nombre des divisions dans la détermination d'un plus grand commun diviseur. *Comptes Rendus de l'Académie des Sciences de Paris*, 19:867–870, 1844.

[181] Klaus-Jörn Lange, Birgit Jenner, and Bernd Kirsig. The logarithmic alternation hierarchiy collapses: A sigmaˆc_2 = A piˆc_2. In *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 531–541. Springer, 1987.

[182] Kasper Green Larsen. Higher cell probe lower bounds for evaluating polynomials. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 293–301, 2012.

[183] Kasper Green Larsen, Omri Weinstein, and Huacheng Yu. Crossing the logarithmic barrier for dynamic boolean data structure lower bounds. *SIAM J. Comput.*, 49(5), 2020.

[184] Leonid A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[185] Jiatu Li and Tianqi Yang. $3.1n$ - $o(n)$ circuit lower bounds for explicit functions. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1180–1193. ACM, 2022.

[186] Rudolf Lidl and Harald Niederreiter. *Finite fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, second edition, 1997.

[187] Nutan Limaye, Srikanth Srinivasan, and Sébastien Tavenas. Superpolynomial lower bounds against low-depth algebraic circuits. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 804–814. IEEE, 2021.

[188] Nutan Limaye, Srikanth Srinivasan, and Sébastien Tavenas. Guest column: Lower bounds against constant-depth algebraic circuits. *SIGACT News*, 53(2):40–62, 2022.

[189] Richard Lipton. New directions in testing. In *Proceedings of DIMACS Workshop on Distributed Computing and Cryptography*, volume 2, pages 191–202. ACM/AMS, 1991.

[190] Richard J. Lipton. Straight-line complexity and integer factorization. In *International Algorithmic Number Theory Symposium*, pages 71–79, 1994. Cited at 124.

[191] Shachar Lovett. Unconditional pseudorandom generators for low degree polynomials. In *40th ACM Symp. on the Theory of Computing (STOC)*, pages 557–562, 2008.

[192] Chi-Jen Lu, Shi-Chun Tsai, and Hsin-Lung Wu. Improved hardness amplification in NP. *Theor. Comput. Sci.*, 370(1-3):293–298, 2007.

[193] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. of the ACM*, 39(4):859–868, October 1992.

[194] O. B. Lupanov. A method of circuit synthesis. *Izv. VUZ Radiofiz.*, 1:120–140, 1958.

[195] Wolfgang Maass and Amir Schorr. Speed-up of Turing machines with one work tape and a two-way input tape. *SIAM J. on Computing*, 16(1):195–202, 1987.

[196] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theoretical Computer Science*, 107:121–133, 1993.

[197] Yossi Matias and Uzi Vishkin. Converting high probability into nearly-constant time-with applications to parallel hashing. In *23rd ACM Symp. on the Theory of Computing (STOC)*, pages 307–316, 1991.

[198] W. D. Maurer and John L. Rhodes. A property of finite simple non-abelian groups. *Pacific Journal of Mathematics*, 16(2):491–495, 1965.

[199] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[200] Pierre McKenzie and Stephen A. Cook. The parallel complexity of abelian permutation group problems. *SIAM J. Comput.*, 16(5):880–909, 1987.

[201] Eric Miles and Emanuele Viola. Substitution-permutation networks, pseudorandom functions, and natural proofs. *J. of the ACM*, 62(6), 2015.

[202] Peter Bro Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 625–634. ACM, 1994.

[203] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *J. of Computer and System Sciences*, 57(1):37 – 49, 1998.

[204] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[205] David A. Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *J. of Computer and System Sciences*, 38(1):150–164, 1989.

[206] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[207] Wolfgang Mulzer. Five proofs of chernoff's bound with applications. *Bull. EATCS*, 124, 2018.

[208] Cody Murray and R. Ryan Williams. Circuit lower bounds for nondeterministic quasi-polytime: an easy witness lemma for NP and NQP. In *STOC*, pages 890–901. ACM, 2018.

[209] J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. In *22nd ACM Symp. on the Theory of Computing (STOC)*, pages 213–223. ACM, 1990.

[210] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 458–467, 1997.

[211] E. I. Nechiporuk. A boolean function. *Soviet Mathematics-Doklady*, 169(4):765–766, 1966.

[212] Valery A. Nepomnjaščiĭ. Rudimentary predicates and Turing calculations. *Soviet Mathematics-Doklady*, 11(6):1462–1465, 1970.

[213] NEU. From RAM to SAT. Available at http://www.ccs.neu.edu/home/viola/, 2012.

[214] Ilan Newman. Private vs. common random bits in communication complexity. *Information Processing Letters*, 39(2):67–71, 1991.

[215] Noam Nisan. The communication complexity of threshold gates. In *Combinatorics, Paul Erdős is Eighty, number 1 in Bolyai Society Mathematical Studies*, pages 301–315, 1993.

[216] Noam Nisan and Avi Wigderson. Hardness vs randomness. *J. of Computer and System Sciences*, 49(2):149–167, 1994.

[217] Noam Nisan and Avi Wigderson. Lower bounds on arithmetic circuits via partial derivatives. *Comput. Complexity*, 6(3):217–234, 1996/97.

[218] Ryan O'Donnell. Hardness amplification within NP. *J. of Computer and System Sciences*, 69(1):68–94, August 2004.

