Pseudorandomness: Applications to Cryptography and Derandomization

March 15, 2019

1 The Desire for Pseudorandom Generators

1.1 Motivation for Cryptography

Consider the following canonical example in cryptography. Suppose Alice would like to securely send a message m to Bob. Crucially, she does not want that message to be read by an eavesdropper named Eve. Alice could do this by first sharing a key k with Bob and then sending the ciphertext $c = m \oplus k$. Here, the $\oplus$ symbol stands for XOR. Bob could decrypt the ciphertext by taking the XOR of c with k:

$$c \oplus k = (m \oplus k) \oplus k = m$$

This encryption/decryption scheme is called a one-time pad. Note two things about this construction. First, the key k must be as long as the message. Second, the key must be randomly chosen. Suppose, the key were not random and instead was the binary encoding of some Billboard Top 100 song. Then, Eve—if she knew the key was a Billboard Top 100 song—could try decrypting with all 100 possible keys and see which output made most sense.

One-time pads have limited effectiveness in practice, however, because of two problems. First, generating a truly random key as long as the message is oftentimes infeasible.[1] Second, the security of this procedure depends on Alice and Bob being able to securely communicate the key to each other; but doing that is just as hard as being able to securely communicate the message. We can avoid this problem by using pseudorandom generators. Alice and Bob could share a short seed s with each other and then apply a PRG G to get a much longer, seemingly random key equal to $G(s)$. In section 2, we will explore how to construct pseudorandom generators for cryptography using one-way functions.

[1] As discussed in class, it takes \$60 to buy a million random digits.

1.2 Motivation for Derandomization

Randomized algorithms can be extremely powerful, often besting their deterministic counterparts in simplicity, efficiency, or both. Because of that power, randomized algorithms are widely used. Unfortunately, their correctness depends on the ability of the algorithm to access truly random bits, which computers don't have access to. This begs the question: Is randomness necessary to efficiently solve problems?

As a first cut, we observe that all languages in **BPP** are also in $\mathbf{EXP} = \bigcup_c \mathbf{DTIME}(2^{n^c})$. A language with randomized algorithm A that runs in time T can only access at most T random bits. To simulate A on a deterministic machine, we run A on all 2^T possible random strings and count how often A returns 1. Our deterministic algorithm then accepts if A accepts on more than half of the random strings and rejects otherwise. Essentially, on input x, this deterministic algorithm exactly computes $\mathbf{Pr}[A(x) = 1]$, which based on the definition of **BPP** is more than $\frac{1}{2}$ iff x is in the language.

Unfortunately, enumerating over all possible random strings of length T is too slow to be of practical use. That's where pseudorandom generators become useful. We hope to enumerate over all random seeds of length only $O(\log T)$ and use a PRG to extend those seeds so that they appear like length T random strings. Since there are only $\text{poly}(T)$ random strings of length $O(\log T)$, enumerating over all of them can be done in polynomial time. The hard part is finding a PRG that takes in seeds of length $O(\log T)$ and outputs strings of length T that appear so random they successfully derandomize all algorithms in **BPP**. In section 3, we'll explore how to create such PRGs contingent on the (currently unknown) existence of sufficiently hard functions.

2 Cryptography

2.1 Cryptographic Pseudorandom Generators

The goal of the crytography portion of this paper is to provide a construction for pseudorandom generators that can be used for encrypting messages. In this section, we provide a series of definitions that will be helpful moving forward. First, we define a negligible function:

Definition (Negligible Function)**.** *A function $\epsilon : \mathbb{N} \to [0, 1]$ is considered negligible if for every c and sufficiently large n,*

$$\epsilon(n) < n^{-c}$$

In other words, $\epsilon(n)$ very quickly goes to 0 and can be safely ignored for most purposes.

Given the definition of a negligible function, we are now equipped to define cryptographic pseudorandom generators:

Definition (Cryptographic Pseudorandom Generator)**.** *Let G be a polynomial-time function, and $\ell : \mathbb{N} \to \mathbb{N}$ satisfy $\ell(n) > n$ for all n. G is a cryptographic pseudorandom generator with stretch $\ell(n)$ if $|G(x)|= \ell(|x|)$ and if for every probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function ϵ such that*

$$\left|\boldsymbol{Pr}[\mathcal{A}(G(U_n)) = 1] - \boldsymbol{Pr}\left[\mathcal{A}\left(U_{\ell(n)}\right) = 1\right]\right| < \epsilon(n)$$

for all $n \in \mathbb{N}$. In the above expression, U_n denotes a random string in $\{0,1\}^n$.

This definition intuitively captures what we would desire from a pseudorandom generator. Recall that one-time pads encrypt messages by XOR'ing them with a random key shared between Alice and Bob. Unfortunately, while one-time pads provide perfect security, they are impractical because they require the key to be as long as the message. Crytographic PRGs avoid this problem, while still maintaining a one-time pad's benefits.

With a PRG G, Alice and Bob can share a small seed s with each other, apply a polynomial stretch PRG, and use the result as their shared key. For example, s could be a random 2048-bit seed that G blows up into a billion bit key. Even though $G(s)$ is pseudorandom—and not truly random—no efficient adversary can tell that $G(s)$ is not truly random, so the security properties of a one-time pad still apply.

2.2 One-Way Functions

From time immemorial, humanity has gotten frequent, often cruel, reminders that many things are easier to do than to reverse. — Leonid Levin

Creating a cryptographic pseudorandom generator is not a trivial task. A central theme of this paper is that constructing such generators requires the use of "hard" functions (with varying notions of hardness). In the case of cryptographic PRGs, we will make use of one-way functions: functions which are easy to compute, but which are hard to invert.

Definition (One-Way Function)**.** *A polynomial-time function $f : \{0,1\}^* \to \{0,1\}^*$ is a* one-way function *if for every probablistic polynomial-time algorithm $\mathcal{A}$ there is a negligible function $\epsilon(n)$ such that for every n*

$$\underset{\substack{x\in\{0,1\}^n \\ y=f(x)}}{\boldsymbol{Pr}}[\mathcal{A}(y) = x' s.t. f(x') = y] < \epsilon(n).$$

In particular, the following are two examples of one-way functions used in modern cryptography:

- Cryptographically secure hash functions (e.g. SHA512): Unlike regular hash functions, these are infeasible to invert. A cryptographic hash of a message is commonly appended to the message as a message authentication code (MAC) to guarantee the receiver that the message has not been tampered with after it left the sender's hands.

- Multiplication of large primes: A very common protocol for achieving public-key encryption (a type of encryption that allows two parties to communicate without sharing a key) is RSA. The RSA protocol is considered hard to breach largely because we believe it is hard to factor the product of two large primes.

