Entropy Lost: Nintendo's Not-So-Random Sequence of 32,767 Bits

Trang Ngo
tqn1@williams.edu
Williams College
Williamstown, MA, USA

Aaron Williams
aaron.williams@williams.edu
Williams College
Williamstown, MA, USA

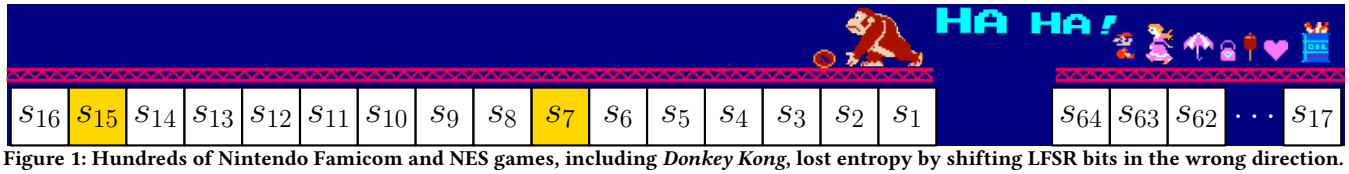


Figure 1: Hundreds of Nintendo Famicom and NES games, including *Donkey Kong*, lost entropy by shifting LFSR bits in the wrong direction.

ABSTRACT

Early video games didn't have hardware support for random number generation, so developers used software-based RNG. We show that hundreds of games in the Nintendo Famicom / NES library regenerate the same pseudorandom sequence of 32,767 bits, although the machine code for doing so comes in more than one hundred variants. We identified the disparate implementations using a simple regular expression that matches the following "fingerprint" of operations: `LDA`, `AND`, `STA`, `LDA`, `AND`, `EOR`, `CLC`, `BEQ`, `SEC`, `ROR`. These instructions implement a classic 15-bit linear feedback shift register associated with $x^{15} + x^7 + 1$ (i.e., tap the 15th and 7th bits). However, many games devoted more memory to the LFSR's state. For example, *Donkey Kong* (1983) used 8 bytes (or $8 \cdot 8 = 64$ bits), *The Legend of Zelda* (1986) used 13 bytes, and *Super Mario Bros. 3* (1988) used 9 bytes. In each case, additional entropy is lost by a simple programming error: the bits are shifted in the wrong direction.

KEYWORDS

Nintendo, Famicom, NES, RNG, linear feedback shift register, bug

1 INTRODUCTION

In the iconic first stage of *Donkey Kong* (1981), the eponymous ape rolls barrels down girders and ladders in an effort to thwart *Mario*'s ascent to *Pauline*[1]. The challenge uses randomization: There is a 25% chance that a barrel will roll downward from the top of a ladder.

1.1 *Donkey Kong's* Backwards ~~Barrels~~ Bits

Nintendo designed their first cartridge-based home console — the *Famicom* (, 1983) / *Nintendo Entertainment System* (, 1985) — so that it could play an accurate version of their new arcade hit [3]. Without dedicated hardware for generating random bits, Nintendo (or more precisely, Ikegami [27]) needed to realize this early example of procedural content generation using a software-based method. Their solution was a linear feedback shift register (LFSR) with 8-bytes (or $8 \cdot 8 = 64$ bits) of memory for its state. However, due to a programming error they only obtained 15-bits of randomization. As a result, the game continually cycles through the same sequence of 32,767 random bits, instead of 18,446,744,073,709,551,616 bits if the bytes had been leveraged optimally. Conveniently, the programming error can be understood by an apropos analogy: the barrels (i.e., bits) are rolled in the wrong direction.

[1]Or vice versa in the *Donkey Kong: Pauline Edition* (2013) hack by Mike Mika [28].

We show that the LFSR code in *Donkey Kong* was modified and optimized in dozens of different ways over the following decade, but the same underlying design (and limitation) rolled over. As a result, hundreds of Famicom/NES titles generate the same sequence of 32,767 "random" bits. Examples span the lifetime of the platform, starting with Famicom launch titles *Donkey Kong* and *Popeye* (1983), and ending with the NES's final commercial release *Wario's Woods* (1994). Many of these games appear to have had higher ambitions, as *Donkey Kong* (1983), *The Legend of Zelda* (1986), and *Super Mario Bros. 3* (1988), all dedicate at least 8 bytes to the LFSR. The flaw was also hard-coded in the BIOS of Nintendo's Japan-only peripheral, the *Famicom Disk System (FDS)* (1986).

1.2 Entropy Lost

We obtained our results by searching the catalog of Famicom/NES/FDS games for a particular "fingerprint" of machine code instructions. More specifically, we constructed a regular expression and used the command-line tool `bgrep` by `nneoneo` [34], which is a binary counterpart to the familiar text-based tool `grep`.

Our results point to a loss of entropy in several ways.

(1) A sequence isn't really "random" if hundreds of games use it.
(2) There is an upper-limit to the content that can be generated.
(3) We can pinpoint the randomization code and turn it off.

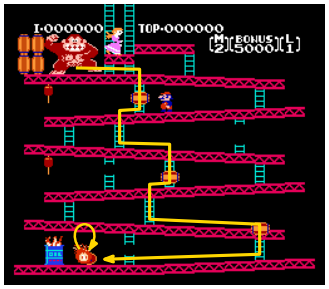
The first point may lead to RNG manipulation by speed-runners. The second point has been observed in *Dr. Mario* (1990) as there are 32,767 different puzzles per difficulty level [32, 33] [43] [31, 45]. The third point is seen in Figure 2, where "zeroing out" the LFSR state provides insight into how these classic games used RNG.

1.3 Outline

The rest of the paper is organized as follows.

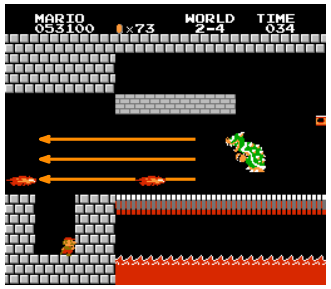
- Section 2 discusses the mathematics behind LFSRs, and the pursuit of an adjustable two-tap LFSR for the NES.
- Section 3 shows how an LFSR can be programmed correctly.
- Section 4 illustrates how a handful of games generate the same sequence of 32,767 bits in different ways.
- Section 5 provides the regular expression that we used to search the Famicom/NES/FDS libraries.
- Section 6 concludes with a summary of our results and methodology, additional results, future research, and additional resources found in an associated `git` repository [46].

We aim to make this material accessible, so sections begin with non-technical summaries. Links to technical results are also provided.



(a) *Donkey Kong* with PAR code `0018:00`. The barrels fall down every ladder that Mario is not blocking. The fireballs vibrate but do not move.

(b) *Duck Hunt* with PAR code `05EC:00`. Each duck follows one of two flight patterns alternating for even/odd ducks. The second pattern is shown above.



(c) *Super Mario Bros.* with PAR code `07A7:00`. Bowser's flames always proceed along the lowest of the three possible heights.

(d) *Excitebike* with PAR code `0018:00`. The CPU racers (cyan and purple above) do not change between the four lanes of the track, except to avoid dirt patches.

