1 Pseudorandomness

Pseudorandomness is the study of how randomized algorithms can be simulated deterministically. Every randomized algorithm admits a natural deterministic simulation which can be obtained by running the algorithm over all possible settings of the randomness and looking at the distribution of outputs. However such simulations are very inefficient; can we do anything better?

A priori, it might seem that the answer should be no. Just as we believe that the best deterministic simulation of NP should require one to look at all possible witnesses, it might seem that simulating a BPP algorithm should require one to look at all possible settings of the randomness. Somewhat surprisingly, we show that this is *not* the case — if we assume a very plausible conjecture about the existence of certain hard functions:

Theorem 1. *Suppose there is a decision problem g that can be decided in time $2^{O(n)}$, but cannot be decided by any circuit family of size $2^{o(n)}$ (on almost all input lenghts). Then* $\mathrm{P} = \mathrm{BPP}$.

We think of g as a "hard" function, like the XOR function is hard for AC^0 circuits. The theorem says that if a certain type of hard function exists, then every BPP computation can be performed deterministically. In fact, as we will see the proof of Theorem **??** gives something even stronger. It shows that there is a universal efficient deterministic simulation for all efficient randomized algorithms of the BPP type.

The proof of Theorem **??** splits in two parts, which we state as separate theorems below.

Theorem 2. *Suppose there is a decision problem g that can be decided in time $2^{O(n)}$ but cannot be decided by any circuit family of size $2^{o(n)}$. Then there is a decision problem f that can be decided in time $2^{O(n)}$ and such that for some $\delta > 0$ and every circuit family C of size $2^{\delta n}$ and sufficiently large n*

$$\Pr_{x \sim \{0,1\}^n}[C(x) = f(x)] < 1/2 + 2^{-\delta n}.$$

This theorem gives a way to turn a hard problem g into a very hard problem f. We assume that g cannot be computed by sufficiently small circuits, meaning that on every input length there is *some* input on which g is not computed correctly. Given such an g, we obtain a f such that no circuit can determine f on much more than half the inputs. In other words, if we think of the input x itself as random, then no small circuit can do much better at predicting the value $f(x)$ better than guessing at random.

Theorem 3. *Suppose there is a decision problem f that can be decided in time $2^{O(n)}$ and such that for every $\delta > 0$ and every circuit family C of size $2^{\delta n}$ and sufficiently large n*

$$\Pr_{x \sim \{0,1\}^n}[C(x) = f(x)] < 1/2 + 2^{-\delta n}.$$

Then $\mathrm{P} = \mathrm{BPP}$.

We now turn to proving Theorem **??**. So let's assume the existence of the function f with the desired properties.

2 Pseudorandom generators

The basic strategy of the proof is as follows. To prove that BPP = P, we show a way to simulate every BPP algorithm A deterministically in an efficient way. In this setting, it will be helpful of A not as a function of its actual input x, but as a function of its randomness r. The condition that A is a BPP algorithm for decision problem L tells us that

$$\begin{aligned} x \in L &\implies \Pr_{r\sim\{0,1\}^m}[A(x,r)=1] \geq 2/3 \\ x \notin L &\implies \Pr_{r\sim\{0,1\}^m}[A(x,r)=1] \leq 1/3. \end{aligned}$$

When the input is of length n, we assume that the algorithm uses $m = m(n)$ bits of randomness. Since A runs in polynomial time, m grows polynomially in n.

The idea is to replace the random string r used by A by a pseudorandom string $G(s)$ which uses much less randomness to generate, yet doesn't affect by much the success probability of A. If $G(s)$ can be generated efficiently using $k = O(\log m)$ bits of randomness, then we could simulate A on input x deterministically by enumerating all possible outputs of $G(s)$ and observing what fraction of the time $A(x, G(s))$ accepts. If A were to accept, a majority of the outputs $G(s)$ should yield accepting computations; if A were to reject, the majority of them should yield rejecting computations. That is, we want

$$\begin{aligned} x \in L &\implies \Pr_{s\sim\{0,1\}^k}[A(x,G(s))=1] > 1/2 \\ x \notin L &\implies \Pr_{s\sim\{0,1\}^k}[A(x,G(s))=1] < 1/2. \end{aligned}$$

