Computer Science 2: Advanced Algorithms Fall 2026

Lecture 12 — Sep 28, 2026

Prof. Jarosław Błasiok *Scribe: Ábel Nagy, Anh Chi Tuan, Sayan Dutta*

1 Overview

Almost every previous lecture of this course used randomness somewhere: a random evaluation point for identity testing, a random starting vector for the power method, a random walk for *st*-connectivity, ... In each case the randomized algorithm was short and usually correct. Today we ask a question about randomness itself:

Is randomness a genuine computational resource, or only a convenience?

In other words, can every efficient randomized algorithm be replaced by an efficient deterministic one? The answer we will see is a conditional "yes", and the condition is surprising. If there exists a problem that is really hard (hard for small Boolean circuits), then randomness gives no extra power at all, i.e. $\mathsf{P} = \mathsf{BPP}$. This is the hardness versus randomness paradigm, and its strongest form is the theorem of Impagliazzo and Wigderson [3].

2 Do we need randomness?

2.1 A motivating example: polynomial identity testing

Recall the very first problem of the course.

Polynomial identity testing (PIT). We are given an arithmetic circuit computing a polynomial $p \in \mathbb{F}[x_1, \ldots, x_n]$ of degree d and we must decide whether $p \equiv 0$.

We cannot just expand the polynomial: the circuit may be small while the expansion is huge. For instance $\prod_{i=1}^{n}(x_i + y_i)$ has a circuit with $O(n)$ gates but 2^n monomials. The randomized solution is two lines long.

Lemma 1 (Schwartz–Zippel [7, 10])**.** *Let $p \not\equiv 0$ have degree d and let $S \subseteq \mathbb{F}$ be finite. Then $\mathbb{P}_{a \sim S^n}\left[p(a) = 0\right] \leq \frac{d}{|S|}$.*

So we evaluate p at one random point $a \in S^n$ with $|S| \geq 3d$ and answer "$p \equiv 0$" iff $p(a) = 0$. Geometrically, the zero set of a non-zero polynomial is "thin", and a random point misses it. The error is one-sided, and repeating makes the error probability as small as we like.

And yet, no deterministic polynomial-time algorithm for PIT is known.

2.2 The class BPP

Definition 2 (BPP)**.** *A decision problem L is in* BPP *(bounded-error probabilistic polynomial time) if there is a deterministic polynomial-time algorithm $A(x, r)$, taking an input x and a string of random bits r, such that*

$$x \in L \;\Rightarrow\; \mathop{\mathbb{P}}_{r}\big[A(x,r)=1\big] \geq \tfrac{2}{3},$$
$$x \notin L \;\Rightarrow\; \mathop{\mathbb{P}}_{r}\big[A(x,r)=1\big] \leq \tfrac{1}{3}.$$

Remark 3 (The constant 2/3 does not matter)**.** If we run A independently k times and output the majority answer, then by a Chernoff bound the error probability drops to $e^{-\Omega(k)}$. So 2/3 could be replaced by any constant larger than 1/2, or even by $1 - 2^{-n}$, without changing the class.

$P \subseteq$ BPP holds because one can create a function $A(x, \cdot)$ ignoring the randomness. BPP $\subseteq$ EXP holds because the the function could iterate through all exponentially many r and return the majority. Some famous problems were first solved with randomness and only much later deterministically: primality testing had randomized algorithms in 1977 [9] and a deterministic one in 2002 [1] (with the running time later improved [4]); undirected st-connectivity in small space was solved by a random walk and derandomized by Reingold [6] (Lecture 5).

What we believe. P = BPP: every randomized algorithm can be simulated deterministically with only polynomial overhead.

Today we see a generic theorem that would derandomize everything at once.

2.3 Today's theorem

Theorem 4 (Impagliazzo–Wigderson [3])**.** *Suppose there is a language L decidable in time $2^{O(n)}$ that requires Boolean circuits of size $2^{\delta n}$, for some $\delta > 0$. Then*

$$\mathsf{P} = \mathsf{BPP}.$$

The theorem gives rise to the idea of Pseudo-Random Generators (PRG). Intuitively, if there is a decision problem none can solve in subexponential time, i.e., distinguish the pseudo-random bits from true randomness, then one can replace the random bits with pseudo-randomness and reuse A from Definition 2.

