SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

RNG and Derandomized Algorithms

av

Wictor Zawadzki

2020 - No K42

RNG and Derandomized Algorithms

Wictor Zawadzki

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Olof Sisask

2020

Abstract

Randomness is heavily relied upon in different computation situations across many industries, but generating a lot of random numbers can be quite resource intensive. As a result, an argument could be made in favor of derandomizing algorithms into deterministic form whenever possible. The goal of this thesis is to investigate random number generation, and the use of randomness in algorithms. We first look at theoretical construction of pseudo-random number generators, statistical tests, and cryptographic random number generators, as well as some practical examples for these. The second part of the thesis focuses on the differences in method and performance between random algorithms and their derandomized counterparts. After looking at specific random algorithms, we conclude in this thesis that deterministic algorithms seem to often suffer a few disadvantages from not using random numbers. Two examples of significant drawbacks are the existence of pathological inputs, as well as that some algorithms may fundamentally require randomness to function as intended, for instance cryptographic systems.

Acknowledgements

I am very grateful to my supervisor Olof Sisask, who had the patience to oversee my work these past months, provided helpful advice, and proofread my work. I would also like to express my gratitude towards Boris Shapiro, who likewise took the time to proofread my work.

Contents

1 Introduction

In practice, there are many purposes for which one may wish to use random number generation. For instance, simulations may require an element of randomness to account for unpredictable variables in what they aim to simulate. In statistics, it may be important to take a number of random and independent samples of a particular body of data that is too big to analyze fully. In computer security, we may desire to create some cryptographic system using secret keys chosen in a way that potential attackers couldn't predict, much like how locksmiths should want to make it difficult to guess what the key to a specific lock might be.

However, generating random numbers may be easier said than done. There are myriad ways to generate random numbers, and all methods are not equally good. As if that wasn't enough to consider, many situations have particular constraints on available time, space and resources, making it even more crucial to choose the right *random number generator* (RNG) for the right purpose. A random number generator is some mechanism which can be used to generate a random number, two types of which will be discussed in this project. It is because of the previously mentioned reasons that the increasing demand of RNG for use in modern technology has led to the study of random number generation becoming increasingly more important in the last several decades.

From this brief introduction, it should be clear that random numbers do not exist in a vacuum. From finding ways to generate random numbers and making sure that they are random enough, to designing algorithms that use RNG and measuring how effective they are, the study of randomness covers many fields of mathematics and computer science, such as number theory, statistical analysis and algorithm theory. As a result, this work will take something of an interdisciplinary approach to consider randomness on both theoretical and practical levels, from generation to application.

I should note that there is one resource in particular, *The Art of Programming* by Donald Knuth, which was used throughout this project. I have opted to cite it only whenever a significant result from it is being used in the text.

2 Random Number Generation

2.1 Background and Terminology

The purpose of a random number generator is to output a number, often in the form of a bit sequence, in an unpredictable manner and on demand. Since randomness does not simply appear out of thin air, such generators will require some form of input.

Devices which generate a random number via some unpredictable physical event are known as hardware random number generators, or *true random number generators* (TRNG). Some such devices may generate random numbers by measuring radioactive decay, or by counting keystrokes on a keyboard. A fairly simple one that may appear in everyday situations is a coin toss, resulting in heads or tails at random.

The alternative to true random number generation is *pseudo-random number generation* (PRNG). Rather than a device that outputs numbers based on some unpredictable phenomenon, it is an algorithm which takes some number input known as a seed and then outputs a seemingly unpredictable number - we use the word "seemingly" here because such an algorithm is typically completely deterministic, which is the motivation behind putting "pseudo" in the name. Simple examples of such algorithms are linear congruential generators, which using a seed x will output a number of the form $f(x) = ax + b \mod c$ for sufficiently good choices of a, b, c. A PRNG is typically used in sequence, where instead of creating a new seed (re-seeding) every time you want your PRNG to generate a random number, you will use the last obtained random number to generate the next one, which formulated recursively is $x_{n+1} = f(x_n)$.

We quickly run into an issue, however. How can we convince ourselves that a random number generator is sufficiently random? This is especially important in cryptographic context as we certainly want to give no clues to a potential attacker. However, even for non-cryptographic RNG, usually referred to as *statistical RNG* for their use in statistical algorithms, enough bias can skew results in unwanted ways and lead to e.g. incorrect/biased simulation [5]. One guiding idea is that of an ideal random number generator, as seen in [18], which is a generator with uniformly distributed and independent outputs. It is not difficult to see how such a generator would provide the titular ideal random numbers.

Luckily, there is a number of methods to formalize how random a generator is, and we will be defining three such concepts here. Let us consider the random number generator as a random variable X with its potential outputs being its associated probability space, where in the case of a TRNG we let this consist of the distribution of outputs of the device. In the case of a PRNG we need to think somewhat out of the box, as they by definition are strictly deterministic and hold no inherent randomness, only the "appearance of randomness". Until we discuss PRNGs more thoroughly, we can consider them to be the distribution of the outputs of the algorithm assuming the input is ideally generated. In other words, we will look at a PRNG $Y = f(X)$ as a random variable defined as

the function of an ideally random variable X, with the additional and rather vague property of the output "seeming random" compared to the input. We will introduce this idea by looking at three types of bias, taken from *True Random Number Generators* [21, page 5] and the literature *Fundamentals in Information Theory and Coding* [3, page 11].

2.2 Bias

The first type of bias is perhaps the simplest, and only considers the distribution of individual bits over time. We can call it *simple binary bias*, and we define it as follows:

Definition 2.1 *A simple binary bias b for a random number generator X over a Bernoulli distribution, in other words a two-point distribution over $\{0, 1\}$, as*

$$b := \frac{p(1) - p(0)}{2},$$

where $p(0)$ is the probability of a 0-bit being generated, and $p(1)$ the same for a 1-bit.

This simple definition can be extended to sequences of n bits instead of just individual bits, in which case the distribution is over 2^n points corresponding to each possible bit sequence. For $n = 2$, each possible sequence $00_2, 01_2, 10_2, 11_2$ correspond to $0, 1, 2, 3$. The denominator for the general case would then be 2^n, so for this example case where $n = 2$, the expression describing the bias would be $\frac{p(0)+p(1)+p(2)+p(3)}{4}$.

The most natural example of a generator that definitely fails the simple bias test is one that outputs only 1-bits or only 0-bits, which will yield biases of $b = 1/2$ and $b = -1/2$ respectively. We may additionally desire that knowledge of previously generated random bits does not give hints on what "nearby" bits may be. In other words, there is no short-range correlation. This can for instance be modeled with serial autocorrelation coefficients:

Definition 2.2 *Serial autocorrelation coefficients a_k for a sequence of N bits $(b_1, \ldots, b_N)$ with the mean value $\bar{b}$ is defined as*

$$a_k := \frac{\sum_{i=1}^{N-k}(b_i - \bar{b})(b_{i+k} - \bar{b})}{\sum_{i=1}^{N-k}(b_i - \bar{b})^2},$$

where k denotes the number of bits of "lag", or the distance between the bits tested for correlation.

Such coefficients could for instance detect if every eighth bit generated is more likely to be a 1 than a 0, in which case a_8 would be noticeably positive rather than approximately zero. An example of a generator that would pass the simple bias test but would fail autocorrelation is on that outputs an alternating sequence $101010\ldots$. While the binary bias is $b = 0$, we have for instance $a_1 = -1$ and $a_2 = 1$. The third concept, information entropy, is a measure of bias that can be used for full generated numbers rather than individual bits. It can be viewed as the average bias across all possible outputs.

Definition 2.3 *The entropy $H(X)$ of a random number generator X, with the probability space consisting of all its possible outputs x_i and their associated probabilities $p(x_i)$, is defined as follows:*

$$H(X) := E[-\log_2(p(X))] = -\sum_i p(x_i)\log_2 p(x_i),$$

in which $E[\cdot]$ denotes expected value. The unit of entropy is bits, and could more elaborately be called "average number of bits worth of information".

Among discrete random variables X with n values, the entropy attains its maximum $H(X) = n\log_2 n$ for the uniform distribution, and its minimum $H(X) = 0$ for a single-point distribution $p(a) = 1$ for a point a. More generally, it increases if there are many, approximately equally improbably outputs in X, and it decreases if amongst these outputs there are a few which are significantly more likely. Since we do not want any specific numbers to be especially likely to appear when we randomly generate numbers, one can generally say that a generator with higher entropy is "more random" or "less biased" than one with lower entropy. Naturally, higher entropy is desirable when we try to maximize randomness.

These three concepts are some ways to measure of bias, and will later be integrated into the more general concept of statistical tests. The amount and type of bias depends on the generator, and there may sometimes be a trade-off between low bias and, say, high efficiency. Other methods of measuring and testing the amount of bias in a generator will be discussed further in section 3.1.

2.3 TRNG and its Insufficiency

True random number generators have a few advantages over pseudo-random number generators, perhaps most obviously that TRNGs do not rely on a seed to produce a random number. Indeed, this mean that you need a seed from a TRNG for a PRNG to function effectively. Additionally, two numbers generated by a TRNG are (typically) independent of each other, whereas for a PRNG, a random number $x_{n+1} = f(x_n)$ depends completely on the number x_n used to generate it.

The question may then strike the reader. Why do we bother with PRNGs if TRNGs are better? There are two primary answers to this question, leading to PRNGs being the predominantly relied upon method of producing random numbers in practice.

Problem 1: Rate of generation. Since a TRNG generates numbers by measuring some physical phenomenon, it can only generate a certain number of random bits per second. Depending on the TRNG, attempting to exceed this limit is either impossible, or will lead to significant dependence between measurements which can manifest as increased bias or other issues. A PRNG, in comparison, only requires TRNG seeds occasionally

for re-seeding, and otherwise can generate new random numbers as quickly as the system allows it to. Even using modern methods trying to increase rate of generation, PRNGs are almost always much more efficient.

Part of the reason behind the inefficiency of TRNG is the fact that whatever phenomenon they measure to generate random bits is typically far from ideal. This can be in the form of both non-uniform distribution and dependency between generated numbers. To counter this, an algorithm to turn the distribution uniform and keep the outputs independent is necessary. This process, in some cases referred to as debiasing, comes at the cost of additional inefficiency that PRNGs do not experience.

Problem 2: Cost of components. If one is dead set on using primarily TRNG to generate random numbers, it may be tempting to counter the first problem by simply using the best available TRNG components. However, this will also increase both the total price of the components as well as the amount of physical space taken up, whereas a PRNG is both without monetary price and doesn't require installing components that take up physical space.

By no means can PRNG completely substitute TRNG, but as we can deduce from the previously mentioned issues with TRNGs, it can be a necessary complement to it.

2.4 PRNG and Derandomization

The conclusions on the drawbacks of TRNGs in section 2.2 merits a closer look into PRNGs, and we shall start with an introduction to the concept of derandomization.
Derandomization is the process of using a deterministic algorithm to substitute some random algorithm, as to remove the need for the random input that the random algorithm requires. As PRNGs are deterministic by definition, and serve to be used to partially replace TRNGs which function on random input, the use of PRNGs can be seen as a derandomization of TRNGs.

Let us define a PRNG more explicitly. This is more difficult than it might seem, as its defining characteristic is that we expect the output to "seem random" while we very well know that it is not. A common way of defining this is as an algorithm with outputs that cannot be distinguished from TRNG by any efficient testing algorithm, also known as being computationally indistinguishable [7] from randomness.

We will settle on defining PRNGs as being indistinguishable from true randomness by any polynomial-time random algorithm, in this context often referred to as statistical tests if they use statistical analysis to "test" how random it is. As complexity will be important later, we will define it here based on how it is described in *An Introduction to Mathematical Cryptography* [10, pages 78–80], in the context of time complexity.

Definition 2.4 *If an algorithm $A(x)$ with input x has a running time of $T(x)$, then it has time complexity $O(f(x))$ with respect to x for some function $f(x)$ if there is some constant c and point C such that for all $x \geq C$, $T(x) \leq cf(x)$. This is denoted by "$T(x) = O(f(x))$", meaning that $T(x)$ is bounded from above by some $f(x)$, not necessarily in value but in its rate of growth.*

Let us consider an example. Consider an algorithm that takes a natural number $x \in \mathbb{N}$, and then performs a specific action x times. Assuming that this action always takes a constant unit of time u to perform, this algorithm will take $T(x) = ux$ time. Since u is constant, ux is bounded from above by cx for all $c > u$, and so $ux = O(x)$ with respect to input x.

The use of $O(f(x))$-snotation in this manner is also applicable to functions not related to running time, but will primarily be used for that in this work. It is important to take note of precisely what an algorithm's running time is taken to be "with respect to", in other words what its variable is. When discussing time complexity of algorithms, one typically does not consider the running time based on the input number itself, but rather as a function $T(n)$ of the "size of the input", usually in terms of its number of bits n, where we let $T(n)$ be the running time for length n. For instance, the decimal number 5 has the binary representation 101_2 which is $n = 3$ bits long, which is why this is often referred to as "bit length".

Going back to our earlier example of an algorithm that performs an action x times for input x, if x is n bits long then we have $x \leq 2^n$. Then see that the running time $ux \leq u2^n = O(2^n)$ with respect to the bit length n. In a scenario like this, we can say that the input is $O(n)$ bits in "size", since it is upper bounded by $n = O(n)$. To make comparisons between algorithms of very different time complexities, there exist several "classes" of time complexity. Let us look at some of the most important ones.

Definition 2.5 *Let an algorithm A with an input size of $O(n)$ have a running time of $T(n)$. We say that*

(i) *$A(x)$ has constant time complexity if $T(n) = O(1)$, meaning it doesn't depend on n;*

(ii) *$A(x)$ has polynomial complexity if $T(n) = O(n^k)$ for some $k > 1$;*

(iii) *$A(x)$ has exponential complexity if $T(n) = O(e^{kn})$ for some positive k;*

(iv) *$A(x)$ has sub-exponential complexity if $T(n) = O_\epsilon(e^{\epsilon n})$ for **every** positive ϵ.*

For inputs being natural numbers, input size means bit length, but for algorithms that take other inputs, like lists, size could be something else, such as the length of the list. Polynomial-time algorithms are considered to be "fast", and is what we usually mean by "efficient", whereas sub-exponential ones are considered to be fairly slow, and exponential ones very slow. Once more, going back to our example about an algorithm repeating an action x times, since $T(n) = O(2^n)$, and $2^n = O(e^{kn})$ for $k = 1$, then that algorithm

is exponential with respect to the bit length, meaning that it is slow. This should be fairly intuitive if we think of it in this manner: for every additional bit in the input, the number of repetitions x will approximately double, meaning that it grows exponentially.

