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

Dan Rowe
Department of Math and Computer Science
Northern Michigan University
Marquette, Michigan, United States
Email: darowe@nmu.edu

***Abstract*—In this paper, we discuss various ways to extend the Thue-Morse Sequence [1] when used as a fair-share sequence. Included are N definitions of the original sequence, M extensions to n players, and proofs of equality for all definitions. In the appendix are several complexity analyses for both time and space of each definition.**

***Index Terms*—TBA**

I. Introduction

Make sure to add that while this paper does not deal with negative or fractional bases, many of the definitions are trivially extendable to that domain, and at least one of them already has been in another paper.

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

Definition	Base	Entries Tested
$T_{2,12}$	2	$2^{17} = 131,072$
Not $T_{2,12}$		$2^{27} = 134,217,728$
$T_{n,*}$	3 thru 8	
	9 thru 16	$2^{26} = 67,108,864$
	41 thru 56	
	17 thru 40	$2^{25} = 33,554,432$
	57 thru 64	
	65 thru 96	$2^{24} = 16,777,216$

II. The Original Sequence

A. Definition 1 - 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 - Invert and Extend

Appears in [1]

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:

$$\begin{aligned} \text{inv}(\mathbf{x}) &= \begin{cases} 0, & \text{if } x_i = 1 \\ 1, & \text{if } x_i = 0 \end{cases} \\ &\qquad \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ t(0) &= \langle 0 \rangle \\ t(n) &= t(n-1) \parallel \text{inv}(t(n-1)) \end{aligned} \tag{2}$$

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?

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

C. Definition 3 - Substitute and Flatten

This definition appears in [1, 2, 4]

$$\begin{aligned} s(n) &= \begin{cases} \langle 0, 1 \rangle, & \text{if } n = 0 \\ \langle 1, 0 \rangle, & \text{if } n = 1 \end{cases} \\ t(0) &= \langle 0 \rangle \\ t(n) &= \Big\|_{i=0}^{2^{n-1}-1} s(t(n-1)_i) \\ T_{2,3}(n) &= t(\lceil \log_2(n+1) \rceil)_n \end{aligned} \tag{4}$$

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

$$
\begin{aligned}
t(0) &= \langle 0 \rangle \\
t(1) &= \prod_{i=0}^{0} s(t(0)_i) = \langle 0, 1 \rangle \\
t(2) &= \prod_{i=0}^{1} s(t(1)_i) = \langle 0, 1, 1, 0 \rangle \\
T_{2,3}(3) &= t(\lceil \log_2(3+1) \rceil)_3 \\
&= t(2)_3 \\
&= \langle 0, 1, 1, 0 \rangle_3 \\
&= 0
\end{aligned} \tag{5}
$$

D. Definition 4 - Recursive Rotation

Another way to phrase the above definition is as recursive rotation. If we decompose s, we can instead represent it as:

$$
\begin{aligned}
r(\mathbf{x}, i) &= \langle x_{0+i \bmod |\mathbf{x}|}, x_{1+i \bmod |\mathbf{x}|}, \dots \rangle \\
&\quad \text{for } \mathbf{x} = \langle x_0, x_1, \dots, x_{n-1} \rangle \\
t(0) &= \langle 0 \rangle \\
t(1) &= \langle 0, 1 \rangle \\
t(n) &= \prod_{i=0}^{1} r\left(t(n-1), i \cdot 2^{n-2}\right) \\
T_{2,4}(n) &= t(\lceil \log_2(n+1) \rceil)_n
\end{aligned} \tag{6}
$$

E. Definition 5 - Recursion

This definition appears in [1, 4]

$$
\begin{aligned}
T_{2,5}(0) &= 0 \\
T_{2,5}(2n) &= T_{2,5}(n) \\
T_{2,5}(2n+1) &= 1 - T_{2,5}(n) \pmod 2
\end{aligned} \tag{7}
$$

F. Definition 6 - Highest Bit Difference

This definition appears in [5]

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_d06():
    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
```

$$
\begin{aligned}
T_{2,6}(0) &= 0 \\
T_{2,6}(n) &= \begin{matrix} \lfloor \log_2(n \oplus (n-1)) \rfloor \\ + T_{2,6}(n-1) \pm 1 \end{matrix} \pmod 2
\end{aligned} \tag{8}
$$

G. Definition 7 - Floor-Ceiling Difference

Appears in [1]

$$
\begin{aligned}
b(n) &= \begin{cases} n & \text{if } n \le 1 \\ b\left(\left\lceil \frac{n}{2} \right\rceil\right) - b\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{otherwise} \end{cases} \\
T_{2,7}(n) &= \frac{1 - b(2n)}{2} \pmod 2
\end{aligned} \tag{9}
$$

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

H. Definition 8 - Odious Number Derivation

Definition appears in [1]

Another way to generate the Thue-Morse Sequence is to take the sequence of Odious Numbers [6] 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 [7] for this is as follows:

```
from itertools import count

def seq_p2_d08():
    for i in count():
        if i.bit_count() % 2:
            yield (i + 1) % 2
```

$$
T_{2,8}(n) = \text{Odious}(n) + 1 \pmod 2 \tag{10}
$$

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

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

I. Definition 9 - Evil Numbers Derivation 1

Appears in [1]

The Evil Numbers [8] 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() % 2 == 0:
            yield i

def p2_d09():
    for n, i in enumerate(evil()):
        yield (i - 2 * n) % 2
```

$$
T_{2,9}(n) = \text{Evil}(n) - 2n \pmod 2 \tag{11}
$$

J. Definition 10 - Evil Numbers Derivation 2

Appears in [9]

A second, more-efficient derivation from the Evil Numbers is as follows, where $ce()$ is the count of Evil Numbers less than n [10], and $p()$ is the function defined in Equation 1.

$$\begin{aligned} ce(n) &= \left\lfloor \frac{n+1}{2} \right\rfloor + p(n+1) \cdot (n+1 \bmod 2) \\ T_{2,10}(n) &= 1 - ce(n+1) + ce(n) \end{aligned} \tag{12}$$

K. Definition 11 - Odious & Evil Numbers Derivation

Appears in [9]

A second, more-efficient derivation from the Evil Numbers is as follows, where $ce()$ is the count of Evil Numbers less than n [10], and $p()$ is the function defined in Equation 1.

$$\begin{aligned} oe(n) &= \begin{cases} \text{Odious}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{if } n \bmod 2 = 0 \\ \text{Evil}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) & \text{if } n \bmod 2 = 1 \end{cases} \\ T_{2,11}(n) &= 1 - oe(n) \pmod 2 \end{aligned} \tag{13}$$

L. Definition 12 - Gould's Sequence Derivation

Appears in [1]

Gould's Sequence [11] are the number of odd entries in a given row of Pascal's Triangle. Note that this is by far the least computationally efficient definition in this paper (see Tables XXIII & XXIV).

Why mod 3? Everything else is mod 2. This definition is unlikely to be extendable, unless you interpret this as the hamming weight of a given row, and extend the idea of hamming weight to those other bases. If so, I predict that the final mod is mod $(n+1)$

$$\begin{aligned} T_{2,12}(n) &= \text{Gould}(n) - 1 \bmod 3 \\ &= \left(\sum_{k=0}^{n} \left(\binom{n}{k} \bmod 2 \right) \right) - 1 \bmod 3 \end{aligned} \tag{14}$$

M. Definition 13 - Derivation from Blue Code

Appears in [1]

In the OEIS, this sequence [12] 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:

$$\begin{aligned} \text{A001317}(n) &= \sum_{k=0}^{n} \left(\binom{n}{k} \bmod 2 \right) \cdot 2^k \\ \text{A193231}(n) &= \bigoplus_{i=0}^{\lceil \log_2(n) \rceil} \text{A001317}\left(i \cdot \left\lfloor \frac{n \bmod 2}{2^i} \right\rfloor \right) \\ T_{2,13}(n) &= \text{A193231}(n) \pmod 2 \end{aligned} \tag{15}$$

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 of all such resulting values. Note that since $\text{A001317}(0) = 0$, 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 efficient compute or prove such a result

N. Summary

III. Proving Equivalence Between Standard Definitions

A. Correlating Definition 1 and Definition 5

Note that in Equation 7 where we define $T_{2,5}$, we are working in mod 2, where $+1$ and -1 are logically equivalent. We can therefore simplify its definition to be:

$$\begin{aligned} T_{2,5}(0) &= 0 \\ T_{2,5}(n) &= n + T_{2,5}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \pmod 2 \end{aligned} \tag{16}$$

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

IV. The Extensions

A. Definition 1 - Modular Digit Sums

Definition appears in [13–15]

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 $n \in \mathbb{Z}_{\geq 2}$.

$$\begin{aligned} p_n(0) &= 0 \\ p_n(x) &= x + p_n\left(\left\lfloor \frac{x}{n} \right\rfloor\right) \pmod n \end{aligned} \tag{17}$$

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

$$T_{n,1}(x, s) = p_s(x) \tag{18}$$

Note that this definition is trivially extensible to non-integer bases, though that is beyond the scope of this paper.