3 Boolean circuits

To state the hypothesis precisely we need the right notion of "hard". It turns out to be circuit size, not running time.

Definition 5 (Boolean circuit)**.** *A Boolean circuit on n inputs is a directed acyclic graph with*

- *n input nodes (sources), labelled $x_1, \ldots, x_n$;*
- *internal nodes (gates) labelled $\wedge$, $\vee$ or $\neg$;*
- *one output node.*

The value of a gate is obtained by applying its operation to the values of its in-neighbours. The ***size*** *of the circuit is its number of nodes.*

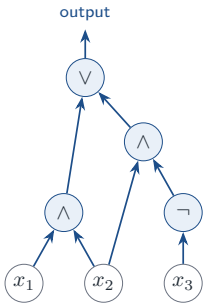


Figure 1: A circuit of size 7 computing $f(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge \neg x_3)$.

Every circuit computes a unique function $f : \{0,1\}^n \to \{0,1\}$, and every function is computed by some circuit (for instance, write f as an OR over its 1-inputs, which costs about $n2^n$ gates). The interesting question is how big the smallest circuit must be.

A circuit has a fixed number of inputs, so to solve a problem on inputs of all lengths we need one circuit per length.

Definition 6 (Circuit families)**.** *A problem L is solved by a circuit family of size $T : \mathbb{N} \to \mathbb{N}$ if for every n there is a circuit C_n of size at most $T(n)$ that decides L on all inputs of length n.*

Theorem 7 (essentially Cook–Levin)**.** *A problem solvable in time $T(n)$ is solvable by a circuit family of size $O(T(n) \log T(n))$.*

The idea is to unroll the computation: one small circuit computes a single step of the machine, and we wire $T(n)$ copies of it in sequence.

Consequence. Circuit lower bounds are stronger than running-time lower bounds: if L needs circuits of size $S(n)$, it also needs time roughly $S(n)$. The converse fails, since a circuit family may even compute something uncomputable: the family is allowed to be different for each n, with no algorithm producing it. For example, the unary language $\{1^n : \text{the } n\text{-th Turing machine halts}\}$ has circuits of size $O(n)$ (for each n, the circuit is either "always 0" or "AND of all inputs"), although it is undecidable. This feature is called non-uniformity.

3.1 Should we believe that hard problems exist?

Yes, by counting (Shannon [8]). There are 2^{2^n} functions $\{0,1\}^n \to \{0,1\}$. A circuit of size S is described by choosing, for each gate, its type and its (at most two) inputs, so there are at most $(3(S+n)^2)^S = 2^{O(S \log S)}$ circuits of size S. If $S \log S \ll 2^n$ there are far fewer circuits than functions, so almost every function needs circuits of size $\geq 2^n/(10n)$. This is the probabilistic method at its simplest.

Example 8 (A concrete count)**.** Take $n = 10$, so there are 2^{1024} functions. Circuits with at most 60 gates number at most $60 \cdot (3 \cdot 70^2)^{60} < 2^{837}$. Hence a uniformly random function of 10 variables needs more than 60 gates with probability at least $1 - 2^{-187}$. Yet nobody can point to an explicit family of functions that provably needs, say, $10n$ gates.

But we need an explicit hard function. The hard function must be computable in time $2^{O(n)}$ (otherwise it is useless for building an efficient generator). By the time hierarchy theorem there are problems in E that need time $2^{\Omega(n)}$, but for circuit size the best known lower bound for an explicit function is only about $5n$ [2]. So the hypothesis of Theorem 4 is a conjecture: plausible, widely believed, wide open, and far beyond current techniques.

Assumption 9 (Hardness)**.** *There is a family $f = \{f_n\}$, $f_n : \{0,1\}^n \to \{0,1\}$, computable in time $2^{O(n)}$, and a constant $\delta > 0$ such that no circuit of size $2^{\delta n}$ computes f_n.*

Let us read it carefully.