One-way functions are definitionally hard to invert, but their connection to notions of computational hardness runs deeper than that, as the following theorem shows:

Theorem 1. *If $\boldsymbol{P} = \boldsymbol{NP}$, then one-way functions cannot exist.*

Proof. Suppose for the sake of contradiction that there existed a one-way function $f : \{0,1\}^* \to \{0,1\}^*$ despite $\mathbf{P}$ equaling $\mathbf{NP}$. We will construct a deterministic, polynomial-time algorithm $\mathcal{A}$ that inverts f without error.

First, consider the language

$$L_f = \{(1^n, x', y) \mid \exists x \text{ s.t. } |x| = n, x' \text{ is a prefix of } x, \text{ and } f(x) = y\}.$$

L_f is in $\mathbf{NP}$, as x is a polynomial-length certificate.[2] The verifier checks the following: (1) $|x| = n$, (2) x' is a prefix of x, and (3) $f(x) = y$. Since $\mathbf{P} = \mathbf{NP}$, there exists a polytime algorithm $\mathcal{D}$ that decides L_f.

We can use $\mathcal{D}$ to construct another algorithm $\mathcal{A}$ that inverts f. $\mathcal{A}$, when given n and y, starts with the tuple $(1^n, \epsilon, y)$ (i.e. with the empty word as a prefix) and iteratively adds bits to x', checking whether the added bit was correct using $\mathcal{D}$. This process continues until $\mathcal{A}$ finds an x' such that $f(x') = y$. At most $2n = 2|x|$ calls to $\mathcal{D}$ are made, so $\mathcal{A}$ runs in polytime. Moreover, since $\mathcal{A}$ inverts f with probability 1 (which is greater than a negligible function), f must not be a one-way function, which contradicts our assumption. $\square$

As an aside, one-way functions are heavily used in cryptography for purposes other than constructing PRGs. As such, if $\mathbf{P}$ is found to equal $\mathbf{NP}$, then a significant amount of modern cryptography would be theoretically insecure. However, this does not imply practical insecurity: the existence of polynomial-time algorithms to break one-way functions does not necessitate that such algorithms are efficient enough to provide adversaries significantly greater efficiency compared to the *status quo*. For example, such algorithms might take time $O(n^{100})$, making them too slow on inputs of length more than a few bits. However, it would still shatter confidence in modern cryptosystems and necessitate finding an alternative set of tools.

[2] x is polynomial in the length of the input since L_f must be passed a unary encoding of $|x|$. If 1^n was not provided, then there is no guarantee that x would be polynomial in the length of x' and y.

2.3 Pseudorandom Generators from One-Way Functions

2.3.1 Yao's Theorem

In a previous section, we provided a formal definition of cryptographic PRGs. However, we now provide an alternative characterization of PRGs based on their unpredictability that will simplify their analysis.

Definition (Unpredictable). *Let $G : \{0,1\}^* \to \{0,1\}^*$ be a generator with stretch $\ell(n)$. G is unpredictable if for every probabilistic polynomial-time $\mathcal{B}$, there is a negligible function ϵ such that*

$$\mathop{\boldsymbol{Pr}}_{\substack{x\in\{0,1\}^n\\ y=G(x)\\ i\in\{1,\ldots,\ell(n)\}}} [\mathcal{B}(1^n, y_1, \ldots, y_{i-1}) = y_1] \leq \frac{1}{2} + \epsilon(n)$$

The randomness in the above expression is over both x and the index i. The 1^n is only there so that $\mathcal{B}$ knows the length of the input x. Essentially, the above definition claims that G is unpredictable if every polynomial-time algorithm struggles to predict the i^{th} bit from the first $i-1$ bits.

It is not hard to see that an pseudorandom generator is also an unpredictable generator. If the output $y_i, \ldots, y_{\ell(n)}$ is generated by randomly sampling bits (or at least appears to be to all adversaries), then it must be impossible for an adversary to predict y_i from the previous $i-1$ bits. Surprisingly, the converse is also true: an unpredictable generator is a pseudorandom generator.

Theorem 2 (Yao's Theorem). *If G is unpredictable, then it is a pseudorandom generator.*

Proof. We will prove the contrapositive of the statement: if G is not a pseudorandom generator, then G is predictable. If G is not pseudorandom, then there must be some algorithm $\mathcal{A}$ that can distinguish between outputs of G and a uniformally sampled string with non-negligible probability $\delta(n)$:

$$\Pr\left[\mathcal{A}(G(U_n)) = 1\right] - \Pr\left[\mathcal{A}(U_{\ell(n)}) = 1\right] \geq \delta(n)$$

That is, $\mathcal{A}$ is more likely to return 1 on outputs of the generator than on uniform strings.[3] We construct another algorithm $\mathcal{B}$ that leverages $\mathcal{A}$ to determine the i^{th} bit from the first $i-1$ bits. $\mathcal{B}$ operates as follows:

- Given $y_1, \ldots, y_{i-1}$, create a new string $h = y_1, \ldots, y_{i-1}, z_i, \ldots, z_{\ell(n)}$ where the z bits are drawn i.i.d. from a uniform distribution
- Compute $a = \mathcal{A}(h)$.

[3]The astute reader will notice that in the above expression we dropped the absolute values that were present in the original definition of a PRG. This is not a big deal, because if $\mathcal{A}$ were to be more likely to return 0 on outputs of the generator, we could simply have $\mathcal{A}$ flip its output.

- If $a = 1$, then $\mathcal{B}$ guesses that $y_i = z_i$. Otherwise, if $a = 0$, $\mathcal{B}$ guesses $y_i = 1 - z_i$.

The above construction is motivated by the hope that if $\mathcal{A}$ has some advantage in distinguishing pseudorandom outputs and truly random outputs, then maybe it has some chance of knowing what the i^{th} bit in the pseudorandom output should be.

Now, we analyze why $\mathcal{B}$ breaks the unpredictability of our generator. Note that $\mathcal{B}$ correctly guesses y_i in two scenarios: (1) if $a = 1$ and $z_i = y_i$ or (2) if $a = 0$ and $z_i = 1 - y_i$. The probability that one of these two scenarios occurs is

$$\mathbf{Pr}(z_i = y_i)\mathbf{Pr}(a_i = 1 \mid z_i = y_i) + \mathbf{Pr}(z_i = 1 - y_i)\mathbf{Pr}(a_i = 0 \mid z_i = 1 - y_i).$$

Since z_i is randomly chosen, the probability that it equals y_1 is $1/2$. Using that and simplifying, we get that $\mathcal{B}$ succeeds with probability

$$1/2 + 1/2\left(\mathbf{Pr}(a = 1 \mid z_i = y_i) - \mathbf{Pr}(a = 1 \mid z_i = 1 - y_i)\right)$$

To prove that G is predictable then, we need to show that $\mathbf{Pr}(a = 1 \mid z_i = y_i) - \mathbf{Pr}(a = 1 \mid z_i = 1 - y_i)$ is larger than a negligible function. To show that, we will have to introduce a techinique called the *hybrid argument*.

