1 Algebraic formulas

Consider a complicated polynomial like

$$q(x,y) = x^5 + 5x^4y + 10x^3y^2 - x^3 + 10x^2y^3 - 6x^2y + 5xy^4 - 12xy^2 + y^5 - 8y^3. \qquad (1)$$

If you look at it, it might seem hopelessly unstructured, and there's no obvious way of simplifying it; for example, it turns out that q is *irreducible*, meaning it cannot be factored as a product of simpler polynomials. Nonetheless, there *is* a simpler way of representing q, since one can check that

$$q(x,y) = (x+y)^5 - (x+2y)^3. \qquad (2)$$

Intuitively, the representation (2) is clearly "simpler" than the representation (1). The first topic we will explore in this class, which we will shortly turn to, is a way of making this precise, by giving formal definitions for what a "representation" of a polynomial is, and of how "complicated" such a representation is.

However, there are more fundamental issues raised by this example: given a polynomial as in (1), can we find a simpler representation like (2)? If not, can we at least determine if a simpler representation exists? Can we prove that for certain polynomials, there are no simple representations?

To start studying these questions, let us formalize them. For the rest of the class, we will only work with polynomials with real coefficients, but we remark that almost everything we do works in greater generality.

Definition 1.1. An *algebraic formula* is any way of representing a polynomial using the following symbols: variables, constants (i.e. elements of $\mathbb{R}$), parentheses, $+$, and $\times$.

More formally, we can define an algebraic formula by the following recursive rules:

- Any constant $\alpha \in \mathbb{R}$ and any variable x is an algebraic formula.
- If F_1, F_2 are algebraic formulas, then so are $F_1 + F_2$ and $F_1 \times F_2$.

The *size* of an algebraic formula is the number of operations $+, \times$ that appear in it.

Every algebraic formula represents some polynomial—by expanding out all of the parentheses, we end up with some sum of monomials, which is a polynomial. Conversely, by writing a polynomial as a sum of monomials, we see that every polynomial can be represented by some algebraic formula. However, any given polynomial may have many different algebraic formulas computing[1] it. For example, here are two algebraic formulas computing

[1] I will generally use the words "compute" and "represent" interchangably.

the polynomial $q(x, y)$ above, corresponding to the two representations (1) and (2) above:

$$F_1 : \quad \begin{aligned} &(x \times x \times x \times x \times x) + (5 \times x \times x \times x \times x \times y) + (10 \times x \times x \times x \times y \times y) \\ &\quad + (-1 \times x \times x \times x) + (10 \times x \times x \times y \times y \times y) + (-6 \times x \times x \times y) \\ &\quad + (5 \times x \times y \times y \times y \times y) + (-12 \times x \times y \times y) \\ &\quad + (y \times y \times y \times y \times y) + (-8 \times y \times y \times y) \end{aligned}$$

$$F_2 : \quad \begin{aligned} &(x + y) \times (x + y) \times (x + y) \times (x + y) \times (x + y) \\ &\quad + (-1) \times (x + 2 \times y) \times (x + 2 \times y) + (x + 2 \times y) \end{aligned}$$

The size of F_1 is 49 and the size of F_2 is 19 (assuming I counted correctly). Thus, the size of a formula[2] is a reasonable way of quantifying the idea that some representations are more efficient than others—the size of F_2 is much smaller than that of F_1, corresponding to the intuitive idea that (2) is a simpler representation than (1).

Definition 1.2. The *formula complexity* of a polynomial p, denoted by $\mathsf{L}(p)$, is the size of the smallest formula computing p.

Thus, for the polynomial $q(x, y)$ above, we find that $\mathsf{L}(q) \leqslant 19$, because F_2 represents q and has size 19. I have no idea what the exact value of $\mathsf{L}(q)$ is (it might be 19, but I wouldn't be surprised if there is a smaller formula computing q). In general, it seems quite difficult to compute $\mathsf{L}(p)$ *exactly* for any polynomial p that is not extremely simple. And in general, we are not really interested in determining $\mathsf{L}(p)$ exactly; we instead care about getting a sense of *roughly* how big $\mathsf{L}(p)$ is.

One reason for this is that the exact value of $\mathsf{L}(p)$ depends a lot on the precise definition of algebraic formulas. For example, you could wonder why we don't allow subtraction in our algebraic formulas. If we did allow subtraction, we could represent q even more efficiently by replacing the multiplication by -1 in F_2 by a minus sign. However, as you will show on the homework, the value of $\mathsf{L}(p)$ changes by at most a factor of 2 if we allow minus signs, for any polynomial p. In other words, while allowing minus signs might change the exact value of $\mathsf{L}(p)$, it has no significant effect on the rough value—in this class, as in all of theoretical computer science, we will usually not care too much about extra constant factors like this 2.

With that understanding, let's see if we can get an idea of how large $\mathsf{L}(p)$ should be for a general polynomial p. To warm up, let's start with some of the simplest polynomials imaginable.

Proposition 1.3. *For the polynomial x^d, we have $\mathsf{L}(x^d) = d - 1$.*

Proof. We will prove both that $\mathsf{L}(x^d) \leqslant d - 1$, and that $\mathsf{L}(x^d) \geqslant d - 1$. The first inequality is pretty straightforward, since we have the formula

$$\underbrace{x \times x \times x \times \cdots \times x}_{d \text{ factors}}$$

[2] I will usually stop writing "algebraic formula" and simply write "fomula" from now on.

representing x^d, using $d-1$ multiplications.

However, the second inequality seems harder: there are infinitely many formulas representing x^d, and how can we be sure that none of them is smaller? To prove this, we will actually prove the following lemma.

Lemma 1.4. *If a formula F has size s and computes a polynomial p, then* $\deg(p) \leqslant s+1$.

We'll prove Lemma 1.4 in a moment, but first let's finish the proof that $\mathsf{L}(x^d) \geqslant d-1$. Indeed, suppose for contradiction that there is some formula F computing x^d with size $s < d-1$. By Lemma 1.4, we conclude that $\deg(x^d) \leqslant s+1 < d$, a contradiction. □

We now turn to the proof of Lemma 1.4. This proof will demonstrate one of the most commonly used proof techniques in this subject, which is induction on the structure of the formula. Indeed, recall that formulas are defined in a recursive fashion: we defined formulas with no operations (size 0) to be just constants and variables, and then every other formula is of the form $F_1 + F_2$ or $F_1 \times F_2$ for some simpler formulas F_1, F_2. This structure naturally lends itself to inductive proofs.

Proof of Lemma 1.4. We prove the lemma by (strong) induction on s. The base case is $s = 0$. By definition, a formula of size 0 is either a variable or a constant. Thus, the polynomial p it computes has degree 0 (if it's a constant) or 1 (if it's a variable), and the inequality $\deg(p) \leqslant s+1$ certainly holds for $s = 0$.

We now turn to the inductive step, so let us suppose we have proved the result for all numbers at most $s-1$. Let F be a formula of size s. By the definition of formulas, there exist formulas F_1, F_2 such that $F = F_1 + F_2$ or $F = F_1 \times F_2$. Let F_1, F_2 compute polynomials p_1, p_2, respectively, and have sizes s_1, s_2, respectively. Note that $s = s_1 + s_2 + 1$, since F is obtained from F_1, F_2 by joining them with a single extra operation.

First, suppose that $F = F_1 \times F_2$. Then the polynomial p computed by F satisfies $p = p_1 \times p_2$, and in particular $\deg(p) = \deg(p_1) + \deg(p_2)$. By the inductive hypothesis, we have that $\deg(p_1) \leqslant s_1 + 1$ and $\deg(p_2) \leqslant s_2 + 1$, and therefore

$$\deg(p) = \deg(p_1) + \deg(p_2) \leqslant (s_1 + 1) + (s_2 + 1) = (s_1 + s_2 + 1) + 1 = s + 1,$$

completing the proof of the inductive step.

Now, suppose instead that $F = F_1 + F_2$ so that $p = p_1 + p_2$. Adding polynomials cannot increase the degree, therefore

$$\deg(p) \leqslant \max\{\deg(p_1), \deg(p_2)\} \leqslant \max\{s_1 + 1, s_2 + 1\} \leqslant s + 1,$$

where we use the inductive hypothesis in the second inequality, and the fact that $s_1, s_2 \leqslant s$ in the final inequality. □

For the moment, let us stick with one-variable polynomials. Lemma 1.4 implies that if p is a degree-d polynomial, then $\mathsf{L}(p) \geqslant d-1$. How good is this bound for general one-variable polynomials?

Consider a general one-variable polynomial

$$p(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_1 x + a_0.$$

The most natural way to compute such a polynomial with a formula is to compute each term $a_i x^i$ separately, and then to add them all up with d extra additions. We know that $\mathsf{L}(x^i) = i - 1$ by Proposition 1.3, so $\mathsf{L}(a_i x^i) \leqslant i$. Therefore, together with the d additions, we conclude

$$\mathsf{L}(p) \leqslant d + \sum_{i=0}^{d} i = d + \binom{d+1}{2} = \frac{d^2 + 3d}{2}.$$

In particular, this bound is *quadratic* in the degree d, whereas our lower bound from Lemma 1.4 is *linear* in d.