[219] Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

[220] Christos H. Papadimitriou and Mihalis Yannakakis. Optimization, approximation, and complexity classes. *J. Comput. Syst. Sci.*, 43(3):425–440, 1991.

[221] Christos H. Papadimitriou and Stathis Zachos. Two remarks on the power of counting. In *Theoretical Computer Science*, volume 145 of *Lecture Notes in Computer Science*, pages 269–276. Springer, 1983.

[222] Seymour Papert. One AI or Many? *Daedalus*, 117, 1988.

[223] Mihai Pătraşcu. Succincter. In *49th IEEE Symp. on Foundations of Computer Science (FOCS)*. IEEE, 2008.

[224] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *ACM Symp. on the Theory of Computing (STOC)*, pages 603–610, 2010.

[225] Wolfgang J. Paul, Nicholas Pippenger, Endre Szemerédi, and William T. Trotter. On determinism versus non-determinism and related problems (preliminary version). In

IEEE Symp. on Foundations of Computer Science (FOCS), pages 429–438, 1983.

[226] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. of the ACM*, 26(2):361–381, 1979.

[227] Pavel Pudlák, Vojtěch Rödl, and Jiří Sgall. Boolean circuits, tensor ranks, and communication complexity. *SIAM J. on Computing*, 26(3):605–633, 1997.

[228] C. Radhakrishna Rao. Factorial experiments derivable from combinatorial arrangements of arrays. *Suppl. J. Roy. Statist. Soc.*, 9:128–139, 1947.

[229] Anup Rao and Amir Yehudayoff. *Communication complexity*. 2019.

[230] Ran Raz. The BNS-Chung criterion for multi-party communication complexity. *Computational Complexity*, 9(2):113–122, 2000.

[231] Alexander Razborov. Lower bounds on the dimension of schemes of bounded depth in a complete basis containing the logical addition function. *Akademiya Nauk SSSR. Matematicheskie Zametki*, 41(4):598–607, 1987. English translation in Mathematical Notes of the Academy of Sci. of the USSR, 41(4):333-338, 1987.

[232] Alexander Razborov and Steven Rudich. Natural proofs. *J. of Computer and System Sciences*, 55(1):24–35, August 1997.

[233] Alexander A. Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992.

[234] Omer Reingold. Undirected connectivity in log-space. *J. of the ACM*, 55(4), 2008.

[235] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021.

[236] J. M. Robson. An O(T log T) reduction from RAM computations to satisfiability. *Theoretical Computer Science*, 82(1):141–149, 1991.

[237] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 1962.

[238] Gian-Carlo Rota. From cardinals to chaos. In Nigel G. Cooper, editor, *From Cardinals to Chaos*, page 26. Cambridge University Press, Cambridge, 1989.

[239] Eyal Rozenman and Salil P. Vadhan. Derandomized squaring of graphs. In *Workshop on Randomization and Computation (RANDOM)*, pages 436–447, 2005.

[240] Rahul Santhanam. On separators, segregators and time versus space. In *IEEE Conf. on Computational Complexity (CCC)*, pages 286–294, 2001.

[241] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[242] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: a compendium. *SIGACT News, Complexity Theory Column*, 2002.

[243] Arnold Schönhage. On the power of random access machines. In *ICALP*, volume 71 of *Lecture Notes in Computer Science*, pages 520–529. Springer, 1979.

[244] Arnold Schönhage. Storage modification machines. *SIAM J. Comput.*, 9(3):490–508, 1980.

[245] Yakov Shalunov. Improved bounds on the space complexity of circuit evaluation. *arXiv preprint*, 2025.

[246] Adi Shamir. Factoring numbers in o(log n) arithmetic steps. *Information Processing*

Letters, 8(1):28–31, 1979.

[247] Adi Shamir. IP = PSPACE. *J. of the ACM*, 39(4):869–877, October 1992.

[248] Claude Shannon. Communication theory of secrecy systems. *Bell Systems Technical Journal*, 28(4):656–715, 1949.

[249] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Tech. J.*, 28:59–98, 1949.

[250] Alexander A. Sherstov. Communication complexity theory: Thirty-five years of set disjointness. In *Symp. on Math. Foundations of Computer Science (MFCS)*, pages 24–43, 2014.

[251] Victor Shoup. New algorithms for finding irreducible polynomials over finite fields. *Mathematics of Computation*, 54(189):435–447, 1990.

[252] Amir Shpilka and Avi Wigderson. Depth-3 arithmetic circuits over fields of characteristic zero. *Comput. Complex.*, 10(1):1–27, 2001.

[253] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Found. Trends Theor. Comput. Sci.*, 5(3-4):207–388, 2010.

[254] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. on Computing*, 33(3):505–543, 2004.

[255] Michael Sipser. A complexity theoretic approach to randomness. In *ACM Symp. on the Theory of Computing (STOC)*, pages 330–335, 1983.

[256] Michael Sipser. *Introduction to the theory of computation, 3rd ed.* PWS Publishing Company, 1997.

[257] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In *19th ACM Symp. on the Theory of Computing (STOC)*, pages 77–82. ACM, 1987.

[258] Roman Smolensky. On representations by low-degree polynomials. In *34th IEEE IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 130–138, 1993.