Consider the following $\ell(n) + 1$ distributions: $D_0, \ldots, D_{\ell(n)}$. The distribution D_i is sampled by choosing x randomly from $\{0,1\}^n$, computing $y = G(x)$, and then outputting $y_1, \ldots, y_i, z_{i+1}, \ldots, z_{\ell(n)}$, where each z_j is an independent random bit. The name "hybrid argument" comes from the fact that these distributions D are a hybrid between the output y and the random bits z. Note that $D_0 = U_{\ell(n)}$ and $D_{\ell(n)} = U(G(n))$. Moreover, define p_i as $P(\mathcal{A}(D_i) = 1)$. We know by assumption that $p_{\ell(n)} - p_0 \geq \delta(n)$. Expanding the above equation into a sum of differences we get that

$$\begin{aligned}(p_{\ell(n)} - p_{\ell(n-1)}) + (p_{\ell(n)-1} - p_{\ell(n)-2}) + \ldots + (p_1 - p_0) &\geq \delta(n)\\ \mathop{\mathbf{E}}_{i \in \{1,\ldots,\ell(n)\}}[p_i - p_{i-1}] &\geq \frac{\delta(n)}{\ell(n)}\end{aligned}$$

How does this relate to what we had done earlier? Note that $\mathbf{Pr}(a = 1 \mid z_i = y_i)$ is exactly equal to p_i because the first i bits passed to $\mathcal{A}$ are from i and the remainder are from z. Similarly, $\mathbf{Pr}(a = 1 \mid z_i = 1 - y_i)$ is equal to p_{i-1}. Thus, we have shown that $\mathcal{B}$ succeeds with probability

$$\geq 1/2 + 1/2(p_i - p_{i-1})$$

On expectation (since G must be unpredictable for a random i), this is greater than

$$\begin{aligned}&\geq 1/2 + 1/2(\delta(n)/\ell(n)),\\ &> 1/2 + \text{negligible function}\end{aligned}$$

This implies that G is predictable, as required. $\square$

2.3.2 Goldreich-Levin Theorem: stretching by one bit

We now have done enough groundwork to present a pseudorandom generator that extends the input by one bit—a nontrivial feat.

Theorem 3 (Goldreich-Levin Theorem)**.** *Suppose that $f : \{0,1\}^* \to \{0,1\}^*$ is a one-way permutation (i.e. a one-way function that is also one-to-one). Then, for every probablistic polynomial-time algorithm $\mathcal{A}$, there is a negligible function ϵ such that*

$$\Pr_{x,r\in\{0,1\}^n}[\mathcal{A}(f(x),r) = x \odot r] \leq 1/2 + \epsilon(n)$$

where $x \odot r = \sum\limits_{i=1}^{n} x_i r_i \pmod 2$.

The interested reader should refer to Arora-Barak for a proof of the above theorem. Here, we will only provide some intuition. For the adversary Eve to determine $x \odot r$, she would like to know the value of x (she already knows the value of r). However, it is hard for Eve to determine x from $f(x)$ precisely because f is a one-way function.

Theorem 3 immediately implies that

$$G(x,r) = f(x), r, x \odot r \qquad \text{(commas imply concatenation)}$$

is a pseudorandom generator that stretches its input by 1 bit. Yao's Theorem will provide us an easy technique to verify this: we simply need to show that G is unpredictable. Note that x and r are randomly chosen, so no adversary would be able to guess the bits of $f(x)$ or r from the preceding bits. The adversary's only hope is in guessing $x \odot r$. However, the Golreich-Levin Theorem squashes any such hope. Thus, we have successfully constructed a generator that stretches its input by 1 bit.

2.3.3 Achieving a polynomial stretch

In the previous section, we showed how to construct a generator with a one-bit stretch. The goal in this section is to produce a generator with arbitrarily long stretch. Thankfully, we can largely recycle the machinery we previously developed.

Theorem 4. *Let f be a one-way permutation. Moreover, let x and r be randomly sampled from $\{0,1\}^n$. Then,*

$$G(x,r) = r, f^{n^c}(x) \odot r, f^{n^c-1}(x) \odot r, \ldots, f^1(x) \odot r$$

is a pseudorandom generator with stretch $\ell(2n) = n + n^c$.

Proof. Suppose for the sake of contradiction that G is not a pseudorandom generator. Then, by Yao's Theorem, G must be predictable. This implies that there exists an efficient algorithm $\mathcal{A}$ that for a random x and r and a random index $i \in \{1, \ldots, n^c\}$ can predict $f^i(x) \odot r$

from the preceding bits with probability greater than random chance plus a non-negligible function $\delta(n)$:

$$\mathbf{Pr}[A(r, f^{n^c}(x) \odot r, \ldots, f^{i+1}(x) \odot r) = f^i(x) \odot r] \geq \frac{1}{2} + \delta(n)$$

We will now construct another algorithm $\mathcal{B}$ that leverages $\mathcal{A}$ to predict $x \odot r$ from $f(x)$ and r with probability greater than random chance plus a non-negligible function. Since the Goldreich-Levin Theorem proved that was impossible, we will have reached a contradiction.

Algorithm $\mathcal{B}$ operates as follows when given $y = f(x)$ and r:

- Pick a random index $i \in \{1, \ldots, n^c\}$
- Compute the values $f^{n^c - i}(y), \ldots, f(y)$
- Compute $a = A(r, f^{n^c - i}(y) \odot r, \ldots, f(y) \odot r)$

By assumption, $\mathcal{A}$ will predict $f^{-1}(y) \odot r$ with probability $\geq 1/2 + \delta(n)$. Why is that useful to us? The expression $f^{-1}(y) \odot r$ is equivalent to $x \odot r$ because $f(x) = r$. Thus, we have violated the Goldreich-Levin Theorem, which completes the contradiction. $\square$

2.4 Summary of pseudorandomness and cryptography

In this portion of this paper, we described how to construct a cyrptographic pseudorandom generator that stretches a seed into a polynomially-bigger output that can be used as a key in an encryption scheme. Along the way, we covered some of the biggest hits in the space: one-way functions, Yao's Theorem, and the Goldreich-Levin Theorem.

