The Hidden Order of the Collatz Problem

Canonical Predecessors, Terminal Seeds, and Structural Classification of Odd Integers

Krishnaswamy Bharati Gandhi Mohan
Independent Researcher

krishna.gandhimohan@gmail.com
kmohan@stravoris.com

DOI: 10.5281/zenodo.18119832

Abstract

The Collatz map is one of the simplest unsolved problems in mathematics to state and one of the most resistant to global analysis. While its forward iteration appears chaotic and its backward structure exhibits unbounded branching, the dynamics remain governed by explicit local rules.

In this paper we develop a **complete symbolic framework** for the accelerated Collatz map that reorganizes both backward and forward dynamics into finite, reversible, and computable structures. Rather than analyzing numerical trajectories directly, we encode Collatz dynamics using finite symbolic words, canonical representatives, and exact decoding rules.

We show that every odd integer admits a unique symbolic predecessor chain terminating at a **canonical terminal seed** $m_0 \equiv 3, 9, 15 \pmod{24}$. These seeds index disjoint **terminal families** that partition the odd integers. Infinite backward predecessor trees are shown to collapse into finite symbolic descriptions via compressed 01-lift structure, yielding exact **orbit codes** over a finite alphabet.

Forward accelerated dynamics are reinterpreted at the family level: successor steps move within a terminal family, while normalization induces discrete transitions between families. The resulting **terminal chains** τ record the ordered sequence of terminal families actually traversed by an orbit. Reversing these chains yields reverse τ-chains, which assemble into a rooted **terminal family tree** anchored at the forward-terminal family 1. This tree provides a canonical phase space for accelerated Collatz dynamics at the family level, revealing a strict hierarchical descent in canonical depth even when numerical values increase.

The symbolic framework is extended to all positive integers via **universal orbit codes**, which encode both accelerated odd dynamics and 2-adic structure. We show that universal orbit codes admit complete symbolic decoding, allowing the full classical Collatz orbit of any integer to be reconstructed exactly without iterating the Collatz map. As a consequence, the classical stopping time becomes a directly computable symbolic quantity.

While this work does not resolve the Collatz conjecture, it transforms the problem from an opaque iteration process into a **finite, symbolically organized system** with explicit structure. Global convergence is reduced to a precise structural condition on canonical terminal seeds, localizing any potential obstruction to a discrete, indexed set. The framework provides a new foundation for structural, computational, and probabilistic approaches to Collatz dynamics.

Table of Contents

Chapter 1

Introduction

The Collatz problem is one of the simplest unsolved problems in mathematics to state and one of the most resistant to analysis. Starting from any positive integer N, the Collatz map applies the rule

- divide by 2 if N is even,
- replace N by $3N + 1$ if N is odd,

and repeats. The conjecture asserts that this process eventually reaches 1 for every starting value.

Despite decades of effort, no proof or counterexample is known. The difficulty of the problem does not lie in the local behavior of the map, which is completely explicit, but in the apparent lack of global structure. Forward iteration produces trajectories that appear irregular and chaotic, while backward analysis reveals explosive branching that quickly becomes unmanageable. As a result, most existing approaches rely either on numerical experimentation or on partial analytic results tied to specific residue classes, stopping-time statistics, or probabilistic heuristics.

This paper develops a different perspective. Rather than studying Collatz trajectories as raw numerical sequences, we introduce a **symbolic framework** that captures the full structure of Collatz dynamics using finite words, canonical representatives, and reversible encoding rules. The goal is not to prove the conjecture directly, but to **organize the Collatz graph** into a form that is globally navigable, structurally explicit, and computationally exact.

1.1 Accelerated dynamics and structural compression

A central obstacle in Collatz analysis is the dominance of trivial even steps. Any even integer undergoes repeated divisions by 2 until an odd value is reached, after which a single odd transformation occurs. This motivates the study of the **accelerated Collatz map**, which collapses each even run into a single step and acts only on odd integers.

Acceleration simplifies forward dynamics, but backward analysis remains highly nontrivial: each odd integer admits infinitely many predecessors, forming unbounded vertical towers of even lifts.

The key insight of this work is that these infinite backward structures admit a **finite symbolic description**. By identifying canonical representatives, predecessor classes, and compression rules, we show that every odd integer belongs to a uniquely determined symbolic lineage that can be recorded as a finite word over a small alphabet.

This symbolic compression is exact: no information about the Collatz dynamics is lost.

1.2 Terminal seeds and symbolic lineage

At the heart of the framework lies the concept of a **canonical terminal seed**. These are odd integers m_0 satisfying

$$m_0 \equiv 3, 9, 15 \pmod{24},$$

which serve as terminal bases for accelerated predecessor chains.

