Random Number Generators in Common Applications

Giovi Moriarty

Abstract

This paper discusses the use of Random Number Generators in software applications and dives into the definitions and history of different types of generators. Humans often misunderstand the idea of randomness due to a lack of understanding of probability coupled with human instinct to look for patterns. This proves to be a problem when individuals are seeking out randomness, especially in the form of a shuffle algorithm, such as that in music streaming apps. There are many different types of random number generators, but very few used in everyday applications produce truly random numbers, as there is usually a code to calculate the next random number in a sequence. This approach has been used since the 1940s, but new algorithms are being developed in order to appease critics and prioritize distribution over true randomness.

1 | Introduction

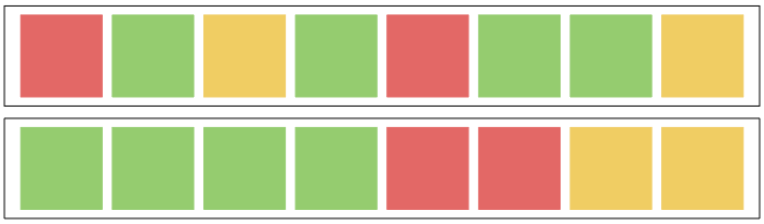
Random numbers are used everywhere in daily life, from gambling to listening to music. Since the iPod Shuffle came out in 2005, shuffling has been used to create seemingly random sequences of songs on many different streaming platforms and software applications. Humans have a subconscious desire for surprise, which can actually be the reason they have a negative opinion of random number generating and shuffle algorithms.

2 | Human Misunderstanding

Have you ever put a playlist on shuffle and wondered if it was actually shuffled? Maybe multiple songs by the same artist come up back-to-back or the first couple songs played are all in the same genre and you doubt that your music is being truly randomized like you intend it to be when you pressed the “shuffle now” button. The truth is, humans tend to see patterns where there are none, and we as a population are bad at accepting truly random events. To us, truly random does not feel random. This inability is due to a couple of factors. First, sometimes a truly random sequence is actually in order! This is confusing for individuals who do not understand permutations and probability. The second reason is that humans often have a mistaken belief that a random event that already occurred will not happen again, which is called the Monte Carlo fallacy.

2.1 | Permutations and Probability

One reason humans have a hard time accepting and understanding random events is because they fail to realize that any permutation of a sequence of independent numbers/events is equally likely. This includes, of course, a permutation in which every event is in order, reverse order, or in another seemingly un-random order. Because the event of a particular song being played is independent from any other song being played, each song has an equal chance of being played in each location in the sequence. The probability of song 1 being played at any given time is 1/n, where n is the number of songs in the playlist being shuffled. The probability of song 2 being played is also 1/n, and this is true for all n songs in the playlist. Therefore, any ordering of songs is equally likely.



Assume pink, green, and yellow squares are 3 different subgroups of a playlist (genre, artist, mood, etc.). Displayed are two possible outcomes of truly random shuffle that may not appear random to a listener/viewer.

For example, say you have a playlist of length 3 with songs ABC. Using statistics, we know that there are 3!, or 6, possible orderings of the 3 songs in this playlist, since the order of the songs matters. All possible permutations of this playlist are as follows:

ABC, ACB, BCA, BAC, CAB, CBA

Now, if you shuffle this playlist of 3 songs, you have a 1/6 chance of the playlist being played exactly in order. This remains true as the playlist length gets larger, so the playlist being played in order will always be a valid outcome.

2.2 | Monte Carlo Fallacy

Another human misconception when encountering a sequence of random events is known as the Monte Carlo Fallacy, or the Gambler's Fallacy. This is the incorrect belief that if something happens more frequently than normal in the past then it is less likely to happen again in the future, or vice versa. This is most commonly seen in gambling, which can lead to big losses and false confidence. For example, if a fair coin is flipped 3 times and lands on heads each time, usually observers would assume that the next flip will land on tails. It seems extremely unlikely that a coin would land on heads four times in a row, but because each coin flip is an independent event, it is a fair possibility. The truth is, the coin is just as likely to land on heads as it is tails, however unlikely it seems or how frequently it has happened in the past.

