Extending The Thue-Morse Sequence

Olivia Appleton-Crocker
TMW Center for Early Learning + Public Health
University of Chicago
Chicago, Illinois, United States
ORCID: 0009-0004-2296-7033

***Abstract*—In this paper, we discuss various ways to extend the Thue-Morse Sequence [1] when used as a fair-share sequence. Included are 15 definitions of the original sequence, 8 extensions to n players, for a total of 23 definitions. Also included are proofs of equality for all definitions, as well as an examination of several properties of the Thue-Morse Sequence and their presence in the Extended Thue-Morse Sequence. In the appendix are several complexity analyses for both time and space of each definition.**

***Index Terms*—Combinatorics, Generating Functions, Thue-Morse, Prouhet-Thue-Morse, Formal Languages, Number Theory, Periodicity, Aperiodicity, Rotation, Concatenation**

I. Introduction

The Thue-Morse Sequence is a fundamental object in combinatorics and theoretical computer science, widely studied for its remarkable properties and diverse applications. It is often presented as an infinite sequence starting with 0, as shown below:

$$T=\langle 0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,\ldots\rangle$$

This sequence arises in various fields, including automata theory, formal languages, number theory, and signal processing, due to its connection to binary operations, periodicity, and complexity. Its structure has been examined in relation to notions of minimality and non-repetitiveness, making it a natural object of study in mathematical logic and computational theory.

Over the past two centuries, the Thue-Morse Sequence has garnered attention for its role in constructing infinite words with specific properties (such as non-repetition over large substrings) and for its use in the design of error-correcting codes, pseudorandom number generators, and tiling problems. More recently, its applications have extended to the analysis of dynamical systems, coding theory, and the study of complexity within algorithms.

This survey primarily aims to explore the various definitions of the Thue-Morse Sequence found in the literature, with an emphasis on identifying distinct formulations and categorizing them. Additionally, this work seeks to extend these definitions from the binary alphabet $\{0,1\}$ to larger alphabets $\{0,1,\ldots,n-1\}$, often referred to as "integer bases." This work aims to explore how properties such as non-repetition, periodicity, and complexity are preserved under these extensions. By systematically analyzing these extensions and their associated properties, this overview hopes to present a deeper understanding of the Thue-Morse Sequence's generalizations and their potential applications in broader contexts.

This paper is divided into 8 sections. In Section 1, we introduce the concepts built upon in this paper. In Section 2 we present each of the 15 definitions of the standard Thue-Morse Sequence as seen in the literature today. In Section 3 we prove their equivalence with each other in the standard base-2 domain. In Section 4 we present 8 definitions that extend into larger domains. Most of these can only construct sequences with integer value elements, though a select few can be extended into even broader domains, such as rational bases. In Section 5 we prove that the extensions are equivalent to each other in the domain of positive integer bases. In Section 6 we examine the preservation (or failure) of properties for the standard Thue-Morse Sequence when extended to larger bases. Sections 7 and 8 are acknowledgments and the appendix respectively. The appendix contains complexity analyses (both time and space) for the different definitions. These show which may be the most computationally efficient ways to calculate the Thue-Morse Sequence or Extended Thue-Morse Sequence, given your available resources.

II. The Original Sequence

A. Definition 1 of 15 — Parity of Hamming Weight

This definition appears in [1–3].

The Hamming Weight, as typically defined, is the digit sum of a binary number. In other words, it is a count of the high bits in a given number. A common way to generate the Thue-Morse Sequence is to take the parity[1] of the Hamming Weight[2] for each natural number. We can define that as follows:

$$\begin{aligned} p(0) &= 0 \\ p(n) &= n + p\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \pmod 2 \\ T_{2,1}(n) &= p(n) \end{aligned} \tag{1}$$

The subscript indicates that we are using 2 players (writing in base 2) and that we are using the first definition laid out in this paper. Note that when we extend to n players, the T function will get a second parameter for the number of players, so it will look like $T_{n,d}(x,s)$, where s is the size of the player pool, and therefore the base we use to define the sequence.

[1]Whether a number is odd or even
[2]The count of 1s in the binary representation of a number

B. Definition 2 of 15 — Root of Unity

This appears in [4].

$$T_{2,2}(n) = \frac{1-(-1)^{p(n)}}{2} \tag{2}$$

C. Definition 3 of 15 — Invert and Extend

This definition appears in [1, 5].

This definition is more natural to think about as extending a tuple that contains the sequence. We will give a recurrence relation below, but to build an intuition we will work in this framework first.

Let $t(n)$ be the first 2^n elements of the Thue-Morse Sequence. Given this, we can define[3]:

$$\begin{aligned} \text{inv}(\mathbf{x}) &= \begin{cases} 0, & \text{if } x_i = 1 \\ 1, & \text{if } x_i = 0 \end{cases} \\ &\quad \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ t(0) &= \langle 0 \rangle \\ t(n) &= t(n-1) \parallel \text{inv}(t(n-1)) \end{aligned} \tag{3}$$

Given the above, we can define a recurrence relation that will give us individual elements. It will be less efficient to compute, but will allow proofs of equivalence to be easier.

Should the below have a +1 inside the log?

$$\begin{aligned} T_{2,3}(0) &= 0 \\ T_{2,3}(n) &= T_{2,3}\left(n - 2^{\lfloor \log_2(n) \rfloor}\right) + 1 \pmod 2 \end{aligned} \tag{4}$$

D. Definition 4 of 15 — Substitute and Flatten

This definition appears in [1, 2, 6].

$$\begin{aligned} s(n) &= \begin{cases} \langle 0, 1 \rangle, & \text{if } n = 0 \\ \langle 1, 0 \rangle, & \text{if } n = 1 \end{cases} \\ t(0) &= \langle 0 \rangle \\ t(n) &= \Big\|_{i=0}^{2^{n-1}-1} s(t(n-1)_i) \\ T_{2,4}(n) &= t(\lceil \log_2(n+1) \rceil)_n \end{aligned} \tag{5}$$

So for example, calculating $T_{2,3}(3)$ would look like:

$$\begin{aligned} t(0) &= \langle 0 \rangle \\ t(1) &= \Big\|_{i=0}^{0} s(t(0)_i) = \langle 0, 1 \rangle \\ t(2) &= \Big\|_{i=0}^{1} s(t(1)_i) = \langle 0, 1, 1, 0 \rangle \\ T_{2,4}(3) &= t(\lceil \log_2(3+1) \rceil)_3 \\ &= t(2)_3 \\ &= \langle 0, 1, 1, 0 \rangle_3 \\ &= 0 \end{aligned} \tag{6}$$

[3]Using $\parallel$ to mean concatenation, so $\langle 0 \rangle \parallel \langle 1 \rangle = \langle 0, 1 \rangle$

E. Definition 5 of 15 — Recursive Rotation

Another way to phrase the above definition is as recursive rotation. To the best of our knowledge, this definition is original to this paper. If we decompose s, we can instead represent it as:

$$\begin{aligned} r(\mathbf{x}, i) &= \langle x_{0+i \bmod |\mathbf{x}|}, x_{1+i \bmod |\mathbf{x}|}, \cdots \rangle \\ &\quad \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ t(0) &= \langle 0 \rangle \\ t(1) &= \langle 0, 1 \rangle \\ t(n) &= \Big\|_{i=0}^{1} r\left(t(n-1), i \cdot 2^{n-2}\right) \\ T_{2,5}(n) &= t(\lceil \log_2(n+1) \rceil)_n \end{aligned} \tag{7}$$

F. Definition 6 of 15 — Recursion

This definition appears in [1, 6].

$$\begin{aligned} T_{2,6}(0) &= 0 \\ T_{2,6}(2n) &= T_{2,6}(n) \\ T_{2,6}(2n+1) &= 1 - T_{2,6}(n) \end{aligned} \tag{8}$$

G. Definition 7 of 15 — Highest Bit Difference

This definition appears in [7].