Regardless of whether the reader remembers the specific content covered, we hope they take away two themes. First, there is a deep connection between modern cryptographic techniques and complexity theory. Second, cryptography rests upon the existence of "hard problems": in our case, we detailed the importance of one-way functions in constructing PRGs. This theme will carry over in the next section of the paper, where we will describe how "hard functions" are also essential for derandomization.

3 Derandomization

3.1 Sufficiently powerful Pseudorandom Generators imply P = BPP

In section 1.2 we sketched an argument for how we could derandomize **BPP** if sufficiently powerful PRGs exist. We will define Derandomization PRGs, which will be similar to the Crytpographic PRGs defined in section 2.1 aside from three keys differences:

1. Cryptographic PRGs must only appear random to uniform machines but Derandomization PRGs must appear random to non-uniform machines. To remind the reader, non-uniform machines (i.e. circuits) are allowed to have a different algorithm for every input size.
2. Cryptographic PRGs aim to appear random to adversaries that are allowed to use more computational time than the generator[4], but Derandomization PRGs are allowed to use more time than the adversary they fool. In particular, we will allow our Derandomization PRGs to take $2^{O(\ell)}$ time where ℓ is seed length.
3. We are particularly interested in Derandomization PRGs that allow for exponential stretch. That is, they convert a seed of length ℓ to one of length $2^{O(\ell)}$. However, exponential PRGs are useless in a cryptographic setting because an adversary can distinguish them in time that is a polynomial of their output length by just trying all possible seeds.

We will point out the exact reasons for these differences after the proof of Theorem 5 which proves that sufficiently powerful "Derandomizaiton" PRGs imply that **BPP** = **P**.

Definition ((S, ϵ) pseudorandom)**.** *A distribution R over $\{0,1\}^m$ is (S, ϵ) pseudorandom if for every circuit C of size at most S:*

$$|\boldsymbol{Pr}[C(R) = 1] - \boldsymbol{Pr}[C(U_m) = 1]| < \epsilon$$

where U_m denotes the uniform distribution over $\{0,1\}^m$. Furthermore, we will call the expression $|\boldsymbol{Pr}[C(R) = 1] - \boldsymbol{Pr}[C(U_m) = 1]|$ the advantage of circuit C over R.

Definition (Derandomization Pseudorandom Generator)**.** *Let G be computable in time $2^{O(n)}$ and $L : \mathbb{N} \to \mathbb{N}$ satisfy $L(\ell) > \ell$ for all ℓ. G is a derandomization pseudorandom generator of stretch $L(\ell)$ if $|G(x)| = L(|x|)$ and the distribution $G(U_\ell)$ is $(L(\ell)^3, \frac{1}{10})$ pseudorandom.*

For example, if a derandomization pseudorandom generator has stretch ℓ^{10}, it stretches seeds of size ℓ to pseudorandom strings of size ℓ^{10}, and that output must appear nearly random to circuits of size $(\ell^{10})^3 = \ell^{30}$.

[4]Specifically, the cryptographic PRG is still secure even if the adversary uses a greater amount of polynomial time. However, it is not secure if the adversary uses exponential time.

The constants are arbitrary. 3 could be changed to any constant strictly greater than 2 and $\frac{1}{10}$ could be changed to any constant strictly less than $\frac{1}{2}$. For the remainder of this section, we will use "pseudorandom generators" to refer to Derandomization Pseudorandom Generators, and we will use the shorthand $L(\ell)$-pseudorandom geneator to refer to pseudorandom generators with stretch $L(\ell)$. Now, we are able to tackle the derandomization of **BPP**:

Theorem 5. *If, for some constant $\epsilon > 0$, there exists an $2^{\epsilon\ell}$-pseudorandom generator, then* ***P*** $=$ ***BPP***.

We prove a proof sketch here and, for the interested reader, a full proof in the appendix. An algorithm in **BPP** uses only poly(n) bits of randomness, and an $2^{\epsilon\ell}$ PRG can stretch seeds of length $O(\log n)$ to poly(n) length pseudorandom strings. A deterministic algorithm can approximately simulate a randomized algorithm by enumerating over all $2^{O(\log n)}$ seeds of length $O(\log n)$, stretching that to poly(n) bits of pseudorandomness, and running the **BPP** algorithm with the true randomness replaced with that generated pseudorandomness. Then, the determinstic algorithm just accepts iff the randomized algorithm accepts on the majority of these simulated runs.

In order to show that the above algorithm is correct, we argue by contradiction. If the algorithm outputs the wrong answer on any input x, then we can construct an algorithm that distinguishes the output of the PRG from fully random (see appendix for details). That would contradict the properties of a PRG.

This proof sketch gives insight on the three primary differences between derandomization and crytographic PRGs.

1. The algorithm that distinguishes the PRG from random uses x as non-uniform advice, which is why we required Derandomization PRGs to be indistinguishable to circuits, not just Turing machines.
2. When we construct the deterministic algorithm that simulates a randomized algorithm, we anyways have to loop over all 2^ℓ seeds (where ℓ is the seed length). Thus, if the generator itself takes $2^{O(\ell)}$ time, it only affects the runtime by a polynomial amount, so we were ok with slower derandomization PRGs than cryptographic PRGs.
3. We also stated that we are particularly interseted in PRGs with exponential stretch. This is because our deterministic simulation must enumerate over all random seeds of length ℓ, so it takes a minimum of 2^ℓ time. In order for this to be poly(n) time, we need $\ell = O(\log n)$. So if we want n bits of randomness, we need exponential stretch. However, it turns out that less stretch can still provide useful derandomization results.

Fact 1. *The following two results also hold:*

1. *If, for some constant $\epsilon > 0$, there exists a 2^{l^ϵ}-pseudorandom geneator, then* ***BPP*** $\subseteq$ ***QuasiP*** $=$ ***DTIME***$(2^{polylog(n)})$

2. *If, for every $c > 1$, there exists an l^c-pseudorandom generator, then* $\boldsymbol{BPP} \subseteq \boldsymbol{SUBEXP} = \bigcap_{\epsilon>0} \boldsymbol{DTIME}(2^{n^\epsilon})$

The proof of Fact 1 is essentially the same as the proof of Theorem 5 except the seeds must be larger.

3.2 Constructing PRGs from Hard Functions

In this section, we will describe how to construct PRGs from sufficiently hard functions. All the functions that we consider will have range $\{0, 1\}$, so they can be reasoned about as languages. In particular, if we say that a function f is in a particular complexity class, we mean that the corresponding language is in that complexity class. Here, we will assume the existence of functions that are hard in the average case.