To achieve this, it is sufficient that for all x,

$$\left|\Pr_{r\sim\{0,1\}^m} A(x,r)=1] - \Pr_{s\sim\{0,1\}^k}[A(x,G(s))=1]\right| < 1/6.$$

The next step is a bit unusual, but crucial: Instead of looking to construct a specific G that works for our algorithm A, we will give a generic G that satisfies the above condition with respect to all algorithms that run in fixed polynomial time $t(n)$. In fact, it will be even more convenient to think of $A(x,r)$ as a collection of circuits $C_x(r)$ – where the input x of A is hardwired into the circuit C_x – of size $S(n)$, where $S(n)$ grows polynomially in n. Then the above condition can be written as

$$\left|\Pr_{r\sim\{0,1\}^m} C_x(r)=1] - \Pr_{s\sim\{0,1\}^k}[C_x(G(s))=1]\right| < 1/6.$$

This motivates the following definition.

Definition 4. $G : \{0,1\}^k \to \{0,1\}^m$ is an *ϵ-pseudorandom generator* (in short, *ϵ-pseudorandom*) against size S circuits if for every circuit C of size S,

$$\left|\Pr_{r\sim\{0,1\}^m} C(r)=1] - \Pr_{s\sim\{0,1\}^k}[C(G(s))=1]\right| < 1/6.$$

So to simulate every BPP algorithm deterministically in an efficient way, it is sufficient to give an "efficient" construction of a family of pseudorandom generators G against circuit families of every polynomial size $S(n)$ with "short" seed length k. What does it mean for k to be short and G to be efficient? If we want to enumerate all 2^k outputs of G in time polynomial in m, we must have $k = \Theta(\log m)$, and G must run in time polynomial in m. Since $m = 2^{O(k)}$, this means that G runs in time $2^{O(k)}$ – exponential in the length of its input s.

To summarize, for every polynomial growing size bound $S(n)$ we want a family of 1/6-pseudorandom generators $G : \{0,1\}^k \to \{0,1\}^m$ against size $S(n)$, where $k = O(\log m)$ and G is computable in time $2^{O(k)}$. (Without loss of generality, we will assume that $m \geq n$.)

3 Turning hardness into pseudorandomness

We want to construct G just based on the assumption that there exists a certain "hard" function f. How do we turn a hard function into a pseudorandom generator? The key idea here is that since the function f is hard to compute – with very small advantage over random – for circuits of a given size, the output of f must "look random" to such circuits. Another way of saying this is that conditioned on s, $f(s)$ is a random-looking string for such circuits, or equivalently, the string $(s, f(s))$ behaves much like a random string of length $k+1$ for such circuits. So it seems that the function $G(s) = (s, f(s))$ should be a good pseudorandom generator. This is indeed the case.

Lemma 5. *Suppose that for every circuit C of size S,*

$$\Pr_{s \sim \{0,1\}^k}[C(s) = f(s)] < 1/2 + \epsilon.$$

Then the function $G : \{0,1\}^k \to \{0,1\}^{k+1}$ given by $G(s) = (s, f(s))$ is ϵ-pseudorandom against size $S - O(1)$.

The proof of this lemma is not technically difficult, but it contains an interesting idea.

Proof. We argue by contradiction – we assume that G is not ϵ-pseudorandom against size S'; that is, there is a circuit C of size S' such that

$$\left|\Pr[C(s, f(s)) = 1] - \Pr[C(s, b) = 1]\right| \geq \epsilon.$$

Here, s is chosen randomly from $\{0,1\}^k$ and b is a random bit independent of s. We want to use C to construct another circuit C'' of size $S' + O(1)$ that computes f with advantage ϵ over random.

The first step is to get rid of the absolute value. By taking either $C'(s) = C(s)$ or $C'(s) = \overline{C(s)}$, we have that

$$\Pr[C'(s, f(s)) = 1] - \Pr[C'(s, b) = 1] \geq \epsilon.$$

where C' has the same size as C (recall that NOT gates don't count towards circuit size).