The text below is from Wiki and needs to be entirely rewritten. I was able to derive the formula on my own from translating their code. This method leads to a fast method for computing the Thue-Morse Sequence: start with t0 = 0, and then, for each n, find the highest-order bit in the binary representation of n that is different from the same bit in the representation of n - 1. If this bit is at an even index, tn differs from tn - 1, and otherwise it is the same as tn - 1.

```
from itertools import count

def p2_d07():
    value = 1
    for n in count():
        # Assumes that (-1).bit_length() == 1
        x = (n ^ (n - 1)).bit_length() + 1
        if x & 1 == 0:
            # Bit index even, so toggle value
            value = 1 - value
        yield value
```

Should there be a +1 inside that call to log2?

$$\begin{aligned} T_{2,7}(0) &= 0 \\ T_{2,7}(n) &= \begin{matrix} \lfloor \log_2(n \oplus (n-1)) \rfloor \\ + T_{2,7}(n-1) \pm 1 \end{matrix} \pmod 2 \end{aligned} \tag{9}$$

H. Definition 8 — Floor-Ceiling Difference

This definition appears in [1].

$$\begin{aligned} b(n) &= \begin{cases} n & \text{if } n \le 1 \\ b\left(\left\lceil \frac{n}{2} \right\rceil\right) - b\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases} \\ T_{2,8}(n) &= \frac{1-b(2n)}{2} \pmod 2 \end{aligned} \tag{10}$$

This seems very similar to the highest bit difference definition, and I think it may be what that was derived from

I. Definition 9 — Odious Number Derivation

This definition appears in [1].

Another way to generate the Thue-Morse Sequence is to take the sequence of Odious Numbers [8] mod 2. Odious numbers are those with an odd number of 1s in their binary representation. Note that the player numbers in this derivation are swapped, so when generating this for testing and extension, we add 1 to the result. Some simple generating code [9] for this is as follows:

```
from itertools import count

def seq_p2_d09():
    for i in count():
        if i.bit_count() & 1:
            yield (i + 1) & 1
```

In mathematical terms, this can be translated to:

$$\begin{aligned} no(n) &= \begin{cases} n & \text{if } p(n) \bmod 2 = 1 \\ no(n+1) & \text{otherwise} \end{cases} \\ o(n) &= \begin{cases} 1 & \text{if } n = 0 \\ no(o(n-1)+1) & \text{if } n > 0 \end{cases} \\ T_{2,9}(n) &= o(n) + 1 \pmod 2 \end{aligned} \tag{11}$$

Aren't Odious Numbers exactly the numbers where the parity of the hamming weight is 1? So doesn't that mean that the Thue-Morse Sequence selects which numbers are Odious? From cursory testing, it seems to. There's something to be had there.

A possible way to extend this would be to reinterpret this as where the digit sum is not n-even

A related definition on OEIS [1] is

$$\begin{aligned} T(n) + \text{Odious}(n-1) + 1 &= 2n \text{ for } n \geq 1 \\ T(n) &= 2n - \text{Odious}(n-1) - 1 \end{aligned}$$

J. Definition 10 — Evil Numbers Derivation 1

This definition appears in [1].

The Evil Numbers [10] are those who have an even number of 1s in their binary representation. Note that this is the opposite of the Odious Numbers referenced above.

```
from itertools import count

def evil():
    for i in count():
        if i.bit_count() & 1 == 0:
            yield i

def p2_d10():
    for n, i in enumerate(evil()):
        yield (i - 2 * n) & 1
```

In mathematical terms, this can be translated to:

$$\begin{aligned} ne(n) &= \begin{cases} n & \text{if } p(n) \bmod 2 = 1 \\ ne(n+1) & \text{otherwise} \end{cases} \\ e(n) &= \begin{cases} 1 & \text{if } n = 0 \\ ne(e(n-1)+1) & \text{if } n > 0 \end{cases} \\ T_{2,10}(n) &= e(n) - 2n \pmod 2 \end{aligned} \tag{12}$$

K. Definition 11 — Evil Numbers Derivation 2

This definition ppears in [4].

A second, more-efficient derivation from the Evil Numbers is as follows, where $ce()$ is the count of Evil Numbers less than n [11], and $p()$ is the function defined in Equation 1.

$$\begin{aligned} ce(n) &= \left\lfloor \frac{n+1}{2} \right\rfloor + p(n+1) \cdot (n+1 \bmod 2) \\ T_{2,11}(n) &= 1 - ce(n+1) + ce(n) \end{aligned} \tag{13}$$

L. Definition 12 — Odious & Evil Numbers Derivation

This definition appears in [4].

A second, more-efficient derivation from the Evil Numbers is as follows, where $ce()$ is the count of Evil Numbers less than n [11], and $p()$ is the function defined in Equation 1.

$$\begin{aligned} oe(n) &= \begin{cases} o\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{if } n \bmod 2 = 0 \\ e\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{if } n \bmod 2 = 1 \end{cases} \\ T_{2,12}(n) &= 1 - oe(n) \pmod 2 \end{aligned} \tag{14}$$

M. Definition 13 — Gould's Sequence Derivation

This definition appears in [1].

Gould's Sequence [12] are the number of odd entries in a given row of Pascal's Triangle. Note that this is by far one of the least computationally efficient definition in this paper (see Tables XXIX & XXX).

Why mod 3? Everything else is mod 2. This definition is unlikely to be extendable, and the obvious routes fail

$$\begin{aligned} T_{2,13}(n) &= \text{Gould}(n) - 1 \bmod 3 \\ &= \left(\sum_{k=0}^{n} \left(\binom{n}{k} \bmod 2 \right) \right) - 1 \bmod 3 \end{aligned} \tag{15}$$

N. Definition 14 — Derivation from Blue Code

This definition appears in [1].

In the OEIS, this sequence [13] is defined as the "binary coding of a polynomial over GF(2), substitute x+1 for x". There are a number of ways to generate it. One of the more computationally-accessible ones is:

$$
\begin{aligned}
\text{A001317}(n) &= \sum_{k=0}^{n} \left(\binom{n}{k} \bmod 2 \right) \cdot 2^k \\
f(n,i) &= \left\lfloor \frac{n}{2^i} \bmod 2 \right\rfloor_{\lceil \log_2(n) \rceil} \\
\text{A193231}(n) &= \bigoplus_{i=0} (\text{A001317}(i \cdot f(n,i)) \cdot f(n,i)) \\
T_{2,14}(n) &= \text{A193231}(n) \pmod 2
\end{aligned}
\tag{16}
$$

Translated into words, this function computes the value of Sierpiński's triangle for the index of each high bit, then takes the bitwise exclusive or[4] of all such resulting values. Note that since $f(n,i) = 0$ if and only if the ith bit of n is low, each low bit can be simplified out when calculating.

It seems to me that this might be extendable by using GF(n) instead of GF(2), though I don't know of a way to efficiently compute or prove such a result

O. Definition 15 — Generating Function

This definition appears in [1, 3].

$$
T_{2,15} = \mathcal{G}.\mathcal{F}. \lim_{k_{\max} \to \infty} \frac{\frac{1}{1-x} - \prod_{k \geq 0}^{k_{\max}} (1 - x^{2^k})}{2} \tag{17}
$$

The logic of this construction is fairly clear. The component that is $\frac{1}{1-x}$ will generate coefficients of all 1s. This is then subtracted by a sequence that counts the high bits of a given index, evaluating to -1 if the number of bits is even, and 1 if it is odd. This means that the only possible outputs of the top part are 0 or 2, so the divisor corrects for that.