Figure 2: Removing randomization allows us to observe the baseline behavior in many NES / Famicom games: *Donkey Kong*'s barrels normally fall down ladders with 25% probability; the ducks in *Duck Hunt* continually change course; Bowser's flames in *Super Mario Bros.* follow different trajectories; rival racers *Excitebike* change lanes. These behaviors stop when randomization is turned off. More specifically, a single Pro Action Replay (PAR) code that zeroes out one byte of the LFSR's state in RAM will cause every random bit to be 0.

For background results on retrogame archeology and procedural content generation, see [5] and [20] [40]. The Famicom and NES use a modified MOS 6502 CPU, and the Easy6502 tutorial [30] is an excellent resource for those interested in quickly learning 6502 assembly programming. The mathematics behind LFSRs can be found in Golomb [19]. We use NES as a shorthand for Famicom/NES/FDS.

2 LINEAR FEEDBACK SHIFT REGISTERS

A linear feedback shift register is a simple and elegant method for generating *pseudo*random bits. The *pseudo* refers to the fact that the bits are not truly random. By way of analogy, suppose that we flipped a coin seven times, and the results were as follows:

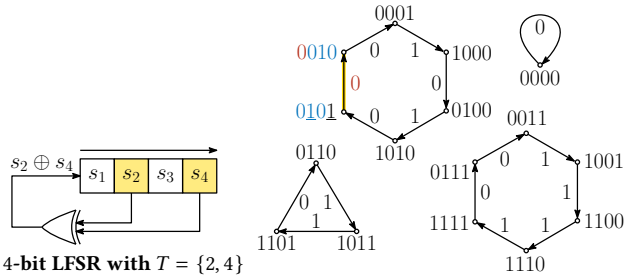
tails, tails, heads, tails, heads, heads, heads.

An observer may interpret this sequence as random. However, their opinion may start to change if the next seven flips yield the same sequence, as does the following seven flips, and so on. Similarly, every LFSR eventually starts repeating its sequence of bits. Applications choose to use a specific LFSR based on the length of sequence it generates, the speed in which it can generate each bit, and the number of bits of memory n its state requires. Unfortunately, there is no simple method for creating an LFSR with the maximum sequence length[2] for a given n. Instead, programmers use trial-and-error, or refer to previously generated results found in tables (e.g,. [41]).

LFSRs have been used for procedural content generation in video games dating back to *Pitfall!* (1982) on the Atari 2600 [29] [5]. In this pioneering example, David Crane used an 8-bit LFSR to generate the elements on 255 different screens. Since then, his code has been optimized and modified to create new adventures [44] [22].

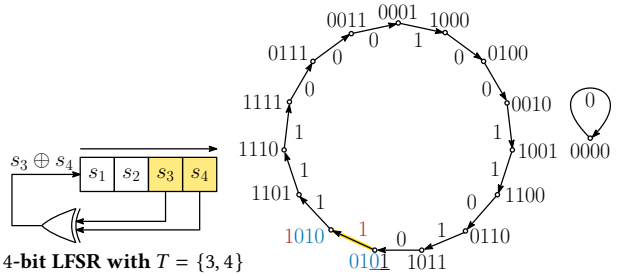
In Section 2.1 we illustrate LFSR concepts with examples, and compare them to de Bruijn sequences. Section 2.2 then considers two strategies for creating 'adjustable' LFSRs; Strategy A is *useless* and we'll see it again in Section 4. Section 2.3 provides the classic 15-bit LFSR that is the basis for the RNG in the games we study.

[2]Maximum-length LFSRs use *primitive polynomials* over $\mathbb{F}_2$ which are *irreducible* (i.e., like prime numbers they cannot be factored). Irreducible polynomials seem to be distributed randomly, although Turán's problem [6] suggests structure [12, 13, 18, 25].



A 4-bit LFSR with $T = \{2, 4\}$ (i.e., tap the 2nd and 4th bits). State graph w/ cycle lengths 1, 3, 6, 6.

(a) A 4-bit LFSR with polynomial $x^4 + x^2 + 1$. Starting at seed state 0101 it repeatedly generates the sequence 001010 of new random bits. (The sequence comes from the first bit of each state along the cycle.)



A 4-bit LFSR with $T = \{3, 4\}$ (i.e., tap 3rd and 4th bits). State graph w/ cycle lengths 1, 15.

(b) A maximum-length 4-bit LFSR with polynomial $x^4 + x^3 + 1$. Starting at seed state 0101 it repeatedly generates 111100010011010. Note: This sequence is only one bit shy of a de Bruijn sequence of order $n = 4$: Simply replace 000 with 0000 to obtain 1111000010011010.

Figure 3: An LFSR's behavior depends on its taps (shown in gold). Here (a) and (b) contain two 4-bit LFSRs and their state graphs. To see how they work, consider the transitions from state $s = s_1s_2s_3s_4 = 0101$. In (a) the next state is $s' = s'_1s'_2s'_3s'_4 = 0010$ because the new bit s'_1 equals $s_2 \oplus s_4 = 1 \oplus 1 = 0$. In (b) the next state is $s' = s'_1s'_2s'_3s'_4 = 1010$ because the new bit s'_1 equals $s_3 \oplus s_4 = 0 \oplus 1 = 1$. Note that the new red bit is computed from the tapped bits, and the blue bits shift right. These two transitions are highlighted (with tapped bits underlined) as edges in their state graphs above. Only (b) is a maximum-length LFSR, since its non-zero states form a single cycle of length $2^4 - 1 = 15$.

2.1 Definition and Examples

An *n-bit linear feedback shift register*[3] consists of an n-bit binary string $s = s_1 s_2 \cdots s_n$ and a set of *tap indices* $T \subseteq \{1, 2, \ldots, n\}$. The LFSR's *state* is the value of the binary string. An LFSR is *incremented* to the *next state* by updating the state to $s' = s'_1 s'_2 \ldots s'_n$ as follows

$$s'_{i+1} = s_i \text{ for all } 1 \le i < n \text{ and } s'_1 = \bigoplus_{i \in T} s_i \tag{1}$$

In other words, each bit in s is shifted into the next index in s', and a new value enters as the first bit in s'. The value of the *new bit* s'_1 is the xor of the tapped bits in s. An n-bit LFSR with that taps its highest bit (i.e., $n \in T$) can be specified by an associated polynomial $(\sum_{i \in T} x^i) + 1$ over $\mathbb{GF}(2)$. These ideas are illustrated in Figure 3.

Each increment produces one new *random bit*. Typically, the new bit s'_1 is treated as the random bit, and the remaining bits are then a recording of previously generated bits. For example, if the current state is $s = s_1 s_2 s_3 s_4 = 0111$, then 0 is the most recent random bit.

2.1.1 State Graphs and Cycle Lengths. Repeated increments of an n-bit LFSR will eventually return it to a previous state. This is visualized by an LFSR's *state graph*. The vertices of this graph are the n-bit binary strings, and there is a directed edge from state s to its next state s' that is labeled with the new bit s'_1. The edge labels along each cycle give the sequence of new bits generated by the LFSR. For example, the LFSR in Figure 3a cycle generates 001010 when starting from 0101, whereas Figure 3b generates 111100010011010. The state graph always contains a *loop* (i.e., a cycle of length one) around the all-zero string $s = 00 \cdots 0$ because its tapped bits are zero, and so $s'_1 = 0$ by (1). More broadly, we are interested in the list of *cycle lengths* in the state graph of an LFSR.