Every odd integer admits a unique finite backward chain terminating at such a seed. The backward path from 1 to m_0 can be encoded symbolically using a finite word $\omega(m_0)$ over the alphabet $\{\mathsf{A}, \mathsf{C}, \mathsf{K}\}$. This word records, in compressed form, the full predecessor structure leading to m_0. Crucially, these symbolic encodings are **canonical**: they do not depend on arbitrary choices, numerical bounds, or truncation depth. The set of terminal seeds, together with their orbit codes, forms a discrete index set that partitions the odd integers into **terminal-indexed families**.

1.3 Universal orbit codes

While terminal orbit codes describe accelerated dynamics on odd integers, the classical Collatz map acts on all positive integers. To bridge this gap, we introduce the **universal orbit code**

$$\omega^*(N) = \omega(n)\,\mathbf{.}\,z,$$

where $N = 2^z n$ with n odd.

The suffix z records the 2-adic height of N above its odd core, while the symbolic body $\omega(n)$ encodes the entire accelerated structure. Together, these data locate N uniquely within the full Collatz graph. A central result of the paper is that $\omega^*(N)$ admits a **complete symbolic decoding**: from the code alone, one can reconstruct the **entire classical Collatz orbit of N**, including all intermediate even values, without applying the Collatz map directly. The encoding and decoding procedures are exact inverses.

1.4 Symbolic reconstruction and stopping times

Once Collatz orbits are available symbolically, new quantitative information becomes accessible. In particular, we show that the **Collatz stopping time** of any integer N can be computed directly from its universal orbit code. Each symbol contributes a fixed number of steps, and the total stopping time becomes a simple symbolic sum. This yields a new invariant: the **symbolic distance** from N to 1. No numerical iteration is required. The stopping time is determined entirely by the finite symbolic representation.

1.5 The Terminal Seed Atlas

To support both theoretical analysis and reproducibility, we construct a **Terminal Seed Atlas**: a database of canonical terminal seeds, their orbit codes, and the indexed families of odd integers that terminate at each seed. The Atlas makes explicit the global partition of the odd integers induced by the symbolic framework. It also provides a concrete computational artifact that can be queried, extended, and independently verified. Finite excerpts of the Atlas are provided as supplementary material, together with reference implementations for its construction.

1.6 Scope and contribution

This paper does not claim to resolve the Collatz conjecture, nor does it propose a new heuristic, numerical pattern, or partial verification strategy. Instead, it develops an exact symbolic framework in which Collatz dynamics are encoded, reconstructed, and analyzed without iteration. Its contribution is structural. We provide:

- a canonical symbolic encoding of accelerated Collatz dynamics,
- a reversible compression of infinite predecessor trees,
- a universal coding scheme for all positive integers,
- a symbolic method for reconstructing full Collatz orbits,
- a direct symbolic formula for stopping times,
- and a concrete Atlas organizing the Collatz graph into terminal-indexed families.

Taken together, these results transform the Collatz problem from an opaque iteration process into a **finite, symbolic, and globally organized system**. Whether this structure can be leveraged toward a proof of convergence remains an open question. But it provides a new foundation on which such arguments can be built.

1.7 Computational Resources

All code, data artifacts, and reference implementations associated with this work are available at:
https://github.com/stravoris-tech/collatz-hidden-order

Chapter 2

Groundwork and Preliminaries

2.1 Successor and Predecessor Sequences

We begin with a simple example. The classical Collatz sequence generated by the integer 7 is:

$$[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1].$$

Repeated division by 2 reveals that all even values eventually reduce to an odd number. Thus it is natural to extract only the odd terms:

$$[7,\ 11,\ 17,\ 13,\ 5,\ 1].$$

We call this the **(odd) successor sequence** of 7. Every step applies the accelerated Collatz operation

$$n_{t-1} = \frac{3n_t + 1}{2^x},$$

for some integer $x \geq 1$, determined uniquely by the power of 2 dividing $3n_t + 1$.

Reversing the View: Predecessor Sequences

For later analysis it will be convenient to reverse this odd-only successor sequence:

$$[1,\ 5,\ 13,\ 17,\ 11,\ 7].$$

This is the **predecessor sequence** of 7: each term is an odd integer whose accelerated successor is the previous term. For example:

- 5 is a predecessor of 13,
- 13 is a predecessor of 17,
- 17 is a predecessor of 11,

and so on. At this stage we emphasize that a number may have **multiple** potential predecessors; we will derive the precise predecessor formula later in this chapter.

Examples

For the integer 17:

- 11 is a predecessor, because

$$\frac{3 \cdot 11 + 1}{2^1} = 17, \quad \text{equivalently} \quad 11 = \frac{2^1 \cdot 17 - 1}{3}.$$

- 13 is a successor, because

$$\frac{3 \cdot 17 + 1}{2^2} = 13.$$

Formal Definitions

Let n_t be an odd integer.