This fallacy applies to listening to music as well. If you listen to a particular song a lot and then shuffle your playlist and the song comes on first, you may think that your music is not shuffling. Similarly, if many songs of the same genre have already played, you may expect a song from a different genre to play. If this does not happen, you may think that the pattern you observe indicates that your playlist is not truly being shuffled. This misconception led listeners to believe that there was an issue with the Spotify shuffle algorithm when it was first launched, causing Spotify to change their approach.

3 | Types of Random Number Generators

In the real world, random numbers can be generated using a dice roll, picking out of a hat, shuffling a deck and picking a card, and many other ways, but random number generation works differently in technology. Computers need a code or algorithm for every operation they perform, since they are machines and do not have a mind of their own or any way to toss a die. Because of this, there are many different algorithms that have been created over time to produce a random

number or sequence of random numbers. There are two types of random number generators in software. The first is called a True Random Number Generator (TRNG) and the second is a Pseudorandom Number Generator (PRNG). Both will be explained below.

3.1 | TRNGs

True Random Number Generators are based on random events that occur in nature. These are truly random because events in nature are variable, making the generation indeterministic and therefore unpredictable. These random events can include climate changes, the cosmic microwave background, atmospheric noise, thermal noise, etc. This program will receive data from one of these physical phenomena, account for any bias in the measurement process, and then return a random number based on the data. This type of random number generation is generally slow because it relies on events in the physical world which have no time constraint and cannot be controlled. It is also quite expensive due to the instruments that need to be used and the cost of data acquisition. However, one very important upside to using true random number generators is their ability to provide security. Because they are not deterministic or predictable, it is almost impossible for someone to know what number is being generated. Most computer software programs do not use true random number generators because of the inconvenience of their speed and cost, but they are very useful for data encryption and gambling, as these are two areas that would be negatively impacted by someone knowing the generated number.

3.2 | PRNGs

Pseudorandom number generators are the more commonly used of the two generators. These are computer algorithms that start with a key/seed and then generate a random number based on a mathematical equation. This type of generator is efficient because it can produce many numbers in a short amount of time. Basic arithmetic is generally easy for computers to do quickly and without using much memory. The sequence of numbers generated is able to be reproduced at a later date if the seed is known and it is easy to predict future numbers in the sequence, which makes the generator deterministic. This can be considered both a good and a bad thing. In the instance of music software, deterministic generators are often good because it can produce a sequence of numbers that is able to jump easily from one to the next and then back again. For example, if you are listening to a playlist and want to listen to a song you heard 2 songs back, it is easy to skip back to that song because the paths are encoded by the generator. It also makes it easy to jump forward. However, deterministic generators can pose a problem for applications such as gambling, because if someone knows the seed and the algorithm, they can easily figure out the rest of the sequence and bet accordingly. The last major quality of PRNGs is that they are periodic. This means that if the algorithm goes on for long enough, the sequence will eventually repeat itself. This is not usually a desirable characteristic, but thankfully it is usually not an issue due to modern PRNGs' ability to have a large enough period. PRNGs are usually used for simulation, modeling, and other basic software applications.

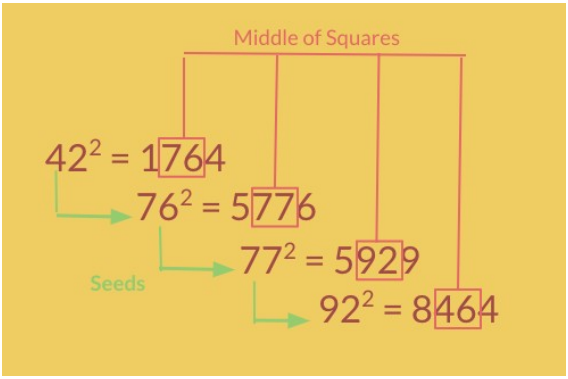
3.2.1 | Early PRNGs

Random Number Generators have been used as early as the 1940s and 1950s, but methods to manually compute random numbers were in use long before then.