- "Computable in time $2^{O(n)}$": the whole truth table of f_n can be written down in time $2^{O(n)}$, which is polynomial in its length 2^n.
- "Requires size $2^{\delta n}$": δ may be tiny, and the two exponents need not match.
- This is **worst-case** hardness: a circuit "fails" as soon as it is wrong on a single input.

The plan: turn this single hard function into pseudorandom bits.

4 Pseudorandom generators

4.1 Indistinguishability

Definition 10 (ε-indistinguishability)**.** *Distributions $\mathcal{D}_1, \mathcal{D}_2$ over $\{0,1\}^N$ are ε-indistinguishable by a class $\mathcal{F}$ of functions $\{0,1\}^N \to \{0,1\}$ if*

$$\left| \mathop{\mathbb{E}}_{x \sim \mathcal{D}_1} f(x) - \mathop{\mathbb{E}}_{x \sim \mathcal{D}_2} f(x) \right| < \varepsilon \qquad \text{for every } f \in \mathcal{F}.$$

Think of each $f \in \mathcal{F}$ as a distinguisher, or a statistical test: it looks at a sample and says 1 ("looks like $\mathcal{D}_2$") or 0. Since f is Boolean, $\mathbb{E}\, f(x) = \mathbb{P}[f(x) = 1]$, so the quantity above is the difference in acceptance probabilities. If some f usually says 0 on samples from $\mathcal{D}_1$ and usually 1 on samples from $\mathcal{D}_2$, this difference is large and f tells the distributions apart.

How strong the notion is depends entirely on $\mathcal{F}$.

- $\mathcal{F}$ = all functions: this is the total variation distance. Far too strong for our purposes (see below).
- $\mathcal{F}$ = circuits of size at most T: computational indistinguishability. The two distributions may be miles apart; they only have to look alike to a bounded observer.

A pseudorandom generator is a way to produce a distribution that looks uniform to every bounded observer. When we later plug its output into a randomized algorithm, the algorithm itself (written as a circuit) is the observer. Since it cannot tell the fake coins from real ones, it cannot behave differently on them.

4.2 Pseudorandom generators

Definition 11 ((ε, T)-PRG)**.** *A function $g : \{0,1\}^s \to \{0,1\}^N$ is an (ε, T)-pseudorandom generator if $g(X)$, for X uniform on $\{0,1\}^s$, is ε-indistinguishable from $\mathrm{Unif}(\{0,1\}^N)$ by all circuits of size*

at most T. The input X is called the ***seed****.*

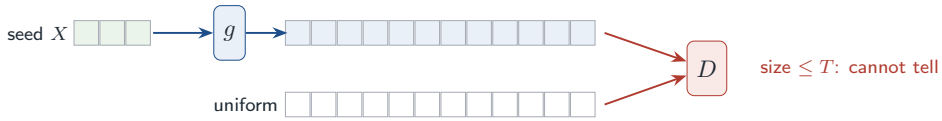


Figure 2: A PRG turns s truly random bits into $N \gg s$ bits that no distinguisher D of size $\le T$ can tell apart from N truly random bits.

Note that $g(X)$ takes at most 2^s values, a tiny fraction of $\{0,1\}^N$. So $g(X)$ is very far from uniform in total variation: the (huge) test "is the string in the image of g?" distinguishes perfectly. Pseudorandomness only makes sense against bounded observers.

4.3 A short seed is all we need

The reason we care about PRGs is the following lemma: a PRG with a logarithmic seed derandomizes BPP.

Lemma 12. *Suppose that for every N there is a $(1/10, N)$-PRG $g : \{0,1\}^{O(\log N)} \to \{0,1\}^N$ computable in time* $\mathrm{poly}(N)$*. Then* $\mathsf{P} = \mathsf{BPP}$*.*

Intuition

A seed of length $O(\log N)$ has only $\mathrm{poly}(N)$ possible values, so we can try them all. Since the algorithm cannot tell pseudorandom bits from truly random ones, its majority vote over all seeds is still correct.