- A **predecessor** of n_t is an odd integer n_{t+1} for which

$$\frac{3n_{t+1} + 1}{2^a} = n_t \quad \text{for some integer } a \geq 1$$

  or equivalently,

$$n_{t+1} = \frac{2^a n_t - 1}{3}.$$

- A **successor** of n_t is an odd integer reached under the accelerated map

$$n_{t-1} = T(n_t) = \frac{3n_t + 1}{2^b},$$

  where $b = \nu_2(3n_t + 1)$.

Reference implementations are given in Appendix D.Sec-2.1

2.2 Binary Patterns and Predecessors

Recall from §2.1 that a **predecessor** of an odd integer n_t is any odd integer n_{t+1} satisfying

$$\frac{3n_{t+1} + 1}{2^a} = n_t \qquad (a \geq 1).$$

Equivalently,

$$n_{t+1} = \frac{2^a n_t - 1}{3}.$$

To illustrate the behavior of these predecessors, we examine several examples.

Example 1: Predecessors of the Odd Integer 1367

Consider all odd integers n_{t+1} for which a single accelerated Collatz step maps n_{t+1} to 1367. Since $\frac{2^a n_t - 1}{3}$ grows strictly with a, only finitely many predecessors appear below any fixed cutoff on a. A sample computation produces the first several predecessors:

$$[911,\ 3645,\ 14581,\ 58325,\ 233301,\ 933205,\ 3732821,\ 14931285,\ 59725141,\ 238900565].$$

The binary expansion of 1367 is $0b10101010111$, while the smallest predecessor has binary form $911 = 0b1110001111$.

Observation 1: The exponents form an arithmetic progression
For these predecessors, the valid exponents a are

$$1, 3, 5, 7, \ldots$$

This arithmetic progression persists throughout the list.

Observation 2: Predecessors arise by appending trailing 01-blocks
Each predecessor is obtained by appending one or more trailing 01 blocks to the binary expansion of the smallest predecessor. Symbolically,

$$\text{binary}(n_{t+1}) = \text{binary}(911)\ 01\ 01\ \ldots\ 01.$$

For example, repeated appending yields the binary string

$$0b111000111101010101010101010101010101,$$

corresponding to the large predecessor 3914146862421.

Why does this happen?
If n is any predecessor of 1367, then so is $4n + 1$, since

$$3(4n + 1) + 1 = 4(3n + 1),$$

which increases the 2-adic valuation by exactly 2. Each application of this transformation appends an additional trailing 01 block to the binary expansion of n. Thus a single predecessor generates an infinite family via

$$n \longmapsto 4n + 1 \longmapsto 4(4n + 1) + 1 \longmapsto \cdots.$$

Example 2: The Special Case of the Integer 1

Now consider all odd integers that map to 1 under a single accelerated Collatz step. The first several such integers are

$$[1,\ 5,\ 21,\ 85,\ 341,\ 1365,\ 5461,\ 21845,\ 87381,\ 349525,\ \ldots].$$

This case is exceptional. For a general odd integer such as 1367, the predecessor list does not include the integer itself. For 1, however, the list includes 1, reflecting the familiar loop

$$1 \to 4 \to 2 \to 1$$

in the original Collatz iteration. The appearance of 1 as its own predecessor will play an important structural role in later chapters.

Summary of Findings

From these examples we observe:

1. Every odd integer has infinitely many immediate predecessors.
2. Once a single predecessor is known, all others are generated by repeatedly applying $n \longmapsto 4n+1$.
3. In binary, this corresponds to appending trailing 01 blocks.
4. The special case of 1 follows the same pattern, with the additional feature that 1 is a predecessor of itself.

These observations form the foundation for the structural analysis developed in the following sections.

A reference implementation for generating predecessor lists is provided in Appendix D.Sec-2.2.

2.3 Count of Trailing Zeros (ν_2)

Given a positive integer n, the quantity

$$\mathrm{ctz}(n) = \nu_2(n)$$

counts the number of trailing zeros in the binary expansion of n. Equivalently, $\nu_2(n)$ is the **2-adic valuation** of n: the largest integer k such that $2^k \mid n$.

In the accelerated Collatz update

$$n_{t-1} = \frac{3n_t + 1}{2^x},$$

the exponent x is exactly the 2-adic valuation of $3n_t + 1$, namely

$$x = \nu_2(3n_t + 1).$$

Thus ν_2 plays a central structural role in determining successor steps under the accelerated map.