[259] P. M. Spira. On time hardware complexity tradeofis for boolean functions. In *Proceedings of the Fourth Hawaii International Symposium on System Sciences*, pages 525–527, 1971.

[260] A. Spivak. Brainteasers b 201: Strange painting. *Quantum*, page 13, 1997.

[261] Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis II. Hierarchies of memory limited computations. In *SWCT*, pages 179–190. IEEE Computer Society, 1965.

[262] Larry Stockmeyer and Albert R. Meyer. Cosmological lower bound on the circuit complexity of a small problem in logic. *J. ACM*, 49(6):753–784, 2002.

[263] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[264] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[265] Volker Strassen. Die rcrechnungskomplcxittit von elementarsvmmetrischen funktionen und von interpolationskoetfiziearten. *Numer. Math.*, 20:238–251, 1973.

[266] Volker Strassen. Polynomials with rational coefficients which are hard to compute. *SIAM J. Comput.*, 3:128–149, 1974.

[267] B. A. Subbotovskaya. Realizations of linear functions by formulas using +, *, -. *Soviet*

Mathematics-Doklady, 2:110–112, 1961.

[268] Madhu Sudan, Luca Trevisan, and Salil Vadhan. Pseudorandom generators without the XOR lemma. *J. of Computer and System Sciences*, 62(2):236–266, 2001.

[269] Xiaoming Sun. A 3-party simultaneous protocol for SUM-INDEX. *Algorithmica*, 36(1):89–111, 2003.

[270] Róbert Szelepcsényi. The moethod of focing for nondeterministic automata. *Bull. EATCS*, 33:96–99, 1987.

[271] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[272] Justin Thaler. Proofs, arguments, and zero-knowledge. *Found. Trends Priv. Secur.*, 4(2-4):117–660, 2022.

[273] Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM J. on Computing*, 20(5):865–877, 1991.

[274] B. A. Trakhtenbrot. Turing computations with logarithmic delay. *Algebra i Logika*, 3(4):33–48, 1964. In Russian; original paper on complexity-gap style results.

[275] Boris A. Trakhtenbrot. A survey of russian approaches to perebor (brute-force searches) algorithms. *IEEE Ann. Hist. Comput.*, 6(4):384–400, 1984.

[276] Luca Trevisan. Personal communication via salil vadhan. Email correspondence, 2006. Communicated to the author through Salil Vadhan.

[277] Luca Trevisan. The program-enumeration bottleneck in average-case complexity theory. In *CCC*, pages 88–95. IEEE Computer Society, 2010.

[278] Vladimir Trifonov. An o(logn loglogn) space algorithm for undirected st-connectivity. *SIAM J. Comput.*, 38(2):449–483, 2008.

[279] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[280] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

[281] Valiant. On non-linear lower bounds in computational complexity. In *ACM Symp. on the Theory of Computing (STOC)*, pages 45–53, 1975.

[282] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *6th Symposium on Mathematical Foundations of Computer Science*, volume 53 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1977.

[283] Leslie G. Valiant, Sven Skyum, S. Berkowitz, and Charles Rackoff. Fast parallel computation of polynomials using few processors. *SIAM J. Comput.*, 12(4):641–644, 1983.

[284] Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. *Theor. Comput. Sci.*, 47(3):85–93, 1986.

[285] J. H. van Lint. *Introduction to coding theory*. Springer-Verlag, Berlin, third edition, 1999.

[286] Dieter van Melkebeek. A survey of lower bounds for satisfiability and related problems. *Foundations and Trends in Theoretical Computer Science*, 2(3):197–303, 2006.

[287] Dieter van Melkebeek and Ran Raz. A time lower bound for satisfiability. *Theor. Comput. Sci.*, 348(2-3):311–320, 2005.

[288] Emanuele Viola. New lower bounds for probabilistic degree and AC0 with parity gates. *Theory of Computing*. Available at http://www.ccs.neu.edu/home/viola/.

[289] Emanuele Viola. The complexity of constructing pseudorandom generators from hard functions. *Computational Complexity*, 13(3-4):147–188, 2004.

[290] Emanuele Viola. The complexity of hardness amplification and derandomization. *Ph.D. thesis, Harvard University*, 2006.

[291] Emanuele Viola. Gems of theoretical computer science. Lecture notes of the class taught at Northeastern University. Available at http://www.ccs.neu.edu/home/viola/classes/gems-08/index.html, 2009.

[292] Emanuele Viola. On approximate majority and probabilistic time. *Computational Complexity*, 18(3):337–375, 2009.

[293] Emanuele Viola. On the power of small-depth computation. *Foundations and Trends in Theoretical Computer Science*, 5(1):1–72, 2009.

[294] Emanuele Viola. The sum of d small-bias generators fools polynomials of degree d. *Computational Complexity*, 18(2):209–217, 2009.

[295] Emanuele Viola. Reducing 3XOR to listing triangles, an exposition. Available at http://www.ccs.neu.edu/home/viola/, 2011.

[296] Emanuele Viola. Bit-probe lower bounds for succinct data structures. *SIAM J. on Computing*, 41(6):1593–1604, 2012.

[297] Emanuele Viola. The complexity of distributions. *SIAM J. on Computing*, 41(1):191–218, 2012.

[298] Emanuele Viola. The communication complexity of addition. *Combinatorica*, pages 1–45, 2014.