We notice that if we are looking at the pure input x instead of its size n, then if the complexity with respect to the size is $O(f(n))$, then the one for the pure input is $O(f(\log x))$. Reiterating the importance of knowing what our algorithm is "with respect to", our example algorithm has exponential time $O(2^n)$ with respect to bit length, but with respect to pure input x is $O(2^{\log_2 x}) = O(x)$, meaning polynomial-time with respect to the pure input.

With a definition of polynomial-time complexity, we can now formally define a PRNG as to distinguish it from just any bitwise function. We wish to remind ourselves that the purpose of a PRNG is to be an extension of a TRNG; in other words, we take some smaller number of random bits from a TRNG as a seed to generate a larger number of "seemingly random" bits using this PRNG, as described in [7].

Definition 2.6 *An algorithm which calculates a function $G : \{0,1\}^n \to \{0,1\}^m$, $n < m$ is a pseudo-random number generator if it is deterministic, and if for all polynomial-time random algorithms $A : \{0,1\}^m \to \{0,1\}$, an ideally randomly chosen seed $x \in \{0,1\}^n$ and an ideally randomly chosen number in bit form $r \in \{0,1\}^m$, the absolute difference $|\Pr(A(G(x) = 1)) - \Pr(A(r) = 1)|$ is negligibly small.*

In the definition, if we specifically crafted A to be an algorithm to test randomness we could for instance let its output being either 1 or 0 be a judgment on whether the input is truly random or not. However, for the sake of generality, we make no such assumption.
Put another way, the second part of the definition means that any random algorithm A would be essentially just as likely to consider the PRNG output as random, as it would for an ideally random bit sequence. The primary strength of this definition is that we practically assume that there is no efficient random algorithm test that can run on a computer program which can tell apart a PRNG's output from a TRNG's. However, in practice, things are not quite so simple. This is partially because there exists no single universally accepted standard of PRNGs, so PRNGs weaker than this definition are still in use, and partially because there is no way to know for sure whether a given PRNG is truly indistinguishable from TRNG, while older PRNGs are routinely being proven as weaker than expected by new statistical tests.

A few additional things worth mentioning about the use of PRNGs which have little effect on the definition of PRNGs but matter in practice. It is not strictly necessary for a PRNG to use only one seed s_0 as input. It may in fact use several - however, several seeds $s_a, s_b, s_c, \ldots$ can be simply concatenated into a single seed $s_a \| s_b \| s_c \| \cdots = s_0$. For instance, if we have two seeds 111_2 and 000_2, they could be concatenated as $111_2 \| 000_2 = 111000_2$. We may also recall that a primary strength of PRNG is the fact that we can use it iteratively while getting its next input - often called a state s_i - from the last output.

Alternatively, when a PRNG G consists of a family of functions G_k for $k \in \{0, 1 \dots\}$, for instance G_0 may be used to generate the input state for the next iteration, and G_1 may generate the intended random number output. Specifically, a PRNG G taking a state $s_i \in \{0,1\}^n$ as input may contain a function G_0 such that the input state used for the next iteration s_{i+1} is $G_0(s_i) = s_{i+1}$. In such a case, we can say that the PRNG is an algorithm of the form $G(s_i) = (s_{i+1}, y)$, such that $G : \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^{m-n}$, where y is the pseudo-random number meant to be outputted. The length $m - n$ of y is called its *block size*, referring to the "block" of random bits that the PRNG outputs. For a PRNG to be functional, it requires either that that the block size is at least one bit per iteration, or that the output is simply each state s_i.

2.5 Cryptography and CSPRNG

While general PRNGs were described in some detail previously, PRNGs for cryptographic use have been mostly ignored so far, but we will introduce those here. Properties often desired from PRNG that may influence which algorithm one chooses to implement include high speed of generation, greater length of output, low storage size, and such. For *Cryptographically Secure PRNG* (CSPRNG), we have higher demands however, and most PRNGs simply will not be sufficient. This is because CSPRNGs require not only randomness, but also keeping the states secret. Due to our fairly strong definition of PRNGs in the previous chapter, which a lot of PRNGs in practice do not fulfill, there are only a few things to add when we define a CSPRNG.

Whenever cryptographic security is concerned, it is perhaps good to first consider Kerckhoffs's principle, which effectively states that a cryptographic system should be secure even if everything about the system, except the secret key, is public knowledge. The motivation behind this principle is that some attacker may have insider info about the system we use, or could have in other ways deduced parts about it. If the only security lies in keeping the inner workings of the system a secret, then if an attacker would somehow find out about it, the entire cryptosystem will collapse - much like how a secret language is only secret while nobody else knows how to speak it. If such an event were to occur, it would be much easier to simply generate a new secret key, instead of having to create a whole new system that the attacker is unaware of. And so, Kerckhoff's principle is often considered to be a very good basis for constructing cryptographic systems.
A way to interpret it more plainly is as a rule of thumb stating that you should never assume that a potential attacker doesn't understand how your system works. When applied to random number generation states that even if a potential attacker were to hold complete understanding of the inner workings of the generator device/algorithm, it should grant them no help in determining what numbers will be generated by it.

With this principle in mind, we should assume that any potential attacker is aware of what CSPRNG we are using. We can safely call this CSPRNG an algorithm $G : \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^k$, such that $G(s_i) = (s_{i+1}, y_i)$ and the block size of each y_i is

$k > 0$. It should be so that having knowledge of the values $y_1, y_2, \ldots, y_i$ would not give any significant insight into what $s_0, s_1, \ldots, s_i, s_{i+1}$ are, where s_0 is the seed. A more specific property that a CSPRNG in practice should have as an additional safety measure is called *Forward Secrecy* and is described in [7]. This property revolves around what would happen if an attacker somehow getting a hold of a state.

Definition 2.7 *Let $G : \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^k$, $k > 0$ be a pseudo-random number generator, such that $G(s_i) = (s_{i+1}, y_i)$ for all $i \geq 1$. This generator has Forward Secrecy if it fulfills the following property: if the seed $s_0 \in \{0,1\}^n$ is uniformly random, then for any $i \geq 1$, we have that the sequences $(y_1, y_2, \ldots, y_i, s_{i+1})$ and $(r_1, r_2, \ldots, r_i, s_{i+1})$ for some ideally random numbers in bit sequence form $r_1, \ldots, r_i \in \{0,1\}^n$ are computationally indistinguishable.*

In practice, Forward Security implies that if an attacker were to get a hold of a state s_{i+1}, it would not give them any hints on what the previous output blocks $y_1, \ldots, y_i$ were. Forward Secrecy also has a variant in the opposite direction, known as Backward Secrecy or Future Secrecy, in the sense that if Forward Secrecy means that past keys are kept secret in the case of a leak, Future Secrecy means that future keys will remain secret as well.

One example of a type of PRNG that does not have Forward Secrecy or Future Secrecy is a PRNG of block length equal to the state length, and where $G(s_i) = (s_{i+1}, s_i)$, in other words each output block $y_i = s_i$, as the sequence $(s_1, s_2, \ldots, s_i, s_{i+1})$ would be very easy to confirm as correct, assuming the attacker knew the algorithm used.

We let the properties mentioned in this section be sufficient to give an overall idea of what a CSPRNG is, as most CSPRNGs will need to fulfill them. More specific applications of RNG may have more precise, non-universal demands of the used CSPRNG.

3 PRNG in Practice

With knowledge about what precisely we will refer to as a PRNG and CSPRNG respectively, we have an abstract understanding of what such generators are. However, it tells us nothing about how these generators function in practice. In addition to the running time of such algorithms, we don't know what the previously mentioned "statistical tests" actually are, nor what they imply about PRNGs. Let us first lay out a basic form of what PRNGs look like in practice. Rather than a strict definition, it is more of an observation that can be used to easier conceptualize how specific PRNGs work.

As previously stated, the input of some PRNG is called a state, but it is not necessarily just one integer used as a seed for the next number, but can also include other specifications. In particular, the state s_i, other than the state value V_i derived from the previous iteration and used to generate the next random number, often includes some set of parameters P, which could either be in the form of a constant that doesn't change

except for potentially during reseeding, or some key that changes during the algorithm. and often also a reseed counter which is increased by one every time a new number is generated, and upon reaching a specified amount, the algorithm will call for a reseeding. These three can collectively be called a working state $s_i = (V_i, P, reseed\_counter)$. Some PRNGs also allow for optional "additional input", as a sort of soft reseeding for potentially increased security.

There are in fact usually three parts to a PRNG's algorithm. First, initiating the generator, much like how you start the engines before you can get a plane moving. Then, there is a number generator process. Thirdly, there is a reseeding process, which is often more useful than restarting the generator completely (in the same way fueling a plane in mid-air could be useful) as stopping and reinitializing a generator takes time. Initialization basically involves using one or several seeds, as well as other specifications like desired security strength, and only serves to create the first usable working state s_0. The reseeding part is also quite simple, with a working state and new seed as input, possibly with additional input, it outputs a new working state s_{i+1}. We will focus on the algorithm for the RNG itself:

input : Working state $s_i = (V_i, P, reseed\_counter)$, requested random number bit length $reqlen$, additional input $addin$.
output: Next state s_{i+1}, random number r_i of length $reqlen$, status.

Reseed check;
if *reseed_counter is above some limit* **then**
| Break and return a "reseed required" indicator as the status.
end

Additional input check;
if $addin \neq Null$ **then**
| For some function f;
| $s_i \leftarrow f(s_i, reqlen, addin)$
end

Random number generation;
For some function g;
$(s_{i+1}, r_i) \leftarrow g(s_i, r_i, reqlen)$;
Return (s_{i+1}, r_i) with a status of success.

The functions f, g in the algorithm could also be their own algorithms. Now when we have a basic idea of what a PRNG typically looks like, and that there is some dedicated initialization and reseeding procedure involved, we will use this section to look at practical examples, as well as some theory.

3.1 Statistical PRNG

A statistical PRNG is a non-cryptographic PRNG, meaning that we do not necessitate security properties such as Forward Secrecy, but still generates sufficiently uniform and independent bits to be indistinguishable from true randomness by polynomial algorithms.

Some primary ways we test indistinguishability are by testing standard biases, but that is typically not sufficient for modern standards, because of the many ways that dependencies between bits can occur depending on the used PRNG. In addition to standard biases, we can use many other statistical tests to see whether the generator seems to be close enough to the ideal one.

A common way to test generators is by using a test battery, which is a sort of package containing several specific tests which are performed on some output of the PRNG. A classic example of a battery is the diehard set of tests, consisting of various statistical tests such as the Birthday Spacing test based on the statistical phenomenon known as the "Birthday Paradox", to the craps test which simply involves simulating a game of craps to see if the game follows a realistic distribution. The invention of new useful statistical tests, and the existence of certain limitations in its original formulation, make newer test suites more appropriate for modern use. One often used suite is TestU01 [14], which contains a few different batteries of varying levels of strictness. The three most important batteries are "SmallCrush" which performs 10 tests, "Crush" which performs 96 tests, and the most intense battery "BigCrush" which performs 160 tests.

In general, statistical tests use hypothesis testing to determine whether some properties of the generator's output follow the statistical distribution that ideal randomness would provide - and if the generator's output deviates too much, with some significance level, the hypothesis is rejected. This is what it means to fail a statistical test. We previously mentioned the Birthday Spacing test, which we now will look at in further detail as an example of a test, as it is described amongst the diehard tests.

Birthday Spacing test. Assume that a year consists of n days. Now randomly choose m birthdays amongst these n days using the generator, and make a list out of the $m-1$ spacings between consecutive birthdays. E.g., if the first birthday is on the 5th day and the next is on the 8th day, the first spacing on the list will be $8-5=3$. The total number of values on this list is of course $m-1$, and let j be the number of values which occur more than once on the list. If the generator is ideally random, then the distribution of j is approximated by a Poisson distribution $Po(\lambda)$ with mean $\lambda = \frac{m^3}{4n}$. In the original diehard tests, the parameters used are $n = 2^{24}, m = 2^9$, and 500 samples of j are taken. Specifically, a chi-square "goodness of fit" test is used with significance level 0.05.

Example 3.1 *Let $n = 16$, and $m = 8$, so we generate 32 random bits from the PRNG to be tested. We generate*

$$0100 \quad 1100 \quad 1101 \quad 1001 \quad 1011 \quad 1100 \quad 0111 \quad 1100$$

which correspond to, when sorted, $4, 7, 9, 11, 12, 12, 12, 13$. *The spacings between consecutive birthdays are* $3, 2, 2, 1, 0, 0, 1$ *respectively, where the values that repeat are 0, 1 and 2. Therefore,* $j = 3$.

With statistical testing in mind, let us look at a few common statistical PRNGs. When

we talk about the running time, memory use, and which statistical tests that the example generators pass, the source used is [15, pages 6–12].

3.1.1 Linear Congruential RNG

One of the most common PRNG types in use is Linear Congruential RNG [24], which describes any PRNG of the form of

$$V_{i+1} \equiv aV_i + c \mod m$$

with the seed V_0, and a modulus $m \geq 1$, multiplier $0 < a < m$ and increment $0 \leq c < m$. Predictably, the state s_i is made up of $(V_i, (a, c, m))$ with a, c, m constant. The random output is some amount of the most significant bits of V_{i+1}, typically a multiple of 32. The modulo m acts as an upper bound for not only the other constants but also the values V_i, so we denote the bit length of m as N. The generation function consists of only two operations, making it very fast in absolute time, and with a complexity depending on multiplication algorithm used, the best known of which is the Harvey-Hoeven algorithm with complexity $O(n \log n)$. This generator is very fast and has polynomial complexity. It is also fairly compact, with each state s_i only requiring at the very most $4N$ bits of storage.

Parameter Choice. Clearly key is the specific choices of parameters a, c, m, where different types of choices of parameters will have different advantages and drawbacks. Most common is choices with $c \neq 0$, which is what we will focus on here. A common goal when choosing parameters is maximizing the period length, in other words the number of possible values for V_i. For instance, $V_{i+1} \equiv V_i + 1 \mod 4$ with seed $V_0 = 0$ has possible values $\{0, 1, 2, 3\}$ and thus a period of 4, whereas $V_{i+1} \equiv V_i + 2 \mod 4$ with the same seed only has the possible values $\{0, 2\}$ with a period of 2. A greater period length means more possible values for our PRNG, meaning that for equivalent runtime and space use, we are getting significantly more unpredictability, and we can expect to go on for longer without having to reseed. When $c \neq 0$, the Hull-Dobell theorem [11, page 233] tells us that we will have a maximized period length, in other words the period is equal to m, when the following three conditions are true:

Theorem 3.2 *A sequence generated by a Linear Congruential RNG with parameters (a, c, m) with increment $c \neq 0$ has full period m if and only if*

(i) *c is relatively prime to m,*

(ii) *$a \equiv 1 \mod p$ for all prime factors p of m,*

(iii) *$a \equiv 1 \mod 4$ if 4 is a factor of m.*

The proof of this theorem is somewhat long, but in short first shows that this obviously holds for $a = 1$, and then shows that $a \neq 1$ if and only if the conditions hold. Since $a = 1$ may not be a good choice for a multiplier, we may instead want to make sure that m is a non-prime, and considering property (iii), it would expand our possible choices of a if we

should have m be divisible by 4. As a price for its simplicity and speed, LCG suffers from several flaws, albeit oftentimes dependent on the particular parameter choice.