2.1.2 Maximum-length LFSRs. An n-bit LFSR is *maximum-length* if its cycle length list is $1, 2^n - 1$. That is, every non-zero string is in one cycle. Maximum-length LFSRs always have two properties:

(1) $n \in T$ (i.e., the highest state bit is tapped);
(2) $|T|$ is even (i.e., the number of taps is even).

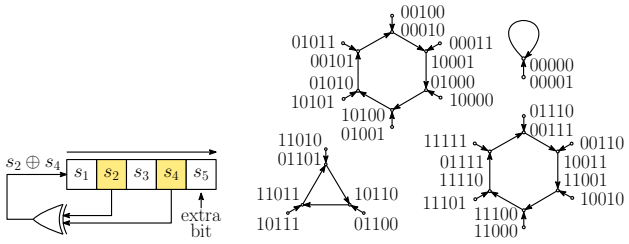
These conditions alone are not sufficient, as seen in Figure 3a.

2.1.3 De Bruijn Sequences. A *de Bruijn sequence of order n* is a cyclic sequence of length 2^n containing each n-bit binary string exactly once as a substring. Unlike LFSRs, most constructions and algorithms for generating these pseudorandom sequences are easily "adjustable" (i.e., they work for all n) so their use could have avoided the errors found in this article. This includes the *granddaddy de Bruijn sequence* (as named by Knuth [23]) from the 1930s [26] which can be generated efficiently [14, 35]. There are also new approaches [1, 2] [10, 11] [21, 38, 39] and frameworks [16, 17] whose sequences appear more or less random [15] or intentionally include fewer strings [36, 42] [8] [9, 37]. A maximum-length LFSR's non-zero sequence is one bit shy of a de Bruijn sequence (see Figure 3b).

2.2 Towards 'Adjustable' LFSR Randomization

Some games require more robust randomization than others, so a function providing an 'adjustable' amount of randomization would be useful. In a low-memory environment like the NES, the function should use less memory when generating less robust randomization.

[3] Our discussion will be limited to *Fibonacci* (cf. *Galois*) LFSRs.



A 5-bit LFSR with $T = \{2, 4\}$ (i.e., tap the 2nd and 4th bits) | **State graph w/ grove lengths 1, 3, 6, 6 and heights $h = 1$.**

(a) Strategy A: add one useless bit to the LFSR in Figure 3a. The same type of growth from cycles to groves occurs starting from Figure 3b.

A 5-bit LFSR with $T = \{3, 5\}$ and cycle lengths 1, 31 (cf. 1, 3, 6, 6). | **A 5-bit LFSR with $T = \{4, 5\}$ and cycle lengths 1, 3, 7, 21 (cf. 1, 15).**

(b) Strategy B: add one risky bit to the LFSRs in Figure 3.

Figure 4: Illustrating two potential strategies for creating an "adjustable" LFSR in which the taps stay in the same relative positions. Strategy A adds bits above the highest tap, while Strategy B adds bits below the lowest tap. These strategies are illustrated here by adding a single bit to the 4-bit LFSRs in Figure 3. If the goal is to increase the pseudorandom sequence length by adding bits, then Strategy A is useless (see Figure 5), while Strategy B is risky.

This goal leads to the following idea: Start with an existing n-bit LFSR, and add *extra* bits of memory if requested. More specifically, we could initialize an $(n+e)$-bit LFSR, which is identical to a known n-bit LFSR, but it uses e extra bits of memory, where e is a parameter set by the game. The extra bits could be added in one of two places.

Strategy A: After the highest tap. These extra bits will be *useless*.

Strategy B: Before the lowest tap. These extra bits will be *risky*.

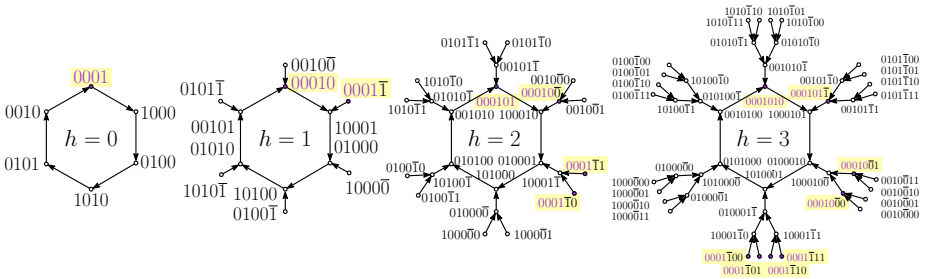
Either way, the taps shift together and stay in the same relative positions. This is analogous to playing a piano chord on different octaves, as your fingers (i.e., the taps) keep the same relative positions. Now we consider the effectiveness of the two strategies.

2.2.1 ***Strategy A***. This strategy is useless for gaining randomization. To start understanding this point, consider Figures 3a–4a. Both state graphs have four subgraphs, which are cycles of length 1, 3, 6, 6 in Figure 3a with appendages in Figure 4a. As a result, both LFSRs ultimately generate repeating sequences of length at most 6. Further explanation requires some basic graph theory terms [7].

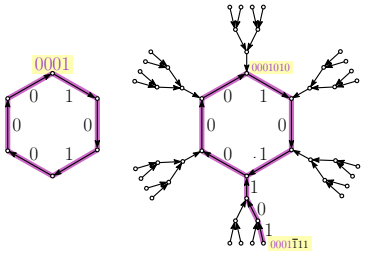
If an n-bit LFSR taps the highest bit (i.e., $n \in T$), then its state graph partitions into directed cycles (e.g., see Figure 3). Otherwise, its state graph partitions into more complicated subgraphs. A *grove of length ℓ and height h* is a directed cycle of length ℓ in which the root of a perfect binary tree[4] of height $h - 1$ is attached to each vertex of the cycle, with additional edges directed towards it. A grove of height $h = 0$ has no appendages and is simply a cycle.

If an n-bit LFSR has highest tap $t^* = \max(T)$ with $t^* \le n$, then its state graph partitions into groves of height $h = n - t^*$. Furthermore, the grove lengths are the same as the cycle lengths in the t^*-bit

[4] A *perfect binary tree of height h* contains 2^h leaves and $2^{h+1} - 1$ total vertices, with each leaf having distance h from the root. When $h = 0$ it is simply a single root vertex.



(a) Groves of height $h = 0, 1, 2, 3$ are generated by the 4-bit LFSR $x^4 + x^2 + 1$ with h useless bits. The $h = 0$ grove is a cycle in Figure 3a, while the $h = 1$ grove is in Figure 4a. Each grove has one highlighted state with prefix 0001 on its cycle; it is a cyclic substring of its cycle's sequence 000101. The other highlighted states differ in a suffix of length $\leq h$ (with the first differing bit overlined).



(b) The sequences generated from non-cycle vertices are the same as the sequences generated from cycle vertices. For example, the highlighted strings in (a) generate the sequence 101000 101000 $\cdots$.

Figure 5: Strategy A does not increase randomness: (a) the useless bits add height to the groves; (b) the generated bit sequences are unchanged.

LFSR with taps T (i.e., the LFSR without the extra bits). For example, consider Figures 3a–4a. Note that the 4-bit version of the LFSR has cycle lengths 1, 3, 6, 6 while the 5-bit version of the LFSR has grove lengths 1, 3, 6, 6. See Figure 5a for examples of taller groves.

Although taller groves contain more states than shorter groves, they do not increase the lengths of the random sequences generated by an LFSR. More specifically, the sequence generated starting from a state outside of the cycle simply duplicates the sequence generated from one of the states on the cycle. This point is explained in Figure 5b. For this reason, Strategy A is both predictable and useless.