[299] Emanuele Viola. Lower bounds for data structures with space close to maximum imply circuit lower bounds. *Theory of Computing*, 15:1–9, 2019. Available at http://www.ccs.neu.edu/home/viola/.

[300] Emanuele Viola. Non-abelian combinatorics and communication complexity. *SIGACT News, Complexity Theory Column*, 50(3), 2019.

[301] Emanuele Viola, 2022. https://emanueleviola.wordpress.com/2022/09/14/myth-creation-the-switching-lemma/.

[302] Emanuele Viola. Correlation bounds against polynomials, a survey. 2022.

[303] Emanuele Viola and Avi Wigderson. Norms, XOR lemmas, and lower bounds for polynomials and protocols. *Theory of Computing*, 4:137–168, 2008.

[304] Emanuele Viola and Avi Wigderson. One-way multiparty communication lower bound for pointer jumping with applications. *Combinatorica*, 29(6):719–743, 2009.

[305] Avi Wigderson. *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press, 2019.

[306] Wikipedia contributors. List of PSPACE-complete problems. `https://en.wikipedia.org/wiki/List_of_PSPACE-complete_problems`, 2025. Accessed: 2025-10-10.

[307] Ryan Williams. *Algorithms and Resource Requirements for Fundamental Problems*. PhD thesis, Carnegie Mellon University, 2007.

[308] Ryan Williams. Non-uniform ACC circuit lower bounds. In *IEEE Conf. on Computa-*

tional Complexity (CCC), pages 115–125, 2011.

[309] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. *SIAM J. on Computing*, 42(3):1218–1244, 2013.

[310] Ryan Williams. Nonuniform ACC circuit lower bounds. *J. of the ACM*, 61(1):2:1–2:32, 2014.

[311] Ryan Williams. Simulating time in square-root space. *Electron. Colloquium Comput. Complex.*, TR25-017, 2025.

[312] Andrew Yao. Theory and applications of trapdoor functions. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 80–91. IEEE, 1982.

[313] Andrew Yao. Separating the polynomial-time hierarchy by oracles. In *26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 1–10, 1985.

[314] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 222–227. IEEE Computer Society, 1977.

[315] Andrew Chi-Chih Yao. Some complexity questions related to distributive computing. In *11th ACM Symp. on the Theory of Computing (STOC)*, pages 209–213, 1979.

[316] Andrew Chi-Chih Yao. On ACC and threshold circuits. In *IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 619–627, 1990.

"All that he does seems to him, it is true, extraordinarily new, but also, because of the incredible spate of new things, extraordinarily amateurish, indeed scarcely tolerable, incapable of becoming history, breaking short the chain of the generations, cutting off for the first time at its most profound source the music of the world, which before him could at least be divined. Sometimes in his arrogance he has more anxiety for the world than for himself." [161]

Appendix A

Table of complexity classes

NC^0			
AC^0		FO	
AC^0-Mod-2			
AC^0-Mod-6			
TC^0		FOM	
NC^1			
BrL		L	
AC^1			
NC^2			
		P	
		ZPP	RP
CktP		BPP	
			NP
	NP	Σ_2P	
		PH	
		BP$\cdot \oplus \cdot$P	
			MajP
		PSpace = IP	
		Exp	Nexp

Appendix B

Math facts

Of all escapes from reality, mathematics is the most successful ever.

Here we collect mathematical definitions and facts that are used in the main text.

B.1 Statistical distance

Definition B.1. Let P and Q be two distributions P and Q over a set X. The *statistical distance* between P and Q is

$$\frac{1}{2}\sum_{x\in X}|P(x)-Q(x)| = \max_{f:X\to[2]}|\mathbb{P}_P[f(P)=1]-\mathbb{P}_Q[(Q)=1]|.$$

The equivalence is left as exercise.

If two distributions have statistical distance ϵ then they can be typically used interchangeably up to an error of ϵ.

B.2 Logic

Fact B.1. For any logical statements P and Q, $\neg(P \vee Q) = \neg P \wedge \neg Q$.

B.3 Integers

Fact B.2. $\gcd(x,y) = \gcd(x \mod y, y)$.

Fact B.3. [Prime number theorem] $\lim_{n\to\infty}(\text{Number of primes} \leq n)/(n/\log_e n) = 1$.

B.4 Sums

Fact B.4. $1+\alpha+\alpha^2+\cdots+\alpha^\ell = \frac{1-\alpha^{\ell+1}}{1-\alpha}$ for any $\alpha \neq 1$, $\ell \in \mathbb{N}$. In particular, if $\alpha \in [0,1/2]$ the lhs is ≤ 2.

Proof. The trick is to multiply the lhs by $1-\alpha$ and note that every term cancels except 1 and $-\alpha^{\ell+1}$:

$$\left(1+\alpha+\alpha^2+\cdots+\alpha^{\ell}\right)(1-\alpha) = 1-\alpha^{\ell+1}.$$

Now if $\alpha \neq 1$ we can divide by $1-\alpha$, concluding the proof. **QED**

B.5 Basic inequalities

Fact B.5. $\binom{n}{k} \leq c2^n/\sqrt{n}$, for all k.

Fact B.6. $(n/k)^k \leq \binom{n}{k} \leq (en/k)^k$.

Fact B.7. $1+\alpha \leq e^{\alpha} \leq 1+\alpha+\alpha^2$, for all $\alpha \leq 1$.