For instance, property (ii) states that $a - 1$ should be divisible by every prime factor of m, however we should take care as to not make a divisible by more factors of m than necessary, lest the number generation becomes more predictable. In *The Art of Programming* [13, page 24], it is simply stated that multipliers of the form $a = 2^k + 1 < m$ for binary computers should be avoided, but for the sake of illustration, we can simulate some such scenarios to see exactly what goes wrong. Let us use some parameters that fulfill the Hull-Dobell theorem, for instance $a = 2^k + 1, m = 64, c = 3$ for $k = 2, 3, 4$ and 5, such that they can have full period. If the advice should hold in this case, we would expect the randomization for $a = 2^2 + 1 = 5$ to look the most random of the bunch.

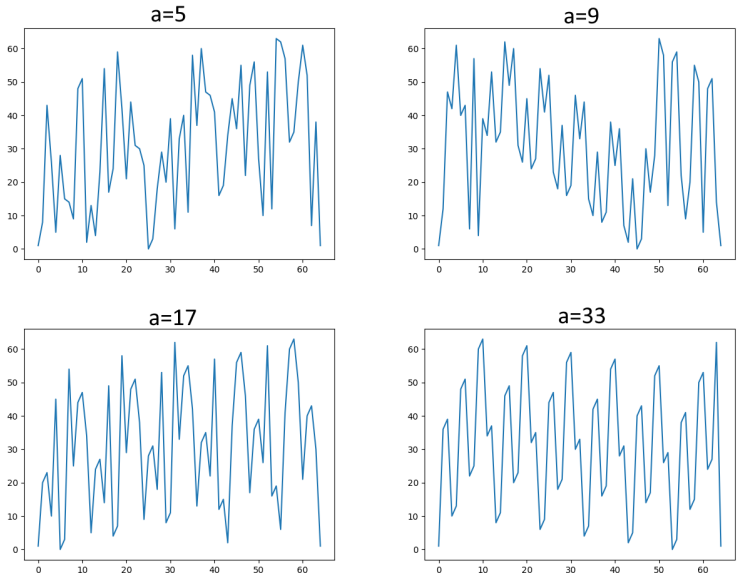

Figure 1: LCG with $c = 3, m = 64$ and seed $s_0 = 1$, for the purpose of comparing the multiplier parameter choices $a = 5, 9, 17$, and 33.

We observe in Figure 1 that indeed, as the recommendation stated, the LCG with $a = 5$ appears to be the most unpredictable, whereas those a such that $a - 1$ is "unnecessarily divisible" by factors of m seem to have very clear patterns that deviate from what we would expect from uniform, independent randomness.

Statistical Tests. What we just saw is one way that bad parameter choices could doom an LCG. Even when good parameters are chosen, it often happens that LCGs fail several types of statistical tests in commonly used suites, and due to its tendency to form "patterns" is considered particularly badly suited for data sampling in larger numbers of dimensions as these patterns become very obvious for the amount of random numbers

required for arbitrary-dimensional sampling. This issue in LCGs is primarily tested by statistical testing suites using what is known as a Spectral Test, which involves checking how obvious the lines or hyperplanes are in the output when plotted in several dimensions. Even strong LCGs will typically fail this test if their output is too patterned.

However, for particularly large bit length parameters such as $N = 96$ with only the 32 most significant bits outputted as the RNG, which is the one featured in the aforementioned paper, LCGs can pass even the TestU01 suite's notorious BigCrush battery. While quite fast even at that level of quality, there are other PRNGs which also pass BigCrush but use significantly shorter parameters.

Memory Use. The fact that a $N = 96$ bit LCG, assuming no additional methods of saving storage space are being utilized, uses space of up to $4N = 384$ bits for 32 random bits per iteration - a ratio of $12 : 1$. Specific implementations of LCGs may not contain the parameters in the state and are instead stored directly in the generator's code, and in such a case the state has a size of only up to $N = 96$ bits and a ratio against the output of only $3 : 1$. Even then however, there are other PRNGs that pass all BigCrush statistical tests while requiring even less memory.

3.1.2 Mersenne Twister

The most widely used statistical PRNG is the Mersenne Twister, which is the default PRNG in many programming languages such as Python and R as well as scientific software such as MATLAB. It gets its name from the fact that its period length is what's called a Mersenne prime, which is a prime of the form $2^n - 1$ for some integer n. It is a linear-feedback shift register (LFSR) type PRNG, roughly meaning that it uses bitwise functions such as shifts and XOR-operators to generate new states. Unlike general formula PRNGs such as linear congruential generators, the Mersenne Twister PRNG is specific and has fixed parameters, but it does have a few variants. The standard 32-bit output variant is known as MT19937, and while there are versions with longer bit length outputs as well as specialized use variants, we will be focusing on this standard MT19937 generator as it serves the most general purpose use.

Memory Use. The Mersenne twister MT19937 generator uses a series of computations, including matrix calculations, to generate random numbers of bit length 32. Unfortunately, due to the nature of the generator, the state consists of an $n \times w$ array, containing n values of bit length w. These coefficients are specifically $w = 32, n = 624$, meaning that each state s_i consists of $32 \cdot 624 = 19968$ bits. There is a reason for this sort of state, which is that it allows the generator to enjoy an incredibly long period, namely $2^{19937} - 1$, adding more unpredictability to the 32-bit random number output and avoiding some problems typical to short periods. There exists an alternative to MT19937 known as TinyMT, which requires a mere 127 bits of state rather than $2.5KiB$, but has a period of "only" $2^{127} - 1$, which is a size comparable to the period of other similarly performing PRNGs.

Running Time. Since the Mersenne Twister uses fixed coefficients, the output is precisely a 32 bit random number, making time complexity somewhat meaningless, but since generating 64 bits takes twice as long as 32 bits, its time complexity is $O(N)$ for number of bits equal to N. Instead, we can compare its runtime to other PRNGs. In the paper, [15, page 6] MT19937 took a little over 6 seconds to run SmallCrush, of which a little over 4 seconds are constant time taken to perform the tests themselves, meaning that it took about 2 seconds to generate the numbers to be tested. In comparison, an LCG with 96-bit keys and 32-bit outputs (which, recall, passed all BigCrush tests) took a little less than one second. Overall the performance of MT here is fairly average amongst the tested generators.

Statistical Tests. So far, we have seen that the standard MT19937 takes up quite a lot of space in exchange for a longer period. Unfortunately, this does not make it very statistically random. This version of the Mersenne Twister not only fails BigCrush, but it also fails Crush. Since its parameters and algorithm are preset and cannot be changed, if you aim to use a PRNG which does pass these two test batteries, you will simple have to use a different one.

3.1.3 XorShift.

There is a class of PRNGs known collectively as Xorshift RNGs [23], which much like the Mersenne Twister PRNG is an LFSR PRNG. XorShift PRNGs are characterized by low memory use and fast processing speed, often at the cost of some of the less significant bits being not entirely random. One such generator is XorShift64*. While not the best performing within the family (which would probably be the xoshiro/xoroshiro generators) XorShift64* was featured in the same test as the previously mentioned PRNGs in the form of a 32-bit RNG from 64-bit state, removing the weaker bits. Much like the Mersenne Twister, its parameters are fixed, although it does not require any special operations and its code occupies very little space.

Specifications. Not only is its state small at a mere 64 bits, meaning a $2:1$ ratio between state and output size, but it is also fast. While time complexity is not particularly meaningful here, its runtime generating numbers for the SmallCrush battery (excluding the time taken performing statistical tests) was just about one second, which was slower than the 96-bit LCG with 32-bit outputs, but almost twice as fast as the Mersenne Twister.

Many XorShift generators do not pass every test in BigCrush, such as the regular XorShift64* due to the low significant bits being weak (low entropy). The implementation of XorShift64* with 32-bit outputs however does in fact pass BigCrush, and it manages this precisely because it does not output its weak bits, which however leads to a lower rate of generating random bits [15, page 7].

3.2 Cryptographically Secure PRNG

While statistical PRNGs are quite useful for general PRNG, they generally lack some previously discussed properties desired for cryptographic contexts, typically because designing a PRNG around such properties tends to complicate the algorithm and increase its runtime. For this reason, it's common to develop specific PRNGs for only cryptographic use, and some such CSPRNGs will be featured in this section.

As an example of a PRNG that is unsuitable for cryptographic use, we need to look no further than LCG. For instance, take $a = 5, c = 3, m = 64$, and it outputs the whole generated number $r_i = 5s_i + 3 \mod 64$, where thus $s_{i+1} = r_i$. If an attacker gets a hold of r_i then they can calculate every successive value on their own, since $r_{i+1} \equiv 5r_i + 3 \mod 64$. Even if we choose to only output, say, the rightmost 3 bits of data, if the attacker finds one r_i then there will be $2^3 = 8$ possible values of s_{i+1}. For instance with $r_i = 011_2$, then the candidates c for s_{i+1} include $000011_2, 001011_2, \ldots 111011_2$. If the attacker also gets a hold of, for instance r_{i+1}, then the number of candidates are further reduced to only the candidates c such that $r_{i+1} \equiv 5c + 3 \mod 64$, which in practice is typically a drastic decrease.

Since CSPRNGs are especially intended for situations where secrecy must be guaranteed, it is important that the CSPRNG (and its creator) is trustworthy. Poor design is undesirable in its own right, but even worse would be a maliciously designed PRNG that could compromise security. Perhaps due to this fairly higher standard of trust demanded of the creator, some organizations have taken it upon themselves to make standardizations of CSPRNG, which must follow strict specifications to gain their approval. One such organization is the US National Institute of Standards and Technology (NIST) which has published a large number of documents, aiming to standardize various systems within technology, including cryptographic RNG, as in their special publication *NIST SP 800-90A* [2] and subsequent revisions.

There are several types of CSPRNGs depending on what sort of algorithm they employ to generate random numbers, where two common types include hash-based and counter-mode block ciphers. For the sake of simplicity, we will be looking at two hash-based CSPRNGs documented in the NIST standard, since it provides a lot of details into the algorithms and discussion of their differences, and two generators based on the same principles of randomization are easier to compare directly.

3.2.1 NIST Hash_DRBG

The first CSPRNG featured in NIST's previously mentioned publication is the hash-based algorithm Hash_DRBG, where DRBG stands for deterministic random bit generator and is another term for PRNG. A hash function is a function which maps arbitrarily large inputs to specific-length outputs, and many such functions intentionally map values in a very unpredictable manner for cryptographic purposes such as digital signatures, making hashes like this an interesting possibility for use in RNG. Hash functions deemed

suitable for such situations are sometimes called cryptographic hash functions. They are expected to have specific properties such as unpredictability, non-reversibility and collision resistance, meaning that finding two inputs that output the same hash value is very difficult.

The NIST standard does not mention what specific hash function should be used for the algorithm, and instead allows the implementer to choose any FIPS-approved cryptographic hashing function depending on for instance how secure it needs to be, and how large they want their randomly generated numbers to be. Many aspects of the algorithm, such as its security, rely overwhelmingly on the hash-function and its unpredictability, so choosing the right function is important. One common choice of hash function is the SHA family of functions, which is what NIST lists as standard.

We will not go through the reseeding check or additional input check as the former doesn't affect the generated number and the additional input check only involves a fairly trivial way to modify the used input value. For some hash function `Hash(x)`, the outputs of which are *blocklen* bits long, Hash_DRBG random number generation functions as follows:

input : Working state $s_i = (V_i, C, reseed\_counter)$, requested random number bit length *reqlen*, additional input *addin*.
output: Next state s_{i+1}, random number r_i of length *reqlen*, status.

Reseed check, Additional input check;

Random number generation;
$data \leftarrow V_i$;
$W \leftarrow NULL$;
for $i \leftarrow 1$ **to** $\lceil \frac{reqlen}{blocklen} \rceil$ **do**
  $W \leftarrow W \| \texttt{Hash}(data)$;
  $data \leftarrow (data + 1) \mod 2^{blocklen}$
end
$r_i \leftarrow \texttt{leftmost}(W, reqlen)$;
$V_{i+1} \leftarrow (V_i + C + \texttt{Hash}(\texttt{0x03} \| V_i)) \mod 2^{blocklen}$;
$s_{i+1} \leftarrow (V_{i+1}, C, reseed\_counter)$;
Return (s_{i+1}, r_i) with a status of success.

Algorithm 1: Hash_DRBG generator algorithm.

To put the main generation step into words, the algorithm uses a hash function `Hash(x)`, the value V_i, and a counter counting up to generate enough *blocklen* length blocks, which are concatenated, such that you can output a *reqlen* length random number consisting of the *reqlen* leftmost bits. For instance, if the hash function produced random bits in blocks of 4, and the desired random output is 10 bits long, this algorithm would generate three blocks totaling in 12 random bits, and then output the 10 leftmost bits as the random number r_i.
Afterwards, the hash function is then used to generate the value V_{i+1} for the next state. For reference, the `0x03` is hexadecimal representation for the number 3, which in binary

becomes 00000011_2. The notation of `0x0N` is also used in other algorithms, but not in such a way that hexadecimal notation will require further explanation.

Choice of hash function. We can now understand why it's so important that our hash function has to be particularly good: since the algorithm relies on a counter, using a hash function with outputs that aren't very unpredictable for inputs "close to" each other will generate subsequent blocks that don't seem very random, which not only means the r_i won't be very random-looking, but also that an adversary looking at the blocks of r_i in sequence could get additional information to brute-force the working state, compromising security in the process.

Running time. Since the main chunk of the algorithm involves generating a fixed number of blocks with quickly calculated inputs, this algorithm is easily parallellizable, meaning that each of these blocks can be calculated at the same time by separate processes. The speed of the algorithm ultimately becomes most dependent on the speed of the `Hash(x)` function, which means that its running time can be significantly lowered if implemented well. Assuming that the loop step can be fully parallellized for some fixed working state and requested number of random bit output *reqlen*, if the hash function has an expected running time u, the total run time of Hash_DRBG will scale along $2u$ with respect to the hash function's running time.