1) Proof of Equivalence with Original Definition 1: It is clear from visual inspection that p_2 is identical to our original definition of p.

$$\begin{aligned} p_2(x) &= p(x) \\ x + p_2\left(\left\lfloor \frac{x}{2} \right\rfloor\right) &= x + p\left(\left\lfloor \frac{x}{2} \right\rfloor\right) \\ x + \left\lfloor \frac{x}{2} \right\rfloor + p_2\left(\left\lfloor \frac{x}{2^2} \right\rfloor\right) &= x + \left\lfloor \frac{x}{2} \right\rfloor + p\left(\left\lfloor \frac{x}{2^2} \right\rfloor\right) \\ x + \left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{x}{2^2} \right\rfloor + \ldots &= x + \left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{x}{2^2} \right\rfloor + \ldots \end{aligned} \tag{19}$$

B. Definition 2 - Increment and Extend

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 $n \in \mathbb{Z}_{\geq 2}$.

$$\begin{aligned} \text{inc}(\mathbf{x}, n) = &\ x_i + 1 \pmod{n} \\ &\text{for } \mathbf{x} = (x_0, x_1, \ldots, x_{n-1}) \end{aligned} \tag{20}$$

$$\begin{aligned} t(0, n) &= \langle 0 \rangle \\ t(1, n) &= \langle 0, 1, \ldots, n-1 \rangle \\ t(x, n) &= t(x-1, n) \cdot \text{inc}(t(x-1, n), n) \end{aligned} \tag{21}$$

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.

$$\begin{aligned} T_{n,2}(0, s) &= 0 \\ T_{n,2}(x, s) &= T_{n,2}\left(x - s^{\lfloor \log_s(x) \rfloor}, s\right) + 1 \pmod{s} \end{aligned} \tag{22}$$

1) Proof of Equivalence with Original Definition 2:

C. Definition 3 - Substitute and Flatten

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.

$$\begin{aligned} b(s) &= \langle 0, 1, \cdots, s-2, s-1 \rangle \\ r(\mathbf{x}, i) &= \begin{array}{l} \langle x_{0+i \bmod |\mathbf{x}|}, x_{1+i \bmod |\mathbf{x}|}, \ldots \rangle \\ \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \end{array} \\ s(x, s) &= r(b(s), x) \\ t(0) &= \langle 0 \rangle \\ t(x, s) &= \Big\|_{i=0}^{2^{x-1}-1} s(t(x-1)_i, s) \\ T_{n,3}(x, s) &= t(\lceil \log_s(x+1) \rceil, s)_x \end{aligned} \tag{23}$$

1) Proof of Equivalence with Original Definition 3:

D. Definition 4 - Recursive Rotation

$$\begin{aligned} r(\mathbf{x}, i) &= \begin{array}{l} \langle x_{0+i \bmod |\mathbf{x}|}, x_{1+i \bmod |\mathbf{x}|}, \ldots \rangle \\ \text{for } \mathbf{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle \end{array} \\ t(0, s) &= \langle 0 \rangle \\ t(1, s) &= \langle 0, 1, \ldots, s-1 \rangle \\ t(x, s) &= \Big\|_{i=0}^{s-1} r\left(t(x-1, s), i \cdot s^{x-2}\right) \\ T_{n,4}(x, s) &= t(\lceil \log_s(x+1) \rceil, s)_x \end{aligned} \tag{24}$$

1) Proof of Equivalence with Original Definition 4:

E. Definition 5 - Recursion

Need to reference eq 16 in definition

1) Proof of Equivalence with Original Definition 5:

F. Definition 6 - Highest Digit Difference

$$\begin{aligned} \text{XOR}_n(a, b) &= \sum_{i=0}^{\lceil \log_n(\max(a,b)+1) \rceil} n^i \left(\left\lfloor \frac{a}{n^i} \right\rfloor - \left\lfloor \frac{b}{n^i} \right\rfloor \bmod n \right) \\ T_{n,6}(0, s) &= 0 \\ T_{n,6}(x, s) &= \begin{array}{l} \lfloor \log_s(\text{XOR}_s(x, x-1)) \rfloor \\ + T_{n,6}(x-1, s) + 1 \end{array} \pmod{s} \end{aligned} \tag{25}$$

Substitute n for 2, then simplify, plus a bit

1) Proof of Equivalence with Original Definition 6:

G. Summary

V. Proving Equivalence Between Extended Definitions

A. Summary

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

A. Use as a Fair-Share Sequence

B. Aperiodicity

C. Palindrome

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

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

VIII. Appendix

A. Complexity of Original Definition 1

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

$$T_{2,1}(n) = \left(\sum_{i=0}^{\lceil \log_2(n+1) \rceil} \left\lfloor \frac{n}{2^i} \bmod 2 \right\rfloor \right) \bmod 2 \tag{26}$$

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 I
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 II
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 1

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

B. Complexity of Original Definition 2

TABLE III
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 2

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE IV
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 2

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

C. Complexity of Original Definition 3

TABLE V
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 3

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE VI
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 3

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

D. Complexity of Original Definition 4

TABLE VII
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 4

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE VIII
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 4

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

E. Complexity of Original Definition 5

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 IX
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 5

	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 X
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 1

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

F. Complexity of Original Definition 6

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 XI
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 6

	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 XII
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 6

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

2) Space Complexity:

G. Complexity of Original Definition 7

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 XIII
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 7

	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 XIV
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 7

		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	$O\left(n^2\right)$	$O\left(n^2 \cdot \log(n)\right)$

H. Complexity of Original Definition 8

TABLE XV
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 8

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XVI
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 8

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

I. Complexity of Original Definition 9

TABLE XVII
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 9

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XVIII
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 9

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

J. Complexity of Original Definition 10

TABLE XIX
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 10

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XX
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 10

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

K. Complexity of Original Definition 11

TABLE XXI
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 11

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXII
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 11

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

L. Complexity of Original Definition 12

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 XXIII
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 12

		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 XXIV
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 12

		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:

M. Complexity of Original Definition 13

TABLE XXV
TIME COMPLEXITY SUMMARY OF STANDARD DEFINITION 13

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXVI
SPACE COMPLEXITY SUMMARY OF STANDARD DEFINITION 13

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

N. Complexity of Extension Definition 1

In an idealized case, this definition will simplify to:

$$T_{n,1}(x,s) = \left(\sum_{i=0}^{\lceil \log_s(x+1) \rceil} \left\lfloor \frac{x}{s^i} \bmod s \right\rfloor \right) \bmod s \tag{27}$$

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 XXVII
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 XXVIII
SPACE COMPLEXITY SUMMARY OF EXTENDED DEFINITION 1

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

O. Complexity of Extension Definition 2

TABLE XXIX
TIME COMPLEXITY SUMMARY OF EXTENDED DEFINITION 2

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXX
SPACE COMPLEXITY SUMMARY OF EXTENDED DEFINITION 2

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

P. Complexity of Extension Definition 3

TABLE XXXI
TIME COMPLEXITY SUMMARY OF EXTENDED DEFINITION 3

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXXII
SPACE COMPLEXITY SUMMARY OF EXTENDED DEFINITION 3

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

Q. Complexity of Extension Definition 4

TABLE XXXIII
TIME COMPLEXITY SUMMARY OF EXTENDED DEFINITION 4

	Fixed Size	Arbitrary Size
Per Element		
In Total		

1) Time Complexity:

TABLE XXXIV
SPACE COMPLEXITY SUMMARY OF EXTENDED DEFINITION 4

	Fixed Size	Arbitrary Size
Per Element		
In Total		

2) Space Complexity:

R. Complexity of Extension Definition 5

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 XXXV
TIME COMPLEXITY SUMMARY OF EXTENDED DEFINITION 5

	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 XXXVI
SPACE COMPLEXITY SUMMARY OF EXTENDED DEFINITION 5

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

S. Complexity of Extension Definition 6

TABLE XXXVII
TIME COMPLEXITY SUMMARY OF EXTENDED DEFINITION 6

	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 XXXVIII
SPACE COMPLEXITY SUMMARY OF EXTENDED DEFINITION 6

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

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

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

[7] 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.

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

[9] 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.

[10] OEIS Foundation Inc. The on-line encyclopedia of integer sequences (oeis). Accessed: 2024-10-24, 2024. OEIS Sequence ID: A159481.

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

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

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

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

[15] 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.

Definitions From OEIS
G.f.: (1/(1 - x) - Product_{k >= 0} (1 - x^(2^k)))/2. - Benoit Cloitre, Apr 23 2003

For n >= 0, a(A004760(n+1)) = 1 - a(n). - Vladimir Shevelev, Apr 25 2009

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).

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

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

a(n) = (1 + (-1)^HW(n))/2. - Lorenzo Sauras Altuzarra, Mar 10 2024

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
See infinite product def on wiki. Might shed some light on other inf. prods. in here