This equation can be interpreted in the following way: If we think of $C'(s, b)$ in terms of the input b, then C' is more likely to accept when $b = f(s)$ than when b is completely random. So it seems that the output of $C'(s, b)$ when b is random should make a good predictor for the value $f(s)$: If $C'(s, b) = 1$, then b is more likely to be $f(s)$, and if $C'(s, b) = 0$, then b is more likely to be $\overline{f(s)}$. This suggests the following randomized procedure C'' for predicting f:

C'' : On input s,
Choose $b \sim \{0,1\}$ at random
If $C'(s,b) = 1$, output b
If $C'(s,b) = 0$, output $\overline{b}$.

We will show that

$$\Pr_{s,b}[C''(s,b) = f(s)] \geq 1/2 + \epsilon \tag{1}$$

so in particular there exists a choice of b for which

$$\Pr_{s}[C''(s,b) = f(s)] \geq 1/2 + \epsilon.$$

If we fix this choice of b into the circuit C'', we obtain a deterministic circuit C'' of size $S' + O(1)$ that contradicts the assumption of the lemma.

It remains to prove (**??**). For this, it is sufficient to show that for every fixed $s \in \{0,1\}^k$:

$$\Pr_{b}[C''(s,b) = f(s)] \geq 1/2 + \Pr[C'(s,f(s)) = 1] - \Pr_{b}[C'(s,b) = 1]. \tag{2}$$

This can be checked by a somewhat tedious case analysis. We fix s and look $C'(s,b)$ as a function of b. There are four possibilities for what this function can be: 0, 1, b or $\overline{b}$.

$C'(s,b)$	$\Pr[C'(s,f(s)) = 1]$	$\Pr_b[C'(s,b) = 1]$	$\Pr_b[C''(s) = f(s)]$
0	0	0	1/2
1	1	1	1/2
b	$f(s)$	1/2	$f(s)$
$\overline{b}$	$\overline{f(s)}$	1/2	$\overline{f(s)}$

We can check that the condition (**??**) holds in all four cases. □

The pseudorandom generator of Lemma **??** falls short on one crucial requirement: It saves only one bit of randomness, namely $k = m - 1$. To achieve $k = O(\log m)$, we need more work.

4 The Nisan-Wigderson generator

One way to improve the construction of Lemma **??** is to use several copies of the generator: Partition the input s into several disjoint substrings and run different copies of the generator on each one of these strings. This idea can save us more bits of randomness, but still falls very short of what we aim to achieve. However, if we allow the substrings of s to have some intersection instead of making them disjoint, then the output of the resulting generator could be potentially much longer than its input.

The problem is that as the substrings of s begin to intersect, the outputs corresponding to them are no longer independent. However if we keep the intersection sizes small, we could hope that

there is enough independence left among the outputs so that it remains pseudorandom. This is the idea behind the Nisan-Wigderson generator, which we describe next. First we need to quantify what we mean by keeping the intersection sizes small.

Definition 6. A collection of sets $T_1, \ldots, T_m \subseteq \{1, \ldots, k\}$ is a *combinatorial design* with set size t and intersection size $t_\cap$ if $|T_i| = t$ for every i and $|T_i \cap T_j| \leq t_\cap$ for every $i \neq j$.

Given a combinatorial design, we define a pseudorandom generator $G : \{0,1\}^k \to \{0,1\}^m$ by

$$G(s) = (f(s|_{T_1}), \ldots, f(s|_{T_m}))$$

where $f : \{0,1\}^t \to \{0,1\}$ is the "hard" function and s_T is the substring of s indexed by the elements of the set T – for instance if $s = s_1 s_2 s_3 s_4$, then $s|_{2,4} = s_2 s_4$. It turns out that combinatorial designs with good parameters can be computed efficiently.

Claim 7. *For every constant $D > 0$ there is a family of combinatorial designs with*

$$k = O(D^2 \log m) \qquad t = D \log m \qquad t_\cap = \log m.$$

Moreover, this family can be constructed deterministically in time $m^{O(D^2)}$.