Proof. Let $L \in \mathsf{BPP}$ be decided by $A(x,r)$ in time $T = \mathrm{poly}(n)$; then A uses at most T random bits.

Step 1: freeze the input. Fix an input x. By Theorem 7, the map $r \mapsto A(x,r)$ is computed by a circuit C_x of size $\tilde{O}(T)$. Take $N = \mathrm{poly}(n)$ at least this size (and at least the number of random bits). By definition of BPP,

$$x \in L \Rightarrow \mathop{\mathbb{E}}_r C_x(r) \ge \tfrac{2}{3}, \qquad x \notin L \Rightarrow \mathop{\mathbb{E}}_r C_x(r) \le \tfrac{1}{3}.$$

Since C_x is a circuit of size $\le N$, the PRG is unanable to distinguish which distribution did the majority of the inputs came fromfools it: $\left|\mathbb{E}_s C_x(g(s)) - \mathbb{E}_r C_x(r)\right| < \frac{1}{10}$. Therefore

$$x \in L \Rightarrow \mathop{\mathbb{E}}_s C_x(g(s)) > \tfrac{2}{3} - \tfrac{1}{10} > \tfrac{1}{2}, \qquad x \notin L \Rightarrow \mathop{\mathbb{E}}_s C_x(g(s)) < \tfrac{1}{3} + \tfrac{1}{10} < \tfrac{1}{2}.$$

Step 2: enumerate the seeds (Figure 3). Run $A(x, g(s))$ for every seed s and output the majority. The displayed inequalities say exactly that a strict majority of seeds gives the right answer, so the majority vote is always correct. There are $2^{O(\log N)} = \mathrm{poly}(N)$ seeds, each $g(s)$ costs $\mathrm{poly}(N)$ and each run of A costs T, so the total running time is polynomial. □

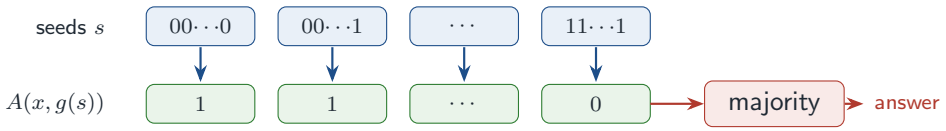


Figure 3: Derandomization by brute force over the seeds.

Two comments. First, the final algorithm is deterministic and always correct; there is no error probability left. Second, it is crucial that the generator fools every small circuit, because we do not know in advance which input x (and hence which test C_x) we will face.

So everything now rests on building g.

4.4 Such generators exist, but that is not enough

Lemma 13. *For every N there exists a $(1/10, N)$-PRG $g : \{0,1\}^{O(\log N)} \to \{0,1\}^N$.*

Proof. Probabilistic method. Pick g at random, i.e. choose 2^s independent uniform strings $g(1), \ldots, g(2^s) \in \{0,1\}^N$. For one fixed circuit C, the average of C over these 2^s strings is within $1/10$ of $\mathbb{E}_r\, C(r)$ except with probability $e^{-\Omega(2^s)}$ (Chernoff). There are only $2^{O(N \log N)}$ circuits of size N, so a union bound works as soon as $2^s \gg N \log N$, i.e. $s = O(\log N)$. □

While existence is easy, the contruction is computationally hard and impractical.

Remark 14 (No circularity)**.** The generator runs in time $\mathrm{poly}(N)$ and only needs to fool circuits of size N, so it is allowed to be slower than its adversary. This is what makes the hardness-vs-randomness approach possible with a mere "exponential-time" hard function. Cryptographic PRGs must instead be faster than the adversaries they fool, which is why they need stronger assumptions (one-way functions).

The last step of the chain, "PRG $\Rightarrow$ P = BPP", is now proved. Next we build the PRG from one function that is hard on average (Nisan-Wigderson). Then we show that a function hard only in the worst case is already enough (hardness amplification).