It seems hopeless to meaningfully improve this upper bound for general one-variable polynomials. After all, Proposition 1.3 says that the "obvious" way of computing x^i is optimal, so how could we compute $p(x)$ other than computing each term in turn and then adding them up?

As it turns out, this is *not at all* the best way to compute one-variable polynomials.

Proposition 1.5 (Horner's rule)**.** *If $p(x)$ is a one-variable polynomial of degree d, then*

$$\mathsf{L}(p) \leqslant 2d.$$

Proof. Let $p(x) = a_0 + a_1 x + \cdots + a_d x^d$. Now, consider the algebraic formula

$$a_0 + x \times (a_1 + x \times (a_2 + x \times (\cdots + x \times (a_d) \cdots))).$$

This formula computes p, and has exactly d multiplications and d additions. □

Combining Lemma 1.4 and Proposition 1.5, we conclude the following theorem.

Theorem 1.6. *If $p(x)$ is a one-variable polynomial of degree d, then*

$$d - 1 \leqslant \mathsf{L}(d) \leqslant 2d.$$

This theorem gives us an essentially complete understanding of how large $\mathsf{L}(p)$ is for one-variable polynomials $p(x)$ (remember that we don't really care about absolute constant factors like the 2 above). Because of this, for most of the rest of the course we will mostly be interested in multivariate polynomials, where things get much more complicated and interesting. But before we do so, we need to introduce another way of computing polynomials.

2 Straight-line programs and algebraic circuits

Although this course is about abstract questions about polynomials, everything we are doing has its roots in theoretical computer science. This is evident in the word "compute" that we use to describe how a formula represents a polynomial. Also, efficient ways of representing polynomials, such as that in Proposition 1.5, are clearly useful if you *actually* want to compute some polynomial using a small number of additions and multiplications.

However, if you were actually computing some polynomial (or, more realistically, programming a computer to compute it), you wouldn't restrict yourself to formulas. In particular, you would allow yourself to run subroutines (i.e. compute some auxiliary polynomials), and then reuse the results of those subroutines multiple times. In "human" terms, this means that you are writing things down in a notebook, and you are always allowed to go back and use whatever you've done previously. The following definition formalizes this notion.

Definition 2.1. A *straight-line program* consists of a sequence $R_1, \ldots, R_m$ of *rows*, such that each row R_k is of one of the following two forms:

- R_k is a variable or a constant $\alpha \in \mathbb{R}$, or
- R_k is of the form $R_i + R_j$ or $R_i \times R_j$, for some indices i, j with $1 \leqslant i, j < k$.

The final row of the program, R_m, is the output: the program computes the polynomial written in row R_m.

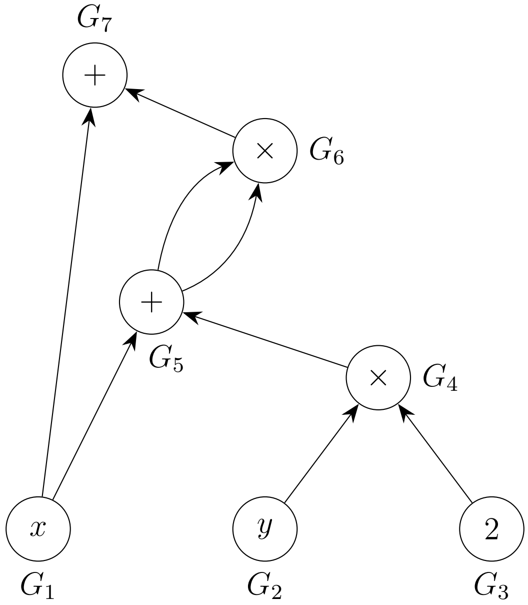
Note that this exactly formalizes the notebook idea above—you are writing down increasingly complex computations, where each computation is obtained from previous ones by either adding or multiplying. Throughout, you have access to the whole history of your computation, and you can use it as you'd like.

Here is an example of a straight-line program. On the left I write the straight-line program itself, and on the right the polynomial computed by every row.

R_1 :	x	x
R_2 :	y	y
R_3 :	2	2
R_4 :	$R_2 \times R_3$	$2y$
R_5 :	$R_1 + R_4$	$x + 2y$
R_6 :	$R_5 \times R_5$	$(x + 2y)^2$
R_7 :	$R_1 + R_6$	$(x + 2y)^2 + x$

Thus, this straight-line program computes the polynomial $(x + 2y)^2 + x$.

It is often convenient to express straight-line programs visually, as a graph. Such a representation is called an *algebraic circuit*. Rather than giving the formal, somewhat cumbersome, definition, let me simply draw the algebraic circuit corresponding to the straight-line program above.



In an algebraic circuit, we have a number of *gates*, labeled G_1–G_7 in the picture above. The gates correspond to the rows of the straight-line program; thus, a gate can either include a variable or a constant $\alpha \in \mathbb{R}$, or it can include one of the two algebraic operations. Every gate marked with an operation has two incoming arrows from gates labeled with a smaller number, corresponding to how a row of a straight-line program can be the sum or product of two earlier rows. Just as repetitions are allowed in a straight-line program, we allow two arrows to go between the same pair of gates. It is hopefully pretty clear how to convert from a straight-line program to an algebraic circuit, and vice versa.

Definition 2.2. The *size* of an algebraic circuit is the number of gates in it marked with $+$ or $\times$. Equivalently, the *size* of a straight-line program is the number of rows that are obtained from earlier rows by adding or multiplying.

The *circuit complexity* of a polynomial p, denoted $\mathsf{S}(p)$, is the minimum size of an algebraic circuit (or, equivalently, straight-line program) computing p.

Note that every algebraic formula computing p can be converted into a straight-line program computing p, with the same size, just by writing every instance of $+$ or $\times$ in the formula as a new row in the straight-line program. Because of this, we conclude that

$$\mathsf{S}(p) \leqslant \mathsf{L}(p) \tag{3}$$

for every polynomial p—the most efficient representation of p as a circuit is at least as efficient as the most efficient representation as a formula.

Perhaps surprisingly, the inequality (3) is often very far from being an equality. The simplest case demonstrating this is the following result, which stands in sharp contrast to Proposition 1.3.

Theorem 2.3. *For the polynomial x^d, we have $\mathsf{S}(x^d) \leqslant 2\lceil \log d \rceil - 1$, where here and throughout the course* log *denotes the base-*2 *logarithm.*

Proof. We begin by proving this when d is a power of 2, say $d = 2^k$. Here is a straight-line program computing x^d with $k = \log d$ multiplications.

$$\begin{array}{rcc|l}
R_1 : & & x & x \\
R_2 : & & R_1 \times R_1 & x^2 \\
R_3 : & & R_2 \times R_2 & x^4 \\
\vdots & & \vdots & \vdots \\
R_{k+1} & & R_k \times R_k & x^{2^k} = x^d
\end{array}$$

In general, suppose d is not a power of 2, and let $k = \lceil \log d \rceil$. By writing d in binary, we can express d as a sum of at most k powers of two, each of which is at most 2^{k-1}. But this means that by multiplying together at most k of the rows $R_1, \ldots, R_k$ above, we can compute x^d. So we had $k-1$ multiplications to produce those rows, plus another at most k multiplications to get to x^d, giving the claimed bound. □

In other words, for the simple polynomial x^d, we have $\mathsf{L}(x^d) = d - 1$, whereas $\mathsf{S}(x^d)$ is much smaller, namely at most[3] $2 \log d$. This demonstrates the power of using straight-line programs over formulas: the reason the *repeated squaring* algorithm above worked is that we could "remember" the powers we have already computed.

Given how much more powerful straight-line programs are, maybe there is an even more efficient way of computing x^d? The following, an analogue of Lemma 1.4 for circuits, shows that the answer is essentially no.

Lemma 2.4. *If a straight-line program has size s and computes a polynomial p, then* $\deg p \leqslant 2^s$.

Proof. As in the proof of Lemma 1.4, we prove this by induction on the structure of the program. The base case $s = 0$ is again clear, since a straight-line program of size 0 can only compute a constant or a variable, and hence $\deg(p) \leqslant 1 = 2^s$ in this case. For the inductive step, let the rows of the straight-line program be $R_1, \ldots, R_m$. We may assume that R_m, where the output of R_m is the polynomial p. If R_m is a constant or a variable, then $\deg(p) \leqslant 1$ again, and we are certainly done. If not, then we have that $R_m = R_i + R_j$ or $R_m = R_i \times R_j$ for some $1 \leqslant i, j < m$. Note that $(R_1, \ldots, R_i)$ is another straight-line program, and it has size at most $s - 1$ because $i < m$ and R_m does an operation. Similarly, $(R_1, \ldots, R_j)$ is a straight-line program of size at most $s - 1$.