2.2.2 ***Strategy B.*** This strategy is risky for gaining randomization. This is shown by Figures 3–4b; note that the extra bit can cause the cycle lengths to increase or decrease. More specifically, the extra bit transforms the not maximum-length 4-bit LFSR in Figure 3a into a maximum-length 5-bit LFSR in Figure 4b (left). On the other hand, the extra bit transforms the maximum-length 4-bit LFSR in Figure 3b into a not maximum-length 5-bit LFSR in Figure 4b (right).

As mentioned earlier, there is no simple way to determine the cycle lengths of an LFSR. Moreover, there is no simple way to predict how adding extra bits change the cycle lengths. In general, we expect more bits to lead to more randomization, but this is not guaranteed. Hence, Strategy B can be beneficial or detrimental.

2.3 A Classic 15-Bit LFSR with Two Taps

When working on an 8-bit system, it makes sense to allocate some number of bytes to an LFSR's state. In particular, a two-byte LFSR has $2 \cdot 8 = 16$ state bits. However, there is a drawback to choosing $n = 16$ bits. Recall that a maximum-length LFSR requires an even number of taps, so the fewest number of taps is two. Reducing the number of taps is helpful, since implementations often use more instructions when more taps are used (see Section 4). Unfortunately, there is no 16-bit LFSR using only two taps, so at least four taps are required. This leads to an appealing alternative choice:

$$n = 15 \text{ with } T = \{7, 15\} \text{ or equivalently, } x^{15} + x^7 + 1. \qquad (2)$$

This two-tap LFSR is maximum-length. Its taps are also *byte-separated* meaning that all of its tap positions differ by full bytes[5]. This property simplifies the LFSR's implementation since the taps are aligned on a single bit (i.e., on b_1 in this case). We name it *Nintendo's classic 15-bit LFSR*, although its exact origin is unknown to the authors.

3 PROGRAMMING AN LFSR

Programming an LFSR on the NES involves two design decisions:

(1) *Bit direction.* The direction in which to shift the bits.
(2) *Byte order.* The order in which to shift successive bytes.

We illustrate these decisions in Section 3.1 by visualizing Donkey Kong rolling barrels along one or more girders[6]. With this analogy, the key to a correct implementation is to roll each barrel over the highest tapped bit last (excluding useless extra bits).

In Section 3.2, we turn our attention to 6502 programming.

3.1 Design Decisions

In this subsection we consider implementing an LFSR in a high-level data structure, and then in 8-bit assembly language.

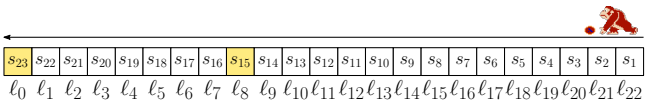
3.1.1 List Implementation: Bit Shift Direction. Let's first consider a simplified setting for implementing an LFSR: Each bit of the state $s = s_1 s_2 \cdots s_n$ (1-based indexing) is stored as an entry in list $\ell = [\ell_0, \ell_1, \ldots, \ell_{n-1}]$ (0-based indexing). In this setting, we only need to make the first design decision: the direction to shift the bits/entries. If we shift right, with the newly computed bit entering on the left, then we need to place the highest tap on the right side. Otherwise, if we shift left, then we need to put the highest tap on the left side. These two options are illustrated in Figure 6.

3.1.2 8-Bit Assembly Implementation: Byte Order. When we implement an LFSR in an 8-bit assembly language, we again need to decide the direction in which to shift the bits. However, it requires an additional design decision: the order in which the bytes are shifted. The two decisions lead naturally to $2 \cdot 2 = 4$ correct implementations of an LFSR, as illustrated in Figure 7.

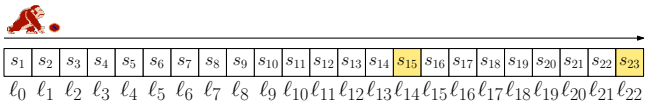
We also note that the byte order can't be implemented incorrectly in the classic 15-bit LFSR from Section 2.3. This is because the taps are byte-separated (i.e., in the same positions on each state byte) and there is a tap on each byte. Thus, each byte plays the same role. It's possible that the flexibility in this special case contributed to errors when programmers added extra bytes to the LFSR.

[5]This LFSR is byte-separated since its taps are one byte apart, i.e., $15 - 7 = 8 = 1 \cdot 8$.

[6]In these images, the barrels roll in one direction, not back-and-forth as in the game.

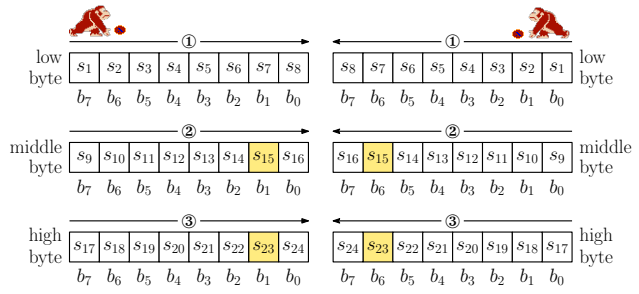


(a) Shifting bits to the left through a list of length $n = 23$.



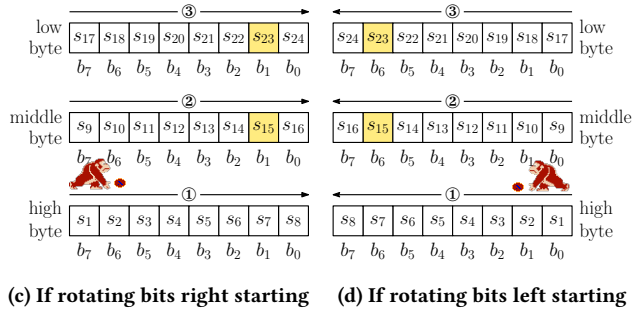
(b) Shifting bits to the right through a list of length $n = 23$.

Figure 6: When implementing an LFSR in a high-level data structure like a list, we can allocate one entry per bit. Increments can shift the entries to the right or the left, and this decision determines where the taps should be located. These example show the correct state bit locations in $s_1 s_2 \ldots s_{23}$ for the 23-bit LFSR with taps $T = \{15, 23\}$.



(a) If rotating bits starting from the low byte, then tap the b_1 bits on the middle and high bytes.

(b) If rotating bits starting from the low byte, then tap the b_6 bits on the middle and high bytes.



(c) If rotating bits right starting from the high byte, then tap b_1 on the middle and low bytes.

(d) If rotating bits left starting from the high byte, then tap b_6 on the middle and low bytes.

Figure 7: Four correct ways to implement a 23-bit LFSR with taps $\{15, 23\}$ on an 8-bit CPU like the MOS 6502. The implementations can shift each byte's bits to the right or left, and can perform the three shifts starting from the lowest or highest addressed byte.

3.2 6502 Programming

3.2.1 Instructions. A 6502 *instruction* consists of an *operation* together with *operands*. Most instructions read or write the *accumulator register* `A` or an *auxiliary register* `X` and `Y`. A *status byte* includes the *carry flag* bit. The following operations will be used when we consider specific LFSR implementations in Section 4.