5 The Nisan–Wigderson generator

5.1 One hard function, many output bits

Let $f : \{0,1\}^n \to \{0,1\}$ be hard. The idea is to apply f to many different pieces of the seed. From a seed $x \in \{0,1\}^m$ we output

$$g(x) = f(x|_{S_1})\, f(x|_{S_2}) \cdots f(x|_{S_N}), \qquad S_i \subseteq [m],\ |S_i| = n,$$

where $x|_S \in \{0,1\}^{|S|}$ denotes the bits of x in the positions of S. How should we choose the sets S_i? The two extremes both fail.

- Disjoint sets. The output bits are independent and each one is a fresh evaluation of f, so they are as good as they can be. But then $m = nN$: the seed is longer than the output, and there is no stretch.
- Equal sets $S_i = S_j$. The seed is short, but the two output bits are identical, which a tiny circuit notices immediately.

The compromise is to take many sets, each of size n, inside a universe of size only $m = O(n)$, with small pairwise overlaps. A small overlap means that, once the bits in S_i are what we care about, the other output bits carry little information about them.

5.2 Combinatorial designs

Definition 15 (Design)**.** *A family $\mathcal{S}$ of subsets of $[m]$ is an (m, n, a)-design if $|S| = n$ for all $S \in \mathcal{S}$, and $|S_1 \cap S_2| \le a$ for all $S_1 \neq S_2$ in $\mathcal{S}$.*

Target: a $(cn, n, \delta n)$-design with $2^{\Omega(n)}$ sets, for constants c and δ.

A first attempt: designs from low-degree polynomials. Let $\mathbb{F}_q$ be a field and take the universe $[m] = \mathbb{F}_q \times \mathbb{F}_q$. For each polynomial p with $\deg p < d$ put

$$S_p = \{(z, p(z)) : z \in \mathbb{F}_q\},$$

the graph of p (Figure 4). Then:

- $|S_p| = q$;
- $|S_p \cap S_{p'}| \le d - 1$ for $p \neq p'$, because two distinct polynomials of degree $< d$ agree in at most $d - 1$ points (this is Schwartz–Zippel in one variable, Lemma 1, which we also used in Lecture 11);
- the number of sets is q^d.

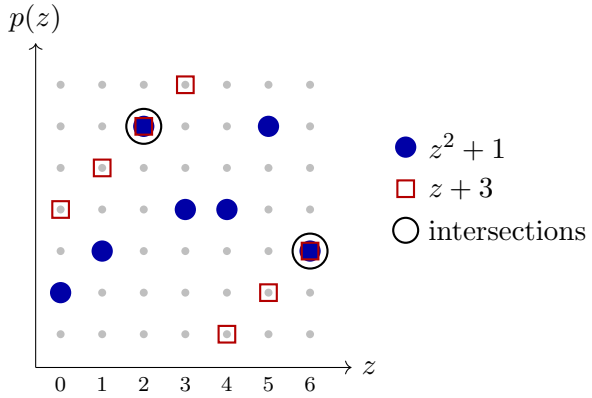


Figure 4: The polynomial design with $q = 7$, $d = 3$: the graphs of $z^2 + 1$ and $z + 3$ over $\mathbb{F}_7$ meet in exactly $2 = d - 1$ points ($z = 2$ and $z = 6$).

Which regime do we need? With $q = n$, the polynomial design has sets of size n but a universe of size $q^2 = n^2$, so the seed has length $m = n^2$ instead of $O(n)$. As we will see, $n = O(\log N)$, so this

gives a seed of length $O(\log^2 N)$ and hence derandomization in quasi-polynomial time $2^{O(\log^2 N)}$. That is already a nice result, but for $\mathsf{P} = \mathsf{BPP}$ we need a universe of linear size: $m = O(n)$, $|S_i| = n$, $a = \delta n$, and $2^{\Omega(n)}$ sets.