Therefore, if we let p_i, p_j be the polynomials computed at R_i, R_j, respectively, we have by the inductive hypothesis that $\deg(p_i), \deg(p_j) \leqslant 2^{s-1}$. Now, if $R_m = R_i \times R_j$, then $p = p_i \times p_j$, and hence

$$\deg(p) = \deg(p_i) + \deg(p_j) \leqslant 2^{s-1} + 2^{s-1} = 2^s,$$

and if $R_m = R_i + R_j$ then $p = p_i + p_j$, and hence

$$\deg(p) \leqslant \max\{\deg(p_i), \deg(p_j)\} \leqslant 2^{s-1} < 2^s,$$

and in either case we have proved the inductive step. □

[3] I will start being lazy with keeping track of floor and ceiling signs; just as we don't care about absolute constant factors, we don't really care about adding or subtracting 1 to make sure the rounding is OK.

Corollary 2.5. *If* $\deg(p) = d$, *then* $\mathsf{S}(p) \geqslant \log d$.

Proof. If $s < \log d$, then any straight-line program of size s must compute a polynomial of degree at most $2^s < d$ by Lemma 2.4, and hence cannot compute p. □

In particular, this result shows that $\mathsf{S}(x^d) \geqslant \log d$. Thus, Theorem 2.3 is essentially best possible.

To summarize, what we have shown is that for any one-variable polynomial p of degree d, we have

$$\log d \leqslant \mathsf{S}(p) \leqslant \mathsf{L}(p) \leqslant 2d,$$

where the lower bound is by Lemma 2.4 and the upper bound by Proposition 1.5. This is a rather large gap; can we hope to close it?

Rather astonishingly, we are essentially at the limits of human knowledge. In particular, the following remains a major open problem.

Open problem 2.6. *Give an example of a one-variable polynomial* $p(x)$ *with degree* d *and* $\mathsf{S}(p) \not\leqslant O(\log d)$.

A few remarks about this open problem are in order:

- First, we recall the meaning of big-O notation; one says that $f \leqslant O(g)$ if there is an absolute constant C such that $f(x) \leqslant Cg(x)$ for all x. In particular, one way of rephrasing Theorem 2.3 is that $\mathsf{S}(x^d) \leqslant O(\log d)$.
- Because of the presence of the implicit constant hidden in the big-O, Open problem 2.6 is not "really" about a single polynomial p: every polynomial p has some finite $\mathsf{S}(p)$, and thus its circuit complexity is of the form $O(\log d)$, if we are allowed to pick the implicit constant sufficiently large.

  Thus, the correct way to understand Open problem 2.6 is as asking for a *infinite family* of polynomials, say $p_1, p_2, \dots,$ where $\deg(p_d) = d$ and $\mathsf{S}(p_d) \not\leqslant O(\log d)$. That is, we want the function $\mathsf{S}(p_d)$ to tend to infinity faster than $\log d$ as $d \to \infty$.
- A result of Hrubeš and Yehudayoff implies that there exist polynomials p of degree d and $\mathsf{S}(p) \geqslant \Omega(\sqrt{d})$. Here, the big-$\Omega$ is the "opposite" of the big-O; this says that, for some absolute constant $c > 0$, we have $\mathsf{S}(p) \geqslant c\sqrt{d}$. In particular, since $\sqrt{d}$ grows much faster than $\log d$, their result does yield polynomials p with $\mathsf{S}(p) \not\leqslant O(\log d)$. However, this is a purely *existential* result: they show that such a polynomial must exist, but give no clue as to what an example of such a polynomial is.

3 Factorials vs. factoring

Although Open problem 2.6 remains open, there are several natural candidates of polynomials whose circuit complexity is expected to be much larger than $\log d$. One such example are the *factorial polynomials*, defined by

$$f_d(x) \coloneqq (x-1)(x-2)(x-3)\cdots(x-d).$$

Note that f_d has degree d; it is widely believed that $\mathsf{S}(f_d) \not\leqslant O(\log d)$. One reason to believe this is the following remarkable result of Lipton.

Theorem 3.1 (Lipton)**.** *If $\mathsf{S}(f_d) \leqslant O(\log d)$, then there is an efficient algorithm for integer factorization.*

Before we see the proof of this result, let's understand what it means. By integer factorization, we mean the following problem: given a composite integer N, output a non-trivial factor of it (that is, some factor m with $1 < m < N$). Note that if you have an efficient algorithm for this problem, then by recursing on m and N/m, you can obtain the full factorization of N.

What does *efficient* mean here? Note that it takes $\log N$ binary digits to write the integer N, so the size of the input to this problem is $n \coloneqq \log N$. As is usual in computer science, an efficient algorithm is one that runs in polynomial time in the input, i.e. time at most $n^C = (\log N)^C$ for some constant C. Note that a naive algorithm for factoring N, namely trying every number between 2 and $\sqrt{N}$, has runtime roughly $\sqrt{N} = 2^{n/2}$, which is exponential in the input length n.

While better algorithms are known, it is widely believed that there is no polynomial-time algorithm for integer factoring. In fact, essentially the entire world economy rests on this assumption: the RSA cryptosystem, which is used all over the internet and other forms of digital communication, is only thought to be secure because of this assumption. If you could efficiently factor integers, then you could break most of the world's encryption. As such, while Theorem 3.1 does not prove that $\mathsf{S}(f_d) \not\leqslant O(\log d)$, it gives fairly compelling evidence.

Proof sketch of Theorem 3.1. Suppose we have an algebraic circuit which computes f_d with $O(\log d)$ operations, for every d. We are given a composite integer N of length $n = \lceil \log N \rceil$ as input, and let's say that $N = ab$ for some integers $1 < a < \sqrt{N} < b < N$.

Let $d = \sqrt{N}$, and note that $\log d = \frac{1}{2} \log N = \frac{n}{2}$. We now run the following (randomized) algorithm:

1. Pick a random integer $1 \leqslant x \leqslant N$.

2. Using our algebraic circuit, evaluate $f_d(x)$. This uses $O(\log d) = O(n)$ algebraic operations, hence this computation can be done in polynomial time, since addition and multiplication can be done in polynomial time.

3. Using the Euclidean algorithm (another efficient, polynomial-time algorithm), compute $m \coloneqq \gcd(f_d(x), N)$.

4. If $1 < m < N$, we have found a non-trivial factor of N.

Recall that $N = ab$, where $a < d < b$. In particular, the fact that $a < d$ implies that

$$f_d(x) \equiv 0 \pmod{a} \qquad \text{for every } x.$$

Indeed, by definition, we have that

$$f_d(x) \equiv (x-1)(x-2)\cdots(x-d) \pmod{a},$$

and since $a < d$, one of these d terms must be 0 mod a, and hence the whole product is zero. In particular, this implies that regardless of the random outcome of x, we have that $a \mid f_d(x)$. Since $a \mid N$ as well, we conclude that $a \mid m$, where we recall that $m = \gcd(f_d(x), N)$.

For this proof sketch, let us make two unjustified extra assumptions: b is prime, and $b \geqslant 2d$. Since we picked x randomly, its residue $x \pmod{b}$ is a random element of $\{0, 1, 2, \dots, b-1\}$. At most half of these elements are between 1 and d, by our assumption that $b \geqslant 2d$. Therefore, with probability at least $\frac{1}{2}$, we have

$$x \bmod b \notin \{1, 2, \dots, d\}$$

or equivalently

$$f_d(x) = (x-1)(x-2)\cdots(x-d) \not\equiv 0 \pmod{b},$$

since we assumed that b is prime.

In other words, with probability at least $\frac{1}{2}$, we have that $b \nmid f_d(x)$, and thus $b \nmid m$. So with probability at least $\frac{1}{2}$, the number m is a factor of N not equal to 1 (since $a \mid m$) and not equal to N (since $b \nmid m$). That is, with probability at least $\frac{1}{2}$, we output a non-trivial factor m of N. To deal with the fact that we fail with some probability, we simply repeat this algorithm over and over again, with a new random choice of x every time; with high probability, we will very quickly find a "good" x, and thus find a non-trivial factor of N. □

4 Multivariate polynomials

We now move on from single-variable polynomials, and turn to multivariate polynomials. Remember that in Proposition 1.5, we obtained a bound on $\mathsf{L}(p)$ that was linear in the degree of p, and that even earlier, we saw a simple algebraic formula showing $\mathsf{L}(p) \leqslant O(d^2)$ for every one-variable polynomial p with $\deg(p) = d$. It is unreasonable to expect something like this to hold for multivariate polynomials. For example, consider the polynomial

$$q(x_1, \dots, x_n) = \sum_{S \subseteq \{1,2,\dots,n\}} \prod_{i \in S} x_i.$$

This polynomial has n variables, degree n, and 2^n monomials. So the naive way of expressing q as a formula, by summing over all monomials, has size at least 2^n—much larger than n. However, for this specific polynomial, we still have $\mathsf{L}(q) \leqslant O(n)$. Indeed, another way of representing q is as

$$q(x_1, \dots, x_n) = (x_1+1)(x_2+1)\cdots(x_n+1), \tag{4}$$

and this gives an algebraic formula for q with n additions and $n-1$ multiplications.