The Middle Square Method was invented by John von Neumann and first described at a conference in 1949. This method, or some variation of it, may have first been described in the 13th century, but von Neumann was the first to implement and popularize it. The definition of this method is as follows:

> "To generate a sequence of *n*-digit pseudorandom numbers, an *n*-digit starting value is created and squared, producing a 2*n*-digit number. If the result is fewer than 2*n* digits, leading zeroes are added to compensate. The middle *n* digits of the result would be the next number in the sequence and returned as the result. This process is then repeated to generate more numbers"

There are some important caveats to consider in this method. First, *n* must be an even number because if it is odd, there is no guarantee that n^2 will have defined middle digits. For example, 345^2=119025. The middle digits in this situation could be 190 or 902, but then there is an uneven number of digits on each side of the number, making this method flawed. The other important quality of this generator that one needs to consider before implementing it is that for any generator of *n* digits, the period will be 8^n numbers long at the most. This makes the method generally undesirable and seldom used because of the short period, which, with enough repetition, will result in the program generating the same number or cycle to a previous number in the sequence and loop indefinitely. The last disadvantage of this approach to generating numbers is that each number produced has the same number of digits. This contributes to the short period and lack of unique values, but also is not very applicable to many applications that need a large range of numbers outputted in the sequence.



An example of the Middle Squares Method showing the seed, square of the seed, and then the output/new seed.

Another early method used is the Lehmer Generator, named after D.H. Lehmer and published in 1951. This is in the family of Linear Congruential Generators, which are the most common generators used and were published by W.E. Thomson and A. Rotenberg in 1958. The general formula for these algorithms is

$$X_{n+1} = (a * X_n + b) \bmod m, \text{ where}$$
a is called the "multiplier," $0 < a < m$
b is called the "increment," $0 \leq b < m$
X_0 is called the "seed" or "start value," $0 \leq X_0 < m$
m is called the "modulus," $0 < m$

Lehmer Congruential Generators are the specific type of Linear Congruential Generator in which $b = 0$, so there is no increment. Linear Congruential Generators are fast and easy for computers to use because they require little memory. These generators usually have very long periods given the correct parameters, making repetition unlikely. An example of parameters that are often used to create a period of length m is:

1. b is nonzero.
2. m and b are relatively prime, meaning the only common divisor of m and b is 1.
3. $a - 1$ is divisible by all prime factors of m
4. $a - 1$ is divisible by 4 is m is divisible by 4.

Linear Congruential Generators have very few weaknesses and are easily manipulated to create sequences of desired periods and randomness, so they are very commonly used in many modern-day applications.

3.2.2 | Modern PRNGs

Although Linear Congruential Generators are the most common modern generators, there are many other generators used in different applications. Xoroshiro128+ is just one example of the many generators available.

Xoroshiro128+ is a PRNG that uses the functions XOR, rotate, shift, and rotate to produce a sequence of random number generators. This type of generator is in a class known as shift-register generators which were discovered by George Marsaglia and are a subset of linear-feedback shift registers. These generators are able to produce the next number in the sequence by taking the 'exclusive or' of a number with a bit-shifted version of itself repeatedly. These are very good for software applications but cannot be executed on hardware. Parameters can be manipulated to create a long period, but like all pseudorandom number generators, there is risk of repetition given insufficient parameters. These generators are extremely fast, among the fastest of commonly used PRNGs, and have a very simple code. Another advantage of Xoroshiro128+ is that it supports jumping within the sequence in increments of 2^{16}. This jumping can allow multiple sequences to be generated that have no overlap or repetition. Although Xoroshiro128+ can be a very efficient and good algorithm to use, it sometimes does not pass all statistical tests, making it potentially unreliable.

4 | Spotify's approach

Spotify is very forthcoming with information about their engineering approach in their software. When Spotify first launched in 2008, they used an algorithm known as the Fisher-Yates

shuffle to produce shuffled sequences of songs in a given playlist or album. This algorithm got a lot of backlash from users who complained that their songs were not actually being shuffled. As discussed previously, this is likely due to human misconception of shuffling and the inclination towards patterns. However, because of this backlash, Spotify changed their algorithm to make "shuffled" queues less random but seemingly more shuffled by prioritizing uniform distribution above randomness.

4.1 | Fisher-Yates Shuffle

The Fisher-Yates Shuffle was first described in 1938 by Ronald Fisher and Frank Yates. When they created this algorithm, before computers were broadly in use, they performed the shuffle manually. It has since been adapted for computer use to shuffle a given array. When performed manually, you start with a collection of numbers you want to shuffle arranged in spots 1 through n. Then, by drawing out of a hat or rolling a die, pick a random number k such that $1 \leq k \leq p$, where p is the number of unstruck numbers remaining. Strike out the value in spot k and place it at the end of a separate list. Continue to pick a random number and add it to the end of the new list until all the numbers are crossed out and you have a new list of your original numbers in a new, shuffled, order. This returns a random permutation of the original numbers. This can simply be translated into a coding language in order to perform the algorithm in software applications.