Implementation. Since the only special part that this CSPRNG requires is a cryptographic hash function, if there is already a strong hash function stored in the system and accessible by Hash_DRBG, the algorithm can be quite cheap to implement since the hash function does not have to be implemented just for this CSPRNG.

We will compare the properties discussed here with the next featured CSPRNG, which despite its difference is also hash-based.

3.2.2 NIST HMAC_DRBG

This is an alternative to Hash_DRBG featured beside it in the NIST publication. It is somewhat more complicated and uses two different functions. One is a hash function `HMAC(K, x)`, where HMAC refers to a family of hashing algorithms defined in the standardization FIPS 198, and was originally intended for cryptographic message authentication. It uses a secret key K and an message x as input, where K is stored in the working state of the algorithm. Additionally, it involves a sub-algorithm HMAC_Update which will be covered after the main algorithm.

input : Working state $s_i = (V_i, Key, reseed\_counter)$, requested random number bit length $reqlen$, additional input $addin$.
output: Next state s_{i+1}, random number r_i of length $reqlen$, status.

Reseed check, Additional input check;

Random number generation;
$temp \leftarrow NULL$;
while *temp length* $<$ $reqlen$ **do**
  $V_i \leftarrow \texttt{HMAC}(Key, V_i)$;
  $temp \leftarrow temp \| V_i$
end
$r_i \leftarrow \texttt{leftmost}(temp, reqlen)$;
$(Key, V_{i+1}) = \texttt{HMACUpdate}(addin, Key, V_i)$;
$s_{i+1} \leftarrow (V_{i+1}, Key, reseed\_counter)$;
Return (s_{i+1}, r_i) with a status of success.

Algorithm 2: HMAC_DRBG generator algorithm.

This algorithm looks overall fairly similar to the Hash_DRBG algorithm. We can see that one big difference from Hash_DRBG is that the loop that generates the random bits does not use a counter to create the different blocks, and that they are instead generated in a serial manner. This means that HMAC_DRBG is not parallellizable like Hash_DRBG. The part left out is the function we called `HMACUpdate(a, K, V)`:

input : Additional input $addin$, parameter Key, value V.
output: New parameter Key, value V.

$K \leftarrow \texttt{HMAC}(Key, V \| \texttt{0x00} \| addin)$;
$V \leftarrow \texttt{HMAC}(Key, V)$;
if $addin = NULL$ **then**
  return Key, V
end
$K \leftarrow \texttt{HMAC}(Key, V \| \texttt{0x01} \| addin)$;
$V \leftarrow \texttt{HMAC}(Key, V)$;
Return Key, V

Algorithm 3: $HMACUpdate(x)$ value updater algorithm.

This algorithm is also used in the instantiation and reseeding steps of HMAC_DRBG, and is basically used to generate the next step's value and key.

Choice of hash function. The choice of the hash function `HMAC(x)` is much more restricted here than it was for Hash_DRBG. As unintuitive as it might at first seem, however, our choice of hash function is actually less strict than in Hash_DRBG, due to a quirk in the random bit generation. Since there is no counter used in the generation loop, and we instead generate each block in a serial manner, the hash function does not have to be quite as unpredictable for sequential inputs, which is something we had to take into consideration when we chose a hash function for Hash_DRBG. The fact that all we really need is a FIPS-

approved hash function for message authentication means that, if we already have a system that uses message authentication, we can just reuse the cryptographic hash function that we used in that system, meaning that we could potentially save storage space by choosing HMAC_DRBG.

Running time. The flip-side of not using a counter for our generation loop is that it is not parallellizable. Instead, we will need to iterate the loop $N \geq 1$ times, where the exact value of N depends on the length of the blocks generated by `HMAC(x)`, as well as on our requested number of bits *reqlen*, meaning that for every random number r_i we wish to generate, we will need to call our function `HMAC(x)` a total of $N + 1 \geq 2$. Assuming that `HMAC(x)` is expected to take time u to run, then the running time will be at least $3u$, which only occurs when *reqlen* is at most the number of bits that `HMAC(x)` outputs, and we are not using additional input. In practice, according to the publication where they are both specified, the algorithm HMAC_DRBG takes twice as long to generate random bits as Hash_DRBG. It does note, however, that both algorithms are still fairly fast, so depending on the situation, the difference in speed may not be significant enough to take into consideration.

3.3 Bad CSPRNG

From how we have discussed CSPRNGs earlier, it could seem that as long as we base a PRNG on a cryptographic function, and do the bare minimum to obfuscate the output r_i from the value V_i, we have a good CSPRNG. Unfortunately, things are not that easy. Even when a CSPRNG seems to have been designed with security taken into account, it might not be fit for practical use due to some bug or inherent flaw. It might then be interesting to ask oneself, what exactly could a bad CSPRNG be?

Some obvious flaws that could make an algorithm unsuitable for cryptographic use are lacking important features, failing many statistical tests, being exceedingly slow or taking up too much storage. Such algorithms will for instance include most statistical PRNGs, as well as many CSPRNGs based on number-theoretical one-way functions like multiplication on elliptic curves which are typically very slow. Other than such obvious drawbacks, CSPRNGs may have particular bugs that can compromise security in specific scenarios. Many such bugs do not stem from the use of bad algorithms, but rather bad implementation, meaning that this can happen even to good CSPRNGs.

In a nightmare scenario, a CSPRNG could have been intentionally designed with a security flaw that could be exploited by its creators to compromise the working state for those with the required knowledge. There is one infamous case where the public consensus seems to be that this occurred, and it is none other than the (now obsolete) Dual_EC_DRBG featured in earlier versions of the NIST SP 800-90A standardization based on mathematics of elliptic curves. Due to the significance of this case within the world of computer security, we will dedicate a section to explain how the alleged backdoor

functions, as that particular aspect is not only very well known but also surprisingly simple, assuming one is at least somewhat familiar with mathematics of elliptic curves.

3.3.1 Mathematics of Elliptic Curves

An *elliptic curve* is a set of points (x, y) that satisfy the equation $y^2 = x^3 + ax + b$, for integers a, b. Additionally, we also require the curve to be non-singular, meaning there are certain types of points that we do not want on the curve, such as self-intersections. This is luckily easy to check, as an elliptic curve is non-singular if and only if $4a^3 + 27b^2 \neq 0$. The curve can be defined with x, y, a, b over any field, and typical choices are real numbers $\mathbb{R}$, complex numbers $\mathbb{C}$, and finite fields F_p where p is a prime.

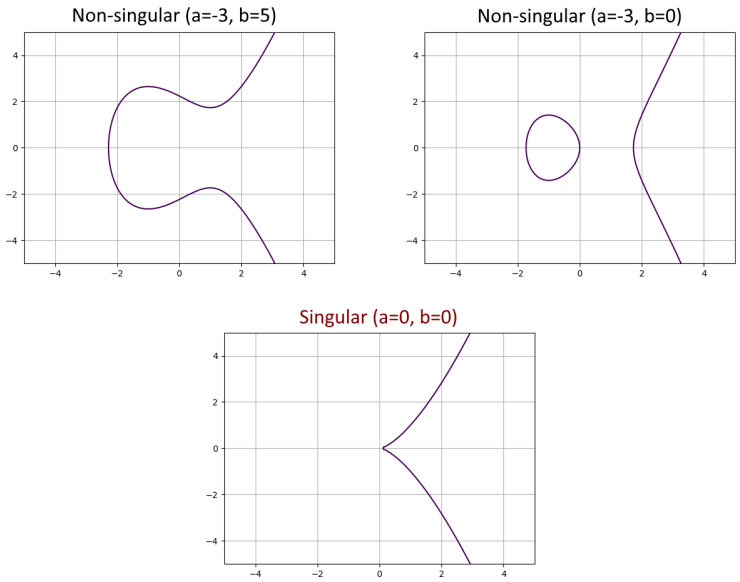

Figure 2: Three elliptic curves defined over the reals, the third curve is singular due to its cusp at the origin.

We construct the abelian group of the points on an elliptic curve. What abelian refers to is practically that addition works like it does for integers, where for instance the order you add elements doesn't affect the sum. We define such an addition between some P, Q, over the reals, informally as the point R' gained by first drawing a line through P, Q, finding the third point $R = (x, y)$ on the curve that the line intersects, and then flipping it over the y-axis to get $R' = (x, -y)$. Before we define it more formally, note two special cases.

First is if we add some $P + P$ together. The answer is simple, namely to let the line be the tangent line at the point P, which will then intersect one other point on the curve Q.

Secondly, what about adding two y-opposite points $P + P'$? Algebraically, there is

no third point on the curve that it intersects. The solution is to consider P' the additive inverse of P, and "including a point at infinity" called O as the additive identity. In this manner, $P + P' = O$. This case also includes when adding $P + P'$ where $P = P'$, which occurs at any "edge points" on the curve, which occurs when $y = 0$.

We more formally describe the addition of points on the elliptic curve with identity point O as follows: [10]

Definition 3.3 *Consider an elliptic curve given by the equation $y^2 = x^3 + ax + b$ for non-singular choices of a, b. We let the group over this curve contain the points $P = (x_P, y_P)$ on the curve, and extend it with an identity element O, such that the inverse of P is $P' = (x_P, -y_P)$. We define the addition of any two points $P = (x_P, y_P), Q = (x_Q, y_Q)$ on the curve as $P + Q = R$ as follows:*

(i) If $P = Q'$, then $P + Q = P + P' = O$;

(ii) If $Q \neq P'$ but $P = Q$, then $P + Q = R = (x_R, y_R)$, where $x_R = k^2 - x_P - x_Q$, $y_R = k(x_P - xR) - y_P$, where $k = \frac{3x_P^2 + a}{2y_P}$;

(iii) Otherwise, exactly the same as for $P = Q$ but with $k = \frac{y_P - y_Q}{x_P - x_Q}$.

Remark 3.4 *Addition of points on an elliptic curve is both commutative and associative.*

In cryptographic contexts, we don't assume the coordinates of the points to be reals, as we define the curve over a finite field F_p modulo a prime p, containing the integers $0, 1, \ldots, p-1$ and following normal modular addition, subtraction, multiplication and division rules. The only difference this makes for our elliptic curve addition is that arithmetic is done modulo p, and that division is defined as the multiplicative inverse modulo p, in other words for some element $x \in F_p$, $1/x = y$ where $xy = 1 \mod p$. Since p is a prime, we know that all such non-zero elements have an inverse.

The final aspect to know about the elliptic curve is the scalar multiplication of points, which is that for some positive number n and point on a curve P, we have that $nP = \sum_1^n P = P + \cdots + P$, that $(-n)P = n(-P) = nP'$, and of course that $0P = O$.

This brings us to the crux of elliptic curves which makes them cryptographically interesting: the *elliptic curve discrete log problem* (ECDLP), which states that for any two distinct P, Q on a curve over some very large field, finding the n such that $nP = Q$ is very difficult, assuming there is such an n. A P which can generate all other points on the curve is called a primitive root, and it is only if the number of points on the curve is a prime that all points on the curve are primitive roots. The point at infinity is never a primitive root. This sort of one-way function $f(x) = xP$ is the reason why elliptic curves have been used in cryptography, such as in the form of elliptic curve Diffie-Hellman exchange.

3.3.2 NIST Dual_EC_DRBG (Obsolete)

We are now prepared to look at the now defunct EC algorithm previously included in the NIST SP 800-90A standard. It involves several functions which will be explained after the algorithm, and its working state primarily consists of a state value s and two points P, Q. The standard described in the publication requires the use of one out of a few specific curves with parameters $(a, b, p, n, seedlen)$, and each specific curve has two specific points P, Q which must be used. If only one curve is used, then the parameters need not be included in the working state, so let us assume that they aren't. As a matter of fact, our point choice P, Q is not meant to be kept secret, and is constant throughout use, so we can presume those are not a part of the working state either, and so the working state consists of only the state value s_i.

Again, we will not look at the additional input and reseed checks, and only consider the generation part of the EC algorithm. One function worth explaining before the algorithm is φ_x. All $\varphi_x(P)$ does is take a point as input, and outputs its x-coordinate x_P in its binary representation.

input : Working state s_i, requested random number bit length $reqlen$, additional input $addin$.
output: Next state s_{i+1}, random number r_i of length $reqlen$, status.

Reseed check, Additional input check;

Random number generation;
$temp \leftarrow NULL$;
$S_0 \leftarrow s_i$;
$S_0 \leftarrow S_0 \bigoplus addin$;
for $k \leftarrow 1$ **to** n *such that* $n = \lceil \frac{reqlen}{blocklen} \rceil$ **do**
  $S_k \leftarrow \varphi_x(S_{k-1}P)$;
  $R_k \leftarrow \varphi_x(S_kQ)$;
  $temp \leftarrow temp \| \texttt{rightmost}(R_k, blocklen)$
end
$r_i \leftarrow \texttt{leftmost}(temp, reqlen)$;
$s_{i+1} \leftarrow \varphi_x(S_nP)$;
Return (s_{i+1}, r_i) with a status of success.

Algorithm 4: Dual_EC_DRBG generator algorithm.

There is quite a lot going on in this algorithm, but let us first point out the functions used. First off, $\bigoplus$ is the bitwise XOR operator, which looks at each pair of bits from each number and if they are the same then the output is a 0 in that place, and if they are different then the output is 1 in that place. As an example, $0111_2 \bigoplus 0001_2 = 0110_2$.
As for the $\texttt{rightmost}(input, bits)$ function, the constant $blocklen$ is specific to the curve used, and is calculated as $seedlen - 16$ where $seedlen$ is the bit length of prime p in bits - making $seedlen$ the length of the states and random numbers derived as coordinates from the curve's points. This means that the output of $\texttt{rightmost}(R_k, blocklen)$ is the

entire coordinate R_k except for its 16 leftmost bits, which is likely a design choice made to obfuscate what R_k is, and by extension, the state S_k that generated R_k. It should be noted that *seedlen* is very large, between 256 and 512 going by the curves that the publication demands be used.

3.3.3 The Dual_EC_DRBG Problem

To assure users that a cryptographic algorithm with specific parameter choices is made in good faith, the creators either properly justify the parameter choices, or in the case that the parameters are randomly chosen, show the method with which they were chosen. No such justification was officially made for the specific curves and points defined for the NIST Dual_EC_DRBG standard, which is where suspicion arose.