Nonetheless, it is widely believed that there should exist "natural" polynomials—we will return shortly to what this means—which have n variables, degree n, and circuit complexity that is exponential in n.

Unfortunately, we are extremely far from proving this. In fact, the following two problems remain wide open.

Open problem 4.1. *Give an example of a polynomial $p(x_1, \dots, x_n)$ with n variables, degree $\deg(p) \leqslant n$, and $\mathsf{S}(p) \nleqslant O(n \log n)$.*

Open problem 4.2. *Give an example of a polynomial $p(x_1, \dots, x_n)$ with n variables, degree $\deg(p) \leqslant 10000$, and $\mathsf{S}(p) \nleqslant O(n)$.*

Note that for the second problem, the polynomial $p(x_1, \dots, x_n) = x_1 + \cdots + x_n$ has degree 1, and it is not hard to show that $\mathsf{S}(p) \geqslant \Omega(n)$. So Open problem 4.2 basically asks for a constant-degree polynomial that is "substantially harder" than this simple example.

But what's up with the $n \log n$ in Open problem 4.1? That comes from the following remarkable theorem.

Theorem 4.3 (Strassen, Baur–Strassen)**.** *The polynomial $t_{d,n}(x_1, \dots, x_n) = x_1^d + \cdots + x_n^d$ satisfies $\mathsf{S}(t_{d,n}) \geqslant \Omega(n \log d)$. In particular, $t_{n,n}$ has n variables, degree n, and $\mathsf{S}(t_{n,n}) \geqslant \Omega(n \log n)$.*

There are two important remarks to make about Theorem 4.3. The first is that, as indicated by Open problem 4.1, this theorem is at the limit of human knowledge: we do not know how to prove a stronger lower bound for any polynomial. The second is that this theorem is *not obvious*.

To indicate what I mean by the second point, let me try to convince you that Theorem 4.3 is obvious. First, we know by Corollary 2.5 that $\mathsf{S}(x_i^d) \geqslant \Omega(\log d)$ for each variable x_i. Second, the n monomials in $t_{d,n}$ each involve a different variable, so there's no way of combining these n computations. Each computation has circuit complexity at least $\Omega(\log d)$, so in total we conclude that $\mathsf{S}(t_{d,n}) \geqslant \Omega(n \log d)$.

This argument is bogus, and specifically the part that's bogus is that we cannot combine the n computations. It turns out that in some cases, there *are* ways of combining seemingly disparate computations, in such a way that the total complexity is less than the sum of its parts. A simple example is the polynomial $x^2 - y^2$. The naive computation of it involves two multiplications (to square x, and to separately sqaure y). But we know that $x^2 - y^2 = (x + y)(x - y)$, which shows that we can instead use only *one* multiplication, by combining the variables in an unexpected way.

A more dramatic example, more closely related to the above, has to do with matrix multiplication. To state this example correctly, we need to extend our definition of S to tuples of polynomials. Let $(p_1, \dots, p_\ell)$ be a tuple of polynomials. We say that an algebraic circuit *computes* $(p_1, \dots, p_\ell)$ if there is a collection of ℓ special gates in the circuit, such that the ith special gate computes p_i. We denote by $\mathsf{S}(p_1, \dots, p_\ell)$ the minimum size of a circuit computing $(p_1, \dots, p_\ell)$. We certainly have

$$\mathsf{S}(p_1, \dots, p_\ell) \leqslant \mathsf{S}(p_1) + \cdots + \mathsf{S}(p_\ell),$$

since we can just take an optimal circuit for each p_i in turn. The bogus argument above suggests that this inequality should always be an equality, but this is not at all the case.

Definition 4.4. The *matrix multiplication tuple* MM_k is a tuple of k^2 polynomials in $2k^2$ variables $\{x_{ij}, y_{ij}\}_{i,j=1}^k$, defined as follows. Make two $k \times k$ matrices of the variables, namely

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k1} & x_{k_2} & \cdots & x_{kk} \end{pmatrix} \qquad Y = \begin{pmatrix} y_{11} & y_{12} & \cdots & y_{1k} \\ y_{21} & y_{22} & \cdots & y_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k1} & y_{k_2} & \cdots & y_{kk} \end{pmatrix}.$$

The k^2 polynomials in the tuple MM_k are then the k^2 entries of the matrix XY.

Thus, for example, the first entry of MM_k is the polynomial

$$x_{11}y_{11} + x_{12}y_{21} + \cdots + x_{1k}y_{k1}.$$

This polynomial has circuit complexity $\Omega(k)$, as does every polynomial in MM_k. As there are k^2 polynomials in this tuple, the intuition above would suggest that $\mathsf{S}(\mathsf{MM}_k) \geqslant \Omega(k^3)$, since we need to do k^2 different computations, each of complexity k. And if you think about how you multiply matrices, this makes sense: whenever I have to multiply two matrices, I go through the entries of the output one-by-one. Nonetheless, the following remarkable theorem is true.

Theorem 4.5 (Strassen, Coppersmith–Winograd, Vassilevska Williams–Xu–Xu–Zhou, and many others). $\mathsf{S}(\mathsf{MM}^k)$ *is much smaller than* k^3. *Concretely,* $\mathsf{S}(\mathsf{MM}_k) \leqslant O(k^{2.371552})$.

In fact, it is widely believed that $\mathsf{S}(\mathsf{MM}_k) \leqslant O(k^{2+\varepsilon})$ for any $\varepsilon > 0$. Note that this is essentially best possible, since MM_k has $2k^2$ variables, so cannot be computed by any circuit of size less than k^2. That is, for the matrix multiplication problem, remarkable "economies of scale" are possible, and many seemingly independent computations can be efficiently combined.

5 The only lower bound we know

As we saw above, Theorem 4.3 is *not* obvious. So let's prove it!

Actually, we won't quite prove it—the proof uses some properties of partial derivatives which are beyond the scope of this class. But we will see a proof of the following closely related theorem.

Theorem 5.1 (Strassen). $\mathsf{S}(x_1^d, \ldots, x_n^d) \geqslant n \log d$.

That is, this theorem shows that (in contrast to other examples, such as matrix multiplication) there really are no "economies of scale" here. The best way to compute the n polynomials $x_1^d, \ldots, x_n^d$ is basically to just compute each of them independently. On the homework, if you are interested, you will see a proof of Theorem 4.3 from Theorem 5.1.

The proof of Theorem 5.1 relies on a foundational theorem of algebraic geometry, known as Bézout's theorem. Before we state it, recall the following two simple facts:

- A one-variable polynomial equation $f(x) = 0$ has at most $\deg(f)$ solutions.

- A system of linear equations has zero, one, or infinitely many solutions.

Bézout's theorem is a common generalization of these two facts. We do not state its strongest version, but a simple consequence that will suffice for us.

Theorem 5.2 (Bézout)**.** *Consider a system of multivariate polynomial equations*

$$\begin{cases} f_1(x_1, \ldots, x_n) = 0 \\ f_2(x_1, \ldots, x_n) = 0 \\ \qquad\qquad\vdots \\ f_m(x_1, \ldots, x_n) = 0 \end{cases}$$

Over $\mathbb{C}$, this system has either infinitely many solutions, or at most $\deg(f_1) \cdots \deg(f_m)$ *solutions.*

Note that the case where there is only one polynomial gives the first bullet point above, whereas the case when all the polynomials have degree 1 gives the second bullet point. We remark too that, since every solution over $\mathbb{R}$ is also a solution over $\mathbb{C}$, the same conclusion holds if we are only interested in solutions over $\mathbb{R}$.

We will not prove Theorem 5.2 in this class. But once we know it, it is not too hard to prove Theorem 5.1.

Proof of Theorem 5.1. Let $(R_1, \ldots, R_m)$ be a straight-line program of size s computing $x_1^d, \ldots, x_n^d$. Let p_k be the polynomial computed at row R_k. Additionally, for every row R_k, we introduce a new variable y_k. We now define a system of $m + n$ polynomial equations, as follows:

- If row R_k is a variable x_i, we add the equation $y_k = x_i$.

- If row R_k is a constant $\alpha \in \mathbb{R}$, we add the equation $y_k = \alpha$.

- If row R_k is of the form $R_i + R_j$, we add the equation $y_k = y_i + y_j$.

- If row R_k is of the form $R_k = R_i \times R_j$, we add the equation $y_k = y_i y_j$.

- If row R_k is one of our special output rows, we add the equation $y_k = 1$.

Note that we have $m + n$ polynomial equations—one equation for each of the m rows, plus n extra equations for our n special output rows. Additionally, at most s of these equations have degree 2, namely those equations corresponding to multiplication rows, and all other equations have degree 1. Call the system of equations we have defined S; S is a system of equations in the variables $x_1, \ldots, x_n, y_1, \ldots, y_m$.