Example: Say you want to shuffle the first 8 letters of the alphabet, A B C D E F G H. An example of how to do that using the Fisher-Yates algorithm is shown below.

Range	Roll (k)	Scratch	Result (new list)
		A B C D E F G H	
1-8	3	A B ~~C~~ D E F G H	**C**
1-7	4	A B ~~C~~ D ~~E~~ F G H	C **E**
1-6	5	A B ~~C~~ D ~~E~~ F ~~G~~ H	C E **G**
1-5	3	A B ~~C~~ ~~D~~ ~~E~~ F ~~G~~ H	C E G **D**
1-4	4	A B ~~C~~ ~~D~~ ~~E~~ F ~~G~~ ~~H~~	C E G D **H**
1-3	1	~~A~~ B ~~C~~ ~~D~~ ~~E~~ F ~~G~~ ~~H~~	C E G D H **A**
1-2	2	~~A~~ B ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~G~~ ~~H~~	C E G D H A **F**
1	1	~~A~~ ~~B~~ ~~C~~ ~~D~~ ~~E~~ ~~F~~ ~~G~~ ~~H~~	C E G D H A F **B**

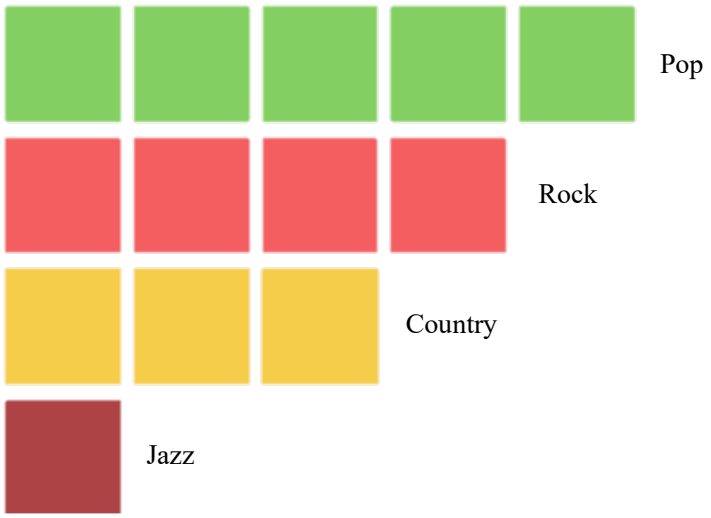
Because the returned sequence of numbers is a permutation and, as previously described, each permutation of the original numbers is equally likely, users were unhappy with this algorithm and its seeming un-randomness. This led Spotify engineers to create a new algorithm, described below.

4.2 | New Algorithm

The new algorithm that Spotify created to appease their critics analyzes the genre, artist, or mood of each song in the playlist and then places each song in a specific spot along the queue. This approach is similar to dithering, a photo tool used to minimize clusters when transforming a colored photo to a black and white photo. While dithering aims to avoid unwanted clumps of black

and white pixels, Spotify's algorithm aims to avoid unwanted clumps of a particular subgroup (genre, artist, etc.) by spreading them out as evenly as possible. The algorithm also incorporates the Fisher-Yates shuffle. It is easiest to understand this approach through an example.

Say you have a playlist of length 13 with 5 pop songs, 4 rock songs, 3 country songs, and 1 jazz song, represented below.



Since our playlist is 13 songs long, we can represent each spot in our playlist by blank squares:

Now that we have our playlist and empty queue set up, we can begin placing songs in their spots. We start with the genre with the highest frequency, which is pop. Since there are 5 pop songs, they should appear about every 20% of the playlist. To make the sequence more random, we can have a buffer zone and say that the pop songs will appear roughly every 15-25% of the playlist. This would be about every 2-3 songs.