The alleged backdoor in the algorithm as described in [19] is based on the idea that P, Q were indeed specifically chosen such that $eQ = P$, where e is known by either the person who installed the backdoor or by some other potential attacker. For the sake of keeping the example simple, we will make a few assumptions. This situation can be seen as a sort of worst case scenario, and the potential for exploiting the backdoor exists even without making these assumptions, but the exploit becomes less straightforward. Let us assume $addin = NULL$, since the standard explicitly states that additional input use is perfectly optional. We also assume that the desired number of bits is exactly *blocklen*, in other words the length of one single generated R. We will naturally also presume that any potential attacker is aware of the curve as well as what points P, Q are being used, by Kerckhoff's principle.

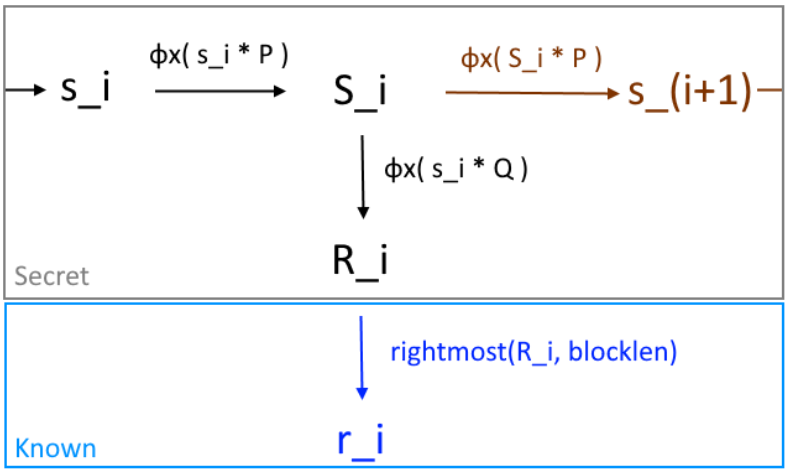

Figure 3: Graph of how Dual_EC_DRBG functions under our assumptions.

With the assumptions we previously made, the algorithm can be simplified as a set of three

calculations when a generation call is made, for an input working state s_i:

$$S_i = \varphi_x(s_i P), \qquad R_i = \varphi_x(S_i Q), \qquad s_{i+1} = \varphi_x(S_i P)$$

as well as, of course, the output random number $r_i = \texttt{rightmost}(R_i, blocklen)$.

Assume that the attacker has access to some r_i. We remind ourselves that a random number was generated via $r_i = \texttt{rightmost}(\varphi_x(s_i Q), blocklen)$, and $s_{i+1} = \varphi_x(s_i P)$. The first step is to get some idea of what R_i could be. Since r_i in practice is several hundred bits long, and is only 16 bits away from the x-coordinate R_i with $2^{16} = 65536$ different values to go through, there aren't that many numbers that R_i could be. If we let $R_i = B \| r_i$ where $0 \leq B < 2^{16}$, we "only" have 65536 values to check. While this sounds like a lot, it is so much less than the total at least around 2^{256} different values R_i could have been if we did not know r_i, and it can be reduced by also ignoring any candidates greater than p, as those are not possible x-coordinates for R_i. We can put in all the possible R_i values into the elliptic curve equation and use modular square roots to see if there is some corresponding y-coordinate such that $A = (R_i, y)$ is on the curve, further reducing the number of possible values of R_i. Out of all these points, one is definitely $(\pm S_i Q)$, and since the attacker knows of an e such that $eQ = P$, they can for each point A simply calculate $\varphi_x(eA)$, knowing that for A such that $A = S_i Q$,

$$\varphi_x(eA) = \varphi_x(e(S_i Q)) = \varphi_x(S_i(eQ)) = \varphi_x(S_i P) = s_{i+1}.$$

If we put all $\varphi_x(eA)$ in a list, we are bound to have gained some duplicates, meaning we have further reduced the number of possibilities, and we now have a list with significantly fewer than 65536 candidates for what s_{i+1} is, reduced by a huge order of magnitude over the (depending on the curve) at least about 2^{256} possible states that the attacker would otherwise have to consider if they were to try and bruteforce the result. Worse is, since the attacker has access to P, Q, they can perform the same calculations on their list of candidates as the generator does for its state, and so for every consecutive random number $r_i, r_{i+1}, \ldots$ that the attacker gets a hold of, they can cross-check which of the states in its list of states yields that particular random number. This means that they could calculate the state of the generator with as little as only two of the random numbers generated by this instantiation of the algorithm, and in fact, they do not have to be consecutive random numbers as long as the attacker has an idea of how many iterations are between them. At that point, all future states until reseeding or de-instantiation will have been compromised, taking the CS out of this CSPRNG. There are various ways to optimize this method, but this is how it works in its simplest form.

3.3.4 Demonstration

For the purpose of a demonstration, I have prepared a situation as previously described, with a specifically chosen curve and parameters. Due to limited computing power, we will be using much smaller numbers than one would in practice. We look at the curve wherein $a = -50, b = 200, p = 4091$ where our states are maximum $seedlen = 12$ bits

long, with $r_1 = \texttt{rightmost}(R_1, 6)$, meaning that $R_1 = B \| r_i$ for $0 \leq B < 2^6$. Without any outside knowledge, the state s_2 could be anything between 0 and $p - 1 \approx 2^{12}$. We let $P = (2937, 1854), Q = (4, 8)$, and the attacker knows that $P = 20Q$.

We can let our secret state be $S_1 = 5$, which of course the attacker won't know. This gives $s_2 = 962$, which is what the attacker is trying to figure out. We end up with $R_1 = (3052, 1246)$, and the 6 rightmost bits of 3052 yields $r_1 = 44 = 101100_2$, which is the random output the attacker sees. With a script, the attacker can check every possible $x = B \| r_i$ for all 6 bit long values $B = 0_2, 1_2, 10_2, \ldots, 111111_2$ such that $B \| r_i < p$, to see which fulfill the elliptic curve equation $y^2 = x^3 - 50x + 200 \mod 4091$ with some solution of y. We can find such y-coordinates via modular square root, which can be done using one of the several available methods, such as the Tonelli-Shanks algorithm which will not be discussed here.

Now we have a list of points $A = (x, y)$ such that A lies on the curve, and the x-coordinate is of the form $B \| r_i < p$. Since we know that $P = 20Q$ then, for all these A, we construct a list consisting of every element $\varphi_x(20A)$. This list contains all possible candidates for the state s_2 based on r_1. This list contains 36 elements which is about half of $2^6 = 64$, which were all the possibilities before filtering based on the points existing on the curve. Indeed, if we check, we can see that the true $s_2 = 962$ is indeed amongst our 36 candidates.

However, 36 candidates may not be few enough for the hacker, and so let us assume they came across r_2. All they need to do is, for all candidates c in the list of possible values of the state s_2, perform $\texttt{rightmost}(\varphi_x(\varphi_x(cP)Q), 6)$, which is the random number r_2 that this candidate would have generated if it actually were the state s_2. We do this for all the candidates, compare them to the actual r_2, and we remove any elements where the random numbers don't match. This time, our list shrinks down to a mere two candidates $1044, 962$, meaning that the state s_2 must be one of these. In fact, we see that the actual state $s_2 = 962$ is in that list of candidates.

As a proof of concept, this demonstration displays the danger of this potential exploit, where the knowledge of just a small amount of random numbers could be enough to completely compromise the working state of this CSPRNG. Naturally, in practice, doing this is quite a bit more resource intensive and may require some extra considerations when it comes to loops and *reqlen* different from *blocklen*, but the problem itself does not go away. A very potent way of either minimizing or even completely eradicating this issue could be to either always or at least very frequently use unpredictable additional inputs when generating numbers. The problem with this, of course, is twofold.

Firstly, where would this frequent "unpredictable" additional input come from? Most likely, to be at all sustainable, we would have to use a TRNG at every step to generate said additional input, and at that point we should perhaps question why we are choosing to rely so heavily on TRNG when we are using an algorithm (a PRNG) with the explicit

purpose to be less reliant on TRNG.
Secondly, we should ask ourselves why we are using a PRNG that requires near constant additional input only to fix an otherwise unavoidable exploit in the code, when we could simply use a different PRNG without this type of exploit. It certainly does not help that Dual_EC_DRBG is much slower than any of the other CSPRNGs featured in the same publication.

Now, with some knowledge of the inner workings of PRNGs both good and bad, we move on to what they are used for.

4 Randomness in Algorithms

When thinking of algorithms, what first springs to mind might be a deterministic algorithm, such as calculator operations or such. Specifically, a deterministic algorithm is one which depends only on the input, with no random variables involved. In particular, given a specific input, we expect them to have the same running time and yield the same output every time.

Many algorithms however, as discussed previously, make use of random numbers in their process. Such algorithms can be formalized as random variables, in which case we more precisely interpret the input of an RNG as a random variable. We are already aware of some of the big perks of derandomization, including predictable runtime as well as by definition reducing reliance on the random number resource.

Much like what we did when we compared the exclusive use of TRNG against de-randomizing RNG to consisting of mostly PRNG, it's worth investigating how and when derandomization can be viable. Random algorithms are so diverse in functionality and in how they use randomness, that it is necessary to look at several different examples to gain an understanding of how random algorithms may be derandomized, and how it may affect their performance.

Before beginning, let us have a short discussion of how complexity functions for random algorithms, as there are two primary types of random algorithms: *Las Vegas algorithms*, which expect to output a "correct" result after a potentially unknown but finite amount of time, and *Monte Carlo algorithms*, which most likely make a "correct" output but could make an incorrect output with some probability after a fixed (or bounded) amount of time [1, pages 9–10].

A Las Vegas algorithm is typically more desirable than a Monte Carlo algorithm due to having a 100% success-rate, however for some problems using a Las Vegas algorithm is either unreasonably time-consuming when there is a much quicker Monte Carlo algorithm that simply yields a "good enough" output, or the problem simply cannot be solved exactly in a finite amount of time.

A key difference between these two types is how exactly they depend on random-

ness. Las Vegas algorithms will always finish with a correct solution, but their running time can depend on whether they get "good" random numbers or not. Monte Carlo algorithms will end after a specific amount of time, but not always give a correct solution, and how good the solution is depends on getting "good" random numbers. For this reason, we consider the time complexity of Monte Carlo algorithms to be based on the upper bound of time it will take. On the other hand, we consider the time complexity for Las Vegas algorithms to be something of a random variable $T(I)$ for an input I that depends on the RNG. For such algorithms, we may divide running time complexity into best-case, average-case, and worst-case complexities, where each of these is the expected value $E[T(I)]$ for inputs I that can be considered best-case, average-case and worst-case inputs. Typically, the worst-case and average-case running times are of greatest interest.

This means, of course, that when we compare the time complexity of Las Vegas algorithms against Monte Carlo or deterministic algorithms, we are speaking of the same time complexity concept as dependent specifically on the input. For instance, with non-input parameters and coefficients fixed, worst-case scenarios in Las Vegas algorithms are typically much more dependent on the random number generation than on the input, whereas Monte Carlo worst cases are mostly or entirely based on the input, and deterministic worst case scenarios are always based entirely on the input.
Something similar also occurs in regards to the error, which often is denoted as $O(f(N))$ for a decreasing function $f(N)$ as N increases, where the "error" that deterministic algorithms may have in their output depends entirely on the input and parameters. Monte Carlo algorithms however have their error $O(f(N))$ depend partially on the input and parameters, as well as on the random numbers, whereas time complexity is not a function of the random numbers.

4.1 Simple Example

To illustrate the difference between deterministic and random algorithms, we will look at a problem, two different random algorithms we could use to solve it, as well as their derandomized variants. This will serve as a practical example to highlight some differences between Monte Carlo and Las Vegas algorithms.

For the problem, imagine the following situation. You are in your household when you notice that there is some large, yet finite, number of cardboard boxes in front of you. You know that under one of these hides a spying neighbor, infiltrating your home to steal your Wi-Fi password. To find where this unwanted guest is, you must check underneath each box until you find him.

Las Vegas Approach. Let us first consider the Las Vegas solution to this problem. Such an algorithm will use some random number generator to tell you which box to look under next, without checking the same box twice. Derandomizing this algorithm is also simple, as we can simply decide on a deterministic way to choose the next box to check,

for instance "the box closest to me that I have not yet checked". At a first glance, there is no major difference between these two algorithms and they should thus be equally good, but upon closer inspection, there are two important differences.

The first point is naturally that the random algorithm requires a source of random numbers. In a system where RNG is already readily accessible, this shouldn't pose a problem, but if there is either little or no access to randomness, this can leave the random algorithm unfeasible.

The second point of difference is what runtime depends on. In the deterministic scenario, when faced with a given situation of boxes, we will check each box in the same time. Our spy may have taken this into consideration, and has hidden himself in the very last box we would check. This type of problem is called a *pathological input*, which is when an input for our algorithm will give us a worst-case scenario. There is no real way to combat this without involving a random element, which causes our algorithm to cease being deterministic. An equivalent worst-case scenario will still be possible for the randomized algorithm variant, but it will no longer depend on the input, and so the spy has to rely on luck to not be found.

Monte Carlo Approach. The Las Vegas algorithm and its corresponding derandomized variant may work just fine if there aren't too many boxes. Imagine however if your neighbor brought an extraordinarily large number of boxes, say several hundred. Checking all of these by hand would not only be nigh impossible, but also much slower than waiting for your neighbor to come out when he inevitably needs a break. However, for good measure, you may decide to check at least a few random boxes anyways, with no guarantee of finding your infiltrator. This is a Monte Carlo algorithm to solve this problem. Such an algorithm would use RNG to tell you which n boxes to look inside.

The deterministic variant would choose n boxes after some predetermined rule, for instance "the n closest to you". These variants differ from each other in the same manner as the Las Vegas algorithm did from its deterministic variant, that is to say that one requires access to RNG and the other has some pathological inputs. However, one big difference is that both of these will take a fixed amount of time depending on n, unlike the Las Vegas algorithm and its deterministic variant. Indeed, the pathological inputs in this case has no effect on running time, only on the likelihood of finding the spy.

4.2 Numerical Integration

Integration of functions is a common mathematical operation in many fields of applied mathematics, such as engineering and statistics. However, many integrals are either very difficult or impossible to compute by explicit formulas, rendering symbolic integration inviable. When dealing with definite integrals, an approach known as numerical integration can be used do approximate the value $F = \int_D f(x)dx$ of said integral over some domain

D, which is sometimes perfectly sufficient. There are myriad methods to integrate functions numerically, and whichever is used typically depends on what function $f(x)$ is being integrated, as well as on the domain D.

4.2.1 Monte Carlo integration.

A common type of Monte Carlo algorithms is what's called the Monte Carlo methods which are used in numerical analysis, defined by taking some number of random samples of a dataset to approximate some function.