A design is a code. Split a universe of size qn into n columns of q cells each, and only use sets that pick exactly one cell per column. Such a set is the same thing as a word $w \in [q]^n$ (the j-th letter says which cell is picked in column j), and $|S \cap S'|$ is the number of positions where the two words agree. So

$$\text{intersections} \le \delta n \iff \text{relative distance} \ge 1 - \delta.$$

The polynomial design above is exactly this picture for the Reed-Solomon code. To get a linear-size universe we just need a different code, and Lecture 11 gives us one.

Lemma 16 (The design we need exists)**.** *For every $\delta > 0$ there are $c = O(1/\delta)$ and $\beta > 0$ such that a $(cn, n, \delta n)$-design with $2^{\beta n}$ sets exists, and can be constructed in time $2^{O(n)}$.*

Proof. Take $q = O(1/\delta)$ and a q-ary code of length n with relative distance at least $1 - \delta$; by the correspondence above, its codewords give the design, with universe size $cn = qn$. Such a code with $2^{\beta n}$ codewords exists by the Gilbert–Varshamov argument: two random words agree in n/q positions on average, so a random word agrees with a fixed one in more than δn positions only with probability $q^{-\Omega(n)}$ (Chernoff), and a union bound over the words chosen so far shows that we can keep adding new words. Done greedily (add words one at a time, scanning all $q^n = 2^{O(n)}$ candidates), this takes time $2^{O(n)}$. This is affordable: we will use it with $N = 2^{\Omega(n)}$, so $2^{O(n)} = \text{poly}(N)$. □

5.3 The generator and the theorem

Definition 17 (Nisan–Wigderson generator)**.** *Given $f : \{0,1\}^n \to \{0,1\}$ and an (m, n, a)-design $S_1, \ldots, S_N \subseteq [m]$, define $g_f : \{0,1\}^m \to \{0,1\}^N$ by*

$$g_f(x) = f(x|_{S_1})\, f(x|_{S_2}) \cdots f(x|_{S_N}).$$

With the design of Lemma 16: the seed has length $m = O(n) = O(\log N)$ and computing g_f takes N evaluations of f, i.e. time $N \cdot 2^{O(n)} = \text{poly}(N)$. A toy example is in Figure 5.

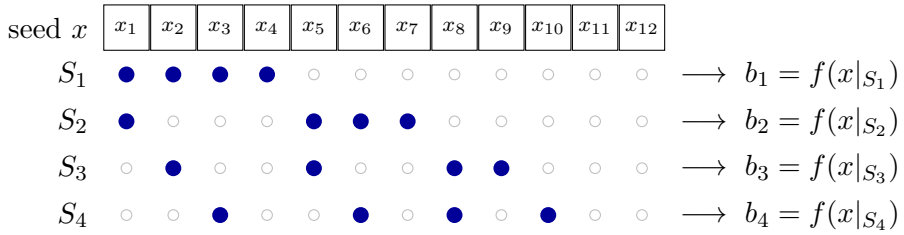


Figure 5: The Nisan-Wigderson generator on a toy example: a seed of $m = 12$ bits and four sets of size $n = 4$ that pairwise share at most $a = 1$ position. Each output bit applies the hard function f to the seed bits picked out by one set.

What kind of hardness do we need? Not worst-case hardness: if a circuit fails on only one input, then $f(x)$ on a random x is predictable almost always, and does not look random at all. We need a function that small circuits cannot even guess much better than a coin flip on a random input. (A constant circuit already gets $\frac{1}{2}$ on a balanced function, so $\frac{1}{2}$ is the natural baseline.)

Theorem 18 (Nisan–Wigderson [5])**.** *Let $f : \{0,1\}^n \to \{0,1\}$ be computable in time $2^{O(n)}$, and suppose that every circuit C of size $2^{\delta n}$ satisfies*

$$\mathop{\mathbb{P}}_{x\sim\{0,1\}^n}\left[C(x) = f(x)\right] \le \tfrac{1}{2} + \gamma, \qquad \gamma = 2^{-\delta n/3}.$$

Then, for a $(cn, n, \frac{\delta n}{2})$-design with $N = 2^{\delta n/4}$ sets, g_f is a $(1/10, N)$-PRG computable in time $\mathrm{poly}(N)$.