Computing $\nu_2(n)$

A standard bit-level identity extracts the lowest set bit of a positive integer n via

$$n \,\&\, (-n).$$

The position of this bit determines the 2-adic valuation. In particular,

$$\nu_2(n) = (n \,\&\, (-n)).\mathrm{bit\_length}() - 1.$$

This yields a concise and efficient implementation as in Appendix D.Core.

2.4 Powers of Two in Predecessor Computations

Section 2.2 showed that once a single predecessor of an odd number is known, infinitely many more can be generated by the map

$$p \longmapsto 4p + 1.$$

This corresponds to appending a binary 01 and increases the accelerated exponent by exactly 2. We now study how the exponent

$$x = \nu_2(3p + 1)$$

behaves when searching for *immediate predecessors* of a fixed odd integer n_t.

The 01-Suffix Property

Theorem (01-Suffix Property).
If an odd integer p satisfies

$$\frac{3p + 1}{2^x} = n_t$$

then

$$4p + 1$$

is also a predecessor of n_t. Equivalently, appending a binary 01 to p yields another predecessor.

Reason.
Multiplying p by 4 and adding 1 gives

$$3(4p + 1) + 1 = 4(3p + 1)$$

which increases $\nu_2(3p + 1)$ by exactly 2, without changing the resulting quotient n_t. Thus the predecessor condition is preserved. This allows an entire *infinite predecessor class* to be generated once a single predecessor is found.

Three Fundamental Cases

When searching for predecessors of a fixed odd integer n_t, the values of $x = \nu_2(3p + 1)$ fall into **three distinct patterns**. We illustrate each case with explicit examples.

Case 1: x runs through odd integers
(Example: predecessors of $n_t = 59$)

We seek all odd p such that

$$\frac{3p + 1}{2^x} = 59.$$

A computation produces:

$$p = 39,\ 157,\ 629,\ 2517,\ 10069,\ 40277,\ 161109,\ 644437, \ldots$$

Each of these values satisfies $\frac{3p+1}{2^x} = 59$, with the corresponding exponents x taking the values $1, 3, 5, 7, \ldots$.

Pattern: The values of x are $1, 3, 5, 7, \ldots$ all *odd.* Each new predecessor is obtained by applying $p \mapsto 4p + 1$, i.e., by appending a binary 01.

Case 2: x runs through even integers
(Example: predecessors of $n_t = 103$*)*

We now solve

$$\frac{3p + 1}{2^x} = 103.$$

We obtain:

$$p = 137,\ 549,\ 2197,\ 8789,\ 35157,\ 140629,\ 562517, \ldots$$

Each of these values satisfies $\frac{3p+1}{2^x} = 103$, with the corresponding exponents x taking the values $2, 4, 6, 8, \ldots$.

Pattern: Here the values of x are $2, 4, 6, 8, \ldots$ all *even.* And again, all of these arise as repeated applications of $p \mapsto 4p + 1$ starting from the smallest predecessor.

Case 3: No predecessors exist
(Example: $n_t = 21$*)*

For some odd integers, the equation

$$\frac{3p + 1}{2^x} = n_t$$

has **no solution** in odd integers p. For example, There is **no** odd integer p and exponent $x \geq 1$ satisfying

$$\frac{3p + 1}{2^x} = 21.$$

In such cases, the number n_t has **no immediate predecessors under the accelerated map**. This phenomenon becomes important later, when we classify which odd values can arise as endpoints of backward Collatz chains.

Summary of the Three Cases

For any odd integer n_t:

1. It may have predecessors with **odd** exponents x.
2. Or predecessors may have **even** exponents x.
3. Or it may have **no predecessors** at all.

These three cases form the starting point for a deeper structural analysis. The next chapter refines this behavior using modular constraints, and Chapter 4 will show how every odd integer is encoded by a unique sequence of A-steps and C-steps.

Chapter 3

Predecessor Analysis

3.1 Modular Classification of Odd Numbers

In this chapter we analyze the predecessor formula in detail. Recall that an odd integer n_{t+1} is a **predecessor** of an odd integer n_t if

$$\frac{3n_{t+1}+1}{2^x} = n_t, \qquad x \geq 1,$$

where the exponent $x = \nu_2(3n_{t+1}+1)$ counts the number of factors of 2 removed to reach the next odd term. Rearranging gives the equivalent formula

$$n_{t+1} = \frac{2^x n_t - 1}{3}.$$

Since n_{t+1} must be an integer, the numerator $2^x n_t - 1$ must be divisible by 3. Thus, for each fixed odd n_t, the existence and nature of its predecessors is governed entirely by the residue of $2^x n_t - 1$ modulo 3.

In this section we examine how the divisibility pattern behaves as x varies. Three distinct cases appear.