Definition (Average-case hardness)**.** *Choose any $f : \{0,1\}^n \to \{0,1\}$. Then the average-case hardness of f, denoted $H_{avg}(f)$ is defined as the largest S such that, for all circuits C with size at most S, the following holds:*

$$\underset{x\in\{0,1\}^n}{\boldsymbol{Pr}}[C(x) = f(x)] < \frac{1}{2} + \frac{1}{S}$$

Thus, average-case hardness refers to the smallest circuit that correctly decides significantly more than half of the inputs. At first glance, it may seem like such functions don't exist. For example, the existence of commercial SAT solvers that work well on most instances suggests that SAT has only polynomial average case hardness. We would hope to instead reason about worst-case hardness which refers to the smallest circuit that correctly decides all inputs, as it seems more likely that worst-case hard problems exist. It turns out, however, that it's possible to construct average case hard functions from worst case hard functions:

Fact 2 (worst-case hardness to average-case hardness. See Theorem 19.27 in Arora-Barak)**.** *Let $f \in \boldsymbol{E} = \boldsymbol{DTIME}(2^{O(n)})$ be a function with worst-case hardness at least $S(n)$. Then there exists a function $g \in \boldsymbol{E}$ and constant $c > 0$ such that the average-case hardness of g is at least $S(\delta n)^\delta$ for constant $\delta > 0$.*

We will not prove Fact 2 but the interested reader should see chapter 19 of Arora-Barak. This fact shows that if we believe that there are functions that are hard in the worst case, then we should believe there are functions that are hard in the average case. In particular, we will later prove that if there is a function in **E** with average case hardness $2^{O(n)}$, then **P** = **BPP**. Fact 2 means we only need search for a worst-case hard function that is in **E**, i.e. computable by a uniform algorithm in $2^{O(n)}$, that requires at least $2^{\Omega(n)}$ time for a non-uniform solution.

3.2.1 Small stretch from Average-Case Hardness

Recall in the crytopgraphy section, we proved Yao's Theorem, which says that if a distribution is next-bit unpredictible the entire distribution is pseudorandom. We restate Yao's Theorem in a slightly different form. The proof of this form is mostly identical to the proof we provided.

Theorem 6. *Let Y be a distribution over $\{0,1\}^m$. Suppose that for every circuit of size at most $2S$ and $i \in [m] = \{1, \dots, m\}$:*

$$\Pr_{r \sim Y}[C(r_1, \dots, r_{i-1}) = r_i] \leq \frac{1}{2} + \frac{\epsilon}{m}$$

Then Y is (S, ϵ)-pseudorandom

We now apply Yao's Theorem to construct a PRG that returns a pseudorandom string two bits longer than its seed:

Lemma 1. *If there exists an $f \in \mathbf{E}$ with average-case hardness at least n^4, then there exists an $(\ell + 2)$-pseudorandom generator.*

Proof. Let ℓ be the length of the seed. For simplicity, assume ℓ is even (essentially the same construction holds for odd ℓ). Let G be the following generator:

$$G(z) = z_1 \cdots z_{\ell/2} \circ f(z_1 \cdots z_{\ell/2}) \circ z_{l/2+1} \cdots z_\ell \circ f(z_{l/2+1} \cdots z_\ell)$$

In the above statement, $\circ$ represents concatenation. In order for G to be a PRG, we need to ensure that its output is $((\ell + 2)^3, \frac{1}{10})$ psuedorandom. By Yao's theorem, it's enough to prove that there is no circuit of size $2(\ell + 2)^3$ and $i \in [\ell + 2]$ such that:

$$\Pr_{r = G(U_\ell)}[C(r_1, ..., r_{i-1}) = r_i] > \frac{1}{2} + \frac{1}{10(\ell + 2)} \tag{1}$$

All of the bits except for $f(z_1 \cdots z_{\ell/2})$ and $f(z_{l/2+1} \cdots z_\ell)$ are completely random and no circuit can predict random bits with probability more than $\frac{1}{2}$. Thus, we only need to check if a circuit can satisfy Equation 1 for $i = \frac{\ell}{2} + 1$ or for $i = \ell + 2$. If a circuit C of size $2(\ell + 2)^3$ could predict bit $i = \frac{\ell}{2} + 1$, then it would satisfy

$$\Pr_{r \in U_{\ell/2}}[C(r) = f(r)] > \frac{1}{2} + \frac{1}{10(\ell + 2)},$$

which contradicts the fact that that f has average-case hardness of n^4. We would like to make a similar claim that predicting the bit $i = \ell + 2$ would also contradict the average-case hardness of f. If a circuit C of size at most $2(\ell + 2)^3$ could predict that bit, then:

$$\Pr_{r, r' \in U_{\ell/2}}[C(r, f(r), r') = f(r')] > \frac{1}{2} + \frac{1}{10(\ell + 2)} \tag{2}$$

Here we could be worried that knowing $f(r)$ could somehow help C predict $f(r')$. Intuitively, it seems that since r and r' are independent, knowing $f(r)$ would not help C predict $f(r')$.

We can make this formal using something called the averaging principle. The averaging principle states that if A is a function of two independent random variables X, Y, then there must some fixed choice for x^* within X such that:

$$\mathop{\mathbf{Pr}}_{Y}[A(x^*, Y)] \geq \mathop{\mathbf{Pr}}_{X,Y}[A(X, Y)]$$

Combining the averaging principle with equation 2, we see that there must exist some fixed choice for r, call it r^*, that satisfies

$$\mathop{\mathbf{Pr}}_{r' \in U_{\ell/2}}[C(r^*, f(r^*), r') = f(r')] > \frac{1}{2} + \frac{1}{10(\ell+2)} \tag{3}$$

where now r is fixed to be r^* and r' is the only randomness. If Equation 3 is true, then we can create a second circuit D that hard-wires in the bits r^* and $f(r^*)$ and computes $C(r^*, f(r^*), r')$, still having size at most $2(\ell+2)^3 + \ell$. That circuit would then be able to predict the output of f significantly more than half of the time, contradicting the fact that f is average-case hard. We conclude that all of the bits of $G(z)$ are unpredictible, so G is a PRG. □

We could repeat the above argument to create an $(\ell + c)$-PRG for any constant c by just breaking up the input into c subsets instead of 2 subsets. To do so, we would have:

$$G(z) = z^1 \circ f(z^1) \circ z^2 \circ f(z^2) \circ \cdots \circ z^c \circ f(z^c)$$

where each z^i is the ith block of ℓ/c bits in z. Unfortunately, such an argument won't even allow for a 2ℓ-PRG, a far cry away from the exponential stretch we need to prove $\mathbf{P} = \mathbf{BPP}$. In order to allow for larger stretches, we need to allow for the blocks to overlap with one another. While the final construction will be more complicated, we first provide an example that still only gives a 2 bit stretch.