Monte Carlo integration, as seen in [4], is precisely when the Monte Carlo method is applied to numerical approximation of an integral. A simple demonstration of this is imagining a 1-by-1 square completely containing a circle of diameter 1. By randomly sampling a large number of points in the form of (x, y)-pairs of numbers in the interval $(0, 1)$, and comparing the amount that land within the circle to the total number of pairs, we can approximate the area of the circle relative to the square, which is one way to approximate π.

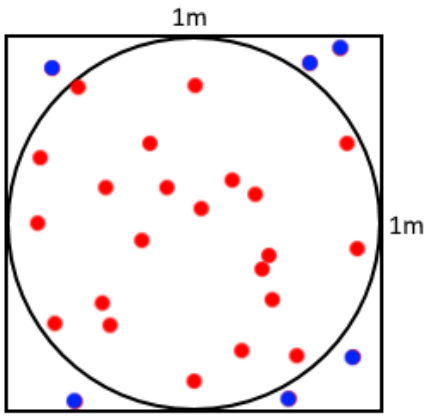

Figure 4: One version of Monte Carlo integration used to approximate the area of a circle of diameter d=1.

The example seen in Figure 4 was sampled by picking 28 random-looking points manually[1]. In this case, 22 landed within the circle, approximating the circle's area as $A_c^* = 0.7857\ldots$, and since the area of a circle is $A_c = \pi r^2$ with $r = 0.5$, we can find an approximation of π by calculating $\pi^* = A_c^*/r^2 = 3.1428\ldots$. Significantly more data points are usually sampled which typically yields a much better approximation.

Monte Carlo integration is particularly attractive for situations where a function is

[1]When implementing a random algorithm of some sort, it is generally ill-advised to substitute the algorithmic PRNG for a person choosing numbers randomly at the top of their head.

difficult or even impossible to integrate theoretically, but where a numerical approximation over an interval is sufficient. The most basic form of a Monte Carlo integration algorithm is a one-dimensional estimator [12], that is when we wish to integrate over an integral in one dimension.

Definition 4.1 *For a function $f(x)$ such that $F = \int_a^b f(x)dx$, the Basic Monte-Carlo Estimator using N uniform random variables (samples) X_i over $[a, b)$ is defined as*

$$\langle F^N \rangle := \frac{b-a}{N} \sum_{i=1}^{N} f(X_i).$$

Naturally, the expected value of $\langle F^N \rangle$ is indeed F, since its probability density function $p(x) = \frac{1}{b-a}$ for $x \in [a, b)$:

$$\begin{aligned} E[\langle F^N \rangle] &= \frac{b-a}{N} \sum_{i=1}^{N} E[f(X_i)] \\ &= \frac{b-a}{N} \sum_{i=1}^{N} \int_a^b f(x)p(x)dx \\ &= \frac{1}{N} \sum_{i=1}^{N} \int_a^b f(x)dx = \int_a^b f(x)dx = F. \end{aligned}$$

To know that this estimator will end up converging towards the expected value as $n \to \infty$, we need look no further than the laws of large numbers. In particular, the strong law of large numbers [22]:

Theorem 4.2 *Let $X_1, X_2, \ldots$ be an infinite sequence of independent and identically distributed random variables with a finite expected value μ in common, and let $\overline{S_n} = \frac{1}{n} \sum_1^n X_i$. Then,*

$$\Pr\left(\lim_{n \to \infty} \overline{S_n} = \mu \right) = 1.$$

The proof of this theorem will be skipped over, but it involves setting $Y_i = X_i - \mu$, and then showing that the sum of all Y_i average out to 0 by employing the generalized Chebyshev inequality.

By the strong law of large numbers, we know that $\langle F^N \rangle$ will converge to its expected value F, and thus that $\langle F^N \rangle$ becomes a better estimator when using a larger number of samples. Thus, faithful to its name, the Monte-Carlo estimator is a random variable which uses samples from N random variables to estimate some integral.

In an arbitrary number of dimensions, the estimator is mostly similar in definition, but we shall show it here for the sake of generality:

Definition 4.3 *For a function $f(\overline{x})$ where $\overline{x} = (x_1, \ldots, x_d)$, and $F = \int_D f(\overline{x})d\overline{x}$ where $D \subseteq \mathbb{R}^d$ with the volume $V = \int_D d\overline{x}$, the General Monte-Carlo Estimator using N uniform random variables (samples) X_i in said d-dimensional domain D, is defined as*

$$\langle F^N \rangle := \frac{V}{N} \sum_{i=1}^{N} f(\overline{X_i}),$$

the expected value of which can be proven to be equal to F in a similar way. While there are various ways to improve upon Monte Carlo integration, such as variance reduction, this will be left out.

Let MC be a random algorithm which uses Monte Carlo integration with an input consisting of a domain $D \subseteq \mathbb{R}^d$, a function $f(\overline{x})$ defined over D, sometimes a specified number of significant digits s in the datapoints and/or output, and a number N of desired data points to be used to estimate the integral $\int_D f(\overline{x})d\overline{x}$. This type of Monte Carlo method is very fast, which we can show by fixing the domain D, the function f and the desired number of significants s for each datapoint, and then counting its steps depending on the desired number of data points N.

Running Time. The first part of MC requires choosing N random datapoints over D of length s, which is done using some PRNG. Assuming each generated data point takes some constant time unit u, generating N random numbers will take about time uN, this part of the MC has polynomial complexity $O(N)$.

The second part of MC involves calculating the estimator, which is done in $2N + 1$ arithmetic operations; one count of division, one of multiplication, N function evaluations and $N - 1$ additions. Each such operation's running time depends on the size of the datapoints s and not the number of datapoints, meaning that we have polynomial complexity $O(N)$.

As so we have established that Monte Carlo integration has a step complexity of $O(N) + O(N) = O(N)$ for number of data points N. Since the time taken by operations in part two depends more on the number of significant digits s than just on the number of data points, we see that either multiplication which can be implemented to have as low as polynomial complexity $O(s \log s)$, or the function f with unknown (and potentially very slow) complexity, may bottleneck the true running time of MC. We can thus consider additional significants s to be more expensive than additional data points N, and if we want a more accurate approximation we might therefore prefer to increase the latter.

Error. Since this algorithm is approximative, it will generally not be completely accurate. Since the variance $\sigma^2 = E[(\langle F^N \rangle - F)^2]$ is a measure of deviation from the expected value, we can look at it as a measure of error. The error can be shown to be proportional to $N^{-1/2}$, in other words, if we wish to reduce the error by a factor of r, we need to increase the number of data points by a factor of r^2. We can here say that the expected accuracy

of MC is $O(N^{-1/2})$. It is important to remember that, unlike deterministic error which acts as more of a strict bound, this error is indeed a probabilistic bound, and not a strict bound.

4.2.2 Deterministic Numerical Integration.

An alternative to the probabilistic Monte Carlo integration method is some deterministic numerical integration, also called quadrature, of which there are several types. A lot of quadrature methods are designed for one-dimensional integrals, but can be generalized to higher dimensions. For the one-dimensional example, an integral $F = \int_a^b f(x)dx$ can be approximated with n points by the following formula:

$$\int_a^b f(x)dx = F \approx \sum_{i=1}^{n} w_i f(x_i),$$

where x_i are points on the interval $[a, b]$, and the coefficients w_i are preset weights to determine the "significance" of certain points on the interval. Specific choices of points and weights are called rules, which is also extended to more advanced forms of quadrature, such as the ones involving derivatives. In its most simple form, weights are taken to be $w_i = \frac{1}{n}$, and the points x_i are equidistant in some manner. For instance, in the Riemann Midpoint rule [20], we let $x_i = a + \frac{2i-1}{2n}(b-a)$.

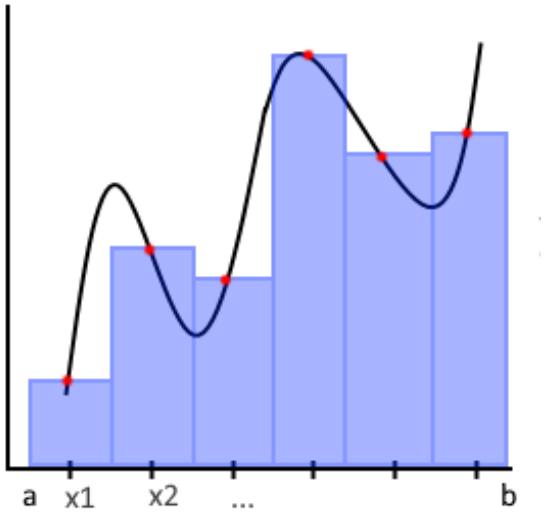

Figure 5: Approximation of a one-dimensional integral using Riemann Midpoint rule integration with $n = 6$ points.

We can see in Figure 3 that, while not a bad attempt, it is not perfect. This can be remedied by using different weights and point distributions, or more advanced formulas such as

higher degree Newton-Cotes formulas. At this level, an algorithm that chooses these points in a deterministic manner and then estimates $F^* = \sum_{i=1}^{n} w_i f(x_i)$ is equivalent to the Monte Carlo integration method discussed earlier in terms of total steps per number of data points.

We can also generalize it for d dimensions with $N = n^d$ points using what's known as the product rule. All this entails is that we fill the space with points distributed just like we would the n points in one dimension, but we do it in every dimension, forming a d-dimensional grid. So, to approximate $F = \int_D f(\overline{x})d\overline{x}$ in a domain D with an $n \times \cdots \times n$ set of points $\overline{x}_a$ indexed by coordinates $a \in \{1, \ldots, n\}^d$, we can use the formula

$$\int_D f(\overline{x})d\overline{x} = F \approx \sum_{\forall a \in \{1,\ldots,n\}^d} w_a f(\overline{x}_a),$$

where w_a is the weight for the d-cube about $\overline{x}_i$. While a good method in theory, we notice what is sometimes called the "curse of dimensionality": The greater the dimension d, the exponentially more points are necessary to "fill the domain" properly. For especially high dimensions, such as $d \geq 10$, this is a terrible restriction. However, for now, let us ignore it and instead compare this to Monte Carlo integration in d dimensions and n^d points.

Running Time. An algorithm RMR using Riemann Midpoint Rule of approximating an integral will have its running time scale at the same rate as Monte Carlo integration. This is because the only real difference between the parts in MC and RMR are the way they get their datapoints as well as the precise appearance of the formula, so the number of steps are approximately the same for any given $N = n^d$ number of used data points.

Error. The error for MC was said to be proportional to $N^{-1/2}$ regardless of the number of dimensions d. For RMR, in one dimension $d = 1$ and a total number of data point evaluations $n = N$, it has an accuracy of $O(N^{-2}) = O(n^{-2})$, which of course is better than the $O(N^{-1/2})$ that MC would have. However, for dimensions $d > 1$ and the same data points "per length" n, the number of data points for the same level of error will be $N = n^d$, and so an equivalent error of $O(n^{-2})$ as a function of dimensions is $O(N^{-2/d})$. For lower dimensions, this is still a better choice than general Monte Carlo sampling, but already at $d = 4$ it is equal to that of MC at $O(N^{-1/2})$, and it quickly starts getting significantly worse. Unintuitive as it may be, MC sits at a steady $O(N^{-1/2})$ regardless of dimension, because its variance is only a function of N.

Pathology. An important point to be made is about pathological inputs, in other words certain inputs that are especially incompatible with the algorithm. Unlike MC, it is a fact that RMR and similar functions will have preset positions and weights, and certain inputs will lead to highly erroneous estimates. This can often be partially remedied by increasing the number of data points, or make other changes to the algorithm, but the fundamental problem of the existence of pathological inputs cannot be completely removed when dealing with deterministic functions, while being significantly rarer in most random

algorithms.

4.3 Sorting Algorithms

Much like how people are often less productive if their papers and notes are a disorganized mess, many algorithms working with large datasets will work slower than usual if the data they are working with is particularly "out of order". In practice, there are user-experience related reasons for why sorting is useful, for instance sorting internet search results by their relevance. It is for reasons such as this that sorting algorithms are commonly used to sort an input data set by imposing some sort of ordering upon its elements by means of rearrangement. Naturally, such algorithms are almost[2] exclusively Las Vegas algorithms, as they involve sorting a finite list of elements until it is completely sorted. Designing a sorting algorithm to have an upper bound to its running time would lead to it occasionally outputting not completely sorted lists, meaning it did not actually sort the list.

4.3.1 Randomized QuickSort.

There are many different sorting algorithms, and it is not uncommon to combine several whenever it is useful. One of the most common sorting algorithms is QuickSort with randomized pivoting, or just Randomized QuickSort.
QuickSort is a comparative sorting algorithm, meaning that it takes some array A as input, and requires some definition of order, typically in the form of an order check, such that for any two elements a, b in the array, it is always possible to determine which is greater. The order check can be some comparison function $f(a, b)$ that will output a 0 if $b < a$, and a 1 if $a < b$. If it is possible for two elements to be equal, there needs to be a third output to signal this.

Randomized QuickSort works by a divide-and-conquer process, splitting the input array A into two smaller subarrays A_1, A_2 where every element in A_1 is smaller than every element in A_2, and repeats the process until the list is sorted - or, alternatively, switches to a different sorting algorithm when the subarrays have become small enough that it's worth it. More formally, Randomized QuickSort RQS is an algorithm using the array A as input. Using lecture notes [16] as a resource, we describe the algorithm as functioning as follows:

[2]Non-Las Vegas sorting algorithms, which are ones that may either never finish or may output an incorrectly sorted list, have little use besides being examples of bad sorting algorithms, such as Bogosort.

input : An array A.
output: The array resulting after sorting A.

check for trivial input;
if $|A| < 2$ **then**
| The array is already sorted or empty, return A.
end

pivot choice;
$p \leftarrow$ randomly chosen element in A;

comparisons;
$B_{less} \leftarrow$ array with elements $\{e \in A |\ e < p\}$;
$B_{equal} \leftarrow$ array with elements $\{e \in A |\ e = p\}$;
$B_{more} \leftarrow$ array with elements $\{e \in A |\ e > p\}$;

recursive assembly;
return $\texttt{RQS}(B_{less}) \| B_{equal} \| \texttt{RQS}(B_{more})$

An example scenario could be an unsorted array if integers $A = [1, 4, 2, 5, 3, 0]$, which we wish to sort from least to greatest. We first randomly choose a pivot point, such as $p = 2$, leaving us with arrays $B_{less} = [1, 0], B_{equal} = [2], B_{more} = [4, 5, 3]$. After the array B_{less} is ran through this algorithm, it will be sorted no matter what pivot point is chosen, leaving us with $[0, 1]$. The array $B_{equal} = [2]$ is already sorted, and we end up with the array $[4, 5, 3]$, where we repeat the process as before and eventually end up with arrays which we can combine into $[3, 4, 5]$. Together, these three arrays combine to $[0, 1, 2, 3, 4, 5]$, which is a fully sorted array.