This means that it is equivalent to $T_{2,1}$ (prove it).

In practice, to get the coefficients up to the nth term, one needs to only expand the product to $k_{\max} = \lfloor \log_2(n+1) \rfloor - 1$.

Note that this is by far the least computationally efficient definition in this paper (see Figure 1 and Tables XXXIII, XXXIV).[5]

P. Summary

Of the 15 we discussed above:

- 13 were found on the OEIS [1, 4, 15–17] (defs. 1-4, 6, 8-15)
- 1 was found in a textbook [7] (def. 7)
- 1 is original to this paper (def. 5)

- 7 utilize recursion (defs. 1-7)
- 6 reference other integer sequences (defs. 9-14)
- 5 utilize floor-division (defs. 1-2, 8, 11-12, 14)
- 4 would be so good to include, but I don't know how
- 3 use operations on strings, not integers (defs. 3-5)
- 2 use combinatoric functions (defs. 13-14)
- 1 utilizes a generating function (def. 15)
- 0 have closed form solutions

[4]Represented in this paper as $\oplus$

[5]Our implementation [9] heavily utilizes [14] for these calculations

III. Proving Equivalence Between Standard Definitions

All definitions are tested computationally, though due to resource limitations, to different extents.

TABLE I: Extent of Testing in Base 2

Definition	Entries Tested
$T_{2,1}$	$2^{34} = 17,179,869,184$
$T_{2,2}$	
$T_{2,3}$	
$T_{2,4}$	$2^{33} = 8,589,934,592$
$T_{2,5}$	
$T_{2,6}$	$2^{32} = 4,294,967,296$
$T_{2,7}$	$2^{34} = 17,179,869,184$
$T_{2,8}$	$2^{26} = 67,108,864$
$T_{2,9}$	$2^{34} = 17,179,869,184$
$T_{2,10}$	
$T_{2,11}$	
$T_{2,12}$	
$T_{2,13}$	$2^{15} = 32,768$
$T_{2,14}$	$2^{29} = 536,870,912$
$T_{2,15}$	$2^{11} = 2,048$
$T_{n,1}$	$2^{32} = 4,294,967,296$
$T_{n,2}$	
$T_{n,3}$	$2^{33} = 8,589,934,592$
$T_{n,4}$	$2^{32} = 4,294,967,296$
$T_{n,5}$	
$T_{n,6}$	
$T_{n,7}$	$2^{31} = 2,147,483,648$
$T_{n,8}$	$2^{32} = 4,294,967,296$

TABLE II: Comparison Matrix of the Standard Definitions

X = done, S = started, O = target

1													
X	2												
X		3											
		O	4										
			S	5									
X					6								
						7							
						O	8						
O								9					
O								O	10				
O							O		O	11			
O											12		
												13	
												O	14
O													O	15

A. Correlating Definition 1 and Definition 2

Proof.

Observation 1: For all integers n, $(-1)^n \in \{1, -1\}$
Observation 2: For $n = \{1, -1\}$, $\frac{1-n}{2} = \{0, 1\}$
Observation 3: For all $n = 2k$, $(-1)^n = 1$
Observation 4: For all $n = 2k+1$, $(-1)^n = -1$
Inference 1: $\frac{1-(-1)^n}{2} = n \mod 2$
Conclusion:

$$T_{2,2}(n) = \frac{1-(-1)^{p(n)}}{2} = p(n) \bmod 2 = T_{2,1}(n) \tag{18}$$

□

B. Correlating Definition 1 and Definition 3

Proof.

Observation 1: $0 \leq n < 2^k \implies t(k+1)_{2^k+n} \equiv t(k)_n + 1$ This is because at each step in the process, you are inverting and extending. Inversion is equivalent to $x + 1 \bmod 2$, and each entry keeps a relative position by adding a power of 2.

Inference 1:

$$\begin{aligned} T_{2,3}(0) &= 0 \\ T_{2,3}(n) &= T_{2,3}(n - 2^{\lfloor \log_2(n) \rfloor}) + 1 \bmod 2 \end{aligned} \tag{19}$$

Observation 2: By subtracting a power of 2, you reduce the Hamming Weight by 1

Inference 2: Since you are adding 1 for each time you reduce the Hamming Weight by 1, this means $T_{2,3}$ is recursively computing the parity of the Hamming Weight.

Observation 3: $T_{2,1} = p(n)$, and $p(n)$ computes the parity of the Hamming Weight of n.

Conclusion: $T_{2,1}(n) = T_{2,3}(n)$ □

C. Correlating Definition 1 and Definition 6

Note that in Equation 8 where we define $T_{2,6}$, we are working in mod 2, where $+1$ and -1 are logically equivalent. We can therefore simplify its definition to be:

$$\begin{aligned} T_{2,6}(0) &= 0 \\ T_{2,6}(n) &= n + T_{2,6}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \pmod 2 \end{aligned} \tag{20}$$

This is identical to Equation 1, where we define $T_{2,1}$.

D. Correlating Definition 4 and Definition 5

Proof.

Observation 1: $r(\mathbf{x}, i) = r(\mathbf{x}, i + k \cdot |\mathbf{x}|)$
Inference 1: $r(\mathbf{x}, i) = r(\mathbf{x}, i \bmod |\mathbf{x}|)$
Observation 2: $s(0) = \langle 0, 1 \rangle = r(s(0), 0)$
Observation 3: $s(1) = \langle 1, 0 \rangle = r(s(0), 1)$
Inference 2: $s(n) = r(s(0), n)$

This is where I run into trouble. I know I can keep going from here, but I'm not quite sure how.

Conclusion: $T_{2,4}(n) = T_{2,5}(n)$ □

IV. The Extensions

A. Extension 1 of 8 — Modular Digit Sums

This definition appears in [16, 18–25].

To extend definition 1 from 2 to n players, we must first map our concept of parity to base n. We can do this by taking the parity equation defined above and replacing the 2s with n, for $n \in \mathbb{Z}_{\geq 2}$.

$$\begin{aligned} p_n(0) &= 0 \\ p_n(x) &= x + p_n\left(\left\lfloor \frac{x}{n} \right\rfloor\right) \pmod n \end{aligned} \tag{21}$$

Under this definition, you can construct the Thue-Morse Sequence using the following, starting at 0:

$$T_{n,1}(x, s) = p_s(x) \tag{22}$$

Note that this definition is trivially extensible to non-integer bases by redefining $p_n()$, though that is beyond the scope of this paper. This has been done in [16, 19]. It has also been extended to negative integer bases [20–22].

Some other works present a more generalized version, where

$$t_{b,m}(n) = p_b(n) \bmod m \tag{23}$$

This allows for increased flexibility, especially when using fractional bases. In this notation, for negative integer bases, $T_{n,1}(x, s) = p_s(x) \bmod |s|$

1) Proof of Equivalence with Original Definition 1:

Proof. It is clear from visual inspection that p_2 is identical to our original definition of p.

$$\begin{aligned} p_2(x) &= p(x) \\ x + p_2\left(\left\lfloor \frac{x}{2} \right\rfloor\right) &= x + p\left(\left\lfloor \frac{x}{2} \right\rfloor\right) \\ x + \left\lfloor \frac{x}{2} \right\rfloor + p_2\left(\left\lfloor \frac{x}{2^2} \right\rfloor\right) &= x + \left\lfloor \frac{x}{2} \right\rfloor + p\left(\left\lfloor \frac{x}{2^2} \right\rfloor\right) \\ x + \left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{x}{2^2} \right\rfloor + \ldots &= x + \left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{x}{2^2} \right\rfloor + \ldots \end{aligned} \tag{24}$$

□

This definition is also trivially extended to negative integer bases.