Combined with Lemma 12, such an f gives $\mathsf{P} = \mathsf{BPP}$.

Remark 19. The required advantage γ is exponentially small because the proof loses a factor N (see Step 1 below). If we only know $\gamma = 1/\mathrm{poly}(n)$, the same argument still works with a seed of length $O(\log^2 N)$, which gives quasi-polynomial derandomization.

6 Hardness amplification

6.1 Worst case is enough

Compare what we assume with what Nisan–Wigderson needs.

- Assumption 9: small circuits cannot compute f on every input. A circuit already counts as failing if it is wrong on a single input. This is **worst-case** hardness.
- Theorem 18: small circuits cannot compute f on a random input, not even with probability noticeably above $\frac{1}{2}$. This is average-case hardness, a much stronger requirement.

Closing this gap is called hardness amplification: given a function f that is hard in the worst case, we build a new function f' that is hard on a uniformly random input. It is easiest to think of the contrapositive: if we can compute f' on most random inputs, then we can compute f on every input.

Theorem 20 (Impagliazzo-Wigderson, amplification [3])**.** *If f is computable in time $2^{O(n)}$ and requires circuits of size $2^{\delta n}$, then there is f' on $n' = O(n)$ bits, computable in time $2^{O(n')}$, such that no circuit of size $2^{\delta' n'}$ agrees with f' on more than a $\frac{1}{2} + 2^{-\delta' n'}$ fraction of inputs.*

6.2 Permanent of a matrix

Definition 21 (Permanent of a matrix)**.** *The permanent of $A \in \mathbb{F}^{n\times n}$ is defined as the sum products of all permutations:*

$$\mathsf{Perm}(A) = \sum_{\pi\in S_n}\prod_{i=1}^{n} A_{i,\pi(i)} \tag{1}$$

Note that the definition is very similar to the determinant, except for the permanent the sign is discarded.

Computing this value is perceived as very hard (#P hard for the integers). We show that if an algorithm is able correctly compute Perm on the majority of the numbers, one can use it to recover $\mathsf{Perm}(\cdot)$ for all inputs. Therefore if the problem turns out to be hard on the majority of inputs (worst case hard), its hard on almost all inputs.

Theorem 22. *Suppose $f(A)$ is efficient and correctly computes Perm for $\frac{3}{4}+\epsilon$ of the matrices:*

$$\mathbb{P}\left[f(A) = \mathsf{Perm}(A) \mid A \sim \mathsf{Uniform}(\mathbb{F}^{n\times n})\right] \geq \frac{3}{4} + \epsilon \tag{2}$$

Then one is able to compute an efficient $\hat{f}^f(\cdot)$ with oracle access to f, which coinsides with Perm with very high probability.

Proof. $\hat{f}$ picks $B \in \mathbb{F}^{n\times n}$ randomly and fixes it. Let us define

$$h(\lambda) = \mathsf{Perm}(A + \lambda B) = \sum_{\pi\in S_n}\prod_{i=1}^{n}(A_{i,\pi(i)} + B_{i,\pi(i)}\lambda) \tag{3}$$

for $n = 1, \cdots, 2n$. Note that this yield an n-degree polynomial in λ. Let the $\#Err$ denote the number of λ values where $h(\lambda) \neq f(A+\lambda B)$. By the linearity of expectation, $\mathbb{E}(\#Err) = 2n(\frac{1}{4}-\epsilon)$. By the Markov inequality $\mathbb{P}(\#Err \geq n) \leq \frac{\mathbb{E}(\#Err)}{n} = \frac{1}{2} - 2\epsilon < \frac{1}{2}$. Therefore Reed-solomons error correction on f will successfully recover the the value of $h(0) = f(A)$ for more than half the cases. By repetiting the algorithm for different B, one can chose the majority value and obtain the correct answer with high probability.

□

Why did this work? Because the values of the permanent have a low-degree structure: restricted to any line, the permanent is a polynomial of degree at most n, so its values along the line form a Reed-Solomon codeword. An algorithm that is right on most random inputs therefore gives us a slightly corrupted codeword, and decoding removes the errors.