There are some improvements that can be made to this basic algorithm. For instance as previously mentioned, when arrays are small enough, instead of resuming with QuickSort, it can be quicker to use a simpler sorting algorithm to sort the elements within. However, as it were, this is the basic setup for Randomized QuickSort.

Running Time. Regarding time complexity, for arrays with $|A| = n$ elements, RQS has the expected running time of $E[T(A)] = O(n \log n)$ comparisons, which is quite fast and in fact polynomial. We remind ourselves of the fact that Las Vegas algorithms are usually considered in terms of expected best-case, average-case and worst-case time complexity. For this type of Randomized QuickSort however, the input array will make no difference in the expected running time, and so the expected worst-, average- and best-case times are identical.

Theorem 4.4 *The expected running time of the Randomized QuickSort algorithm, for an input array A of size n, is $O(n \log n)$ comparisons.*

To demonstrate this, let us consider some input array A of size n. For the sake of simplicity, let us assume that no two elements are equal - the existence of such elements would not negatively affect the running time, and it will simplify our notation.

When we first pick a pivot, we will perform $n-1$ comparisons to split the array. We have divided this array into subarrays B_{less}, B_{more}, where depending on the pivot point, it could be that $|B_{less}| = 0, |B_{more}| = n-1$, or that $|B_{less}| = 1, |B_{more}| = n-2$, and so on. Each such partition has a probability of $1/n$ of occurring, since there were n elements to choose our pivot from. Since we performed $n-1$ comparisons, we are now up to $n-1$ steps so far. With these facts at hand, and because we defined RQS recursively, we can set up a recursion for the expected number of comparisons $T(n)$ as a function of the list length n [1, pages 5–7].

$$\begin{aligned} E[T(n)] &= (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(E[T(i)] + E[T(n-i-1)]) \\ &= (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} E[T(i)] \end{aligned}$$

This recursion follows immediately from the definition of the expected value, since we're considering the expectation of each possible way to partition the array into two. The next part will involve making an educated guess of an expected upper bound for the running time, and then showing that this is indeed the case.

The intuition is as follows: on average, the pivot is expected to split the array in two about equally sized subarrays, halving the size in every recursion step, where each halved list is easier to sort. In other words, for an array B of size $2n$, we would only have to split it one more time than our array A of size n - this progression is clearly logarithmic. Since before each time we split our arrays in two, we need to perform up to n comparisons, we can reasonably guess that $E[T(n)] \leq cn \ln n$ for some constant c. We will show that this is the case with a proof by induction. According to our guess, $E[T(i)] \leq ci \ln i$, with the base case $T(1) = 0$.

$$\begin{aligned} E[T(n)] &\leq (n-1) + \frac{2}{n}\sum_{i=1}^{n-1} ci \ln i \\ &\leq (n-1) + \frac{2}{n}\int_{1}^{n} cx \ln x dx \\ &\leq (n-1) + \frac{2}{n}\left(\frac{cn^2 \ln n}{2} - \frac{cn^2}{4} + \frac{c}{4}\right) \\ &\leq cn \ln n \text{ (for } c = 2) \end{aligned}$$

Thus, we have found that $E[T(n)] \leq 2n \ln n$, implying complexity class of polynomial-time by $O(n \log n)$. Since we made no assumption of what input we have other than the relative worst-case assumption that there are no elements that are equal to each other, we know that each case is necessarily polynomial-time. As stated previously, we could also show that all cases are $O(n \log n)$, but that would require making specific assumptions about equal elements which only complicates notation but leads to the same result, so it shall be skipped.

4.3.2 Deterministic QuickSort.

As for deterministic QuickSort, the only typical difference against Randomized QuickSort is how we choose a pivot. There are a few ways of doing this deterministically with various levels of successfulness [16].

The most simple method is perhaps the most intuitive one: choose the first element in the array. On average, this will be just as effective as randomized QuickSort, since assuming uniformly random input arrays, the first element in the array will be random, as will every subsequent "first element" in the upcoming subarrays. However, this is only in the average case, and since we now do not pick our pivot at random, what type of case we're dealing with is no longer dependent on a random variable's output, but rather on the input array. This means that we have a new worst case scenario.

Worst-Case. What if our input array A is already nearly (or completely) sorted? Equally troubling would it be if our array is reverse-sorted, or in other words, in "descending order" as opposed to the properly sorted "ascending order". In fact, this is a relatively common occurrence in practice. In this nearly sorted scenario, every iterative step of this first-element-pivot algorithm, we will compare said pivot to all other elements $a \in A$ we find that this is the least element in the list, placing all other elements $b \in A$ into B_{more}, and then repeat the process.

Theorem 4.5 *The worst case complexity of the Deterministic QuickSort algorithm, for an input array A of size n, is $O(n^2)$ comparisons.*

For an array of size n, to "sort" an already sorted list, we would in total have to perform $(n-1) + (n-2) + \cdots + 1$ comparisons, which is an arithmetic series with the sum $\frac{n^2-n}{2} = O(n^2)$. Thus, the first-element-pivot QuickSort algorithm has a worst case of $O(n^2)$, and considering that almost sorted and almost reverse-sorted lists are quite common in practice, this worst case will be an uncomfortably common occurrence for this algorithm, while being suitably rare for Randomized QuickSort.

Of course, there are a few ways to remedy this situation by changing the pivot choice step in one of a few ways. For instance, instead of definitely choosing the first element, we may have the algorithm look at three elements, namely the first, last, and middlemost elements of the array, ordering those three elements, and choosing the median out of those to be your pivot. This not only eradicates the problem with both nearly ordered and reverse-ordered lists, but it also improves your pivot choice in the average case; At the cost of sorting three elements every pivot selection, requiring only 2-3 comparisons, you are much more likely to get a pivot further towards the middle, which is optimal because creating two middle-sized arrays is more efficient than one much larger than the other. For especially large lists, it could be worthwhile choosing more than three elements, as the few additional comparisons performed once per iteration is a small price for such a greatly improved average-case.

To illustrate, we can imagine having a list with a very large amount of elements (such as integers) that we intend to sort, and we assume that they appear at random in this list. If we were to choose only one element as our pivot, there is a 1/2 chance that this element is either amongst the 25% smallest integers or 25% greatest integers in the list. If we instead choose between three elements and let the median be pivot, the probability of that median being in those upper and lower fringes is now 1/8, and if we choose between five elements that probability becomes 1/49, with a sacrifice of up to 10 comparisons to sort the pivot candidates.

If we indeed are dealing with an immensely large array of elements, it is fairly obvious that this type of measure would save a lot of time on average. Naturally, there is also no need for Random QuickSort to feel left out, as this type of improvement is possible even then - and for the person in charge of programming it, even simpler than for deterministic QuickSort. Instead of coding a way for the list to choose k elements from specific places in the array, we can simply ask it to generate k distinct random numbers from our RNG, sort them, then pick the median.

4.4 Primality Testing

One can imagine that there are a lot of theoretical, more abstract reasons why finding and testing prime numbers could be interesting. However, there are some practical applications as well, such as when number theoretical restrictions necessitate that some parameter is chosen as a prime. For instance, this was the case for the modulus p for defining elliptic curves over finite fields in section 3.3.1, and we will see a similar cryptographic application in the later section 4.5.2.

What all this means is that it can be important to have a good method or algorithm to locate primes and test primality. As for primality testing in particular, the problem is finding out whether some number n is a prime or not. Other than the naive method - namely, checking if there is any $1 < a < n$ such that $a|n$ - there are a few more efficient methods. Several such methods rely on finding some integer a which is a *witness* to the compositeness of n. For the naive method, the witnesses are exactly the true divisors of n, but better methods have witnesses that are not necessarily divisors of n.

4.4.1 Miller-Rabin Test

A pretty reliable method is known as the Miller-Rabin test [10, pages 130–132], referring to the random algorithm variant. For an input n, the primality of which is to be determined, it needs to take a random sample of some m numbers $1 < a < n$, and they will all be checked to see whether they are witnesses. If even a single number is a witness to the compositeness of n then it proves that n is a composite number, but even if none of the sample numbers are witnesses, it does not necessarily mean that n is prime.

The test is based on the following result. Let p be some prime, and write $p - 1 = 2^k q$ for

an odd q, and let a be a positive integer that does not divide p. By Fermat's little theorem [25], a^{p-1} must be congruent to 1 mod p, meaning that either a^q is congruent to 1 mod p, or there is some $0 \leq i \leq k-1$ such that $a^{2^i q}$ is not congruent to 1 mod p, but its square is, thus $a^{2^i q}$ must be congruent to -1. In other words, if p is a prime and $1 < a < p$, then either
1. $a^q \equiv 1 \mod p$, or
2. there is some $0 \leq i \leq k-1$ such that $a^{2^i q} + 1 \equiv 0 \mod p$.

Thus, if we are testing some number n in the above manner, if neither of the top conditions is fulfilled, we know that n is composite, and then a is a witness to the compositeness of n. However, if one of the conditions is fulfilled, then n could either be prime or composite. If we perform the test using a random number, and it fails to prove compositeness, we can say that n is "composite with some probability x", and if this happens for two different numbers a_1, a_2, we can say the same but with probability x^2. Luckily for us, we have a fairly good idea of what these probabilities are.

Theorem 4.6 *For a composite number $n > 4$, at least 75% of positive integers $a < n$ are witnesses to its compositeness.*

The proof for this [17, Theorem 1] is rather involved and so will be skipped. In either case, for m distinct random numbers less than n, assuming n is composite, the probability that none of m such numbers being witnesses is at most about $1/4^m$. The probability of n being a prime number, as described in [10, page 134], is at least approximately $1 - \frac{\ln n}{4^m}$. Now, let us formulate an algorithm to perform this test. We assume that m is some constant not determined by the input.

input : An input integer $n > 2$ to be tested.
output: Primality status.

if $2|n$ **then**
| return status "n is composite"
end

test for witnesses;
for $num \leftarrow 1$ **to** m **do**
| Let a be a random element $1 < a < n$;
| **if** $1 < \gcd(a, n) < n$ **then**
| | return status "n is composite"
| **end**
| Let us notate $n - 1 = 2^k q$ such that q is odd;
| **if** $a^q \not\equiv 1 \mod n$ **then**
| | **if** *for all* $i \in \{1, \ldots, k\}$, $a^{2^i q} + 1 \not\equiv 0 \mod n$ **then**
| | | return status "n is composite"
| | **end**
| **end**
end

If we have reached this point without returning compositeness;
return status "about $1 - \frac{\ln n}{4^m}$ chance of n being prime"

We are also quickly checking if n is a prime and whether a divides n, before performing the test proper.

Running Time. Skipping over the details, the complexity of the randomized Miller-Rabin test using repeated squaring is $O(m(\ln n)^3)$, however naturally this is only the expected running time, and the true time will depend entirely on n, and what precise numbers a are being tested, and even in what order. In the best case scenario, n is even and a compositeness status can be immediately returned, whereas the worst case is n being a prime and thus none of the m tested numbers are witnesses. However, we will soon see that the running time is not the most interesting part about this problem.

4.4.2 Deterministic Miller Test

The deterministic Miller-Rabin test, in this case a type known as the Miller test, will check $O((\ln n)^2)$ numbers for witnesses. While not often used in practice due to its slow running time, it has the interesting property of potentially being able to prove that a number is prime with certainty, and not just with probability depending on how many numbers were tested. [10, page 136]

Theorem 4.7 *Assuming that the Generalized Riemann Hypothesis is true, then for any composite number n, there is some Miller-Rabin witness a such that $a \leq 2(\ln n)^2$.*

Its algorithm functions identically, but instead of randomly choosing a list with some parametric length k, a list is deterministically chosen as the numbers less than $2(\ln n)^2$.

Other than the fact that it's slow to check about $2(\ln n)^2$ numbers when n is particularly large, there are two other obvious problems.
Firstly, if we choose to improve running time by reducing the number of integers that we check for witnesses, we are no longer guaranteed to be able to prove primality under the assumption of the GRH. We once more fall prey to the issue of pathological inputs.
Secondly, we have the fact that we are basing this entire test on the assumption that the GRH, an unproven hypothesis, is true. It may be unwise to rely on an algorithm when it is not known whether it is theoretically sound or not.

The good news for those deterministically inclined is that there are other algorithms, some in fact faster than the deterministic Miller test, which can prove the primality of any $n > 1$. As an example, there is the AKS primality test which, for every $\epsilon > 0$, will conclusively prove whether a number is a prime or not in no more than $O_\epsilon((\ln n)^{6+\epsilon})$ steps. It is worth mentioning that this complexity is significantly worse than the Random Miller-Rabin test, but with the primary difference that the Miller-Rabin will have some level of confidence less than 100% in the primality of its input, whereas the AKS test will prove it with certainty.

4.5 Necessarily Random Algorithms

We have previously seen examples where deterministic algorithms have been situationally better than random algorithms in certain ways or for certain inputs, but there are cases where use of a random algorithm is inherently necessary, such that a deterministic version is completely out of the question for some fundamental reason. Two examples of such situations involve certain simulations where unpredictability is the entire point, as well as many cryptographic situations.

4.5.1 Shuffling Algorithms

The sorting algorithm was touched upon in an earlier section, so let us now consider a type of algorithm which involves what is arguably the opposite of sorting, namely a shuffling algorithm. We may use such an algorithm if we have some array A, possibly sorted, which we wish to shuffle evenly and efficiently such that every shuffled permutation is equally likely. Another property we want is that, much like when shuffling a deck of cards by hand, we want the result to be independent from the starting position in the sense that shuffling two separate perfectly ordered decks of cards using the same algorithm should still yield two differently shuffled decks. An example of such an algorithm for when we wish to shuffle some array A with size $|A|$ is the optimized Fisher-Yates shuffling algorithm:

input : An array A.
output: The array resulting after shuffling A.

for $i \leftarrow 1$ **to** $|A| - 1$ **do**
  $r \leftarrow$ random number $i \leq r \leq |A|$;
  let the r:th element switch places with the i:th element
end

The way that this algorithm shuffles the elements is quite simple. It first chooses an element at random to place in the first spot in the array, then randomly chooses a previously not picked element to place in the second spot, and so on for the rest of the array. With a running time of $O(n)$ for an array of size $|A| = n$, and a perfectly random shuffle assuming ideally random numbers are used, you would be hard pressed to find a better shuffling algorithm.