B. Extended Definition 2 of 8 — Roots of Unity

$$T_{n,2}(x, s) = \frac{\log\left(\omega_s^{p_s(x)}\right)}{\log\left(\omega_s\right)} \tag{25}$$

1) Proof of Equivalence with Original Definition 2:

Proof. Let's start by substituting s for 2:

$$\begin{aligned} T_{n,2}(x,2) &= \frac{\log\left(\omega_2^{p_2(x)}\right)}{\log(\omega_2)} \\ &= \frac{\log\left((-1)^{p_2(x)}\right)}{\log(-1)} \\ &= \frac{(p_2(x) \bmod 2)\cdot\log(-1)}{\log(-1)} \\ &= \frac{(p_2(x) \bmod 2)\cdot i\pi}{i\pi} \\ &= p_2(x) \bmod 2 \end{aligned} \tag{26}$$

This is identical to $T_{2,1}$, which we earlier proved is equivalent to $T_{2,2}$. □

C. Extended Definition 3 of 8 — Increment and Extend

[26] gives a good example on how to possibly adapt inversion to incrementing

In the original version of this definition, we inverted the elements. In base 2, this is the same thing as adding 1 (mod 2). Given that, let $t(x,n)$ be the first n^x elements of the Extended Thue-Morse Sequence, for $n \in \mathbb{Z}_{\geq 2}$.

$$\operatorname{inc}(\mathbf{x},n) = \begin{array}{l} x_i + 1 \pmod n \\ \text{for } \mathbf{x} = (x_0, x_1, \ldots, x_{n-1}) \end{array} \tag{27}$$

$$\begin{aligned} t(0,n) &= \langle 0 \rangle \\ t(1,n) &= \langle 0, 1, \ldots, n-1 \rangle \\ t(x,n) &= t(x-1,n) \cdot \operatorname{inc}(t(x-1,n), n) \end{aligned} \tag{28}$$

Given the above, we can define a recurrence relation that will give us individual elements. It will be less efficient to compute, but will allow proofs of equivalence to be easier.

$$\begin{aligned} T_{n,3}(0,s) &= 0 \\ T_{n,3}(x,s) &= T_{n,3}\left(x - s^{\lfloor \log_s(x) \rfloor}, s\right) + 1 \pmod s \end{aligned} \tag{29}$$

1) Proof of Equivalence with Original Definition 3: ...

D. Extended Definition 4 of 8 — Substitute and Flatten

This definition appears in [27]

There's a bit of a leap here, since we have to explain why the rotation is equivalent to the binary choice presented in the original. There also might be a better syntax to define the rotation, perhaps using the format used in inv and inc.

$$\begin{aligned} b(s) &= \langle 0, 1, \cdots, s-2, s-1 \rangle \\ r(\mathbf{x}, i) &= \begin{array}{l} \langle x_{0+i \bmod |\mathbf{x}|}, x_{1+i \bmod |\mathbf{x}|}, \ldots \rangle \\ \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \end{array} \\ s(x,s) &= r(b(s), x) \\ t(0) &= \langle 0 \rangle \\ t(x,s) &= \Big\|_{i=0}^{2^{x-1}-1} s(t(x-1)_i, s) \\ T_{n,4}(x,s) &= t(\lceil \log_s(x+1) \rceil, s)_x \end{aligned} \tag{30}$$

1) Proof of Equivalence with Original Definition 4: ...

E. Extended Definition 5 of 8 — Recursive Rotation

$$\begin{aligned} r(\mathbf{x}, i) &= \begin{array}{l} \langle x_{0+i \bmod |\mathbf{x}|}, x_{1+i \bmod |\mathbf{x}|}, \ldots \rangle \\ \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \end{array} \\ t(0,s) &= \langle 0 \rangle \\ t(1,s) &= \langle 0, 1, \ldots, s-1 \rangle \\ t(x,s) &= \Big\|_{i=0}^{s-1} r\left(t(x-1,s), i \cdot s^{x-2}\right) \\ T_{n,5}(x,s) &= t(\lceil \log_s(x+1) \rceil, s)_x \end{aligned} \tag{31}$$

1) Proof of Equivalence with Original Definition 5: ...

F. Extended Definition 6 of 8 — Recursion

$$\begin{aligned} T_{n,6}(0,s) &= 0 \\ T_{n,6}(s \cdot x, s) &= T_{n,6}(x,s) \\ T_{n,6}(s \cdot x + k, s) &= k + T_{n,6}(x,s) \pmod s \end{aligned} \tag{32}$$

1) Proof of Equivalence with Original Definition 6:

Proof. Let's begin by substituting s for 2:

$$\begin{aligned} T_{n,6}(0,2) &= 0 \\ T_{n,6}(2 \cdot x, 2) &= T_{n,6}(x,2) \\ T_{n,6}(2 \cdot x + k, 2) &= k + T_{n,6}(x,2) \pmod 2 \end{aligned} \tag{33}$$

Note that that only values for k that fit in this definition are 0 and 1. This means we can further simplify to:

$$T_{n,6}(2 \cdot x + 1, 2) = 1 + T_{n,6}(x,2) \pmod 2 \tag{34}$$

This is very similar to the definition found in equation 8, except that one is adding and the other subtracting. Fortunately, we know that the only values that $T_{2,6}$ will return are 0 and 1, which means that these operations will be completely equivalent.

$$\begin{aligned} 1 - 0 \bmod 2 &= 1 + 0 \bmod 2 \\ 1 - 1 \bmod 2 &= 1 + 1 \bmod 2 \end{aligned}$$

□

G. Extended Definition 7 of 8 — Highest Digit Difference

$$\begin{aligned} \operatorname{XOR}_n(a,b) &= \sum_{i=0}^{\lceil \log_n(\max(a,b)+1) \rceil} n^i \left(\left\lfloor \frac{a}{n^i} \right\rfloor - \left\lfloor \frac{b}{n^i} \right\rfloor \bmod n \right) \\ T_{n,7}(0,s) &= 0 \\ T_{n,7}(x,s) &= \begin{array}{l} \lfloor \log_s(\operatorname{XOR}_s(x, x-1)) \rfloor \\ + T_{n,7}(x-1,s) + 1 \end{array} \pmod s \end{aligned} \tag{35}$$

Substitute n for 2, then simplify, plus a bit

1) Proof of Equivalence with Original Definition 8:

H. Extended Definition 8 of 8 — Latin Square Constructions

This definition appears in [5], and seemingly only there. It is very clearly equivalent to our Extended Definition 3, but I will need to actually prove that.