- Tapping a bit is accomplished with `AND`. Specifically, tapping bit b_1 is `AND #$02`, while tapping bit b_6 is `AND #$40`.
- Tapped values are combined with `EOR` (xor). This step is simplified in the classic LFSR as its taps are byte-separated.
- A byte shifts left or right with `ROL` or `ROR`, respectively. The carry flag shifts in, and then it is set to the bit shifted out.
- The branching operations `BEQ` and `BNE` jump over some number of bytes (or to a *label*) if `A` is equal or not equal to 0.
- Enter and exit a subroutine with `JSR` and `RTS`; goto with `JMP`.
- The accumulator `A` is loaded with a byte using `LDA`, and its byte is stored in memory with `STA`.

The three letter names (e.g., `ROL`) are an operation's *mnemonic*.

3.2.2 Addressing Memory. The 6502 is an 8-bit CPU but it has 16-bits of addressable memory. If the operands specify a full two-byte address, then the operation uses *Absolute mode*. If the first byte is omitted, then it is assumed to be zero. Operations using this feature use *Zero Page mode* in reference to the first 256 bytes of memory. Other *addressing modes* may use `X` or `Y` as an offset. An operation whose operand is a constant is said to use *Immediate mode*.

4 CLASSIC NINTENDO GAMES

This section examines a handful of important NES games and their LFSRs. Each classic game uses the classic LFSR from Section 2.3 with some number of extra bytes. Due to programming errors, the extra bytes are always useless, rather than risky, as per Section 2 and 3. Section 3.2 can be used as a reference when reading the code.

We begin with a launch title for the Famicom in Section 4.1 and end with the last commercial release for the NES in Section 4.5.

4.1 *Donkey Kong* (1983)

The LFSR increment code for *Donkey Kong* is in Figure 8b. Additional initialization code is provided in Figure 8a for those who wish to run the code. The LFSR allocates 8 bytes of memory for the state at addresses `$18`–`$1F`. The intention was to add 6 extra bytes to the classic LFSR from Section 2.3. However, it places the extra bytes on the wrong side of the taps, thus the extra bytes are useless, and the 15-bit LFSR is generated instead of a 63-bit LFSR.

The same incorrect implementation appears in *Donkey Kong Jr.* (1983) and *Donkey Kong Jr. Math* (1983). It has three easy fixes: change the tapped bits or byte order or shift direction. Each fix appears in 8c–8e. Nearly identical code appears in *Spartan X* ( , 1984) / *Kung Fu* ( , 1985) at a different location, *Excitebike* (1984), and *Clu Clu Land* (1984) with the LFSR state in addresses `$10`–`$17`.

4.2 Launch Titles including *Popeye* (1983)

Among the three launch titles for the Famicom (*Donkey Kong*, *Donkey Kong Jr.*, and *Popeye*), the less "ambitious" *Popeye* is the only one which implemented the LFSR correctly without useless bytes.

One of the NES launch titles, *Duck Hunt* (1985), also adds useless bytes. Figure 9b shows that it deviates from *Donkey Kong* by only wasting 4 bytes, and by storing the state bytes off of the zero-page at `$05EC`-`$05EF`. This same code appears in *Wild Gunman* (1985).

So far for the Famicom/NES's launch titles, we have seen variations to the *Donkey Kong* code such as different location of LFSR code (e.g. *Kung Fu*), different number of state bytes used, and using the non-zero page mode (*Duck Hunt* and *Wild Gunman*).

```
start:
  jsr init
  jmp loop

init:
  LDA #$FF
  STA $18
  STA $19
  STA $1A
  STA $1B
  STA $1C
  STA $1D
  STA $1E
  STA $1F
  RTS

loop:
  jsr LFSR_inc
  jmp loop
```

(a) Initialization. Use this auxiliary code to run (b) in Easy6502.

```
LFSR_inc:
  LDA $18   ; load lowest byte
  AND #$02  ; tap b1 (7th bit)
  STA $00   ; temporary storage
  LDA $19   ; load next byte
  AND #$02  ; tap b1 (15th bit)
  EOR $00   ; xor aligned taps
  CLC       ; clear carry
  BEQ rors  ; test the xor
  SEC       ; set carry
rors:
  ROR $18   ; shift lowest byte
  ROR $19   ; shift next byte
  ROR $1A   ; shift useless byte
  ROR $1B   ; shift useless byte
  ROR $1C   ; shift useless byte
  ROR $1D   ; shift useless byte
  ROR $1E   ; shift useless byte
  ROR $1F   ; shift useless byte
  RTS       ; return
```

(b) Incorrect implementation of the LFSR increment in *Donkey Kong*.

```
LFSR_inc:
  LDA $1E
  AND #$02
  STA $00
  LDA $1F
  AND #$02
  EOR $00
  CLC
  BEQ rors
  SEC
rors:
  ROR $18
  ROR $19
  ROR $1A
  ROR $1B
  ROR $1C
  ROR $1D
  ROR $1E
  ROR $1F
  RTS
```

(c) Fix 1: Tap the highest two bytes not lowest.

```
LFSR_inc:
  LDA $18
  AND #$02
  STA $00
  LDA $19
  AND #$02
  EOR $00
  CLC
  BEQ rors
  SEC
rors:
  ROR $1F
  ROR $1E
  ROR $1D
  ROR $1C
  ROR $1B
  ROR $1A
  ROR $19
  ROR $18
  RTS
```

(d) Fix 2: Start from the highest byte not lowest.

```
LFSR_inc:
  LDA $18
  AND #$40
  STA $00
  LDA $19
  AND #$40
  EOR $00
  CLC
  BEQ rols
  SEC
rols:
  ROL $1F
  ROL $1E
  ROL $1D
  ROL $1C
  ROL $1B
  ROL $1A
  ROL $19
  ROL $18
  RTS
```

(e) Fix 3: Shift left from highest byte; adjust taps.

Figure 8: (a)–(b) *Donkey Kong* (1983) accidentally implements the classic 15-bit LFSR with taps $T = \{7, 15\}$ instead of a 63-bit LFSR with $T = \{55, 63\}$. This is because it adds 6 bytes on the wrong side of the taps (i.e., Strategy A not Strategy B). This error limits the length of the generated pseudorandom sequence to $2^{15} - 1 = 32{,}767$ bits. The code can be fixed in several different ways: (c)–(e) follow Figure 7a, 7c, 7d, respectively.

```
LFSR_inc:
    LDA $18
    AND #$02
    STA $00
    LDA $19
    AND #$02
    EOR $00
    CLC
    BEQ rors
    SEC
rors:
    ROR $18
    ROR $19
    RTS
```

(a) *Popeye* (1983).

```
LFSR_inc:
    LDA $05EC
    AND #$02
    STA $07
    LDA $05ED
    AND #$02
    EOR $07
    CLC
    BEQ rors
    SEC
rors:
    ROR $05EC
    ROR $05ED
    ROR $05EE
    ROR $05EF
```

(b) *Duck Hunt* (1985).

```
LFSR_inc:
  LDX #$00
  LDY #$07
  LDA $07A7
  AND #$02
  STA $00
  LDA $07A8
  AND #$02
  EOR $00
  CLC
  BEQ rors
  SEC
rors:
  ROR $07A7,X
  INX
  DEY
  BNE rors
```

(c) *Super Mario Bros.* (1985).

```
LFSR_inc:
  LDA $00,X ; A = mem[0+X]
  AND #$02  ; A &= 2
  STA $00   ; mem[0] = A
  LDA $01,X ; A = mem[1+X]
  AND #$02  ; A &= 2
  EOR $00   ; A ^= mem[0] (nb. sets Z)
  CLC       ; Clear carry bit (C = 0)
  BEQ rors  ; If Z == 0, branch to rot
  SEC       ; Else, set carry (C = 1)
            ; Thus, C = Z = (A != 0)
rors:
  ROR $00,X ; Rotate byte mem[0+X]
  INX       ; X = X + 1
  DEY       ; Y = Y - 1
  BNE rot   ; If Y == 0, branch to rot
  RTS       ; Return from LFSR_inc
```

(d) *Famicom Disk System BIOS* (1986).

```
LFSR_inc:
  LDA $0580
  AND #$02
  STA $00
  LDA $0581
  AND #$02
  EOR $00
  CLC
  BEQ rors
  SEC
rors:
  ROR $0580
  ROR $0581
  RTS
```

(e) *Wario's Woods* (1994).