Case 1 — Solutions occur for odd x

(Example: $n_t = 65$)

We compute

$$n_{t+1} = \frac{2^x \cdot 65 - 1}{3}$$

for successive values of x. The first several integer predecessors occur at $x = 1, 3, 5, 7, 9$, yielding the values $43, 173, 693, 2773, 11093$, respectively. All valid values of x - $x = 1, 3, 5, 7, \ldots$ - are **odd integers**. Each new predecessor is obtained from the smallest one by repeatedly applying the transformation

$$p \longmapsto 4p + 1,$$

which appends a binary suffix 01. Thus **Case 1** consists of those odd integers whose predecessors occur only for odd values of x.

Case 2 — Solutions occur for even x

(Example: $n_t = 79$)

Now compute

$$n_{t+1} = \frac{2^x \cdot 79 - 1}{3}.$$

The first several integer predecessors occur at $x = 2, 4, 6, 8, 10$, yielding the values $105, 421, 1685, 6741, 26965$, respectively. Here the valid values of x - $x = 2, 4, 6, 8, \dots$ - are **even integers**. Again, each predecessor is obtained from the smallest by the map $p \mapsto 4p + 1$. Thus **Case 2** consists of those odd integers whose predecessors occur only for even values of x.

Case 3 — No value of $x \geq 1$ gives a predecessor

(Example: $n_t = 21$)

Computing

$$n_{t+1} = \frac{2^x \cdot 21 - 1}{3}$$

yields **no integer solutions** for any $x \geq 1$:

- $2^1 \cdot 21 - 1 = 41 \equiv 2 \pmod{3}$
- $2^2 \cdot 21 - 1 = 83 \equiv 2 \pmod{3}$
- $2^3 \cdot 21 - 1 = 167 \equiv 2 \pmod{3}$
- … and so on.

Thus **21 has no predecessors** under the accelerated map. This occurs when neither of the first two numerators, $2n_t - 1$ nor $4n_t - 1$, is divisible by 3.

This motivates the following characterization.

Theorem (Mod-3 Predecessor Obstruction).

If **neither** $4n - 1$ nor $2n - 1$ is divisible by 3, then no value of $k \geq 1$ can make $2^k n - 1$ divisible by 3. Hence n has **no predecessors**.

Proof.

Since $2 \equiv -1 \pmod{3}$, we have

$$2^k \equiv (-1)^k \pmod{3}.$$

Thus

$$2^k n - 1 \equiv \begin{cases} n - 1, & k \text{ even}, \\ -n - 1 \equiv 2n - 1, & k \text{ odd}. \end{cases}$$

Now:

- For **even** k, $2^k n - 1 \equiv n - 1 \equiv 4n - 1 \pmod{3}$, since $4 \equiv 1 \pmod{3}$.

- For **odd** k, $2^k n - 1 \equiv 2n - 1 \pmod 3$.

Therefore, if $3 \mid (2^k n - 1)$, then **either**

- $3 \mid (4n - 1)$ (even k), **or**
- $3 \mid (2n - 1)$ (odd k).

Hence, if **neither** $4n - 1$ nor $2n - 1$ is divisible by 3, then **no** value of $k \geq 1$ can satisfy $3 \mid (2^k n - 1)$. Thus n has **no predecessors**. ■

A diagnostic routine used to scan values of $2^x n - 1$ for divisibility by 3 is provided in Appendix D.Sec-3.1.

3.2 Deriving the Smallest Predecessor Formula

In §3.1 we established a complete modular criterion for determining whether an odd integer n_t has predecessors. In this section we refine that analysis to obtain a **closed-form expression** for the *smallest* predecessor when one exists. The goal is to show:

$$\boxed{n_{t+1} = \left\lfloor \frac{\big((n_t - 3) \bmod 6\big)\, n_t - 1}{3} \right\rfloor}$$

This single expression encodes **all three predecessor cases** in a uniform way, automatically selecting the correct formula based solely on the residue of n_t modulo 6.

Step 1 — The Divisibility Condition

From the predecessor identity

$$n_{t+1} = \frac{2^x n_t - 1}{3},$$

a predecessor exists exactly when the numerator $2^x n_t - 1$ is divisible by 3.

From §3.1 we found:

- If $3 \mid (4n_t - 1)$, then x must be **odd**, and the smallest predecessor is

$$\frac{4n_t - 1}{3}.$$

- If $3 \mid (2n_t - 1)$, then x must be **even**, and the smallest predecessor is

$$\frac{2n_t - 1}{3}.$$

- If neither is divisible by 3, no predecessor exists.

Thus everything depends on the residues of $2n_t - 1$ and $4n_t - 1$ modulo 3.