Let $L(n)$ be the reduced-form Latin Square with a first row of $\langle 0, 1, \ldots, n-1 \rangle$, and where each row progresses from one entry to the next as $L(n)_{a,x+1} \equiv L(n)_{a,x} + 1 \pmod{n}$. For each iteration t_n, substitute each entry x for the string $L(n)_{x,*}$.

$$L(N) = \begin{pmatrix} 0 & 1 & 2 & \ldots & N{-}1 \\ 1 & 2 & \ddots & N{-}1 & 0 \\ 2 & \ddots & N{-}1 & 0 & 1 \\ \vdots & N{-}1 & 0 & 1 & \ddots \\ N{-}1 & 0 & 1 & \ddots & \ddots \end{pmatrix} \tag{36}$$

I. Summary

Of the 8 we discussed above:

- 1 was found on the OEIS ($T_{n,1}$)
- 2 were found in another paper ($T_{n,4}, T_{n,8}$)
- 5 are original to this paper ($T_{n,2\ldots3}, T_{n,5\ldots7}$)

- 7 utilize recursion ($T_{n,1\ldots7}$)
- 4 utilize floor-division ($T_{n,1\ldots2}, T_{n,6\ldots7}$)
- 4 use operations on strings, not integers ($T_{2,3\ldots5}, T_{n,8}$)
- 0 have closed form solutions

V. Proving Equivalence Between Extended Definitions

All definitions are tested computationally, though due to resource limitations, to different extents.

TABLE III: Extent of Testing in Integer Bases > 2

Definition	Bases	Entries Tested
$T_{n,1}$	3 - 256	$2^{30} = 1,073,741,824$
$T_{n,2}$	3 - 16	$2^{29} = 536,870,912$
	17 - 256	$2^{30} = 1,073,741,824$
$T_{n,3}$	3 - 16	
	33 - 208	
	17 - 32	$2^{29} = 536,870,912$
	209 - 256	
$T_{n,4}$	3 - 16	$2^{30} = 1,073,741,824$
	33 - 256	
	17 - 32	$2^{29} = 536,870,912$
$T_{n,5}$	3 - 192	
	193 - 256	$2^{28} = 268,435,456$
$T_{n,6}$	3 - 256	$2^{30} = 1,073,741,824$
$T_{n,7}$		
$T_{n,8}$		

TABLE IV: Comparison Matrix of the Extended Definitions

X = done, O = target

1							
X	2						
O		3					
		O	4				
			O	5			
O					6		
O						7	
		X					8

A. Correlating Definition 1 and Definition 2

Proof.

Observation 1: For all integers n, $\omega_s^n = \omega_s^{n \bmod s}$

Inference 1: $\dfrac{\log(\omega_s^n)}{\log(\omega_s)} = n \bmod s$

Conclusion:

$$\begin{aligned} T_{n,2}(x,s) &= \frac{\log(\omega_s^{p_s(x)})}{\log(\omega_s)} \\ &= \frac{(p_s(x) \mod s) \cdot \log(\omega_s)}{\log(\omega_s)} \\ &= \frac{(p_s(x) \mod s) \cdot 2i\pi s^{-1}}{2i\pi s^{-1}} \\ &= p_s(x) \mod s \\ &= T_{n,1}(x,s) \end{aligned} \tag{37}$$

□

B. Correlating Definition 4 and Definition 8

Proof.

Observation 1: For any given row of $L(N)$, it will start with the index of the row

Observation 2: For any given row of $L(N)$, it will end with 1 less than the index of the row $\pmod N$

Inference 1: $L(N)_{x,*} = r(b(N), x)$

Observation 3: $T_{n,4}$ is defined as substituting $r(b(N), x)$ for each element x in the previous iteration

Conclusion: $T_{n,4} = T_{n,8}$ □

C. Summary

VI. Proving Persistence (Or Lack Thereof) of Original Properties

A. Use as a Fair-Share Sequence

Goal: show that for a variety of value functions, greedy algorithms given this turn order will always minimize inequality. They should at least do so more than the standard turn order. This is going to look like setting up an equation to show

$$eq(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases}$$

$$f(v,p,s) = \lim_{n\to\infty} \sum_{i=0}^{n} v(i) \cdot eq(T_n(i,s),p) \tag{38}$$

$$\forall p : p \in \{1,\ldots,s-1\}$$

$$f(v,0,s) + \epsilon < f(v,p,s) < f(v,0,s) - \epsilon$$

This is for $v(i)$ being a value function and ϵ being an arbitrarily small number. $f()$ is therefore the sum of total value that they will be receiving. For example, one could model a board game as $v(i) = \frac{1}{2^i}$, where $v()$ models the amount each turn contributes to your probability of victory. Note that this may be very hard to show for versions of $v()$ which shrink too quickly, such as $v(i) = \frac{1}{i!}$, so for those cases we must show that it's better than the standard turn order

1) On the value function of 1: It is well known [26] for the Standard Thue-Morse Sequence that

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{T_2(i)}{n+1} = \frac{1}{2} \tag{39}$$

Does this generalize to: ?

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{T_n(i,s)}{n+1} = \sum_{i=0}^{s-1} \frac{i}{s} = \frac{n-1}{2} \tag{40}$$

and

$$\forall x \mid 0 \leq x < s \text{ and } x \in \mathbf{Z}$$

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{eq\left(T_n(i,s),x\right)}{n+1} = \frac{1}{s} \tag{41}$$

2) On more general value functions:

B. Aperiodicity

Not totally sure how this should go, but I think a good approximation would be to show that the distance between the nearest two appearances of a substring grows to infinity much faster than the length of those substrings grows

C. Palindrome

This seems trivially violated for $n > 2$. Example: $t_3(2) = \langle 0,1,2,1,2,0,2,0,1 \rangle$

This doesn't even work for power-of-2 bases, like $t_4(2) = \langle 0,1,2,3,1,2,3,0,2,3,0,1,3,0,1,2 \rangle$

I am unable to think of any base other than 2 where this property could even conceivably hold true

D. Uniform Recurrence

The Thue-Morse Sequence is a uniformly recurrent word: given any finite string X in the sequence, there is some length nX (often much longer than the length of X) such that X appears in every block of length nX

VII. Acknowledgment

We thank Dan Rowe for helping with the initial work on this paper, as well as Randy Appleton and Lydia Crocker who helped greatly with proofreading and editing.

VIII. Appendix

A. General Performance Results

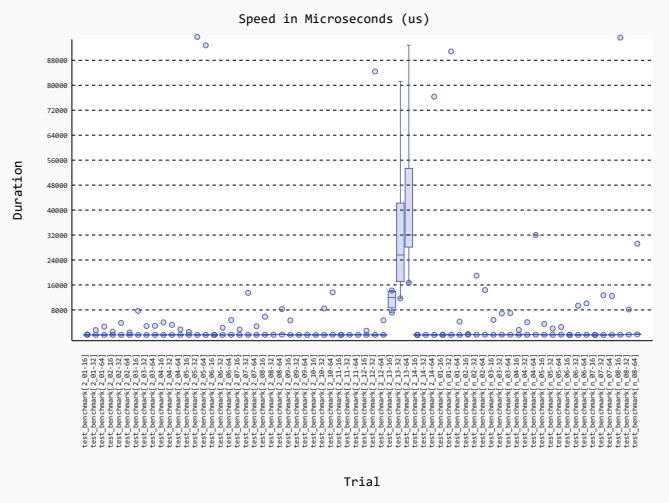

Fig. 1: Benchmark results up to seconds.

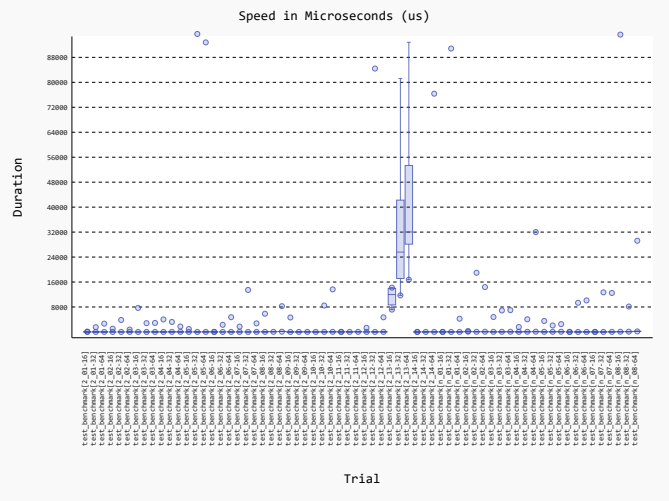

Fig. 2: Benchmark results up to milliseconds.