Figure 9: LFSR implementations in classic games and systems, ranging from a launch title for the Famicom in (a) to the final NES release in (e). Each code block implements the LFSR associated with $x^{15} + x^7 + 1$ and repeatedly generates the same sequence of 32,767 bits. However, that was not the intention of (b)–(d), which all repeat the *Donkey Kong* error in Figure 8. The correct implementations in (a) and (e) use two bytes for the LFSR's state starting at zero-page address `$18` and non-zero-page address `$0580`, respectively. In (b) four bytes starting at `$05EC` are used, while (c) uses eight bytes starting at `$0727` and puts the `ROR`s in a loop. In (d) the `X` and `Y` registers are used as parameters: `X` gives an offset that allows multiple LFSRs to exist simultaneously; `Y` gives the number of bytes for the LFSR state, although all values of `Y` > 2 lead to useless bytes.

4.3 *Super Mario Bros.* (1985)

After the Famicom/NES's launch titles were released, the LFSR code evolved with sophisticated 6502 programming. For example, the repeated `ROR` instructions are rolled into a loop by using the `X` and `Y` registers. This is seen in *Super Mario Bros.* (see Figure 9c) which still makes the same mistake as *Donkey Kong* with its extra bytes.

A close examination of the *Super Mario Bros.* loop shows that the `X` and `Y` registers are used inefficiently: `X` stores an offset to the shifted byte (and is initialized to 0); `Y` stores the number of bytes to shift. An obvious simplification is to initialize `X` to the number of state bytes and ignore `Y`. In fact, this change would also *fix* the LFSR by shifting the bytes starting from the highest byte as in Figure 7c.

4.4 *Famicom Disk System BIOS* (1986)

The LFSR code continued to be refined after *Super Mario Bros.* Instead of hardcoding the X and Y registers, they are now used as parameters to be set before calling the increment code.

(1) The X register continues to provide an offset into the zero page. This is where the LFSR state bytes are stored.
(2) The Y register continues to provide the number of state bytes.

The first point allows the LFSR state to be anywhere on the zero page, and for multiple LFSRs to be active simultaneously. The second point aims to allow for an adjustable LFSR — associated with $x^{8Y-1} + x^{8Y-9} + 1$ instead of $x^{15} + x^7 + 1$ — but again, the code is implemented incorrectly. A version of this code was included in the *Famicom Disk System*'s BIOS, as seen in Figure 9d. To call the increment code, a game could set the X and Y registers as follows.

```
LDX #$17     ; e.g., state bytes start at X = 17
LDY #$0D     ; e.g., the state is 13 bytes long
JSR LFSR_inc ; run increment code
```

4.5 *Wario's Woods* (1994)

In the last NES game, *Wario's Woods* (1994), the LFSR code reverts to the classic 15-bit LFSR using only two bytes, as in Figure 9e.

Interestingly, this LFSR is initialized to zero and stays in the zero-cycle. However, if we edit the state, then it exits the zero-cycle and continually updates. In other words, the LFSR is updated but is never actually used. This suggests that *Nintendo* may have been shifting away from LFSRs by the end of the NES era.

5 SEARCHING THE LIBRARY

In Section 4, we identified specific games that generate the same sequence of pseudorandom bits via the classic LFSR. To search the entire library, we'll listen for games that sing the following song.

> Load AND Store, Load AND Xor, Clear Branch Set Roar!
>
> Nintendo's Random Song

In Section 5.1 and 5.2 we describe the idea behind our search, and how to automate it. In Section 5.3 we report on the matches.

5.1 Fingerprint

The implementations from the previous section are quite varied, but they have one important similarity: Each implementation includes a progression of ten consecutive instructions with the same mnemonics. This is shown in Figure 10a and is sung[7] at the start of Section 5. We refer to this sequence of mnemonics as our *fingerprint*. Our approach is to search the libraries for *matches*, which are consecutive instructions with this fingerprint; see Figure 10b–10c.

5.2 Regular Expressions and bgrep

To search for an individual instruction in a ROM file, we need to consider the different opcodes for the operation and the number of bytes that follow the opcode. For example, an LDA could be specified by opcode 0xA5 followed by one byte, or by 0xAD followed by two bytes. There are eight forms of LDA, as summarized by Table 1. Similarly, there are eight forms of AND, as in Table 2.

```
 1. Load (LDA)            LDA $18 ;A5 18  LDA $0580;AD 80 05
 2. And (AND)             AND #$02;29 02  AND #$02 ;29 02
 3. Store (STA)           STA $00 ;85 00  STA $00  ;85 00
 4. Load (LDA)            LDA $19 ;A5 19  LDA $0581;A9 81 05
 5. And (AND)             AND #$02;29 02  AND #$02 ;29 02
 6. Xor (EOR)             EOR $00 ;45 00  EOR $00  ;45 00
 7. Clear carry bit (CLC) CLC     ;18     CLC      ;18
 8. Branch if equal (BEQ) BEQ #$+1;F0 01  BEQ #$+1 ;F0 01
 9. Set carry flag (SEC)  SEC     ;38     SEC      ;38
10. Rotate right (ROR)    ROR $18 ;66 18  ROR $0580;6E 80 05
```

(a) Fingerprint. **(b) *Donkey Kong*.** **(c) *Wario's Woods*.**

Figure 10: Our song's fingerprint of three-letter mnemonics in (a) with examples of matched instructions and machine code in (b)–(c).

Mode	Opcodes	Bytes
Immediate	$A9	2
Zero Page	$A5	2
Zero Page, X	$B5	2
Absolute	$AD	3
Absolute, X	$BD	3
Absolute, Y	$B9	3
(Indirect, X)	$A1	2
(Indirect), Y	$B1	2

Table 1: LDA instructions.

Mode	Opcodes	Bytes
Immediate	$29	2
Zero Page	$25	2
Zero Page, X	$35	2
Absolute	$2D	3
Absolute, X	$3D	3
Absolute, Y	$39	3
(Indirect, X)	$21	2
(Indirect), Y	$31	2

Table 2: AND instructions.