In particular, for every fixed D we have $k = O(\log m)$, so the seed size is exactly what we were aiming for. Let's now check that G can be computed efficiently (in time $\text{poly}(m)$). To compute G, we first construct the design in time $m^{O(D^2)}$. We then need to evaluate m copies of f, each on an input of size $t = D \log m$. Since we assumed that f is computable in time $2^{O(t)} = m^{O(D^2)}$, the whole computation can be done in time polynomial in m.

It remains to show that G is $1/6$-pseudorandom against circuits of every polynomial size $S(n)$. (Looking ahead, the choice of D will depend on the polynomial bound $S(n)$.) We argue by contradiction: If G is not $1/6$-pseudorandom, then for some circuit C of size $S(n)$ we have

$$\Pr_{s \sim \{0,1\}^k}[C(G(s)) = 1] - \Pr_{r \sim \{0,1\}^m}[C(r) = 1] > 1/6.$$

(As before, we can remove the absolute value in the definition of pseudorandom without loss of generality.) Let's expand this definition:

$$\Pr_{s \sim \{0,1\}^k}[C(f(s|_{T_1}), \ldots, f(s|_{T_m})) = 1] - \Pr_{r_1, \ldots, r_m \sim \{0,1\}}[C(r_1, \ldots, r_m) = 1] > 1/6. \tag{3}$$

This formula does not appear all that useful. To see what is happening, we introduce of the following way of "slowly" going from the pseudorandom distribution $G(s)$ to the random distribution r: At each step, we change one input of C from pseudorandom to random. If C can distinguish $G(s)$ from r, then at some step there must be a noticeable change in the behavior of C.

More formally, we consider the following sequence of *hybrid distributions* on inputs of C:

$$\begin{array}{rcccc}
D_m: & f(s|_{T_1}), & \ldots, & f(s|_{T_{m-1}}), & f(s|_{T_m}) \\
D_{m-1}: & f(s|_{T_1}), & \ldots, & f(s|_{T_{m-1}}), & r_m \\
& \vdots & & \vdots & \vdots \\
D_0: & r_1, & \ldots, & r_{m-1}, & r_m.
\end{array}$$

These distributions are not "real"; we merely use them to help us in the analysis. Condition (**??**) tells us that $\Pr_{r\sim D_m}[C(r)] - \Pr_{r\sim D_0}[C(r)] > 1/6$. Then there must be some j between 1 and m for which $\Pr_{r\sim D_j}[C(r)] - \Pr_{r\sim D_{j-1}}[C(r)] > 1/6m$, that is

$$\Pr_{s\sim\{0,1\}^k, r_{j+1},\ldots,r_m\sim\{0,1\}}[C(f(s|_{T_1}),\ldots,f(s|_{T_j}),\ldots,r_m)=1] \\ - \Pr_{s\sim\{0,1\}^k, r_j,\ldots,r_m\sim\{0,1\}}[C(f(s|_{T_1}),\ldots,r_j,\ldots,r_m)=1] > 1/6m.$$

There must then exist a fixing of the values $r_{j+1},\ldots,r_m$ that maximizes the above difference in probabilities. If we hardwire this fixing into the circuit C, we obtain a circuit C_1 of the same size such that

$$\Pr_s[C_1(f(s|_{T_1}),\ldots,f(s|_{T_{j-1}}),f(s|_{T_j}))=1] - \Pr_{s,r_j}[C_1(f(s|_{T_1}),\ldots,f(s|_{T_{j-1}}),r_j)=1] > 1/6m.$$

Let $s' = s|_{T_j}$ (this is a string of length t). There is now a fixing of all the bits of s outside s' that maximizes the above difference in probabilities. Let's hardwire these bits into C_1 and call the resulting circuit C_2. With respect to this fixing, for every $i < j$, $f(s|_{T_i})$ becomes a function of at most $\log m$ bits in s' (because s' intersects $s|_{T_i}$ in at most $t_\cap = \log m$ positions). Let's call this function $g_i(s')$. We then have

$$\Pr_{s'}[C_2(g_1(s'),\ldots,g_{j-1}(s'),f(s'))=1] - \Pr_{s',r_j}[C_2(g_1(s'),\ldots,g_{j-1}(s'),r_j)=1] > 1/6m.$$