Step 2 — Restricting to Odd Integers (Modulo 6)

Every odd integer lies in exactly one of the congruence classes

$$n_t \equiv 1, 3, 5 \pmod{6}.$$

These correspond to the three cases:

- $n_t \equiv 1 \pmod{6} \Rightarrow n_t \equiv 1 \pmod{3} \Rightarrow 4n_t - 1$ is divisible by 3, but $2n_t - 1$ is not.
- $n_t \equiv 3 \pmod{6} \Rightarrow n_t \equiv 0 \pmod{3} \Rightarrow$ neither expression is divisible by $3 \Rightarrow$ **no predecessor**.
- $n_t \equiv 5 \pmod{6} \Rightarrow n_t \equiv 2 \pmod{3} \Rightarrow 2n_t - 1$ is divisible by 3, but $4n_t - 1$ is not.

Thus:

$n_t \pmod{6}$	Predecessor Condition	Smallest Predecessor
1	$3 \mid (4n_t - 1)$	$(4n_t - 1)/3$
3	none	-1
5	$3 \mid (2n_t - 1)$	$(2n_t - 1)/3$

Step 3 — Encoding the Three Cases Compactly

Instead of using conditional cases, we encode the three outputs using $(n_t - 3) \bmod 6$. Its values are:

$n_t \pmod{6}$	$(n_t - 3) \bmod 6$	Encoded Output
1	4	$4n_t - 1$
3	0	$0n_t - 1 = -1$
5	2	$2n_t - 1$

Thus the expression $((n_t - 3) \bmod 6)\, n_t - 1$ automatically becomes:

- $4n_t - 1$ when $n_t \equiv 1 \pmod{6}$,
- $0 \cdot n_t - 1 = -1$ when $n_t \equiv 3 \pmod{6}$,
- $2n_t - 1$ when $n_t \equiv 5 \pmod{6}$.

Dividing by 3 gives the exact predecessor formulas in the first and third cases.

Step 4 — Why the Floor Function Appears

For $n_t \equiv 1$ or $5 \pmod{6}$, the expressions $(4n_t - 1)/3$ and $(2n_t - 1)/3$ are integers. For $n_t \equiv 3 \pmod{6}$, the numerator is -1, so

$$\frac{-1}{3} = -\frac{1}{3}, \qquad \left\lfloor -\frac{1}{3} \right\rfloor = -1.$$

Thus the floor function makes all three cases consistent:

- returns an integer predecessor when one exists,
- returns -1 when none exists.

Final Unified Formula

For every odd integer n_t,

$$\boxed{n_{t+1} = \left\lfloor \frac{\big((n_t - 3) \bmod 6\big)\, n_t - 1}{3} \right\rfloor}$$

This gives:

1. **Growth:** $n_t \equiv 1 \pmod 6$ → predecessor = $(4n_t - 1)/3$
2. **Termination:** $n_t \equiv 3 \pmod 6$ → no predecessor = -1
3. **Shrinkage:** $n_t \equiv 5 \pmod 6$ → predecessor = $(2n_t - 1)/3$

And once the smallest predecessor is known, all others follow via $p \longmapsto 4p + 1$, which appends a trailing binary 01.

Demonstration (Reminder)

For any odd integer n_t:

- A **predecessor** is an odd integer n_{t+1} satisfying

$$n_{t+1} = \frac{2^a n_t - 1}{3}.$$

- A **successor** is an odd integer n_{t-1} satisfying

$$n_{t-1} = \frac{3n_t + 1}{2^b}.$$

For example:

$$\frac{3 \cdot 11 + 1}{2} = 17, \qquad \frac{2 \cdot 17 - 1}{3} = 11.$$

Hence:

- successor of 11 is 17,
- predecessor of 17 is 11.

Thus the smallest predecessor of any odd n_t is given by the unified formula above.

A reference implementation for generating the smallest predecessor is provided in Appendix D.Core.

3.3 Formula for Multiple Immediate Predecessors

In §3.2 we derived a closed-form expression for the **smallest predecessor** of an odd integer n_t. In this section we show that *all* of its immediate predecessors arise by iterating a simple linear map, and we obtain a closed-form formula for every one of them. Let n_t be an odd integer. Let n_{t+1} denote its **smallest predecessor**, given by the unified expression

$$n_{t+1} = \left\lfloor \frac{\big((n_t - 3) \bmod 6\big)\, n_t - 1}{3}. \right\rfloor$$

We now describe and derive the full infinite class of the immediate predecessors of n_t. If $n_t \equiv 3$ (mod 6), then no predecessor exists and the construction below does not apply.

The Immediate Predecessor Class

Every odd predecessor of n_t is of the form

$$\mathrm{Pred}_k(n_t) = 4^k\, n_{t+1} \; + \; \frac{4^k - 1}{3}, \qquad k = 0, 1, 2, \ldots$$

where:

- $k = 0$ gives the smallest predecessor,
- each increment of k corresponds to undoing two additional halving steps in the forward map.