The same idea for any hard function. An arbitrary worst-case hard function $f : \{0,1\}^n \to \{0,1\}$ has no such structure, so we create it: we encode the truth table of f with a good error-correcting code and use the codeword as the new function. Concretely, take the low-degree extension of f (a Reed-Muller encoding): the unique polynomial $\hat{f} : \mathbb{F}^n \to \mathbb{F}$ of degree at most 1 in each variable such that $\hat{f}(b) = f(b)$ for every $b \in \{0,1\}^n$. Then:

- computing $\hat{f}$ on every input includes computing f on every input;
- $\hat{f}$ has total degree at most n, so on every line it is a polynomial of degree at most n, exactly like the permanent;
- $\hat{f}$ can still be computed in time $2^{O(n)}$.

So the random-line argument for the permanent works for $\hat{f}$ word for word. If a circuit C computes $\hat{f}$ correctly on a $\frac{3}{4}+\varepsilon$ fraction of inputs, then querying C on a random line and decoding computes $\hat{f}(x)$ for every x with high probability. A circuit cannot flip coins, but it does not need to: after enough repetitions, one fixed choice of the random bits works for all inputs at once (union bound), and the circuit can have it hard-wired. This gives a circuit only poly(n) times larger than C

that computes f on every input. Taking the contrapositive gives the main Impagliazzo-Wigderson reduction: if f needs circuits of size S on worst-case inputs, then no circuit of size $S/\mathrm{poly}(n)$ computes $\hat{f}$ correctly on more than a $\frac{3}{4}+\varepsilon$ fraction of random inputs.

Amplifying further. This is still far from what Nisan-Wigderson needs: a Boolean function that no small circuit computes on more than a $\frac{1}{2}+2^{-\Omega(n)}$ fraction of inputs. Closing this gap, while keeping the input length $O(n)$, uses much fancier tools from error-correcting codes. The result is the hardness amplification theorem, which we state without proof.

Theorem 23 (Hardness amplification [3])**.** *Suppose there is $f \in \mathsf{E}$ and $\delta > 0$ such that f_n requires circuits of size $2^{\delta n}$ (worst-case). Then there is $f' \in \mathsf{E}$ and $\delta' > 0$ such that for every n, every circuit of size $2^{\delta' n}$ computes f'_n correctly on at most a $\frac{1}{2}+2^{-\delta' n}$ fraction of inputs.*

Open problems. A deterministic polynomial-time algorithm for PIT. Any $2^{\Omega(n)}$ (or even super-linear) circuit lower bound for an explicit function.

LLM disclosure. LLMs were used to re-write our notes and the lecture slides together, to better understand the material, and to produce the basic draft of the document. The lecture notes were subsequently manually edited by the authors, and finally the model was used for proofreading and formatting.

References

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004. doi:10.4007/annals.2004.160.781.

[2] K. Iwama and H. Morizumi. An explicit lower bound of $5n - o(n)$ for Boolean circuits. In *Proc. MFCS*, LNCS 2420, pages 353–364, 2002. doi:10.1007/3-540-45687-2_29.

[3] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proc. 29th ACM STOC*, pages 220–229, 1997. doi:10.1145/258533.258590.

[4] H. W. Lenstra, Jr. and C. Pomerance. Primality testing with Gaussian periods. *Journal of the European Mathematical Society*, 21(4):1229–1269, 2019. doi:10.4171/JEMS/861.

[5] N. Nisan and A. Wigderson. Hardness vs randomness. *Journal of Computer and System Sciences*, 49(2):149–167, 1994. doi:10.1016/S0022-0000(05)80043-1.

[6] O. Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4), 2008. doi:10.1145/1391289.1391291.

[7] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980. doi:10.1145/322217.322225.

[8] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1):59–98, 1949. doi:10.1002/j.1538-7305.1949.tb03624.x.

[9] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977. doi:10.1137/0206006.

[10] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. EUROSAM*, LNCS 72, pages 216–226, 1979. doi:10.1007/3-540-09519-5_73.