Note that solutions to the system S are rather constrained. Indeed, fix some solution $(\overline{x_1}, \dots, \overline{x_n}, \overline{y_1}, \dots, \overline{y_m})$, where each $\overline{x_i}, \overline{y_m}$ is some complex number, such that this sequence of complex numbers is a solution of S. We claim that, for every k, we have

$$\overline{y_k} = p_k(\overline{x_1}, \dots, \overline{x_n}). \tag{5}$$

Indeed, this is proved by induction on k. For rows R_k which are variable or constant rows, (5) is immediate from the equation $y_k = x_i$ and $y_k = \alpha$, respectively. For the operation rows, we maintain (5) by induction: if $R_k = R_i + R_j$, then $p_k = p_i + p_j$, and therefore the equation $y_k = y_i + y_j$ implies

$$\overline{y_k} = \overline{y_i} + \overline{y_j} = p_i(\overline{x_1}, \dots, \overline{x_n}) + p_j(\overline{x_1}, \dots, \overline{x_n}) = p_k(\overline{x_1}, \dots, \overline{x_n}).$$

Similarly, rows $R_k = R_i \times R_j$ also preserve (5). Finally, recall that at the n special output rows, we have $p_k = x_i^d$, since we assumed this straight-line program computes $(x_1^d, \dots, x_n^d)$. Therefore, our solution also satisfies

$$\overline{x_i}^d = 1 \tag{6}$$

for all $1 \leqslant i \leqslant n$.

Now, consider the system of equations S' defined by

$$S' = \begin{cases} x_1^d = 1 \\ x_2^d = 1 \\ \quad\vdots \\ x_n^d = 1 \end{cases}$$

The argument above shows that solutions to S are in bijection with solutions to S'. Indeed, by (6), every solution to S is also a solution to S'. Conversely, given a solution to S', we may uniquely construct a solution to S by using the rule (5) to define the value of each of the variables y_k. These two operations are inverses, so we conclude that indeed, there is a bijection between solutions of S and solutions of S'.

Now, we observe that S' has exactly d^n solutions over $\mathbb{C}$. In particular, S' has finitely many solutions over $\mathbb{C}$, so S also has finitely many solutions over $\mathbb{C}$. Recall that S consists of at most s equations of degree 2, plus some number of equations of degree 1. By Theorem 5.2, and the fact that S has only finitely many solutions, we conclude that S has at most 2^s solutions over $\mathbb{C}$. But recalling that S and S' have the same number of solutions, we conclude that

$$2^s \geqslant d^n$$

or equivalently

$$s \geqslant n \log d. \qquad \square$$

We stress again that this simple but beautiful argument remains the best lower bound we know on the circuit complexity of *any* polynomial!

6 The determinant

One of the most important polynomials in all of mathematics is the determinant of a matrix. There are a number of different ways to define it; for our purposes, we will define the determinant of a $k \times k$ matrix

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k1} & x_{k_2} & \cdots & x_{kk} \end{pmatrix}$$

by the *Leibniz formula*, namely

$$\det(X) \coloneqq \sum_{\sigma \in S_k} \operatorname{sgn}(\sigma) \prod_{i=1}^{k} x_{i\sigma(i)}. \tag{7}$$

Here, S_k denotes the set of all permutations $\sigma : \{1, \ldots, k\} \to \{1, \ldots, k\}$, and $\operatorname{sgn}(\sigma)$ denotes the *sign* of a permutation. If you've never seen the sign of a permutation, don't worry too much about it; it is a natural way of declaring half of the permutations "positive" and half "negative". Thus, (7) tells us that in order to compute $\det(X)$, we need to run over all "generalized diagonals" of X (i.e. pick one entry from each row and each column of X), multiply the entries on this generalized diagonal, multiply by the sign of the permutation, and add up all of these contributions.

Note that we may view $\det_k \coloneqq \det(X)$ as a degree-k polynomial in the k^2 variables $x_{11}, \ldots, x_{kk}$. The formula (7) also gives an algebraic formula for computing this polynomial, implying that

$$\mathsf{S}(\det_k) \leqslant \mathsf{L}(\det_k) \leqslant O(k \cdot k!).$$

Note that $k!$ grows super-exponentially quickly in k, so this is a very poor bound.

However, if you've ever had to compute determinants by hand, you know it can be done much more quickly by using Gaussian elimination (aka row reduction). This algorithm *should* prove that $\mathsf{S}(\det_k) \leqslant O(k^3)$, but there's a catch: when you do Gaussian elimination, you frequently have to divide by matrix entries, and we do not allow division in our algebraic circuits! There are (at least) three ways of getting around this issue.

- We can say "eh" and allow division in our algebraic circuits.
- Strassen proved that, in fact, division is never necessary! Every algebraic circuit which does allow division but computes a polynomial can be converted into another circuit, not much larger, which has no division whatsoever.
- Berkowitz found a beautiful way to reduce the computation of the determinant to the computation of a bunch of matrix multiplication problems. In particular, his result implies that $\mathsf{S}(\det_k) \leqslant \mathsf{S}(\mathsf{MM}_k) \cdot O(\log k)$. Plugging in Theorem 4.5, we obtain an even better upper bound on $\mathsf{S}(\det_k)$ than the $O(k^3)$ bound given by Gaussian elimination, which uses no division.

If you are interested, you can learn about Berkowitz's algorithm (as well as another algorithm for computing the determinant) on the homework.

Thus, the determinant can be efficiently computed. Big whoop. Why should we care? Well, it turns out that there is a sort of converse to this statement: if a polynomial can be efficiently computed, then it *is* a determinant! To state this more precisely, we need the following definition.

Definition 6.1. A *symbolic matrix* X is a matrix whose entries are elements of $\mathbb{R}$ or variables. All of our symbolic matrices will always be square matrices, and we denote their *size* (i.e. number of rows or columns) by $|X|$.

Note that the determinant of a symbolic matrix is a polynomial in the variables appearing in it. Then the result stated informally above is rigorously stated as follows.

Theorem 6.2 (Valiant)**.** *Let $p(x_1, \ldots, x_n)$ be a polynomial with $\mathsf{L}(p) = s$. There exists a symbolic matrix X_p, of size $|X_p| = O(s)$ and using the variables $x_1, \ldots, x_n$, such that*

$$\det(X_p) = p(x_1, \ldots, x_n).$$

In the language of computational complexity theorem, we would say that "the determinant is VL-complete"; this means precisely that whenever a polynomial can be efficiently computed by an algebraic formula, then it can be efficiently "simulated" by a determinant of a symbolic matrix.

Theorem 6.2 can give an explanation for why determinants appear so frequently in so many different branches of mathematics, including combinatorics, differential equations, knot theory, and statistical physics: if we have a naturally-appearing polynomial which we can efficiently compute, Theorem 6.2 says that it simply *is* a determinant, regardless of where it came from!

For the proof of Theorem 6.2, we will need a few basic facts about the determinant. All of these are well-known, and you may have seen them in your linear algebra class; if there is one that is unfamiliar to you, please try to prove it or come ask me at TAU!

Lemma 6.3. *The determinant satisfies the following properties.*

(a) If $M = (\alpha)$ is a 1×1 matrix whose only entry is α, then $\det(M) = \alpha$.

(b) For a 2×2 matrix, we have

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc.$$

(c) Suppose that M is a square matrix of the form

$$M = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix},$$

where A, B are square matrices (not necessarily of the same size), and 0 denotes a submatrix all of whose entries are 0. Then $\det(M) = \det(A)\det(B)$.

(d) If M' is obtained from M by swapping two rows, then $\det(M') = \pm \det(M)$*, where the sign is determined by the parity of the distance between the two swapped rows.*

Proof of Theorem 6.2. We are given a polynomial p, as well as a formula F computing p, and would like to construct a symbolic matrix X_p satisfying $\det(X_p) = p$. By the same "induction on structure" idea we've seen a number of times, it suffices to figure out how to do the following steps:

1. Define X_p in case F is a formula of size 0, consisting of a constant or a variable.

2. Define X_p in case F is of the form $F_1 \times F_2$. Concretely, this means that given symbolic matrices X_1, X_2, we would like to construct a symbolic matrix $X_\times$ such that $\det(X_\times) = \det(X_1)\det(X_2)$.

3. Define X_p in case F is of the form $F_1 + F_2$. Concretely, this means that given symbolic matrices X_1, X_2, we would like to construct a symbolic matrix X_+ such that $\det(X_+) = \det(X_1) + \det(X_2)$.

Then, so long as we ensure that $X_\times, X_+$ are not much bigger than X_1, X_2, we will have proved the theorem.

The first step is quite straightforward. Indeed, if F consists of a variable x, we can simply define X_p to be the 1×1 symbolic matrix x. Similarly, if F consists of a constant $\alpha \in \mathbb{R}$, then we can again set $X_p = (\alpha)$. By Lemma 6.3(a), this will make the first step work.