After our pop songs are placed, we can begin placing our rock songs. These should go about every 20-30% of the way, so about every 3-4 songs, but adjusting as needed to account for the pop songs already placed. This gives us the following result:

We can do the same for our country and jazz songs, resulting in this shuffled playlist:

Now that we have spots designated to specific genres, we can use Fisher-Yates shuffle to order the specific songs of each genre within their allocated spots in the queue. And there you have it!

5 | Conclusion

Random numbers are used in many different ways and have many different real-world applications, especially in ways we do not even think about. Because of the general population's lack of knowledge of computing, random number generators, and statistics, sometimes these random number generators fall short of expectations of randomness. This is likely also in part because most random number generators that we come across are actually pseudorandom number generators, as true random number generators are extremely expensive and unnecessary. Spotify's early approach to generating random sequences of songs uses the Fisher-Yates shuffle, but when this proved to be insufficient for listeners' experience, they began to use an algorithm that actually makes the sequence less random but more "shuffled".

5.1 | Reflection

I found this project really interesting. I have always been a big music fan and used many different streaming platforms, so I loved learning more about how these platforms actually work and are connected to math. I also received a lot of positive feedback on my presentation, which was encouraging because I have never been confident in my public speaking, presenting, or even math skills. The class seems to feel similarly to me in that they also utilize these kinds of apps and are interested in learning more about how they work and how random number generators may be implemented in other applications they use.

I would love to research more about random number generators based on the feedback I received from the class. Some possible paths to explore that my peers recommended to me are the possibility of a nonuniform distribution of random number generators, whether there are other factors involved in the algorithm, such as listening history, and further research into the psychology of human perception of randomness. It was also fascinating reading thought my classmates' discussion posts and seeing how this intersection of music streaming and math can further be related to so many other fields of study, like history.

References

Arobelidze, Alexander. “Random Number Generator: How Do Computers Generate Random Numbers?” *FreeCodeCamp.org*, FreeCodeCamp.org, 8 June 2021, https://www.freecodecamp.org/news/random-number-generator/.

Arobelidze, Alexander. “Random Number Generator: How Do Computers Generate Random Numbers?” *FreeCodeCamp.org*, FreeCodeCamp.org, 8 June 2021, https://www.freecodecamp.org/news/random-number-generator/.

“The Art of Shuffling Music.” *KeyJs Blog RSS*, http://keyj.emphy.de/balanced-shuffle/.

Bendet, Naomi. “Is Spotify's Random Play Button Really Random?” *Medium*, UX Collective, 8 Feb. 2020, https://uxdesign.cc/randomly-not-random-2fd53536513c.

Bertoldi, David. “Building a Pseudorandom Number Generator.” *Medium*, Towards Data Science, 11 Nov. 2019, https://towardsdatascience.com/building-a-pseudorandom-number-generator-9bc37d3a87d5.

“Fisher–Yates Shuffle.” *Wikipedia*, Wikimedia Foundation, 31 Jan. 2022, https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle.

“Gambler's Fallacy.” *Wikipedia*, Wikimedia Foundation, 19 Mar. 2022, https://en.wikipedia.org/wiki/Gambler%27s_fallacy.

“Lehmer Random Number Generator.” *Wikipedia*, Wikimedia Foundation, 18 Oct. 2021, https://en.wikipedia.org/wiki/Lehmer_random_number_generator.

“Linear Congruential Generator.” *Linear Congruential Generator - Rosetta Code*, https://rosettacode.org/wiki/Linear_congruential_generator.

“Middle-Square Method.” *Wikipedia*, Wikimedia Foundation, 8 Dec. 2021, https://en.wikipedia.org/wiki/Middle-square_method.

Poláček, Published by Lukáš. “How to Shuffle Songs?” *Spotify Engineering*, 17 Mar. 2021, https://engineering.atspotify.com/2014/02/how-to-shuffle-songs/.

“Pseudo Random Number Generator (PRNG).” *GeeksforGeeks*, 6 Sept. 2019, https://www.geeksforgeeks.org/pseudo-random-number-generator-prng/.

“Squares: A Fast Counter-Based RNG.” *Papers With Code*, https://cs.paperswithcode.com/paper/squares-a-fast-counter-based-rng.

“Xoroshiro128+¶.” *Xoroshiro128+ - RandomGen v1.21.2 Documentation*, https://bashtage.github.io/randomgen/bit_generators/xoroshiro128.html.