Since each g_i is a function of at most $\log m$ bits, it can be computed by a circuit of size $O(2^{\log m}) = O(m)$. If we compose the circuit C_2 with the circuits for $g_1,\ldots,g_{j-1}$, we obtain a single circuit C_3 of size $S(n) + O(jm) = S(n) + O(m^2)$ such that

$$\Pr_{s'}[C_3(s',f(s'))=1] - \Pr_{s',r_j}[C_3(s',r_j)=1] > 1/6m.$$

In words, $(s', f(s'))$ is not $1/6m$-pseudorandom for size $S(n)+O(m^2)$; by Lemma **??** it follows that there is a circuit C_4 of size $S(n) + O(m^2)$ such that

$$\Pr_{s'}[C_4(s') = f(s')] > 1/2 + 1/6m.$$

Recall that f is a function on $t = D\log m$ bits, so we have that

$$\Pr_{s'}[C_4(s') = f(s')] > 1/2 + 1/6\cdot 2^{-t/D}$$

where C_4 is a circuit of size $S(n)+O(m^2) \le 2^{ct/D}$ for some constant $c > 0$. If we choose $\delta = c/D$ we have that C_4 is a circuit of size $2^{\delta t}$ that predicts f with advantage $2^{-\delta t}$, contradicting the assumed hardness of f.

5 Randomness and space

We now have some understanding of how randomness (as an external resource) affects time-efficient computation. Randomness can help simplify the design of efficient algorithms, but (under reasonable assumptions) it can ultimately be eliminated without significantly affecting the efficiency of the algorithm.[1] We now turn to investigate how randomness affects space-efficient computation.

[1] Actually, we only proved this is the case for decision problems.

We first need to define our computational model. A *randomized space-bounded* Turing Machine has three tapes: A read-only input tape, a work tape, and a one-way *randomness tape* (where the head always moves left to right). The Turing Machine runs in logarithmic space if the number of cells used on the work tape is logarithmic in the input length for each setting of the randomness tape. Just like for nondeterministic space-bounded computation, the head of the randomness tape cannot move back, since we do not want to allow the Turing Machine to remember its previous random choices for free.

Just like for randomized time-bounded computations, we can consider several variants of randomized logarithmic space computation, depending on whether the computation is typically correct, or typically logarithmic space, and if the error is one-sided or two-sided. For simplicity we will focus our attention on the following kinds of problems:

Definition 8. The class RL consists of those decision problems L for which the exists a randomized logarithmic space and polynomial time Turing Machine M such that if $x \in L$, then $\Pr[M(x) = 1] \geq 1/2$ and if $x \notin L$, then $M(x) = 0$.

Note that we explicitly require that the Turing Machine runs in polynomial time (for all settings of the randomness tape). When we introduced deterministic logarithmic space, there was no need to bound the running time of the Turing Machine – we obtained polynomial running time "for free". In contrast, a randomized logarithmic space Turing Machine could be running in exponential time with high probability. Moreover, such a Turing Machine could be used (fairly trivially) to solve directed graph connectivity, and therefore all problems in NL. This indicates that the polynomial running time requirement is natural.

It follows from the definition that RL = NL. The most natural problem known to be in the class RL concerns the connectivity of undirected graphs:

UPATH: Given an undirected graph G and two vertices s, t in G, is there a path from s to t?

This is the undirected graph version of the problem PATH, which we showed to be complete for NL. In fact, the following simple randomized algorithm solves UPATH:

Algorithm Random Walk
On input (G, s, s'), where G is an undirected graph on n vertices:
    Put a particle at position s.
    Repeat the following for $4n^3$ steps:
        If the particle is at position s', accept.
        Otherwise, move the particle to a random neighbor.
    Reject.

This algorithm performs a *random walk* on G: Starting from s, the particle keeps moving to a random neighbor, with the hope of eventually reaching s'. Clearly, if s is not connected to s', the algorithm always rejects. We will argue that if s is connected to s', then with probability at least $1/2$, s' will be visited by the particle.

The random walk algorithm does not work for directed graphs, even when repeated for any polynomial number of steps. Can you see why?