Similarly, the second step is pretty easy, given Lemma 6.3(c): we can simply set

$$X_\times = \begin{pmatrix} X_1 & 0 \\ 0 & X_2 \end{pmatrix}.$$

Then the size of X is simply $|X_1| + |X_2|$, and we are still in good shape to obtain a good inductive bound on $|X_p|$.

However, the big problem is figuring out how to define X_+. This isn't simply me forgetting some useful property in Lemma 6.3: I am not aware of any way of building, from two general matrices X_1, X_2, a matrix X_+ with $\det(X_+) = \det(X_1) + \det(X_2)$.

This seems like bad news for proving Theorem 6.2, but luckily Valiant came up with a beautiful way of strengthening the induction hypothesis. We will now ensure that all of our matrices X_p have the following form:

$$\begin{pmatrix} * & * & * & * \\ 1 & * & * & * \\ 0 & 1 & * & * \\ 0 & 0 & 1 & * \end{pmatrix} \tag{8}$$

That is, all of our matrices will have unconstrained entries on and above the main diagonal, have all ones on the first sub-diagonal, and have all entries below that 0.

Given this new constraint, we have to go back and redo the steps we've already done, but luckily this is pretty easy. For formulas of size 0, we can simply define

$$X_p = \begin{pmatrix} 1 & 0 \\ 1 & x \end{pmatrix}$$

in case F is a single variable x, and similarly

$$X_p = \begin{pmatrix} 1 & 0 \\ 1 & \alpha \end{pmatrix}$$

in case F is a constant α. Lemma 6.3(b) implies that these two matrices have determinants x and α respectively, and they are of the desired form as in (8). So we have the base case of our induction established.

Similarly, it is not too hard to figure out how to define $X_\times$ and maintain the structure (8). Namely, we define $X_\times$ as before, except we now change a single 0 entry to 1 to maintain the property that we have ones on the first sub-diagonal. For example, if

$$X_1 = \begin{pmatrix} 3 & x & -7 \\ 1 & y & 0 \\ 0 & 1 & 2 \end{pmatrix} \qquad \text{and} \qquad X_2 = \begin{pmatrix} x & z & \pi \\ 1 & y & -2 \\ 0 & 1 & 1 \end{pmatrix},$$

then we define

$$X_\times = \begin{pmatrix} 3 & x & -7 & 0 & 0 & 0 \\ 1 & y & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & \boxed{1} & x & z & \pi \\ 0 & 0 & 0 & 1 & y & -2 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix},$$

where the only difference from before is that the entry in the box is now a 1 and not a 0. As this example shows, and as is not hard to prove in general, this operation maintains the structure (8). We also need to check that $\det(X_\times) = \det(X_1)\det(X_2)$ is still true. To prove this, consider which "generalized diagonals" can use the special, boxed 1. If we pick this entry, then we are not allowed to pick any other entry from its column. But then, if X_1 has k_1 rows, then from the first k_1 rows we must select k_1 entries in different columns, and only $k_1 - 1$ of these columns have any non-zero entries in them. Therefore, if we select the boxed 1, any way of completing to a generalized diagonal will have to take a 0 entry, and thus the product of the entries on the generalized diagonal is 0. This shows that we may as well ignore the generalized diagonal taking this entry. But in that case, we might as well turn this entry back to a 0, bringing us back to the setting of Lemma 6.3(c).

Finally, we need to define X_+. Let's first define another matrix, $\widetilde{X_+}$, which will almost but not quite work, and then we will see how to fix it to obtain X_+. We define $\widetilde{X_+}$ according to the following picture:

$$\widetilde{X_+} := \left(\begin{array}{c|c|c} & \quad 1 & \quad 1 \\ \hline 1 & X_1 & 0 \\ \hline 1 & 0 & X_2 \end{array}\right) = \left(\begin{array}{c|ccc|cccccc} & & & 1 & & & & & & 1 \\ \hline 1 & * & * & * & & & & & & \\ & 1 & * & * & & & & 0 & & \\ & & 1 & * & & & & & & \\ \hline 1 & & & & * & * & * & * & * & * \\ & & & & 1 & * & * & * & * & * \\ & & 0 & & & 1 & * & * & * & * \\ & & & & & & 1 & * & * & * \\ & & & & & & & 1 & * & * \\ & & & & & & & & 1 & * \end{array}\right)$$

That is, we start with a block-diagonal matrix consisting of X_1 and X_2, then add a row and a column to the left and top. The row is all zeroes, except for a 1 at the positions corresponding to the last columns of X_1, X_2. Similarly, the column is all zeroes, except for a 1 at the positions corresponding to the first rows of X_1, X_2.

Let's compute $\det(\widetilde{X_+})$. Any generalized diagonal must use one of the entries in the first column, and if it selects a 0 from the first column then it gives a contribution of 0. So we may restrict our attention to generalized diagonals that select one of the two 1 entries in the first column. Suppose first that such a diagonal uses the first 1. Now, it needs to select something non-zero from the second column, and the only option is the 1 under the main diagonal of X_1—all other entries in the second column are 0, except for one in the row we've already used. Similarly, in the third column, we must again select the 1 under the main diagonal of X_1, and so on. So we select all ones from X_1. Finally, in the final column of X_1, we are forced to select the 1 from the top row. In the remaining rows and columns, we may select any generalized diagonal from X_2, and nothing else. So we end up with a contribution of $\pm\det(X_2)$, where the sign is determined by the parity of the size of X_1.

On the other hand, if we select the second 1 from the first column, then a similar argument shows that our contribution is $\pm\det(X_1)$. So in total, we find that

$$\det(\widetilde{X_+}) = \pm\det(X_1) \pm \det(X_2),$$

where the signs are determined by the parities of $|X_1|$ and $|X_2|$. However, we already know how to fix the signs, since we know how to construct $X_\times$—that is, if we end up with a negative sign on, say, X_1, we simply do the above construction with X_1 replaced by X_1', defined so that $\det(X_1') = -\det(X_1)$ by adding two rows and two columns to X_1. By doing this, we construct a matrix whose determinant is exactly $\det(X_1) + \det(X_2)$.

However, the bigger issue is that $\widetilde{X_+}$ does not have the structure (8), so we cannot keep the induction going. Luckily, there is a simple fix for this, by swapping two rows of $\widetilde{X_+}$, namely the first row and the row corresponding to the top of X_2.

$$\left(\begin{array}{c|ccc|cccccc} & & & 1 & & & & & & 1 \\ \hline 1 & * & * & * & & & & & & \\ & 1 & * & * & & & 0 & & & \\ & & 1 & * & & & & & & \\ \hline 1 & & & & * & * & * & * & * & * \\ & & & & 1 & * & * & * & * & * \\ & & 0 & & & 1 & * & * & * & * \\ & & & & & & 1 & * & * & * \\ & & & & & & & 1 & * & * \\ & & & & & & & & 1 & * \end{array}\right) = \left(\begin{array}{c|ccc|cccccc} 1 & & & & * & * & * & * & * & * \\ \hline 1 & * & * & * & & & & & & \\ & 1 & * & * & & & 0 & & & \\ & & 1 & * & & & & & & \\ \hline & & & 1 & & & & & & 1 \\ & & & & 1 & * & * & * & * & * \\ & & 0 & & & 1 & * & * & * & * \\ & & & & & & 1 & * & * & * \\ & & & & & & & 1 & * & * \\ & & & & & & & & 1 & * \end{array}\right)$$

Doing so does return us to the structure in (8), and either preserves $\det(\widetilde{X_+})$ or negates it, by Lemma 6.3(d). And if it negates it, we already know how to deal with that!

In other words, we define X_+ as follows. First, we check whether it is necessary to add a pair of rows to X_1 or X_2 (or both) to negate their determinant, by computing a product $X_\times$ with the matrix $\left(\begin{smallmatrix} 1 & 0 \\ 1 & -1 \end{smallmatrix}\right)$ corresponding to the size-0 formula computing the constant -1. Then, we build $\widetilde{X_+}$ with these matrices. We then swap the two rows as described above, and, if necessary add two more rows to again negate the determinant. All in all, we end up with $\det(X_+) = \det(X_1) + \det(X_2)$, as desired. And the size of X_+ satisfies

$$|X_+| \leqslant |X_1| + |X_2| + 7,$$

since we add one row in building $\widetilde{X_+}$, and at most three pairs of rows to fix the signs. We also have that

$$|X_\times| = |X_1| + |X_2|$$

and that $|X_p| = 2$ if p is a variable or a constant, i.e. a size 0 formula.

These bounds imply, by induction, that a formula of size s can be represented as a determinant of a matrix of size at most $9s + 2$. □

7 P vs. NP, VP vs. VNP

What does all of this have to do with computer science? At first glance, not much; but it turns out that there is a deep and fruitful connection between algebraic complexity and more "standard" topics within theoretical computer science.