The Fisher-Yates algorithm mentioned above will in total generate $n - 1$ random numbers between 1 and n, meaning it will require $O(n \log n)$ random bits. However, is there a way that we may be able to change the fact that it requires random bits at all? The answer, rather intuitively, is no. If we removed any trace of randomization, then some given input would always end up with the same shuffled result. What we have now is not a shuffling, but something more akin to a PRNG. But as opposed to a PRNG, we mentioned previously that we desire for the input to be shuffled into any possible output with equal likelihood, but that is not possible with a deterministic algorithm as it depends only on the input.

4.5.2 Cryptographic Algorithms

Cryptographic algorithms are typically concerned with concealment of information in some manner too unpredictable to be revealed normally in a timely fashion, but can be efficiently revealed using the associated secret key. One such algorithm is the regular Diffie-Hellman exchange, which is used to generate a shared secret key between two parties for future cryptographic use. The entire point of such algorithms is to provide for some form of secure exchange of information, and the Diffie-Hellman exchange is no exception.

The Diffie-Hellman exchange between two parties Alice and Bob first requires them to agree on two public parameters, the base g and the modulo p. We also require that g is a primitive root modulo p, in other words that g is a generator for all numbers from 0 to $p - 1$. After this, Alice and Bob will independently make randomly generate one private key each, calculate a corresponding public key each, send them to each other, and they will then be able to calculate their shared secret.

input : Public base element g and modulo p
output: Shared secret S

Private exponents;
$a \leftarrow$ random number $1 < a < p - 1$ (Alice's private key);
$b \leftarrow$ random number $1 < b < p - 1$ (Bob's private key);

Public exponents;
$A \leftarrow g^a \mod p$ (Alice's public key);
$B \leftarrow g^b \mod p$ (Bob's public key);
Key exchange as Alice and Bob transmit their public keys to each other;

Shared secret calculation;
Alice's calculation: $B^a \equiv S \mod p$;
Bob's calculation: $A^b \equiv g^{ab} \equiv B^a \equiv S \mod p$;
Shared secret: $A^b \equiv S \equiv B^a \mod p$

The only publicly known elements here are g, m, A, B, from which S is quite difficult to calculate without access to the secret private parameters a, b, making this a useful way to generate some secret number S just in case there may be someone eavesdropping. The fact that S is difficult to calculate from only the public parameters and A, B is referred to as the Diffie-Hellman Problem. Without going too far into the details, we direct our attention to the very first step of the algorithm: generating the private key exponents a, b.

The fact of the matter is that, if an eavesdropper were to know what either a or b are, then calculating the shared secret S would be trivial and hold no security. If this algorithm was somehow derandomized, however, that is precisely what would happen. If a, b were deterministically chosen based on g, m, then all an adversary would need access to is the algorithm and the already public keys g, m for the whole exchange to no longer be secure, and thus would render it pointless.

We can generalize this to any cryptographic algorithm which relies on a private key, which is every cryptographic algorithm. If such an algorithm would be derandomized, it would have to generate its secret keys based on its input. The input in turn would have to consist of public parameters. Now, if the input of a deterministic algorithm that generates a private key consists of public parameters, anyone with access to the algorithm can generate the same private key, and thus by Kerckhoff's principle you can no longer assume that the private key is a secret, and the algorithm necessarily fails at being cryptographic.

4.6 Limitations of Derandomized Algorithms

As thoroughly established previously in this section, using derandomized algorithms is not always preferable to random algorithms. While many derandomized algorithms have their fair share of benefits, there are some things that random algorithms can do with a limited amount of time and space that deterministic algorithms cannot do with the same resources. Additionally, there are a couple of issues that are downright emblematic of derandomized

algorithms, and are much rarer in random algorithms.

Pathological inputs. Something we have seen a few times so far is pathological inputs being a risk factor for deterministic algorithms. Indeed, for most such algorithms, there is often some type of input that is especially time-consuming or gives a particularly large degree of error. For some algorithms, it is reasonably easy to make modifications to reduce the intensity of events like this, or the probability that they will affect the process negatively. Nevertheless, a more straightforward way is sometimes to simply implement random variables in place of some fixed choices somewhere to make the issue less dependent on input - something you have little to no control over - and more dependent on RNG, which is something you can often generate more of should you need it.

Complicated coding. Some algorithms are able to take a variety of inputs that may require different kinds of solutions depending on certain properties of the input. One example of this seen previously involved QuickSort. Both to improve worst-case performance, and to make on average more effective pivot choices for very large lists, deterministic QuickSort would choose a pool of candidates and then pick the median as the next pivot. How large this pool was, and how precisely it would choose the elements to put into the pool, would somewhat depend on the size of the current list. This would be quite a bit simpler to code for a randomized QuickSort algorithm, as now the only thing depending on the size of the list is the size of the pool, whereas the elements in the pool can be chosen at random instead of following some complicated equation.

Situational ineffectiveness. There are situations in which a deterministic decision will yield a worse result on average than the expectation of a random choice. One such example was in regards to numerical integration, for which lower dimensional integrals could be solved with minimal error using deterministic algorithms, whereas for integrals over higher dimensions, error became much more costly to reduce. This particular situation is caused by what is known as the "curse of dimensionality", which is an observation regarding the exponential increase of volume for each additional dimension. There are a few different methods that are known to work with small error despite this terrible curse, one of which is Monte Carlo integration, which relies on random sampling, and is therefore more suitable for estimating higher dimensional integrals.

Inherent determinism. While perhaps much more obvious than other issues, an algorithm may require some degree of randomness simply due to the nature of the problem. Examples featured previously included shuffling algorithms, which cannot be deterministic by how we define a fair shuffle as inherently unpredictable, and cryptographic algorithms, which cannot be deterministic because it relies on choosing private keys unpredictably.

5 Open Problems in Random Algorithms

Those familiar with unsolved problems in mathematics may have heard of the "NP versus P" question, which asks whether or not "$P = NP$". What P and NP refer to are essentially classifications of problems, where P is the class of problems which can be solved using some deterministic polynomial-time algorithm, and NP informally refers to the class of problems such that, if you give it an answer, there is a polynomial-time algorithm that can verify whether this answer is correct. Phrased more informally, the question is whether a problem can be solved quickly if and only if a correct answer to the problem can be verified quickly. The type of problems that these classes refer to are so called *decision problems*, and all that means is that the answer for some given input is either "yes" or "no". It is conjectured that $P \neq NP$, but it has not been proven.

There are many similar open problems in mathematics, which ask whether two classes of problems are overlap exactly. One such open problem which relates to random numbers is "BPP versus P". Here, P again refers to the class of polynomial-time deterministically solvable problems, and BPP refers to problems which are polynomial-time solvable using a specific kind of random algorithm.

The type of random algorithm referred to by BPP is defined by its probability of providing a correct answer. It is related to the class PP, which is short for probabilistic polynomial-time, and the B in BPP stands for Bounded-Error.

Definition 5.1 *A decision problem is a BPP problem if it is solvable by an algorithm A such that*

(i) A is a polynomial-time random algorithm, and

(ii) the probability that the algorithm outputs the correct answer is $\geq 2/3$.

This definition is a rephrasing of the one that can be found on ComplexityZoo [6] at the time of writing. If we consider an algorithm that solves a problem exactly half of the time as neutral, then the fact that BPP problem solving algorithms must have a success rate of at least 2/3, which we note is fairly significantly more than 50% of the time, means that we could informally call BPP algorithms "good". Additionally, by using such an algorithm repeatedly similar to how it was done in the Miller-Rabin test in 4.4.1, the rate of success can be further increased, converging to 100%s.

It is currently conjectured that indeed $P = BPP$ [8, page 4]. What it would mean for derandomization if this result was to be proven is, in very rough terms, that any efficient, good random algorithm can be derandomized to an efficient deterministic algorithm.

References

[1] Aspnes, James *"Notes on Randomized Algorithms"* `https://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf`,

[2] Barker, Elaine; Kelsey, John *"Recommendation for Random Number Generation Using Deterministic Random Bit Generators"* (NIST Special Publication 800-90A, as well as the succeeding version Revision 1),

[3] Borda, Monica *"Fundamentals in Information Theory and Coding"* (2011),

[4] Burkardt, John *"Integration, Quadrature, and Sparse Grids"* (2010) `https://people.sc.fsu.edu/~jburkardt/presentations/sparse_2010_fsu.pdf`,

[5] Click, Timothy; Liu, Aibing; Kaminski, George *"Quality of random number generators significantly affects results of Monte Carlo simulations for organic and biological systems"* (2011),

[6] ComplexityZoo *"BPP"* `https://complexityzoo.uwaterloo.ca/Complexity_Zoo:B#bpp`,

[7] Dodis, Yevgeniy *"Lecture 5 Notes of Introduction to Cryptography (PDF)"*,

[8] Guild, David; van Melkebeek, Dieter *"Lecture 14: Randomized Classes"* `http://pages.cs.wisc.edu/~dieter/Courses/2011f-CS710/Scribes/PDF/lecture14.pdf`

[9] Harvey, David; van der Hoeven, Joris *"Integer multiplication in time O(n log n)"* `https://hal.archives-ouvertes.fr/hal-02070778/document`

[10] Hoffstein, Jeffrey; Pipher, Jill; Silverman, J.H. *"An introduction to mathematical cryptography"*,

[11] Hull, Thomas E.; Dobell, Alan R. *"Random Number Generators"* (SIAM Review Vol. 4, No. 3, July, 1962),

[12] Jarosz, Wojchiech *"Efficient Monte Carlo Methods for Light Transport in Scattering Media, Appendix A"* `https://cs.dartmouth.edu/~wjarosz/publications/dissertation/appendixA.pdf`,

[13] Knuth, Donald *"The Art of Computer Programming"* (Third Edition),

[14] L'Ecuyerm, Pierre; Simard, Richard *"TestU01 User's guide, compact version (PDF)"* (May 16, 2013),

[15] O'Neill, Melissa E. *"PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation"*,

[16] *"QuickSort"*
https://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0123.pdf,

[17] Rabin, Michael O. *"Probabilistic algorithm for testing primality"*

[18] Schindler, Werner; Killmann, Wolfgang *"Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications"*,

[19] Shumow, Dan; Ferguson, Niels *"On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng"*,

[20] Sparling, George A. J. *"The midpoint rule"* (2002)
http://www.math.pitt.edu/~sparling/23021/23022numapprox2/node4.html,

[21] Stipčević, Mario; Kaya Koç, Çetin *"True Random Number Generators"*,

[22] Tracy, Craig A. *"Laws of Large Numbers"*
https://www.math.ucdavis.edu/~tracy/courses/math135A/UsefullCourseMaterial/lawLargeNo.pdf

[23] Vigna, Sebastiano *"An experimental exploration of Marsaglia's xorshift generators, scrambled"*,

[24] Wolfram Alpha's definition of an LCG
https://demonstrations.wolfram.com/LinearCongruentialGenerators/

[25] Wolfram Alpha's statement of Fermat's Little Theorem
https://mathworld.wolfram.com/FermatsLittleTheorem.html

A Dual_EC_DRBG Backdoor Demonstration Script

The following is the script I wrote to generate a list of candidates for the next state. It may not be as well written as it could have been, which I attribute to my inexperience with programming. The script consists of a function which takes in the parameters of the exploit system and outputs the list of candidates for the state. The module "customEC" is primarily used to perform elliptic curve arithmetic, and "modsqrt" is used to perform a modular square root operation.

```
import customEC as ec # self made EC module for creating curves and points, finding points on a given curve, and adding points together.
import math as m
from modsqrt import modular_sqrt as msqrt # a module fortaking  modular square roots, taken from github.

def ECcheck(cparams, Qcoords, e, state1 = None, rem = 0, printout=0):
  """Takes parameters of curve parameters, xy-coordinates of point Q, the state S1, number of bits to be removed, and whether to print extra info"""

  a, b, p = cparams    # curve parameters.
  xQ,yQ = Qcoords      # coordinates for Q. curve class checks if on the curve.
  S1 = state1          # state of the first known random number

  slmax = len( bin(p)[2:]) # seed length
  if rem > slmax: raise Exception("Removed bits \"rem\" is greater than the number of bits \"seedlen\", the bitlength of p.")

  curve = ec.Curve(a, b, p)    # define the curve
  Q = ec.Point(xQ,yQ, curve)   # define point on the curve
  P = Q * e                    # define eQ = P
  s2 = (P * S1)[0]             # just for reference, calculate what the next actual state is. not necessary.

  R1 = Q * S1; R1x = R1[0] # internal RNG coordinates generated
  r1 = bin(R1x)[2:]        # the random number that we see, string form
  while slmax > len(r1):   # formats r1 to usable form, e.g. 00000010 instead of 10
    r1 = "0"+r1
    r1 = r1[rem:]


  V = []   # list of candidates for s2
  for i in range(2**rem):
    if rem>2 and i % 2**(m.floor(rem/3)) == 0 and i != 0: print(100*i/(2**rem), "%") # will sometimes print % progress (upper bounded, may finish early)

    x = int(bin(i)[2:]+r1, 2)
    if x >= p: break # excludes any R1 candidates greater than the modulo p

    z = (x**3 + a*x + b) % p
```

```
    y = msqrt(z, p) # if there is no such mod square root y, it outputs 0.
    if y != 0:
      A = ec.Point(x, y, curve)
      s2c = (A * e)[0]
      if s2c == None: s2c = "0"
      V.append( s2c )      # saves the next state candidate corresponding to x in the list V

  if printout == 1:      # prints out useful information at the end
    print("------------------")
    print("a=", a, "b=", b, "p=", p)
    print("P=", P, "Q=", Q, "e=", e)
    print("S1=", S1, "\nR1=", R1, "r1=", int(r1,2), "\nbin(R1x)=", bin(R1x), "bin(r1)=", r1)
    print("seedlen=", slmax, "rem=", rem)
    print("\ncontents of V is:", V, "\ns2 is actually:", s2)
    print("------------------")

  return V, s2


cparams = [-50, 200, 4091]  # arbitrary non-singular curve (a, b, p)
Qcoords = ec.findpoint(cparams, xmax = 50, ymax = 50) # given a curve, finds a point Q deterministically.
e = 20       # arbitrarily chosen multiplier e to generate eQ = P
state1 = 5   # arbitrarily chosen starting state
rem = 6      # the number of leftmost bits removed from R1 to make r1. larger means safer, but outputs fewer random bits.

outp = ECcheck(cparams, Qcoords, e, state1, rem, printout=1)    # runs the function that finds the candidates using the exploit

if outp[1] in outp[0]: print("Yes, it is one in", len(outp[0]), ", opposed to naive", 2**rem)
else: print("uh oh, nope. s2 is not in Slist.") # this line existed for early troubleshooting
```

Listing 1: The script used to get the list of candidates from one random number r1.