We now give an analysis of the random walk algorithm. First we will give a heuristic (non-rigorous) calculation that explains why the algorithm works. We will then give a precise, but more involved, analysis which (although it gives a weaker bound of $O(n^4)$ for the number of steps) will be useful later on, when we talk about derandomization of space-bounded algorithms.

First let us make some simplifying assumptions of G: We will assume that G is not bipartite and is$d$-regular, i.e. every vertex has the same degree d. We allow G to have loops, and in fact it will be useful later on to assume that G has a loop around every vertex. (These assumptions are not really necessary for the first analysis, although they make things simpler.) We can enforce the assumptions as follows: To make the graph d-regular (in fact, 3-regular), short-cut all vertices of degree 1 and 2, and replace every vertex v of degree $d_v > 3$ by a cycle of length d_v, where each vertex in the cycle will be connected to exactly one of the previously incoming edges. To make G non-bipartite, add a loop around every vertex. After all these transformations, which can be carried out in logarithmic space, we obtain a graph that is 4-regular and non-bipartite, and each vertex in the original graph is represented by some vertex in the new graph in a way that preserves connectivity.

In the following analysis we will assume, without loss of generality, that G is connected.

6 Intuition about random walks

Assume we put the particle at verte s and perform a random walk for t steps. For every t, this experiment induces a probability distribution on the vertices of G. What does this distribution look like as t gets large? That is, after we have walked for a long time, where do we expect to land? It seems reasonable that we should be at a random vertex of the graph — that is, be at every vertex v with probability $1/n$, and this is indeed the case. We sketch an informal argument.

First, it is easy to see that if at any point in time we reach a uniformly random vertex, then after one step of the walk we are again at a uniformly random vertex. But is it not possible that starting from some vertex s, after many steps we reach not a uniform vertex but a vertex following some other (non-uniform) distribution? Consider the following two scenarios: In scenario 1, our particle (let's call it particle 1) starts at s, and in scenario 2 our particle (let's call it particle 2) starts at a uniformly random vertex. Now we claim that if we observe the position of the particle at some very large time t, the two scenarios look exactly alike! This is because at some point in time before t, particle 1 and particle 2 must surely have landed at the same vertex — as t tends to infinity, the probability of this event tends to one[2] — after which they become indistinguishable. Since the position of particle 2 is uniformly random at every step, the position of particle 1 must also eventually become uniformly random.

There is a way to make this argument precise, but instead of doing so now (we'll prove this in a different way later) let's go back and try to understand why the random walk algorithm works. Let (s, u) be any edge in G. What is the expected number of steps that it takes for a random walk that starts at s to reach u? It can do so in one step, but only with probability $1/d$ (with the rest of the probability, it will get sidetracked at some other vertex).

[2]This is the Borel-Cantelli lemma in probability: Informally, if an event *can* ever happen, it *will* eventually happen.

However, we will give a heuristic argument that the expected number of steps to go from s to u is at most dn. To explain why, let's take a random walk, starting from a random vertex, for some very large number of steps t. This walk (except for the very first and last parts) can be divided into sections, where each section starts at s, reaches u exactly once, and goes back to s. Let r be the number of such sections and p_ℓ be the probability that such a section is ℓ steps long. Since all of the sections are independent of one another, if t is very large, we expect that the number of sections that are ℓ steps long will occur about $p_\ell r$ times.[3] Since such a section contains ℓ edges, we get that $\sum_{\ell=1}^{\infty} \ell p_\ell r$ should approach t as t gets large. So the expected length of each section should be about t/r. On the other hand, when t is large, we expect that the edge (u, v) will occur about t/dn times (since each edge in the walk is random). Since the edge (u, v) can occur at most once in every section, we get that the expected number of sections can be at most dn.

Now take any simple path $s = v_0, v_1, \ldots, v_k = s'$. Let us require that the random walk from s to t goes thru vertices v_0, v_1, up to v_k in that order. Then the expected length of this walk equals the sum of the expected lengths of each stage (v_{i-1}, v_i), which we argued is at most dn. So the expected length of the whole path is at most $dn^2 \leq 2n^3$. By Markov's inequality, the walk will reach s' in at most $4n^3$ steps with probability at least $1/2$.

[3] To make this statement precise we would need to show that the random walk is *ergodic*; we won't do so here.