Lemma 2. *Let f be a function with average case hardness 2^n and G be the following generator:*

$$G(z) = z \circ f(z^1 \circ z^2) \circ f(z^2 \circ z^3)$$

where each z^i is a block of $\ell/3$ bits and $z = z^1 \circ z^2 \circ z^3$. Then, G is a $(\ell+2)$-PRG.

For simplicity, we will assume ℓ is a multiple of 3. This generator is meant to illustrate how we can allow overlap between the inputs to f if the function f is sufficiently hard and the overlap is not too big. The *average-case* hardness requirement of 2^n is a bit strict for practical purposes but makes the proof simpler. In later constructions, we'll see that average case hardness of $2^{\epsilon n}$ for $\epsilon > 0$ is sufficient.

Proof. Once again, we use Yao's Theorem. It is enough for us to prove that none of the bits of $G(z)$ are predictable by a circuit of size $2(\ell+2)^3$. Clearly none of the first $\ell/3$ bits are predictable because they are completely random. If the bit at index $\ell/3+1$ were predictable, then there would be circuit C such that:

$$\mathop{\mathbf{Pr}}_{r_1,r_2,r_3\in U_{\ell/3}}[C(r_1 \circ r_2 \circ r_3) = f(r_1 \circ r_2)] > \frac{1}{2} + \frac{1}{10(\ell+2)}$$

If that were true, by the averaging principal, there would be at least one $r_3^* \in \{0,1\}^{\ell/3}$ that satisfies

$$\mathop{\mathbf{Pr}}_{r_1,r_2\in U_{\ell/3}}[C(r_1 \circ r_2 \circ r_3^*) = f(r_1 \circ r_2)] > \frac{1}{2} + \frac{1}{10(\ell+2)}$$

This allows us to construct a circuit that simulates C with r_3^* hard-wired in and is able to predict the output of f, contradicting the average case hardness of f. The harder part is showing that no circuit, C, of size $2(\ell+2)^3$ can predict $f(z^2 \circ z^3)$ even if it knows $f(z^1 \circ z^2)$, which overlap in many bits. Formally, we want to prove the no circuit C of size $2(\ell+2)^3$ satisfies:

$$\mathop{\mathbf{Pr}}_{r_1,r_2,r_3\in U_{\ell/3}}[C(r_1 \circ r_2 \circ r_3 \circ f(r_1 \circ r_2)) = f(r_2 \circ r_3)] > \frac{1}{2} + \frac{1}{10(\ell+2)}$$

Assume the above is true. Then, by the averaging principle, we can choose fixed $r_1^* \in \{0,1\}^{\ell/3}$ such that:

$$\mathop{\mathbf{Pr}}_{r_2,r_3\in U_{\ell/3}}[C(r_1^* \circ r_2 \circ r_3 \circ f(r_1^* \circ r_2)) = f(r_2 \circ r_3)] > \frac{1}{2} + \frac{1}{10(\ell+2)}$$

Consider the function $g : \{0,1\}^{\ell/3} \to \{0,1\}$ that computes $g(r_2) = f(r_1^* \circ r_2)$. Since this function takes a string of length $\ell/3$ as input, it is trivially computable[5] by a circuit, which we will call D, of size $(\ell/3)2^{\ell/3}$. We can then construct a circuit, which we will call E, which takes as input arbitrary $r_2, r_3 \in \{0,1\}^{\ell/3}$. It uses circuit D to compute $g(r_2) = f(r_1^* \circ r_2)$ and then circuit C to compute $C(r_1^* \circ r_2 \circ r_3 \circ f(r_1^* \circ r_2))$. We then have that:

$$\mathop{\mathbf{Pr}}_{r_1,r_2\in U_{\ell/3}}[E(r_2 \circ r_3) = f(r_2 \circ r_3)] > \frac{1}{2} + \frac{1}{10(\ell+2)}$$

But, circuit E has total size that is the sum of the size of circuits C and D, or only $(\ell+2)^3 + (\ell/3)2^{\ell/3}$, which is less than 2^ℓ. This contradicts the average-case hardness of f. Thus, G is a PRG. □

[5]Any binary function of n inputs is computable by a circuit with $n2^n$ gates. If we allowed unlimited fan-in, then it is trivially computable by a size 2^n+1 circuit. That circuit would have an AND gate for each possible input and then a single OR gate that connects all the AND corresponding to a string the function wants to accept. If we instead use fan-in of 2, then the AND gates need to be repalced with $n-1$ connected AND gates, leading to a bound of $(n-1)2^n + 1 \leq n2^n$ gates.

3.3 Constructing Exponential Stretch

In the previous section, we saw that it is permissible for a PRG to generate the output of some hard function f on intersecting subsets of a random seed. What is important is that the output of f on the subsets not be too correlated. We saw that if f is hard and the subsets have sufficiently small intersection, then the PRG will be hard to predict. We introduce the following construction of PRGs:

Definition. *(Nisan-Wigderson generator) Let $\mathcal{I} = \{I_1, ..., I_m\}$ be a family of subsets of $[\ell]$ where $|I_j|= n$ for every j and $f\{0,1\}^n \to \{0,1\}$ be some function. Then, the $(\mathcal{I}, f)$-NW generator is the function $NW^f_{\mathcal{I}} : \{0,1\}^\ell \to \{0,1\}^m$ that maps every $z \in \{0,1\}^\ell$ to*

$$NW^f_{\mathcal{I}}(z) = f(z_{I_1}) \circ f(z_{I_2}) \circ \cdots \circ f(z_{I_m})$$

where z_I denotes the restriction of z to the coordinates of I.

For an example of how the notation z_I is used, if $z = 0110$ and $I = \{1, 2, 4\}$, then $z_I = 010$. In order to construct PRGs with exponential stretch, we will use an NW-generator with exponentially many subsets. Combinatorial designs will allow us to do this while maintaining a small intersection between every pair of subsets.

Definition (Combinatorial Designs)**.** *Consider any positive integers $\ell > n > d$. A family $\mathcal{I} = \{I_1, ..., I_m\}$ of subsets of $[\ell]$ is an (ℓ, n, d)-design if each subset has n elements and the intersection of any two subsets has at most d elements. I.e., for $|I_j|= n$ for all j and $|I_j \cap I_k|\leq d$ for all $j \neq k$*

For example, the following is $(7, 4, 2)$-design with 7 elements:

$$\mathcal{I} = \left\{ \begin{array}{lccccccr} \{1 & 2 & 3 & 4 & & & & \} \\ \{1 & 2 & & & 5 & 6 & & \} \\ \{1 & & 3 & & 5 & & 7 & \} \\ \{1 & & & 4 & & 6 & 7 & \} \\ \{ & 2 & 3 & & & 6 & 7 & \} \\ \{ & 2 & & 4 & 5 & & 7 & \} \\ \{ & & 3 & 4 & 5 & 6 & & \} \end{array} \right\}$$

It turns out that large designs can be constructed by a simple deterministic algorithm.