The rhs is $\leq 1+2\alpha$ for $\alpha \in [0,1]$, and $\leq 1+\alpha/2$ for $\alpha \in [-1/2, 0]$ (because $\alpha(1+\alpha) \leq \alpha/2 \iff (1+\alpha) \geq 1/2$).

Fact B.8. $(1+\alpha)^r \geq 1+r\alpha$ for all $\alpha \geq -1$ and $r \geq 1$.

Fact B.9. For any $\alpha \in \mathbb{R}$,$1+\alpha \leq \frac{1}{1-\alpha} \leq 1+(1+\epsilon)\alpha$. The first inequality holds for any $\alpha \in \mathbb{R}$, the second for $\alpha \in [0, \epsilon/(1+\epsilon)]$.

For example, $1/(1-\alpha) \leq 1+2\alpha$ ($\epsilon = 1$) for $\alpha \leq 1/2$.

B.5.1 Squaring tricks

Fact B.10. For every real random variable X, $\mathbb{E}^2[X] \leq \mathbb{E}[X^2]$.

Proof. $\mathbb{E}[(X-\mathbb{E}[X])^2]$ is ≥ 0. Expand the square. **QED**

This is a special case of:

Fact B.11. For real random variables X, Y, jointly distributed: $\mathbb{E}^2[XY] \leq \mathbb{E}[X^2]\mathbb{E}[Y^2]$.

Proof. Let

$$Z := X - \frac{\mathbb{E}[XY]}{\mathbb{E}[Y^2]}Y.$$

Since $\mathbb{E}[Z^2] \geq 0$, expanding Z^2 concludes the proof. **QED**

Equivalently, for reals a_i, b_i one has $\left(\sum_i a_i b_i\right)^2 \leq \left(\sum_i a_i^2\right)\left(\sum_i b_i^2\right)$; and for vectors v, w one has $\langle v, w\rangle \leq |v| \cdot |w|$.

B.6 Probability theory

Developing intuition about random variables is one of the hardest skills to master, or even define. To anyone struggling I'd like to mention that my background was null, and in fact I didn't even like the emphasis on randomization, given the status of the field and in particular the grand challenge.

Naturally, with effort I grew to like probability theory. I think of it simply as *normalized counting*, and I do find the normalization useful. Many times when reading a new result I find myself translating the statements in the language of probability to make them more "physical."

I find the joke at the *incipit* of Chapter 3 a good illustration of how elusive the concept of probability is.

Fact B.12. [Linearity of expectation] TBD

Fact B.13. Let X be a real-valued r.v. s.t. $X \geq 0$ always. Then $\mathbb{P}[X \geq t] \leq \mathbb{E}[X]/t$ for every $t > 0$.

Exercise B.1. Prove this. Hint: Use that for any event E, $\mathbb{E}[X] = \mathbb{E}[X|E]\mathbb{P}[E]+\mathbb{E}[X|\text{not } E]\mathbb{P}[\text{not } E]$.

Fact B.14. If X and Y are independent, real-valued random variables then $\mathbb{E}[X \cdot Y] = \mathbb{E}[X] \cdot \mathbb{E}[Y]$.

B.6.1 Deviation bounds for the sum of random variables

Here we discuss probability bounds for the deviation of the sum of random variables from the mean. Such deviation bounds permeate theoretical computer science, and many other fields as well, see section §3.2.

Lemma B.1. Let $X_0, X_2, \ldots, X_{t-1}$ be i.i.d. boolean random variables with $p := \mathbb{P}[X_i = 1]$. Then for any $0 < p \leq q < 1$ we have $\mathbb{P}[\sum_{i\in[t]} X_i \geq qt] \leq 2^{-D(q|p)t}$, where

$$D(q|p) := q \log \left(\frac{q}{p}\right) + (1-q) \log \left(\frac{1-q}{1-p}\right)$$

is the *divergence.*

While the statement involves a complicated-looking quantity – divergence – it is the best bound on this pervasive quantity, and it is flexible: From it one can get a variety of inequalities by bounding divergence for different settings of parameter. We give an example and then state a general bound for D which we use shortly.

Example B.1. Let us toss t fair coins X_i, with $\mathbb{P}[X_i = \text{heads}] = 1/2 = p$. To bound the probability that we get $\geq 0.75 = q$ heads we compute the divergence $D(q|p) = 0.189\ldots$. Thus the probability is $\leq 2^{-D(q|p)t} \leq 2^{-0.189t}$. It decays exponentially fast and will be astronomically small even for moderate values of t.

Fact B.15. $D(q|p) \geq c(p-q)^2$, for any $p, q \in [0,1]$.

Exercise B.2. For $q = 1/2$ and $p = 1/2 - \epsilon$ plot both sides of Fact B.15 as a function of ϵ. (Hint: I used https://www.desmos.com/calculator)

We now return to the proof of Lemma B.1. We use the following facts.