Therefore, an LDA operation followed by an AND operation can be realized by $8 \cdot 8 = 64$ different pairs of opcodes, which can be easily described using a single regular expression. As a starting point, the following regular expression matches any single hex character.

```
h = 0+1+2+3+4+5+6+7+8+9+A+B+C+D+E+F
```

For those unfamiliar with regular expressions, each + can be interpreted as "or", so the full expression is "0 or 1 or … or F". Regular expressions can be grouped together using parentheses, and concatenation is implied when two expressions are written next to each other. Thus, the following regular expression matches the byte sequence of an instruction with mnemonic LDA. Specifically, the left side matches the two-byte versions of the operation, and the right side matches the three-byte versions.

```
LDA = (A9 + A5 + B5 + A1 + B1)hh + (AD + BD + B9)hhhh
```

We can similarly define expressions for every 6502 operation. This is the regular expression for our operation fingerprint.

```
(LDA)(AND)(STA)(LDA)(AND)(EOR)(CLC)(BEQ)(SEC)(ROR)
```

To search for regular expressions in the Famicom and NES libraries, we use nneonneo's bgrep [34], a binary variant of grep.

A bgrep regular expression for LDA appears below, where | and [] are forms of "or" (like +), while . matches one byte (like hh).

```
([\xA9\xA5\xB5\xA1\xB1].|[\xAD\xBD\xB9]..)
```

Similarly, we can construct bgrep regular expressions for each of the other 6502 operations. Concatenating the operations in the fingerprint gives the result in the left column in Table 3.

To search an individual NES file for the operation fingerprint, it is helpful to save the above pattern in a file named `fingerprint.re`.

[7] The first words in this list give the ten words in *Nintendo's Random Song* at the start of this section, with some artistic license taken in the last step.

([\xA9\xA5\xB5\xA1\xB1].|[\xAD\xBD\xB9]..)	LDA
([\x29\x25\x35\x21\x31].|[\x2D\x3D\x39]..)	AND
([\x85\x95\x81\x91].|[\x8D\x9D\x99]..)	STA
([\xB5\xA5\xA9\xA1\xB1].|[\xAD\xBD\xB9]..)	LDA
([\x29\x25\x35\x21\x31].|[\x2D\x3D\x39]..)	AND
([\x49\x45\x55\x41\x51].|[\x4D\x5D\x59]..)	EOR
\x18	CLC
\xF0.	BEQ
\x38	SEC
(\x6A|[\x66\x76].|[\x6E\x7E]..)	ROR

Table 3: bgrep regular expression for our fingerprint (left) and the corresponding three-letter mnemonics of the operations (right). For readability, each operation's expression is on a separate line; it must be written on one line—without a new line—when used with bgrep.

Again, the pattern should be saved on a single line, without a newline character at the end of the file[8]. Then the following command will search for the fingerprint in a ROM file named `game.nes`.

```
bgrep -E -f fingerprint.re game.nes
```

The `-E` denotes extended patterns (including `[]`), and `-f` signifies a search pattern file. Matches output the file name, the location in the file, the bytes that match the pattern (in **red**) with additional context (in **black**). For example, running the command on *Donkey Kong* (Famicom) gives the following output.

```
Donkey Kong (Japan).nes:00003504: 009d31038e300360a5
1829028500a5192902450018f00138661866619661a661b661c
```

Note that the matched **red** bytes appear in Figure 10b.

More broadly, we can run `bgrep` on all files in a `ROMS/` folder with the following command: `bgrep -E -f fingerprint.re ROMS/*`

5.3 Number of Matches

Using the `bgrep` command from the previous subsection, we searched all of the relevant ROMs in the well-known *No-Intro Collection* [4].

- 470 matches total matches across 374 different files.
- 179 titles after removing revisions and regional translations.

For example, *Dr. Mario* contributes 24 total matches across 9 different files. This is due to several prototypes being in circulation, including some under the working title *Virus*, and each includes multiple implementations. In addition, a regional variant and revision were released. The third total is our approximation for the number of distinct titles[9] , and *Dr. Mario* contributes +1 to this total.

In the Famicom Disk System library we found the following[10].

- 37 total matches across 33 different files.
- 28 titles after removing revisions and regional translations.

In total, we found 508 matches in 207 different titles.

6 FINAL REMARKS

We traced a specific pseudorandom number generator across hundreds of games in the NES/Famicom/FDS library. The classic linear feedback shift register — associated with $x^{15}x^7 + 1$ — is beautiful. It has maximum-length and two byte-separated taps, which allows it to be incremented elegantly and efficiently. Unfortunately, LFSRs are not naturally adjustable (cf. de Bruijn sequences in Section 2.1.3), and efforts to increase the LFSR's randomness were unsuccessful to the point of comedy. Programmers dating back to *Donkey Kong* (1983) added bytes to the wrong side of the taps, thus limiting the games to the same not-so-random sequence of 32,767 bits.

During the course of the article we illustrated how several games use RNG in Figure 2, explained the mathematics behind useless bits in Section 2.2, and showed how to fix *Donkey Kong*'s error by rotating bits in the other direction (or bytes in the opposite order) in Figure 8. The machine code from multiple games in Figure 9 led us to a song of ten mnemonics and a `bgrep` regular expression that was used to search the library.

6.1 Additional Results

In an extended version of the paper we will add the following details.

We reduced the chance of *false positives* (i.e., matches that do not implement the LFSR) by refining our `bgrep` regular expression. For example, `AND` operands were hardcoded to `#$02`, and the operands for `STA` and `EOR` were forced to be equal. None of our restrictions reduced the number of matches. We also found *false negatives* (i.e., non-matches that implement the LFSR) by relaxing the expression.

Not all programmers made the same mistake. In particular, three titles by *Culture Brain* expand the classic LFSR to $x^{63} + x^7 + 1$ (byte-separated) and correctly rotate the bits and bytes as in Figure 7c.

There are over 100 distinct sequences of machine code bytes in the NES/Famicom/FDS library that trigger a match. We also discovered matches in other console libraries, including the SNES.

6.2 Additional Resources

Files associated with this project can be found in [46]; see Figure 11.

110 in 1 (Asia) (En) (Unl) (Pirate).nes
Aa Yakyuu Jinsei Icchokusen (Japan).nes
Abadox (Japan).nes
Adventures in the Magic Kingdom (USA).nes

(a) `NESTitles.txt`.

10000001000000110000010000
00011100001010000110100001
00000011101001010100110111
10100010011100111101000111

(b) `32767bits.txt`.

Figure 11: Additional materials in our `git` repository [46] include (a) lists of matches, and (b) the pseudorandom sequence they generate.

6.3 Future Work

We hope this work promotes other tool-based studies on the implementation of important subroutines on various gaming platforms. Such studies will help improve CS history, as per Knuth's call [24].

REFERENCES

[1] Abbas Alhakim, Evan Sala, and Joe Sawada. 2021. Revisiting the prefer-same and prefer-opposite de Bruijn sequence constructions. *Theoretical Computer Science* 852 (2021), 73–77.

[2] Abbas M Alhakim. 2010. A simple combinatorial algorithm for de Bruijn sequences. *The American Mathematical Monthly* 117, 8 (2010), 728–732.

[3] Matt Alt. 2020. The Designer Of The NES Dishes The Dirt On Nintendo's Early Days. https://www.kotaku.com.au/2020/07/the-designer-of-the-nes-dishes-the-dirt-on-nintendos-early-days/.

[4] Internet Archive. 2020. [No-Intro] Nintendo - Nintendo Entertainment System. https://archive.org/details/nointro.nes.

[5] John Aycock. 2016. *Retrogame Archeology: Exploring Old Computer Games* (1st ed.). Springer, New York, NY, USA.