Lemma 3 (Deterministic Construction of Designs)**.** *There is an algorithm A that on input $\langle \ell, d, n\rangle$ for $n > d$, $n > 20$, and $\ell \geq 10n^2/d$ takes time $2^{O(\ell)}$ and outputs an (ℓ, n, d)-design $\mathcal{I}$ containing $2^{d/10}$ subsets of $[\ell]$.*

The proof of Lemma 3 is provided in the appendix. The ability to construct exponentially large designs with small overlap will allow us to prove that if certain hardness results hold then PRGs with exponential stretch exist.

Theorem 7. *If there exists an $f \in \boldsymbol{E}$ with average-case hardness at least $2^{\epsilon n}$ for $\epsilon > 0$, then there exists an $2^{\delta \ell}$-PRG for $\delta = \epsilon^2/10000$. By Theorem 5, this implies that $\boldsymbol{P} = \boldsymbol{BPP}$.*

Let G be the generator that first creates $\mathcal{I}$ which is (ℓ, n, d)-design of size $2^{d/10}$ where $n = \epsilon\ell/100$ and $d = \epsilon n/10$. Then, G returns the $(\mathcal{I}, f)$-NW generator. This generator runs in time $2^{O(\ell)}$ and has stretch:

$$2^{d/10} = 2^{\epsilon n/100} = 2^{\epsilon^2 \ell/10000} = 2^{\delta \ell}$$

Thus, we need only prove that the output of G is $((2^{d/10})^3, \frac{1}{10})$ pseudorandom. This proof is extremely similar to the proof of Lemma 2. We will use S to refer $2^{\epsilon n}$, the average-case hardness of f. By Yao's Theorem, it is sufficient to show that no circuit C of size $2(2^{3d/10}) \leq S/2$ can predict the next bit of G. Let $Z \sim U_\ell$. Then suppose for the sake of contradiction that there is a circuit of size $S/2$ and index $i \in [2^{d/10}]$ that satisfies

$$\Pr_{R = NW_{\mathcal{I}}^f(Z)}[C(R_1, ..., R_{i-1}) = R_i] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}}$$

Plugging in the definition for the Nissan-Wigderson generator, we have that:

$$\Pr_{Z \in U_\ell}[C(f(Z_{I_1}), ..., f(Z_{I_{i-1}}) = f(Z_{I_i})] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}}$$

Similarly to how we did in Lemma 2, we want to break the larger random Z into two parts. Without loss of generality, we suppose that $I_i = \{0, 1, ..., n\}$, because we can always permute the indices of Z to ensure this. Then, let Z_1 refer to the first n indices of Z and Z_2 refer to the last $n - d$ indices. The reason we do this is so that later, we can use the averaging principle to fix Z_2. First, we write:

$$\Pr_{Z_1 \in U_n, Z_2 \in U_{\ell - n}}[C(f((Z_1 \circ Z_2)_{I_1}), ..., f((Z_1 \circ Z_2)_{I_{i-1}}) = f(Z_1)] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}}$$

We can now use the averaging principle: There must be at least one fixed choice for Z_2, which we will call z_2^*, such that:

$$\Pr_{Z_1 \in U_n}[C(f((Z_1 \circ z_2^*)_{I_1}), ..., f((Z_1 \circ z_2^*)_{I_{i-1}})) = f(Z_1)] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}}$$

Since $\mathcal{I}$ is a design, we know that for any $j = 1, ..., i-1$, that $|I_j \cap I_i| \leq d$. This means for any such j, the function that maps Z_1 to $f((Z_1 \circ z_2^*)_{I_j})$ depends on at most d indices of Z_1. Thus, there must exist a circuit that computes it with size at most $d2^d$. Call those circuits $D_1, ..., D_{i-1}$. This means that:

$$\Pr_{Z_1 \in U_n}[C(D_1(Z_1), ..., D_{i-1}(Z_1)) = f(Z_1)] \geq \frac{1}{2} + \frac{1}{10 \cdot 2^{d/10}}$$

The function that takes in z and returns $C(D_1(z), ..., D_{i-1}(z))$ can be computed by a circuit that has size equal to size of C plus the sizes of each of the $D_1, ..., D_{i-1}$. This size is at most $\frac{S}{2} + id2^d < S$, contradicting the average-case hardness of f. Thus, we conclude that G must be a PRG. $\square$

3.4 Summary of Derandomization

In this section, we showed that the question of whether **P** equals **BPP** is tied to the power of non-uniformity. We know by the time hierarchy theorem that there are problems that both can be solved in time $2^{O(n)}$ and require at least $2^{\Omega(n)}$ time for uniform Turing machines. If any of those problems require $2^{\Omega(n)}$ size for circuits, or equivalently, for *non-uniform* Turing machines, then $\mathbf{P} = \mathbf{BPP}$. The proof for that fact proceeds as follows:

1. Fact 2 is used to show that we can convert the function that has worse-case hardness of $2^{\Omega(n)}$ to one that has average-case hardness of $2^{\Omega(n)}$.
2. Theorem 7 uses the Nissan Wigderson construction and the average-case hard function to construct a PRG with exponential stretch.
3. Theorem 5 shows that, given a PRG with exponential stretch, we can create a deterministic polytime algorithm for every problem in **BPP**, proving that $\mathbf{P} = \mathbf{BPP}$

An example for a function that is computable in $2^{O(n)}$ and might also satisfy the hardness requirements is SAT. To compute SAT in $2^{O(n)}$, an algorithms can simulate every possible set of inputs. Still, despite much work on SAT, no one has been able to come up with a circuit that solves SAT and uses less than $2^{\Omega(n)}$ gates.

Interestingly, we also know that if SAT is extremely easy, i.e. solvable in **P** , then the polynomial hierarchy collapses and $\mathbf{P} = \mathbf{BPP}$. Thus, the only scenario in which $\mathbf{P} \neq \mathbf{BPP}$ is when SAT is neither too easy nor too hard.

4 Appendix

We provide the interested reader with additional proofs. We do not think that the peer graders should be required to read this section during finals week given that this report is already quite long, but provide it for completeness.

Theorem 5. *If, for some constant $\epsilon > 0$, there exists an $2^{\epsilon \ell}$-pseudorandom generator, then* $\boldsymbol{P} = \boldsymbol{BPP}$.