Proof of Lemma B.1. For $z \geq 1$, to be picked later, the function $x \to z^x$ is increasing. Using this and then Fact B.13 and finally the independence of the X_i, the LHS equals

$$\mathbb{P}[z^{\sum_{i=1}^t X_i} \geq z^{qt}] \leq \frac{\mathbb{E}[z^{\sum_{i=1}^t X_i}]}{z^{qt}} = \frac{\prod_{i=1}^t \mathbb{E}[z^{X_i}]}{z^{qt}} = \left(\frac{pz + 1 - p}{z^q}\right)^t =: b^t.$$

To minimize b we set

$$z := \frac{q(1-p)}{(1-q)p}.$$

This value can be derived using calculus, see Problem B.4, or one can just remember it. Note $z \geq 1$ because $q \geq p$, and obtain

$$b = \frac{\frac{1-p}{1-q}}{z^q} = \left(\frac{p}{q}\right)^q \left(\frac{1-p}{1-q}\right)^{1-q}.$$

QED

The proof of the tail-bound Lemma B.1 is flexible and applies to a variety of useful settings. The most interesting extensions concern *dependent* random variables, where in general the bounds are weaker. In the next exercise we instead explore settings where the bounds in Lemma B.1 continue to hold; note independence is dropped in the last.

Exercise B.3. Prove that the tail bound in Lemma B.1 holds as stated more generally for any independent random variables $X_1, X_2, \ldots, X_t$ distributed in $[0,1]$ with $p := \sum_i \mathbb{E}[X_i]/t$. Guideline: Repeat the same proof as before. Use that $z^x \leq 1 + x(z-1)$ and the arithmetic-mean geometric mean inequality (AM-GM) inequality: for all $a_i \geq 0$: $(\sum_{i \in [t]} a_i)/t \geq (\prod_{i \in [t]} a_i)^{1/t}$.

Now suppose the X_i are more generally distributed in $[a,b]$. For $q = \epsilon + p$ prove a deviation bound of $2^{-c\epsilon^2 t/(b-a)^2}$.

Go back to the the tail bound in Lemma B.1. Prove it holds as stated even if the X_i are not independent, but conditioned on any $X_1, X_2, \ldots, X_{i-1}$, we have $\mathbb{E}[X_i] \leq p$.

Exercise B.4. Derive the minimizing value of z in the proof of Lemma B.1.

B.6.2 Groups

The theory of groups is *rich and pervasive.* A group is a set G equipped with an operation $\cdot$ called *multiplication* mapping $G^2 \to G$ and written xy for $x \cdot y$ which enjoys:

1. *Associativity*: $x(yz) = x(yz)$.
2. *Identity*: There exists an element $1_G \in G$ (also written 1) s.t. $1x = x1 = x$ for every $x \in G$.
3. *Inverse*: For every $x \in G$ there is x^{-1} (also written $1/x$) s.t. $xx^{-1} = x^{-1}x = 1$. We can think of multiplication by $1/x$ as *division* by x.

If in addition the operation $\cdot$ satisfies *commutativity*: $xy = yx$ for every $x, y \in G$ then the group is called *commutative.* In this case we sometimes use 0 for the identity element and $+$ for the operation, and we call $+$ *addition*.

Example B.2. TBD

A group G is *cyclic* if it is generated by a single elements: $G = \{1, g, g^2, g^3, \ldots\}$.

Theorem B.1. [Fundamental theorem of commutative finite groups] A finite group is commutative iff it is a direct product of finite cyclic groups.

A subgroup N of G is *normal*, written $N \lhd G$ if it is invariant under conjugation: $g^{-1}Ng = N$ for every $g \in G$. Normal subgroups allow us to define the *quotient* group G/N, which is the group of cosets of N, with operation $(gN)(hN) := ghN$. A finite group G is *solvable* if there is a series

$$\{1\} = G_1 \lhd \cdots \lhd G_t = G$$

s.t. the quotient groups G_{i+1}/G_i are cyclic.

Fact B.16. Any finite, not solvable group has a non-trivial subgroup whose commutator subgroup (i.e., the subgroup generated by commutators) is itself.

Fact B.17. $g^G = 1$ for every group G and $g \in G$.

B.7 Fields

A field is a set F with two operations, $+$ called *addition* and $\cdot$ called *multiplication* such that:

1. F with $+$ is a commutative group with identity element 0_F (written 0).
2. $F - \{0\}$ with $\cdot$ is a commutative group with identity element 1_F (written 1).
3. *Distributivity* of $\cdot$ over $+$: $x(y + z) = xy + xz$.

By convention, $\cdot$ has precedence over $+$.

Example B.3. The reals $\mathbb{R}$ or the rationals $\mathbb{Q}$ are *infinite* fields. The integers modulo a prime p form a finite field. For $p = 2$ this gives the field with two elements where $+$ is Xor and $\cdot$ is And. For larger p you add and multiply as over the integers but then you take the result modulo p.

Fact B.18. [Finite fields] A finite field of size q exists iff $q = p^t$ where p is a prime and $t \in \mathbb{N} - \{0\}$. This field is unique and denoted $\mathbb{F}_q$.

Elements in the field can be identified with $\{0, 1, \ldots, p-1\}^t$.

Given q, one can compute a *representation* of a finite field of size q in time $(tp)^c$. This representation can be identified with p plus an element of $\{0, 1, \ldots, p-1\}^t$.

Given a representation r and field elements x, y computing $x+y$ and $x \cdot y$ is in $\text{Time}(n \log^c n)$.

Fields of size 2^t are of natural interest in computer science. It is often desirable to have very explicit representations for such and other fields. Such representations are known and are given by simple formulas, and are in particular computable in linear time.