We now derive this formula.

Closure of Predecessors Under $4x + 1$

Let p be an odd predecessor of n_t, i.e. $T(p) = n_t$ under the accelerated map. Consider the transformed value $4p + 1$. Then

$$3(4p + 1) + 1 = 12p + 4 = 4(3p + 1)$$

,

so the accelerated map behaves as:

$$T(4p + 1) = \frac{3(4p + 1) + 1}{2^{\nu_2(3(4p+1)+1)}} = \frac{4(3p + 1)}{4 \cdot 2^{\nu_2(3p+1)}} = n_t.$$

Thus:

$$T(4p + 1) = T(p) = n_t.$$

Conclusion: If p is a predecessor of n_t, then so is $4p + 1$. Hence the map $x \longmapsto 4x + 1$ preserves the entire predecessor set.

Constructing All Immediate Predecessors

Let $p_0 = n_{t+1}$ be the smallest predecessor. Generate a sequence by:

$$p_{k+1} = 4p_k + 1, \qquad k \geq 0.$$

By the closure property above, **every** p_k is a predecessor of n_t. Moreover, every odd predecessor arises this way: iterating $4x + 1$ corresponds exactly to undoing two additional halving steps in the forward Collatz map. Thus the infinite predecessor class is exactly $\{p_k : k \geq 0\}$.

Solving the Recurrence

Expanding the recurrence:

$$\begin{aligned}
p_1 &= 4p_0 + 1, \\
p_2 &= 4p_1 + 1 = 4(4p_0 + 1) + 1 = 4^2 p_0 + (4 + 1), \\
p_3 &= 4p_2 + 1 = 4^3 p_0 + (4^2 + 4 + 1), \\
&\vdots
\end{aligned}$$

Thus the pattern is:

$$p_k = 4^k p_0 \ + \ \left(4^{k-1} + 4^{k-2} + \cdots + 4 + 1\right).$$

The geometric sum evaluates to:

$$1 + 4 + 4^2 + \cdots + 4^{k-1} = \frac{4^k - 1}{3}.$$

Hence

$$p_k = 4^k p_0 + \frac{4^k - 1}{3}.$$

Substituting $p_0 = n_{t+1}$, we obtain the closed-form predecessor formula:

$$\boxed{\mathrm{Pred}_k(n_t) = 4^k n_{t+1} + \frac{4^k - 1}{3}, \qquad k = 0, 1, 2, \ldots}$$

with $k = 0$ giving the smallest predecessor.

Example: $n_t = 13$

Smallest predecessor: $n_{t+1} = 17$. Then:

- $k = 0:\ 17$
- $k = 1:\ 4 \cdot 17 + 1 = 69$
- $k = 2:\ 16 \cdot 17 + 5 = 277$
- $k = 3:\ 64 \cdot 17 + 21 = 1109$
- $k = 4:\ 256 \cdot 17 + 85 = 4437$

Each of these satisfies $T(p_k) = 13$.

The sequence p_k is called a **01-tower** because each step

$$p_{k+1} = 4p_k + 1$$

corresponds in binary to appending the two-bit pattern $\mathbf{01}$ to p_k. Thus the entire immediate predecessor class is created by repeatedly attaching trailing 01-blocks.