At an extremely high level of abstraction, all of computer science is concerned with the following problem: given some input, which we think of as a string of 0s and 1s, evaluate some output, which is itself (say) either a 0 or a 1. That is, computer science is concerned with the evaluation of *Boolean functions*, namely functions $f : \{0,1\}^n \to \{0,1\}$.

Just as every polynomial can be built up from constants, variables, addition, and multiplication, there is a similar structure for Boolean functions: every Boolean function can be expressed in terms of the variables of the input, plus three basic operations: negation ($\neg$), AND ($\wedge$), and OR ($\vee$).

Thus, every Boolean function can be computed by a *Boolean formula*, which is just an expression such as $(x_3 \wedge x_4) \vee (x_5 \vee (\neg x_1))$, consisting of variables, parentheses, and the three basic operations above. It is then natural to define the *formula complexity* of a Boolean function as the minimum size of a Boolean formula computing it. Similarly, every Boolean function can be computed by a Boolean circuit (or, equivalently, Boolean straight-line program), whose gates again correspond to the three basic operations, and we can similarly define the *circuit complexity* of a Boolean function. The "circuit" terminology actually comes from here—Boolean circuits were originally introduced as an abstract representation of what actually goes on in a computer chip, which has physical wires performing these computations.

In computational complexity theory, the fundamental question is to understand which Boolean functions can be efficiently evaluated by a computer, and which ones cannot be. There are many *complexity classes* capturing different levels of difficulty, the most famous and important of which are P and NP. P consists of all problems that can be solved in polynomial time, meaning in a number of operations that depends polynomially on the input length. By contrast, NP consists of all problems whose solution can be efficiently *verified*—it may be difficult to solve them, but if someone shows us the answer, we can check in polynomial time that the answer is indeed correct. A good example is a huge Sudoku puzzle—you may not know how to solve it, and doing it by brute force might take years, but it's not hard to check that a filled-in grid is indeed a correct solution. Speaking less precisely, NP consists of all "natural problems", i.e. problems that we might actually wish to solve in real life. Indeed, how could we even think about solving a problem not in NP—what does it mean to solve a problem if we can't even recognize a correct answer?

The most famous problem in theoretical computer science is the P vs. NP problem, which conjectures that P $\neq$ NP. That is, there should exist "natural" problems that cannot be efficiently solved. Despite decades of intense effort, this conjecture remains wide open.

Moving back to the world of Boolean circuits, one can define the complexity class P/poly to consist of all Boolean functions $f : \{0,1\}^n \to \{0,1\}$ whose Boolean circuit complexity is polynomial in n, the input size. In this language, the P vs. NP problem is essentially the same[4] as the assertion NP $\not\subseteq$ P/poly. That is, there should exist natural Boolean functions that cannot be computed by small (i.e. polynomially-sized) circuits.

In 1979, Valiant introduced analogues of these questions in the world of algebraic complexity, which we now turn to. The first complexity class he defined, now called VP, consists of all polynomials that can be computed by polynomial-sized algebraic circuits. Formally, we make the following definition.

Definition 7.1. Let $\{p_n\}_{n\geqslant 1}$ be a sequence of polynomials. We say that the sequence $\{p_n\}$ lies in the class VP if there exists a polynomial function $s : \mathbb{N} \to \mathbb{N}$ such that the following holds for all n:

- p_n is a polynomial in at most $s(n)$ variables,
- p_n has degree at most $s(n)$, and
- $\mathsf{S}(p_n) \leqslant s(n)$.

Of these conditions, the last one is the important one: it says that the circuit complexity of p_n is at most polynomial in n. The first two conditions simply enforce that the "input size", which in the algebraic world means the number of variables and the degree, are also both polynomial in n.

We remark that there is a closely related complexity class, called VL, which is defined identically except that we now require $\mathsf{L}(p_n) \leqslant s(n)$ rather than $\mathsf{S}(p_n) \leqslant s(n)$. This is a stronger condition, so VL is a smaller complexity class. However, an important result of Hyafil and of Valiant–Skyum–Berkowitz–Rackoff implies that, in a precise sense, $\mathsf{S}(p)$ cannot be "much bigger" than $\mathsf{L}(p)$, and therefore VL is roughly the same as VP.

The class VNP is also not difficult to define, but I will not do so in this course. However, I will give two vague descriptions of VNP. The first is that, just as NP consists of all "natural problems", VNP consists of all "natural polynomials", that is, all polynomials we have any hope of even *trying* to understand. The second is that VNP consists of all polynomials whose coefficients can be efficiently described. More precisely, consider a polynomial

$$p(x_1, \dots, x_n) = \sum_{\alpha_1,\dots,\alpha_n \geqslant 0} c_{\alpha_1,\dots,\alpha_n} x_1^{\alpha_1} \dots x_n^{\alpha_n}.$$

This polynomial is said to be in VNP (roughly) if there is a polynomial-time algorithm which takes as input a sequence $(\alpha_1, \dots, \alpha_n)$ and computes as output $c_{\alpha_1,\dots,\alpha_n}$. This makes precise the idea that VNP contains all polynomials we might hope to understand—if we can't even *describe* the coefficients, we have little hope of saying much about the polynomial.

[4] It is not hard to show that P $\subseteq$ P/poly, since any efficient algorithm can be converted into a small circuit. The reverse inclusion is actually false—circuits are a more powerful computational model than algorithms, because they are *non-uniform*. This is an important, but somewhat subtle, distinction, and I won't dwell on it; come ask me at TAU if you're interested!

It is not hard[5] to show that VP $\subseteq$ VNP. The algebraic analogue of the P vs. NP problem is called *Valiant's hypothesis*, and asserts that VP $\neq$ VNP; that is, there should be "natural" polynomials that cannot be efficiently computed by algebraic circuits.

This question is an analogue of the P vs. NP question, but it also turns out to be intimately related to it. We won't discuss this in any detail, but it turns out that stronger forms of the statement P $\neq$ NP imply the statement VP $\neq$ VNP. Because of this, and because there is so much more algebraic structure in the world of polynomials, many people consider VP vs. VNP a natural first step to solve in the direction of eventually proving P $\neq$ NP.

Our final topics are three examples of simple statements, each of which would imply (essentially) VP $\neq$ VNP. All of these are very promising directions, and it may be that one or two clever new ideas could lead to a major breakthrough.

8 Permanent vs. determinant

Given a $k \times k$ matrix

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1k} \\ x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k1} & x_{k_2} & \cdots & x_{kk} \end{pmatrix} \tag{9}$$

its *permanent* is the polynomial defined by

$$\mathrm{per}(X) := \sum_{\sigma \in S_k} \prod_{i=1}^{k} x_{i\sigma(i)}.$$

Note that this is the same as the definition of $\det(X)$, except that we do not include the pesky signs. The permanent is a less famous polynomial than the determinant, but it still appears in a ton of places in mathematics, including knot theory, graph theory, and statistical mechanics.

Given that there are no signs in the definition of the permanent, it seems like it should be easier to compute than the determinant. Unfortunately, the usual tricks we have for computing the determinant, such as Gaussian elimination, do not directly work for computing the permanent. And in fact, it is widely believed that computing the permanent is *very* hard.

Conjecture 8.1. $\mathsf{S}(\mathrm{per}_k)$ *is exponential in* k.

In particular, it is believed that $\mathrm{per}_k \notin$ VP. However, $\mathrm{per}_k \in$ VNP. In fact, Valiant proved that the permanent is *complete* for VNP, which roughly means that any polynomial in VNP can be efficiently represented as a permanent of a symbolic matrix (in the same way that Theorem 6.2 says that the determinant is VL-complete, i.e. that any polynomial computed

[5] OK, I haven't told you what VNP is, so I suppose it's extremely hard. But hopefully you can convince yourself that if $p \in$ VP, then you can efficiently compute its coefficients.

by a small formula can be efficiently represented as a determinant). Thus, Conjecture 8.1 would imply that VP $\neq$ VNP.

For an integer k, let us define $m(k)$ to be the minimum integer m such that there exists an $m \times m$ symbolic matrix Y such that

$$\det(Y) = \operatorname{per}(X),$$

where X is the $k \times k$ symbolic matrix from (9). In other words, $m(k)$ measures the most efficient way we can represent per_k as a determinant of some symbolic matrix. Note that $m(k)$ is well-defined, since we know by Theorem 6.2 that we can represent per_k as the determinant of *some* symbolic matrix.

A simple example, first observed by Pólya, is that $m(2) = 2$. Indeed, we have that

$$\operatorname{per}_2 = \operatorname{per}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad + bc = \det\begin{pmatrix} a & -b \\ c & d \end{pmatrix}.$$

The determination of $m(k)$ is a purely linear-algebraic question: how big does a symbolic matrix have to be for its determinant to compute the permanent of a smaller, given symbolic matrix? And yet, as proved by Valiant, this question is essentially the same as the VP vs. VNP question. Indeed, since the permanent and determinant are complete for their respective classes, we have the following result[6].

Theorem 8.2 (Valiant). *$m(k)$ is exponential in k if and only if* VP $\neq$ VNP.