Example B.4. We can represent the elements of $\mathbb{F}_{p^t}$ as (the coefficients of) polynomials of degree $< t$ over $\mathbb{F}_p$. Addition is done component-wise, and multiplication occurs modulo an irreducible polynomial of degree t over the base field $\mathbb{F}_p$, i.e., a polynomial that cannot be factored as the product of two non-constant polynomials.

While in general computing irreducible polynomials is non-trivial, there are some very explicit families. For example, it is known that the polynomial $z^t + z^{t/2} + 1$ is irreducible over $\mathbb{F}_2$ for $t = 2 \cdot 3^\ell$ for any ℓ, giving very explicit representations. To illustrate field operations, consider the field elements $z^2 + 1$ and $z^{t-1} + 1$ over such a representation of $\mathbb{F}_{2^t}$. Their sum equals $z^{t-1} + z^2$, and their product equals $z^{t+1} + z^2 + z^{t-1} + 1 = z^{t-1} + z^{t/2+1} + z^2 + z + 1$.

Fact B.19. The multiplicative group of a field $\mathbb{F}$ is *cyclic*, meaning there exists a generator $g \in \mathbb{F} - \{0\} : \forall x \in \mathbb{F} - \{0\}, x = g^i$, for some $i \in \mathbb{N}$.

We can think of i as the *logarithm* of x (with respect to the generator g).

Example B.5. For $\mathbb{F}_5$ we can take $g = 2$: $2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8 = 3$.

B.8 Linear algebra

The only game in town.

First we recall list some basic definitions.

- A vector $v = (v_1, v_2, \ldots, v_n) \in \mathbb{R}^n$.
- Inner product $\langle v, w \rangle = \sum_i v_i \cdot w_i$.
- Two vectors are orthogonal, denoted $v \perp w$, if $\langle v, w \rangle = 0$.
- The length of a vector is $|v| = |v|_2 := \sqrt{\sum_i v_i^2} = \sqrt{\langle v, w \rangle}$.

Fact B.20. [Triangle inequality for vectors] $|v + w| \leq |v| + |w|$ for any vectors v, w.

Fact B.21. If $v \perp w$, then $|v + w|^2 = |v|^2 + |w|^2$.

Proof. The lhs is $\sum_i (v(i) + w(i))^2$. Expand the square and use orthogonality. **QED**

Fact B.22. Let $\alpha_i, i \in [n+1]$ be different elements from a field $\mathbb{F}$. Then the vectors $(\alpha_i^0, \alpha_i^1, \ldots, \alpha_i^n)$ are linearly independent.

Proof. We show it instead for the vectors $(\alpha_0^j, \alpha_1^j, \ldots, \alpha_n^j)$, then appeal to the fact that row rank equals column rank (for the $n+1 \times n+1$ matrix α_i^j). Suppose there are a_j, not all zero, s.t.:

$$\sum_{j \in [n+1]} a_j (\alpha_0^j, \alpha_1^j, \ldots, \alpha_n^j) = (0, 0, \ldots, 0).$$

Then the α_i are roots of the non-zero polynomial $\sum_{j \in [n+1]} a_j x^j$ of degree n. This contradicts Fact B.28. **QED**

Fact B.23. Orthogonal vectors v_i are in particular linearly independent.

Proof. Suppose that $\sum_i a_i v_i = 0$ for some coefficients a_i. Then for any j we have

$$0 = \langle \sum a_i v_i, v_j \rangle = \sum a_i \langle v_i, v_j \rangle = a_j \langle v_j, v_j \rangle,$$

and so $a_j = 0$. **QED**

Definition B.2. Let A be a $n \times m$ matrix, and B be a $n' \times m'$ matrix. The *tensor product* of A and B is is an $n \cdot n' \times m \cdot m'$ matrix defined as $(A \otimes B)_{(iA,iB),(jA,jB)} = A_{iA,jA} \cdot B_{iB,jB}$.

Diagrammatically,

$$A \otimes B = \begin{bmatrix} a_{11}B & \ldots & a_{1n}B \\ \vdots & \ldots & \vdots \\ a_{n1}B & \ldots & a_{nn}B \end{bmatrix}.$$

But the algebraic Definition B.2 makes most sense and is almost always more convenient.

Fact B.24. $\text{rank}(A \otimes B) = \text{rank}(A) \cdot \text{rank}(B)$.

Fact B.25. Every matrix is a root of its characteristic polynomial: If $p_A(x) := \det(xI - A)$ then $p_A(A) = 0$.

B.8.1 The eigenbasis Theorem 12.5

The proof relies on the fundamental theorem of algebra that every polynomial has a complex root. This proves the existence of eigenvectors. Then from first principles one can verify that for symmetric matrices they are real, and one can find an orthonormal basis.

B.9 Polynomials

Fact B.26. Let p and q be multi-variate polynomials over a field. Define the *degree* deg of a polynomial as the maximum sum of exponents of any monomial. Then $\deg(p \cdot q) = \deg(p) + \deg(q)$.