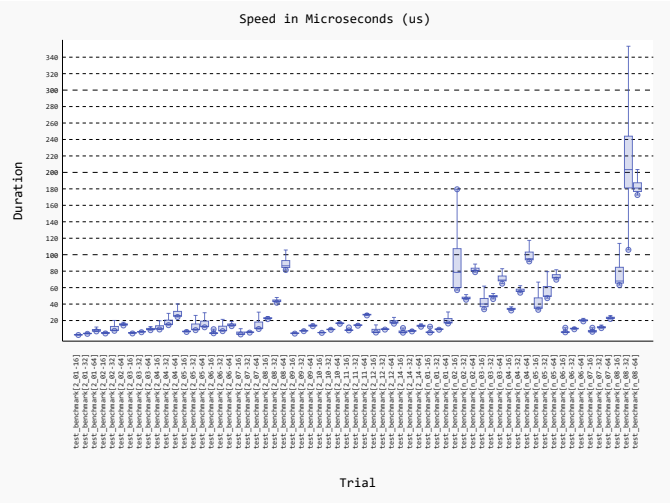

Fig. 3: Benchmark results up to microseconds.

B. Complexity of Original Definition 1

1) Time Complexity: In an idealized case, this definition will simplify to:

$$T_{2,1}(n) = \left(\sum_{i=0}^{\lceil \log_2(n+1) \rceil} \left\lfloor \frac{n}{2^i} \bmod 2 \right\rfloor \right) \bmod 2 \qquad (42)$$

This is pretty explicitly $O(\log(n))$ operations. This means that generating the first n entries will take $O(n \log(n))$ operations.

In languages with dynamically sized integers, this can be slightly more complicated. In the above, we perform $\log(n)$ bit shifts, multiplications, moduli, and additions. Since a bit shift is constant time, calculation will be dominated by multiplication, division, and moduli. Each of these take $O(\log(n) \cdot \log(\log(n)))$, where n is the largest number involved. This means that in such languages, we can expect it to take $O(\log(n)^2 \cdot \log(\log(n)))$ operations per element, for $O(n \cdot \log(n)^2 \cdot \log(\log(n)))$ in total.

TABLE V: Time Complexity Summary of Standard Definition 1

	Fixed Size	**Arbitrary Size**
Per Element	$O(\log(n))$	$O(\log(n)^2 \cdot \log(\log(n)))$
In Total	$O(n \cdot \log(n))$	$O(n \cdot \log(n)^2 \cdot \log(\log(n)))$

2) Space Complexity: This is one of the more space-efficient implementations. Each element takes at most the same size as the passed integer. In languages that use Fixed Size integers, that means it will take $O(1)$ space. In languages like Python that use Arbitrary Size integers, it would take $O(\log(n))$ space, where n is the largest element you intend to calculate. If you intend to store all n elements, it will therefore take $O(n)$ or $O(n \cdot \log(n))$ space.

TABLE VI: Space Complexity Summary of Standard Definition 1

	Fixed Size	**Arbitrary Size**
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(n)$	$O(n \cdot \log(n))$

C. Complexity of Original Definition 2

TABLE VII: Time Complexity Summary of Standard Definition 2

	Fixed Size	**Arbitrary Size**
Per Element		
In Total		

1) Time Complexity:

TABLE VIII: Space Complexity Summary of Standard Definition 2

	Fixed Size	**Arbitrary Size**
Per Element		
In Total		

2) Space Complexity:

D. Complexity of Original Definition 3

TABLE IX: Time Complexity Summary of Standard Definition 3

	Fixed Size	**Arbitrary Size**
Per Element		
In Total		

1) Time Complexity:

TABLE X: Space Complexity Summary of Standard Definition 3

	Fixed Size	**Arbitrary Size**
Per Element		
In Total		

2) Space Complexity:

E. Complexity of Original Definition 4

TABLE XI: Time Complexity Summary of Standard Definition 4

	Fixed Size	**Arbitrary Size**
Per Element		
In Total		

1) Time Complexity:

TABLE XII: Space Complexity Summary of Standard Definition 4

	Fixed Size	**Arbitrary Size**
Per Element		
In Total		

2) Space Complexity:

F. Complexity of Original Definition 5

TABLE XIII: Time Complexity Summary of Standard Definition 5

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XIV: Space Complexity Summary of Standard Definition 5

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

G. Complexity of Original Definition 6

1) Time Complexity: At each step in calculation, the value of n passed to the next recursion is halved. This means that it will take $O(\log_2(n))$ recursive steps. Each recursion involves at maximum 2 subtractions and a bit shift. In most languages with Fixed Size integers, this will take constant time. However, in languages with Arbitrary Size integers these subtractions will typically take $O(\log(n))$, where n is the largest integer in the operation. This means we can expect it to take $O(\log(n)^2$ operations.

TABLE XV: Time Complexity Summary of Standard Definition 6

	Fixed Size	Arbitrary Size
Per Element	$O(\log(n))$	$O(\log(n)^2)$
In Total	$O(n \cdot \log(n))$	$O(n \cdot \log(n)^2)$

2) Space Complexity: This is one of the more space-efficient implementations. Each element takes at most the same size as the passed integer. In languages that use Fixed Size integers, that means it will take $O(1)$ space. In languages like Python that use Arbitrary Size integers, it would take $O(\log(n))$ space, where n is the largest element you intend to calculate. If you intend to store all n elements, it will therefore take $O(n)$ or $O(n \cdot \log(n))$ space.

TABLE XVI: Space Complexity Summary of Standard Definition 6

	Fixed Size	Arbitrary Size
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(n)$	$O(n \cdot \log(n))$

H. Complexity of Original Definition 7

1) Time Complexity: Since this algorithm works sequentially, and cannot perform computation of an arbitrary element without recursing to the base case, the time is equal on a per-element and in-total basis

TABLE XVII: Time Complexity Summary of Standard Definition 7

	Fixed Size	Arbitrary Size
Per Element	$O(n \cdot \log(n))$	$O\left(n \cdot \log(n)^2\right)$
In Total	$O(n \cdot \log(n))$	$O\left(n \cdot \log(n)^2\right)$

TABLE XVIII: Space Complexity Summary of Standard Definition 7

	Fixed Size	Arbitrary Size
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(n)$	$O(n \cdot \log(n))$

2) Space Complexity:

I. Complexity of Original Definition 8

1) Time Complexity: memoization doesn't seem to help in the worst case of $1\ldots1_2$, so you should still end up calculating the value of $b()$ for every positive number less than n

TABLE XIX: Time Complexity Summary of Standard Definition 8

	Fixed Size	Arbitrary Size
Per Element	$O(n)$	$O(n \cdot \log(n) \cdot \log(\log(n)))$
In Total	$O\left(n^2\right)$	$O\left(n^2 \cdot \log(n) \cdot \log(\log(n))\right)$

2) Space Complexity: There are two ways to implement this algorithm in terms of space complexity. They both have equal worst-case time complexity. The first is to take the recursive approach, and the second is to use dynamic programming.

In a recursive approach, you will end up descending $O(\log(n))$ stack frames, each of which will contain at minimum 1 integer. In the dynamic approach, you will keep a table of all the values of $b()$ from 0 through n. The biggest difference between these approaches is that in the recursive approach you may need to repeat calculations.