Proof. Suppose that L is in **BPP** with randomized algorithm running in time $T(n) = O(n^c)$ for constant c. This means there is a deterministic algorithm A such that for all $x \in \{0,1\}^n$:

$$\mathop{\mathbf{Pr}}_{r \in \{0,1\}^{T(n)}}[A(x,r) = L(x)] \geq \frac{2}{3}$$

Where $L(x)$ is defined as 1 if $x \in L$ and 0 otherwise. The main idea is to prove that we can replace the truly random bits r, with the output of G, which is a $2^{\Omega(l)}$-pseudorandom generator. In order to produce $T(n)$ random bits, G only needs a seed of length $O(\log n)$. We will prove the following:

$$\mathop{\mathbf{Pr}}_{s \in \{0,1\}^{O(\log n)}}[A(x,G(s)) = L(x)] > \frac{1}{2} \tag{4}$$

We construct a polytime deterministic algorithm B that exactly computes $\mathop{\mathbf{Pr}}_{s}[A(x,G(s)) = 1]$ and accepts iff that probability is more than $\frac{1}{2}$. If Equation 4 is true, then B must correctly decide L. Since there are only $2^{O(\log n)} = \text{poly}(n)$ choices for $s \in \{0,1\}^{O(\log n)}$, it's possible for B to enumerate over all possible s and simulate $A(x, G(s))$ for each of them. B then accepts if more than half of the simulations accept. From our definition for PRGs, we know that computing $G(s)$ will take poly(n) time, and computing A also takes poly(n), so B is a polytime deterministic algorithm.

Now we prove Equation 4 by contradiction. Suppose that there exists an x where:

$$\mathop{\mathbf{Pr}}_{s \in \{0,1\}^{O(\log n)}}[A(x,G(s)) = L(x)] \leq \frac{1}{2}$$

Then, we can construct an algorithm C that distinguishes the output of G from a fully random string. On input $|r| = T(n)$, algorithm C simply returns $A(x,r)$. If r were truly random, then $A(x,r) = L(x)$ with probability at least $\frac{2}{3}$, and if r is the output of G, then $A(x,r) = L(x)$ with probability at most $\frac{1}{2}$.

This algorithm C can be transformed into a circuit with x hard-wired in of size $O(T(n)^2) = O(|r|^2)$ using the same transformation from the Cook-Levin theorem. Thus, there is a circuit that distinguishes the output of G from a fully random string with probability at least $\frac{2}{3} - \frac{1}{2} > \frac{1}{10}$ contradicting the fact that G is a pseudorandom generator. □

We also will prove Lemma 3, but first need the following well know probability fact.

Fact 3 (Chernoff Bounds)**.** *Let* $X = \sum_{i=1}^{n} X_i$ *where each* X_i *is a Bernoulli random variable (always* 0 *or* 1*). Let* $\mu = \boldsymbol{E}[X]$*. Then the following concentration inequalities hold:*

$$\boldsymbol{Pr}(X \geq (1+\delta)\mu) \leq \exp\left(-\frac{\mu\delta^2}{2+\delta}\right) \quad \textit{for all } \delta > 0$$
$$\boldsymbol{Pr}(X \leq (1-\delta)\mu) \leq \exp\left(-\frac{\mu\delta^2}{2}\right) \quad \textit{for all } 0 < \delta < 1$$

Armed with the above, we are able to prove the correctedness of an algorithm for the construction of designs.

Lemma 3 (Deterministic Construction of Designs)**.** *There is an algorithm* A *that on input* $\langle \ell, d, n\rangle$ *for* $n > d$*,* $n > 20$*, and* $\ell \geq 10n^2/d$ *takes time* $2^{O(\ell)}$ *and outputs an* (ℓ, n, d)*-design* $\mathcal{I}$ *containing* $2^{d/10}$ *subsets of* $[\ell]$*.*

Proof. A will be a very simple greedy algorithm. First, it initializes $\mathcal{I} = \emptyset$ and adds on subset to $\mathcal{I}$ at a time. When $\mathcal{I} = \{I_1, ..., I_m\}$ the algorithm searches for I_{m+1} by searching all subsets of $[\ell]$ and adding the first n-sized subset I that satisfies $I \cap I_j| \leq d$ for $j = 1, ..., m$. Once $2^{d/10}$ subsets have been found this way, the algorithm returns $\mathcal{I}$.

Algorithm A runs in time poly$(m, n, d, 2^{\ell})$ which is in $2^{O(\ell)}$ time, so the A satisfies efficiency requirements. Thus, we need only prove it doesn't get stuck. In order to do that, we need to show that if $\{I_1, ..., I_m\}$ is a collection of n-sized subsets of $[\ell]$ for $m < 2^{d/10}$, then there exists another sized n subset of $[\ell]$ named I that does not intersect any $I_1, ..., I_m$ at more than d indices. We'll do this by showing if we pick such an I randomly, there is a nonzero probability it meets the requirements, which means at least one such I exists.

Suppose we pick I at random by independently adding in each index in $[\ell]$ with probability $2n/\ell$. First, we claim that with probability at least 0.9, the size of I will be at least n. Let X denote the random variable corresponding to the size of I, which is the sum of ℓ Bernoulli random variables, each of which is 1 with probability $2n/\ell$. Thus, $\mu = \mathbf{E}[X] = 2n$, so by Chernoff Bounds:

$$\mathbf{Pr}[|I| \leq n = \mathbf{Pr}[X \leq (1 - \frac{1}{2})\mu] \leq \exp(-\frac{2n}{8}) \leq 0.1$$

So with probability at least 0.9, the size of I is at least n. Now we show that the probability that I intersects any of the $I_1, ..., I_m$ at more than d indices is small. Fix some I_j and let Y be the random variable representing $|I \cap I_j|$. Then, each index has a $2n/\ell$ probability of being in I and a n/ℓ probability of being in I_j, so Y is the sum of ℓ Bernouli variables each of which is 1 with probability $2n^2/\ell$. Thus, $\mathbf{E}[Y] = 2n^2$ In this lemma we assume $\ell \geq 10n^2/d$ and we larger ℓ only make it easier to find subsets, so it is safe to take $\ell = 10n^2/d$. Then, $\mu = \mathbf{E}[Y] = d/5$ and by Chernoff Bounds:

$$\mathbf{Pr}[|I \cap I_j| \geq d] = \mathbf{Pr}[Y \geq (1+4)\mu] \leq \exp(-\frac{32n}{6}) \leq \exp(-4d) \leq 0.5 \cdot 2^{-d/10}$$

Thus, by union bound, we that I does not intersect with any of the $m \leq 2^{d/10}$ sets $I_1, ..., I_m$ at more than d indices and I has at least n indices with probability at least $1-0.1-0.5 = 0.6$, proving that such an I must exist. If such an I exists, then there must also exist an I with exactly n indices in it, because we can remove the extra indices from I. $\square$