Proof. $\leq$ is obvious. $\geq$ is not because some terms may cancel. Define an ordering on monomials where larger degree comes first, and for equal degree we use lexicographic order. (That is, first compare the exponent of x_1, if equal compare the exponents of x_2, and so on.) We claim that the product of the first (in this order) monomial in p times the first monomial in q occurs in no other way, because if $m_1 > m_2$ and $m_3 > m_4$ then $m_1 \cdot m_3 > m_2 \cdot m_4$, where the m_i are any monomials. The result follows. **QED**

Definition B.3. The *elementary symmetric polynomial* of degree d (in n variables):

$$e_d(x_1, x_2, \ldots, x_n) := \sum_{S \subseteq [1..n], |S|=d} \prod_{i \in S} x_i.$$

The *power sum polynomial* of degree d in (n variables):

$$p_d(x_1, x_2, \ldots, x_n) := \sum_{i=1}^{n} x_i^d.$$

Fact B.27. $k \cdot e_k = \sum_{i=1}^{k} (-1)^{i-1} e_{k-i} \cdot p_i$, for all k.

Proof. Let $f(x) := (x-x_1)(x-x_2)\cdots(x-x_n)$. The coefficient of x^i in $f(x)$ is $e_{n,n-i}(x_1, x_2, \ldots, x_n) \cdot (-1)^{n-i}$. We also have $f(x_j) = 0$ for any j. Summing over j proves the claim for $k = n$. To prove it for $n > k$, consider any monomial on the LHS. Set to 0 all the other variables. Now the claim reduces to the case $n = k$. This shows that the coefficient of any monomial on the LHS is the same as that on the RHS, finishing the proof. **QED**

Fact B.28. [Polynomial identity] Let p be a polynomial over a field $\mathbb{F}$ with n variables and degree $\leq d$. Let S be a finite subset of $\mathbb{F}$, and suppose $d < |S|$. The following are equivalent:
p is the zero polynomial.
$p(x) = 0$ for every $x \in \mathbb{F}^n$.
$\mathbb{P}_{x_1, x_2, \ldots, x_n \in S}[p(x) = 0] > d/|S|$.

Proof of Fact B.28.. The implications 1. $\Rightarrow$ 2. $\Rightarrow$ 3. are trivial, but note that for the latter we need $d < |S|$. The implication 3. $\Rightarrow$ 1. is not trivial. We proceed by induction on n.

The base case $n = 1$ is the fact that if p has more than d roots then it is the zero polynomial. This fact in turn can be proved by induction on the degree. The base case $d = 0$ is obvious. For larger d, suppose a is a root of p and use division for polynomials to

write $p = (x-a)q + r$ where q has degree $\leq d-1$ and $r \in \mathbb{F}$. Because a is a root we have $r = 0$, and so $p = (x-a)q$ and q has $d-1$ roots, and by induction $q = 0$ and so $p = 0$.

For larger n write

$$p(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{d} x_1^i p_i(x_2, x_3, \ldots, x_n).$$

If p is not the zero polynomial then there is at least one i such that p_i is not the zero polynomial. Let j be the largest such i. Note that p_j has degree at most $d-j$. By induction hypothesis

$$\mathbb{P}_{x_2,\ldots,x_n \in S}[p_j(x) = 0] \leq (d-j)/|S|.$$

For every choice of $x_2, x_3, \ldots, x_n$ s.t. $p_j(x) \neq 0$, the polynomial p is a non-zero polynomial $q_{x_2,x_3,\ldots,x_n}(x_1)$ only in the variable x_1. Moreover, its degree is at most j by our choice of j. Hence by the $n = 1$ case the probability that q is 0 over the choice of x_1 is $\leq j$.

Overall,

$$\mathbb{P}_{x_1,x_2,\ldots,x_n \in S}[p(x) = 0] \leq (d-j)/|S| + j/|S| = d/|S|.$$

QED

Exercise B.5. Show that the equivalence between 1. and 2. does not hold over small fields such as $\mathbb{F}_2$ and large d.

B.10 Analysis of boolean functions over groups

B.10.1 Abelian groups

Fact B.29. Let D be a distribution over $[2]^n$ that fools degree-1 polynomials over $\mathbb{F}_2$ with error 0 (a.k.a. 0-biased). Then D is uniform.

Proof. By Exercise 11.4, we write for any $x \in [2]^n$,

$$D(x) = \sum_{\alpha} \widehat{D}_\alpha x^\alpha = \widehat{D}_\emptyset x^\emptyset = \widehat{D}_\emptyset = \mathbb{E}_y[D(y)y^\emptyset] = \mathbb{E}_y[D(y)] = \frac{1}{2^n}\sum_y D(y) = \frac{1}{2^n}.$$

QED

B.11 Notes

The introductory quote is from [238].

A tail bound similar to Lemma B.1 was first proved in 1938, see [75] and [76] for a translation. Since then, there has been an explosion of such bounds and proofs in the

literature. Lemma B.1 as stated first appeared in [67]. For a computer-science friendly introduction to this theory see the book [83]. For a number of different proofs of Lemma B.1 see [207].

The reference for Fact B.18 is [251]. For more on finite fields see [186], for the fields $\mathbb{F}_{2^t}$ in Example B.4 see Theorem 1.1.28 in [285].

TBD

For a history of the Polynomial Identity Fact B.28 and related results, see [50]. One can get a sharper bound taking into account the individual degrees of the variables, in addition to the total degree.

Hashing originates from [60] and pervades the computer science literature. The analysis of the best solution in Exercise **??** relies on Lemma 1 in [92].

Index