TABLE XX: Space Complexity Summary of Standard Definition 8

		Fixed Size	Arbitrary Size
Recursive	**Per Element**	$O(\log(n))$	$O(n \cdot \log(n))$
	In Total	$O(n \cdot \log(n))$	$O\left(n^2 \cdot \log(n)\right)$
Dynamic	**Per Element**	$O(n)$	$O(n \cdot \log(n))$
	In Total	$O\left(n^2\right)$	$O\left(n^2 \cdot \log(n)\right)$

J. Complexity of Original Definition 9

TABLE XXI: Time Complexity Summary of Standard Definition 9

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXII: Space Complexity Summary of Standard Definition 9

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

K. Complexity of Original Definition 10

TABLE XXIII: Time Complexity Summary of Standard Definition 10

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXIV: Space Complexity Summary of Standard Definition 10

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

L. Complexity of Original Definition 11

TABLE XXV: Time Complexity Summary of Standard Definition 11

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXVI: Space Complexity Summary of Standard Definition 11

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

M. Complexity of Original Definition 12

TABLE XXVII: Time Complexity Summary of Standard Definition 12

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXVIII: Space Complexity Summary of Standard Definition 12

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

N. Complexity of Original Definition 13

1) Time Complexity: There are two ways one could reasonably calculate this. The first is by building each row of Pascal's Triangle iteratively. This allows you to avoid multiplication whenever possible, and lets you apply a bitmask or modulus operation to take the parity of each entry. The downside is that this version is not parallelizable. Using the bit mask approach, this means that each entry will take $O(n)$ time.

The other is to take advantage of the relation $\binom{n}{k} = \binom{n}{k-1} \cdot \frac{n-(k-1)}{k}$. This allows you to calculate each row independently, using $\frac{n}{2}$ moduli, multiplications, and divisions. This means that each entry will take $O(n)$ operations, each of which take $O(\log(n) \cdot \log(\log(n)))$ if with arbitrary sized integers, totaling $O(n)$ or $O(n \cdot \log(n) \cdot \log(\log(n)))$.

TABLE XXIX: Time Complexity Summary of Standard Definition 13

		Fixed Size	Arbitrary Size
Serial	**Per Element**	$O(n)$	$O(n)$
	In Total	$O\left(n^2\right)$	$O\left(n^2\right)$
Parallel	**Per Element**	$O(n)$	$O(n \cdot \log(n) \cdot \log(\log(n)))$
	In Total	$O\left(n^2\right)$	$O\left(n^2 \cdot \log(n) \cdot \log(\log(n))\right)$

TABLE XXX: Space Complexity Summary of Standard Definition 13

		Fixed Size	Arbitrary Size
Serial	**Per Element**	$O(n)$	$O\left(n \cdot \log(n)\right)$
	In Total	$O\left(n^2\right)$	$O\left(n^2 \cdot \log(n)\right)$
Parallel	**Per Element**	$O(1)$	$O\left(\log(n)\right)$
	In Total	$O(n)$	$O\left(n \cdot \log(n)^2\right)$

2) Space Complexity:

O. Complexity of Original Definition 14

TABLE XXXI: Time Complexity Summary of Standard Definition 14

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXXII: Space Complexity Summary of Standard Definition 14

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

P. Complexity of Original Definition 15

TABLE XXXIII: Time Complexity Summary of Standard Definition 15

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXXIV: Space Complexity Summary of Standard Definition 15

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

Q. Complexity of Extension Definition 1

In an idealized case, this definition will simplify to:

$$T_{n,1}(x,s) = \left(\sum_{i=0}^{\lceil \log_s(x+1) \rceil} \left\lfloor \frac{x}{s^i} \bmod s \right\rfloor \right) \bmod s \tag{43}$$

This is pretty explicitly $O(\log(n))$ operations. This means that generating the first n entries will take $O(n \log(n))$ operations.

In languages with dynamically sized integers, this can be slightly more complicated. In the above, we perform $\log(n)$ multiplications, moduli, and additions. Since additions are simpler, calculation will be dominated by multiplication, division, and moduli. Each of these take $O(\log(n) \cdot \log(\log(n)))$, where n is the largest number involved. This means that in such languages, we can expect it to take $O(\log(n)^2 \cdot \log(\log(n)))$ operations per element, for $O(n \cdot \log(n)^2 \cdot \log(\log(n)))$ in total.

TABLE XXXV: Time Complexity Summary of Extended Definition 1

	Fixed Size	Arbitrary Size
Per Element	$O(\log(n))$	$O(\log(n)^2 \cdot \log(\log(n)))$
In Total	$O(n \cdot \log(n))$	$O(n \cdot \log(n)^2 \cdot \log(\log(n)))$

1) Space Complexity: This is one of the more space-efficient implementation. Each element takes at most the same size as the passed integer. In languages that use Fixed Size integers, that means it will take $O(1)$ space. In languages like Python that use Arbitrary Size integers, it would take $O(\log(n))$ space, where n is the largest element you intend to calculate. If you intend to store all n elements, it will therefore take $O(n)$ or $O(n \cdot \log(n))$ space.

TABLE XXXVI: Space Complexity Summary of Extended Definition 1

	Fixed Size	Arbitrary Size
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(n)$	$O(n \cdot \log(n))$

R. Complexity of Extension Definition 2

TABLE XXXVII: Time Complexity Summary of Extended Definition 2

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXXVIII: Space Complexity Summary of Extended Definition 2

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

S. Complexity of Extension Definition 3

TABLE XXXIX: Time Complexity Summary of Extended Definition 3

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XL: Space Complexity Summary of Extended Definition 3

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

T. Complexity of Extension Definition 4

TABLE XLI: Time Complexity Summary of Extended Definition 4

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XLII: Space Complexity Summary of Extended Definition 4

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

U. Complexity of Extension Definition 5

TABLE XLIII: Time Complexity Summary of Extended Definition 5

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XLIV: Space Complexity Summary of Extended Definition 5

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

V. Complexity of Extension Definition 6

1) Time Complexity: At each step in calculation, the value of n passed to the next recursion is divided by s (the selected base). This means that it will take $O(\log_s(n))$ recursive steps. Each recursion involves at maximum 2 subtractions and a bit shift. In most languages with Fixed Size integers, this will take constant time. However, in languages with Arbitrary Size integers these subtractions will typically take $O(\log(n))$, where n is the largest integer in the operation. This means we can expect it to take $O(\log(n)^2$ operations.

TABLE XLV: Time Complexity Summary of Extended Definition 6

	Fixed Size	Arbitrary Size
Per Element	$O(\log(n))$	$O(\log(n)^2)$
In Total	$O(n \cdot \log(n))$	$O(n \cdot \log(n)^2)$

2) Space Complexity: This is one of the more space-efficient implementations. Each element takes at most the same size as the passed integer. In languages that use Fixed Size integers, that means it will take $O(1)$ space. In languages like Python that use Arbitrary Size integers, it would take $O(\log(n))$ space, where n is the largest element you intend to calculate. If you intend to store all n elements, it will therefore take $O(n)$ or $O(n \cdot \log(n))$ space.