[6] Pradipto Banerjee and Michael Filaseta. 2010. On a polynomial conjecture of Pál Turán. *Acta Arith* 143, 3 (2010), 239–255.

[8] By default, some text editors add a newline character to the end of a file.

[9] For example, we count *Spartan X* (JP, 1984) and *Kung Fu* (US, 1985) as the same title, but not *Downtown Nekketsu Monogatari* (JP, 1989) and *River City Ransom* (US, 1990). We include non-gaming software (e.g., the *FamicomBox* menu) and several multicarts.

[10] These matches do not include games that call the LFSR code found in the FDS BIOS. In other words, these games include their own implementation of the classic LFSR.

[7] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. 1976. *Graph theory with applications*. Vol. 290. Macmillan London.

[8] Ben Cameron, Aysu Gündoğan, and Joe Sawada. 2022. Cut-Down de Bruijn Sequences. *arXiv preprint arXiv:2205.02815* (2022).

[9] Joseph DiMuro. 2019. Classifying rotationally-closed languages having greedy universal cycles. *The electronic journal of combinatorics* 26 (2019), P1.35. Issue 1.

[10] Patrick Baxter Dragon, Oscar I Hernandez, Joe Sawada, Aaron Williams, and Dennis Wong. 2018. Constructing de Bruijn sequences with co-lexicographic order: The k-ary Grandmama sequence. *European Journal of Combinatorics* 72 (2018), 1–11.

[11] Patrick Baxter Dragon, Oscar I Hernandez, and Aaron Williams. 2016. The grandmama de Bruijn sequence for binary strings. In *LATIN 2016: Theoretical Informatics*. Springer, 347–361.

[12] Michael Filaseta. 2014. Is Every Polynomial with Integer Coefficients Near an Irreducible Polynomial? *Elemente der Mathematik* 69, 3 (2014), 130–143.

[13] Michael Filaseta and Michael Mossinghoff. 2012. The distance to an irreducible polynomial, II. *Math. Comp.* 81, 279 (2012), 1571–1585.

[14] Harold Fredricksen and James Maiorana. 1978. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics* 23, 3 (1978), 207–210.

[15] Daniel Gabric and Joe Sawada. 2022. Investigating the discrepancy property of de Bruijn sequences. *Discrete Mathematics* 345, 4 (2022), 112780.

[16] Daniel Gabric, Joe Sawada, Aaron Williams, and Dennis Wong. 2018. A framework for constructing de Bruijn sequences via simple successor rules. *Discrete Mathematics* 341, 11 (2018), 2977–2987.

[17] Daniel Gabric, Joe Sawada, Aaron Williams, and Dennis Wong. 2019. A successor rule framework for constructing k-ary de Bruijn sequences and universal cycles. *IEEE Transactions on Information Theory* 66, 1 (2019), 679–687.

[18] Petar Gaydarov and Konstantin Delchev. 2015. Combinatorial Computations on an Extension of a Problem by Pál Turán. *Serdica Journal of Computing* (2015), 257–268.

[19] Solomon W. Golomb. 1981. *Shift Register Sequences* (1 ed.). Aegean Park Press.

[20] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications* 9, 1 (2013), 1–22.

[21] Cees JA Jansen, Wouter G Franx, and Dick E Boekee. 1991. An efficient algorithm for the generation of DeBruijn cycles. *IEEE Transactions on Information Theory* 37, 5 (1991), 1475–1478.

[22] Thomas Jentzsch. 2017. Pitfall!x256 (was: Pitfall!x16). https://atariage.com/forums/topic/267046-pitfallx256-was-pitfallx16.

[23] Donald E Knuth. 2005. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley Professional.

[24] Donald E Knuth and Len Shustek. 2021. Let's not dumb down the history of computer science. *Commun. ACM* 64, 2 (2021), 33–35.

[25] Gilbert Lee, Frank Ruskey, and Aaron Williams. 2007. Hamming distance from irreducible polynomials over $\mathbb{F}_2$. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) (Jan. 2007). https://doi.org/10.46298/dmtcs.3550

[26] Monroe H Martin. 1934. A problem in arrangements. *Bull. Amer. Math. Soc.* 40, 12 (1934), 859–864.

[27] Damien McFerran. 2018. Feature: Shining A Light On Ikegami Tsushinki, The Company That Developed Donkey Kong. https://www.nintendolife.com/news/2018/02/feature_shining_a_light_on_ikegami_tsushinki_the_company_that_developed_donkey_kong.

[28] Mike Mika. 2013. Why I hacked Donkey Kong for my daughter. *Wired, March* 11 (2013).

[29] Nick Montfort and Ian Bogost. 2009. *Racing the beam: The Atari video computer system*. Mit Press.

[30] Nick Morgan. 2012. Easy 6502 Simulator. https://skilldrick.github.io/easy6502/simulator.html.

[31] Trang Q. Ngo. 2020. Python implementation of Dr. Mario algorithms. https://github.com/trangqngo/Dr-Mario-virus-generation.

[32] nightmareci. 2012. Dr. Mario virus placement. https://tetrisconcept.net/threads/dr-mario-virus-placement.2037.

[33] nightmareci. 2013. Dr. Mario. https://tetris.wiki/Dr._Mario.

[34] Robert Xiao (nneonneo). 2015. bgrep. https://github.com/nneonneo/bgrep.

[35] Frank Ruskey, Carla Savage, and Terry Min Yih Wang. 1992. Generating necklaces. *Journal of Algorithms* 13, 3 (1992), 414–430.

[36] Frank Ruskey, Joe Sawada, and Aaron Williams. 2012. De Bruijn sequences for fixed-weight binary strings. *SIAM Journal on Discrete Mathematics* 26, 2 (2012), 605–617.

[37] Joe Sawada, Aaron Williams, and Dennis Wong. 2016. Generalizing the classic greedy and necklace constructions of de Bruijn sequences and universal cycles. *The electronic journal of combinatorics* (2016), P1–24.

[38] Joe Sawada, Aaron Williams, and Dennis Wong. 2016. A surprisingly simple de Bruijn sequence construction. *Discrete Mathematics* 339, 1 (2016), 127–131.

[39] Joe Sawada, Aaron Williams, and Dennis Wong. 2017. A simple shift rule for k-ary de Bruijn sequences. *Discrete Mathematics* 340, 3 (2017), 524–531.

[40] Noor Shaker, Julian Togelius, and Mark J Nelson. 2016. *Procedural content generation in games*. Springer.

[41] Wayne Stahnke. 1973. Primitive binary polynomials. *Mathematics of computation* 27, 124 (1973), 977–980.

[42] Brett Stevens and Aaron Williams. 2014. The coolest way to generate binary strings. *Theory of Computing Systems* 54, 4 (2014), 551–577.

[43] taotao54321. 2018. NES Dr.Mario hand simulator. https://gist.github.com/taotao54321/4ec019a251fdd8f9759fa8a8b5439559.

[44] Sam Trenholme. 2013. Alternate Pitfall maps. https://www.samiam.org/blog/20130617.html.

[45] Aaron Williams. 2019. Dr. Mario Puzzle Generation: Theory, Practice, & History (Famicom/NES). In *The 22nd Japan Conference on Discrete and Computational Geometry, Graphs, and Games*. 128–129.

[46] Aaron Williams. 2022. Nintendo's Not-So-Random Sequence. https://gitlab.com/combinatronics/nintendos-not-so-random-sequence.