The best known lower bound, due to Mignon and Ressayre, is that $m(k) \geqslant \Omega(k^2)$, and improving this is a major open problem.

9 Polynomial identity testing

The *polynomial identity testing (PIT)* problem is the following problem. Given two algebraic circuits C_1, C_2, determine if they compute the same polynomial. At first glance, this sounds pretty easy: just write down the two polynomials and check whether they are the same! More formally, write each polynomial as a sum of monomials, and see whether the coefficients match.

But actually, this is a lot trickier than it seems. For example, Degen's eight-square identity states that the polynomial

$$\left(a_1^2 + a_2^2 + a_3^2 + a_4^2 + a_5^2 + a_6^2 + a_7^2 + a_8^2\right)\left(b_1^2 + b_2^2 + b_3^2 + b_4^2 + b_5^2 + b_6^2 + b_7^2 + b_8^2\right)$$

[6] Strictly speaking, this is not 100% correct, but it's both morally and almost actually a true statement.

is equal to

$$\begin{aligned}(a_1b_1 - a_2b_2 - a_3b_3 - a_4b_4 - a_5b_5 - a_6b_6 - a_7b_7 - a_8b_8)^2 \\ + (a_1b_2 + a_2b_1 + a_3b_4 - a_4b_3 + a_5b_6 - a_6b_5 - a_7b_8 + a_8b_7)^2 \\ + (a_1b_3 - a_2b_4 + a_3b_1 + a_4b_2 + a_5b_7 + a_6b_8 - a_7b_5 - a_8b_6)^2 \\ + (a_1b_4 + a_2b_3 - a_3b_2 + a_4b_1 + a_5b_8 - a_6b_7 + a_7b_6 - a_8b_5)^2 \\ + (a_1b_5 - a_2b_6 - a_3b_7 - a_4b_8 + a_5b_1 + a_6b_2 + a_7b_3 + a_8b_4)^2 \\ + (a_1b_6 + a_2b_5 - a_3b_8 + a_4b_7 - a_5b_2 + a_6b_1 - a_7b_4 + a_8b_3)^2 \\ + (a_1b_7 + a_2b_8 + a_3b_5 - a_4b_6 - a_5b_3 + a_6b_4 + a_7b_1 - a_8b_2)^2 \\ + (a_1b_8 - a_2b_7 + a_3b_6 + a_4b_5 - a_5b_4 - a_6b_3 + a_7b_2 + a_8b_1)^2 \,.\end{aligned}$$

Good luck checking this by expanding everything out and comparing coefficients! There are 64 monomials in the first expression, which you have to compare with 224 monomials in the second (of which 160 will eventually cancel).

More formally, as we've seen in examples such as the determinant and (4), small algebraic circuits can compute polynomials with a huge number of monomials. In particular, a circuit of size s can compute a polynomial with exponentially many (in s) monomials.

An equivalent formulation of the PIT problem is to determine whether a given algebraic circuit C computes the zero polynomial (simply because $p_1 = p_2$ if and only if $p_1 - p_2 = 0$). As is standard in theoretical computer science, we would hope to have a polynomial-time algorithm for this problem: given a circuit of size s, compute in time polynomial in s whether it computes the zero polynomial or not. And as discussed above, simply expanding out everything in terms of monomials will not work—a circuit of size s can still have exponentially many monomials, so even writing all of them down will take much longer than polynomial time.

Nevertheless, there is an efficient *randomized* algorithm for solving PIT. Namely, given an algebraic circuit C computing some polynomial $p(x_1, \ldots, x_n)$, pick random assignments $a_1, \ldots, a_n$ for each of the variables (say, let each of them be a random real number in the interval $[0, 1]$). Using C, we can efficiently compute $p(a_1, \ldots, a_n)$. If p is the zero polynomial, then $p(a_1, \ldots, a_n) = 0$ with probability 1. However, if p is *not* the zero polynomial, then the probability that $p(a_1, \ldots, a_n) = 0$ is zero. This is perhaps easiest to intuitively see in the case of two variables: in this case, the set of (x, y) such that $p(x, y) = 0$ is some curve in the plane. Although this curve may be very complicated, it's still one-dimensional, so if pick a random point in the square $[0, 1]^2$, the probability that we lie on the curve is 0.

Thus, PIT is a central problem in *derandomization*, which is a field of theoretical computer science that seeks to convert randomized algorithms into efficient deterministic ones. For a long time, people thought of PIT as a good test case for derandomization, since there is so much algebraic structure which seems useful for building clever, efficient algorithms. But to this day, no one has found a polynomial-time algorithm for PIT. And in fact, the following remarkable theorem shows that finding such an algorithm would have *big* consequences.

Theorem 9.1 (Kabanets–Impagliazzo)**.** *If there is a polynomial-time algorithm for PIT, then* VP $\neq$ VNP*.*

OK, as I've stated, this theorem is a slight lie, but it is morally what Kabanets and Impagliazzo proved. And it's pretty amazing! In particular, it shows that if this specific problem—PIT—*can* be efficiently solved, then other problems *cannot* be efficiently solved!

Theorem 9.1 is too advanced to prove in this course, but it is actually not so hard once one develops some basic results in computational complexity theory.

Finally, let's remark that thanks to Theorem 6.2, the following is an (essentially) equivalent formulation of the PIT problem: given a symbolic matrix X, determine whether $\det(X) = 0$. This, in turn, can be equivalently formulated as a linear algebra problem: given a collection of $n \times n$ matrices over $\mathbb{R}$, determine whether some linear combination of them is invertible. Again, if you find an efficient algorithm for this linear-algebraic problem, you have proved VP $\neq$ VNP!

10 Elusive functions

A *polynomial mapping* is a function $\Gamma : \mathbb{R}^n \to \mathbb{R}^m$ each of whose coordinates is a polynomial. The *degree* of Γ is just the maximum degree of its coordinate functions. Thus, a polynomial mapping of degree at most 1 is the same as an affine linear map.

A very simple example of a polynomial mapping is the *moment curve* (also known as the *rational normal curve*), which is the function $f : \mathbb{R} \to \mathbb{R}^m$ defined by

$$f(t) = (t, t^2, \ldots, t^m).$$

For example, if $m = 2$, the image of f is just the parabola $y = x^2$ in the plane. A cool fact about the moment curve, whose proof is an exercise in linear algebra, is that its image does not lie in any affine hyperplane in $\mathbb{R}^m$. For example, in case $m = 2$, this just says that the parabola is not a subset of any line, which is pretty clear; the same statement remains true in higher dimensions.

We can turn this observation into a definition.

Definition 10.1. A polynomial mapping $f : \mathbb{R}^n \to \mathbb{R}^m$ is called *r-elusive* if for every polynomial mapping $\Gamma : \mathbb{R}^{m-1} \to \mathbb{R}^m$ of degree at most r,

$$\operatorname{im}(f) \nsubseteq \operatorname{im}(\Gamma).$$

In this language, the fact above states that the moment curve is 1-elusive. Indeed, if Γ is a polynomial mapping of degree at most 1, then $\operatorname{im}(\Gamma)$ is an affine hyperplane, and we said that the image of the moment curve is not contained in any affine hyperplane.

That's cool, but affine hyperplanes are fairly simple objects to understand. Can we find a 2-elusive polynomial mapping?

Theorem 10.2 (Raz)**.** *Let m grow exponentially with n. If there is an explicit 2-elusive polynomial mapping $\mathbb{R}^n \to \mathbb{R}^m$, then* VP $\neq$ VNP*.*

Here, as in Open problems 2.6, 4.1 and 4.2, the key point is having an explicit example. It is not very hard to show that 2-elusive functions *exist*, but in order to use Raz's result to conclude that VP $\neq$ VNP, we would need an explicit construction of one.

The way Raz proved Theorem 10.2 is roughly as follows. If we pick m appropriately, we can identify $\mathbb{R}^m$ with the space of all polynomials in n variables of degree at most n, simply by identifying a polynomial with its list of coefficients. One can then show, using ideas we have seen (plus some extra ones) that the set of all polynomials computed by circuits of size s is of the form $\text{im}(\Gamma)$, for some Γ of degree 2. That is, the polynomials computed by small circuits have a nice structure, and that structure is "algebraic": thus the set of all such polynomials is some algebraic variety, i.e. a set of the form $\text{im}(\Gamma)$. Finally, if we have an *explicit* 2-elusive function f, we can use it to construct a polynomial in VNP that does not lie on $\text{im}(\Gamma)$: basically, a point in $\text{im}(f)$ that is not on $\text{im}(\Gamma)$ corresponds to some polynomial, and the fact that f is explicit implies that this polynomial is in VNP. Since it does not lie on $\text{im}(\Gamma)$, it cannot be computed by a small circuit, hence VP $\neq$ VNP!

A more detailed (but still incomplete) proof sketch of Theorem 10.2 can be found at `https://n.ethz.ch/~ywigderson/math/static/ElusiveFunctions.pdf`.