TABLE XLVI: Space Complexity Summary of Extended Definition 6

	Fixed Size	Arbitrary Size
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(n)$	$O(n \cdot \log(n))$

W. Complexity of Extension Definition 7

TABLE XLVII: Time Complexity Summary of Extended Definition 7

	Fixed Size	Arbitrary Size
Per Element	$O(n \cdot \log(n))$	$O\left(n \cdot \log(n)^2\right)$
In Total	$O(n \cdot \log(n))$	$O\left(n \cdot \log(n)^2\right)$

1) Time Complexity:

TABLE XLVIII: Space Complexity Summary of Extended Definition 7

	Fixed Size	Arbitrary Size
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(1)$	$O(\log(n))$

2) Space Complexity:

X. Complexity of Extension Definition 8

TABLE XLIX: Time Complexity Summary of Extended Definition 8

	Fixed Size	Arbitrary Size
Per Element	$O(n \cdot \log(n))$	$O\left(n \cdot \log(n)^2\right)$
In Total	$O(n \cdot \log(n))$	$O\left(n \cdot \log(n)^2\right)$

1) Time Complexity:

TABLE L: Space Complexity Summary of Extended Definition 8

	Fixed Size	Arbitrary Size
Per Element	$O(1)$	$O(\log(n))$
In Total	$O(1)$	$O(\log(n))$

2) Space Complexity:

References

[1] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A010060. The Thue-Morse Sequence.

[2] Lukas Spiegelhofer. The level of distribution of the thue–morse sequence. *Compositio Mathematica*, 156(12):2560–2587, 2020.

[3] Jean-Paul Allouche and Jeffrey Shallit. The ubiquitous prouhet-thue-morse sequence. In C. Ding, T. Helleseth, and H. Niederreiter, editors, *Sequences and their Applications*, pages 1–16, London, 1999. Springer London.

[4] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A010059. The inversion of the Thue-Morse Sequence.

[5] Ethan D. Bolker, Carl Offner, Robert Richman, and Catalin Zara. The prouhet–tarry–escott problem and generalized thue–morse sequences. *Journal of Combinatorics*, 7(1):117–133, 2016.

[6] M. Kolář, M. K. Ali, and Franco Nori. Generalized thue-morse chains and their physical properties. *Phys. Rev. B*, 43:1034–1047, Jan 1991.

[7] Jorg Arndt. *Matters computational*. Springer, Berlin, Germany, October 2010.

[8] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A000069. The Odious Numbers.

[9] Olivia Appleton-Crocker. Extending the thue-morse sequence source code. Available at:

https://github.com/LivInTheLookingGlass/Thue-Morse, https://thue.oliviappleton.com, 2024. This repository contains the source code and data for the research presented in this paper.

[10] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A001969. The Evil Numbers.

[11] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A159481. Number of evil numbers ¡= n.

[12] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A001316. The Gould Sequence.

[13] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A193231. Blue code for n.

[14] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[15] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A106400. The Thue-Morse Sequence, substituting $1 \to -1$ and $0 \to 1$.

[16] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A357448. The Extended Thue-Morse Sequence in Base 3/2.

[17] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A001285. The inversion of the Thue-Morse Sequence plus one.

[18] Ricardo Astudillo. On a class of thue-morse type sequences. *Journal of Integer Sequences*, 6, 2003.

[19] F. M. Dekking. The thue-morse sequence in base 3/2. *Journal of Integer Sequences*, 26, 2023.

[20] Jeffrey Shallit, Sonja Linghui Shan, and Kai Hsiang Yang. Automatic sequences in negative bases and proofs of some conjectures of shevelev, 2022.

[21] Vladimir Shevelev. Two analogs of thue-morse sequence, 2017.

[22] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A269027. The Extended Thue-Morse Sequence in base -2.

[23] Štěpán Starosta. Generalized thue-morse words and palindromic richness, 2011.

[24] Olga G. Parshina. *On Arithmetic Index in the Generalized Thue-Morse Word*, page 121–131. Springer International Publishing, 2017.

[25] Gerardo González Robert. Generalized thue-morse continued fractions, 2013.

[26] Yi Cai and Vilmos Komornik. Difference of cantor sets and frequencies in thue–morse type sequences, 2020.

[27] Jin Chen and Zhi-Xiong Wen. On the abelian complexity of generalized thue-morse sequences. *Theoretical Computer Science*, 780:66–73, August 2019.

IX. NOTES

```
Definitions From OEIS
For n >= 0, a(A004760(n+1)) = 1 - a(n). - Vladimir Shevelev, Apr 25 2009
Is this just true of numbers != 10..., or is this true of a class of prefixes?

a(A160217(n)) = 1 - a(n). - Vladimir Shevelev, May 05 2009

G.f. A(x) satisfies: A(x) = x / (1 - x^2) + (1 - x) * A(x^2). - Ilya Gutkovskiy

From Bernard Schott, Jan 21 2022: (Start)
a(n) = a(n*2^k) for k >= 0.
a((2^m-1)^2) = (1-(-1)^m)/2 (see Hassan Tarfaoui link, Concours General 1990). (End)

A004760: numbers not beginning with 10_2
A160217: min incr.ing seq. w/ a(1)=3 + that a(n) & n are both in or not in A003159
A003159: Numbers whose binary representation ends in an even number of zeros.
-----------------------------------------
Definitions from OEIS (n -> 1 - 2 * Thue-Morse(n): PNNPNPPN)
G.f. A(x) makes 0 = f(A(x), A(x^2), A(x^4)) where f(u, v, w) = v^3 - 2*u*v*w + u^2*w

G.f. A(x) satisfies 0 = f(A(x), A(x^2), A(x^3), A(x^6)) where f(u1, u2, u3, u6) = u6*u1^3 - 3*
    u6*u2*u1^2 + 3*u6*u2^2*u1 - u3*u2^3.

G.f.: Product_{k>=0} (1 - x^(2^k)) = A(x) = (1-x) * A(x^2).
(prove = to T_{2,15})

a(n) = B_n(-A038712(1)*0!, ..., -A038712(n)*(n-1)!)/n!, where B_n(x_1, ..., x_n) is the n-th
    complete Bell polynomial. See the Wikipedia link for complete Bell polynomials , and
    A036040 for the coefficients of these partition polynomials. - Gevorg Hmayakyan, Jul 10
    2016 (edited by - Wolfdieter Lang, Aug 31 2016)

a(n) = A008836(A005940(1+n)). [Analogous to Liouville's lambda] - Antti Karttunen

a(n) = (-1)^A309303(n), see the closed form (5) in the MathWorld link. - Vladimir R.

A038712: Let k be the exp. of highest pwr of 2 dividing n (A007814); a(n) = 2^(k+1)-1
A008836: Liouville's function lambda(n) = (-1)^k, where k is number of primes dividing n (
    counted with multiplicity).
A005940: The Doudna sequence
A309303: Expansion of g.f. (sqrt(x+1) - sqrt(1-3*x))/(2*(x+1)^(3/2)).
-----------------------------------------
Definitions from OEIS (inverse: 10010110)
G.f.: (1/2) * (1/(1-x) + Product_{k>=0} (1 - x^2^k)). - Ralf Stephan, Jun 20 2003
(prove = to T_{2,15})

a(n) = HW(A054429(n)) mod 2. - Antti Karttunen, May 30 2017

If A(n)=(a(0),a(2),...,a(2^n-1)), then A(n+1)=(A(n),1-A(n)). - Arie Bos, Jul 27 2022

A054429: Simple self-inverse permutation of natural numbers: List each block of 2^n numbers (
    from 2^n to 2^(n+1) - 1) in reverse order.
-----------------------------------------
Definitions from OEIS (inv add 1: 21121221)

G.f.: (3/(1 - x) - Product_{k>=0} (1 - x^(2^k)))/2. - Ilya Gutkovskiy, Apr 03 2019
(prove = to T_{2,15})
See infinite product def on wiki
```