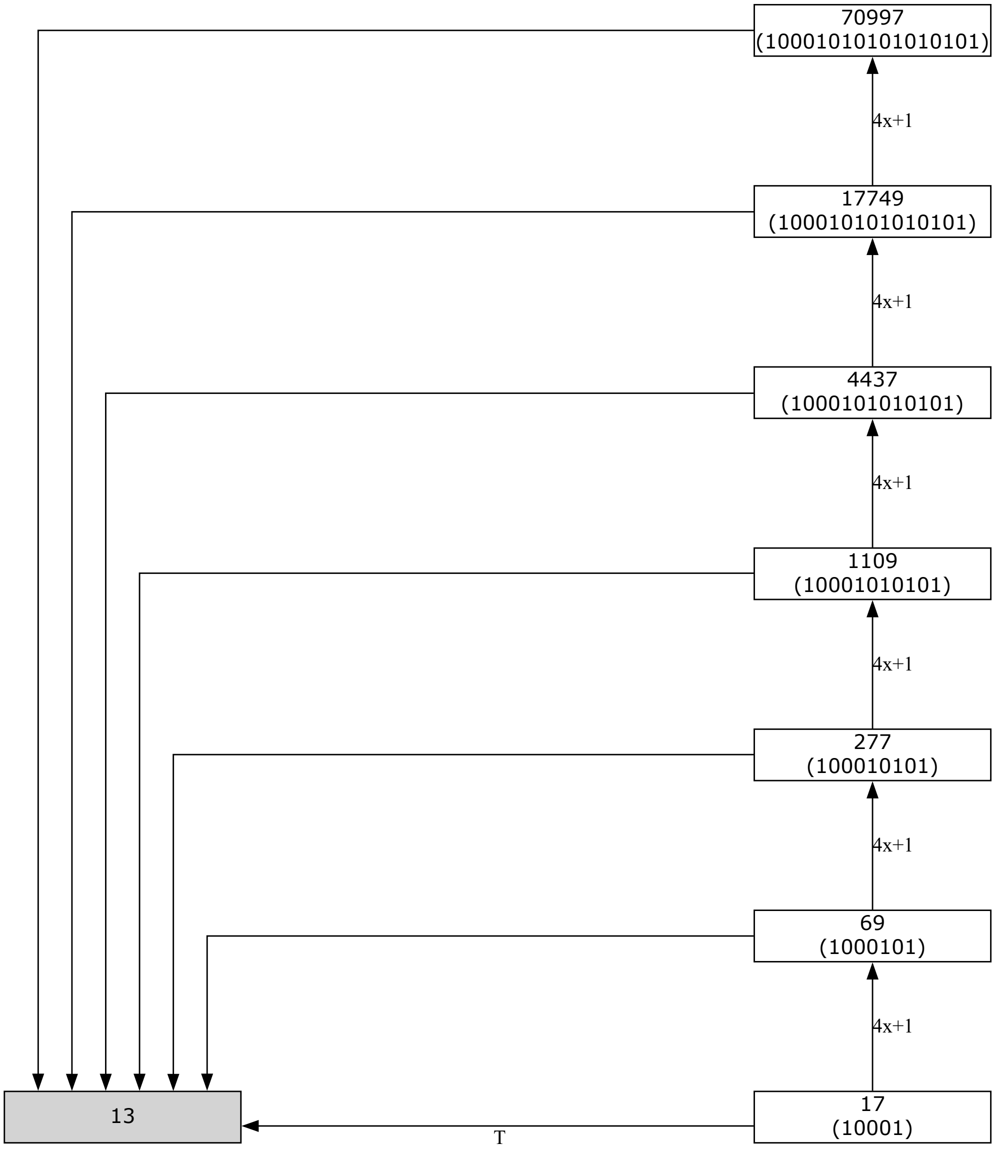

Figure 3.1. *The 01-tower of immediate predecessors of 13.* Each node in the vertical chain is obtained from the one below by the map $p_{k+1} = 4p_k + 1$, starting from the smallest predecessor 17. Every value in the tower satisfies $T(p_k) = 13$, illustrating the general structure proved in §3.3: all immediate predecessors of a fixed odd integer lie on a single infinite 01-tower generated by repeated application of $4x + 1$.

Definition (Immediate Predecessor Class)

For any odd integer n_t, once the smallest predecessor n_{t+1} is known, **all** immediate predecessors are given by

$$\boxed{\mathrm{Pred}_k(n_t) = 4^k\, n_{t+1} + \frac{4^k - 1}{3}, \qquad k = 0, 1, 2, \ldots}$$

and every predecessor arises uniquely in this way.

Reference implementations for generating $\mathrm{Pred}_k(n_t)$ are provided in Appendix D.Sec-3.3.

3.4 Recovering the Smallest Predecessor from Any Predecessor

In a previous section we showed that every odd predecessor of an odd integer n_t has the form

$$p_k \;=\; 4^k\, p_0 \;+\; \frac{4^k - 1}{3} \qquad k \geq 0.$$

where p_0 is the **smallest predecessor** of n_t. Equivalently, every predecessor is obtained by iterating the map $x \longmapsto 4x + 1$.

In this section we solve the *inverse* problem:

Given any predecessor p_k, how can we recover the smallest predecessor p_0?

The answer is completely visible in the **binary expansion** of p_k.

Example: The predecessors of 13

From §3.3, the predecessors of 13 are

$$17,\ 69,\ 277,\ 1109,\ 4437, \ldots$$

Their binary forms are:

```
17      : 0b0000000000000000010001
69      : 0b0000000000000001000101
277     : 0b0000000000000100010101
1109    : 0b0000000000010001010101
4437    : 0b0000000001000101010101
```

Every entry after the first ends with one or more copies of the binary block **01**. This exactly reflects the transformation $p \;\longmapsto\; 4p + 1$, since multiplying by 4 shifts left by two bits and adding 1 appends **01**. Thus **every non-minimal predecessor is obtained by repeatedly appending trailing 01-blocks**. To recover the smallest predecessor, we simply remove those appended blocks.

Two possible outcomes

Let p be an odd predecessor of n_t, and let us repeatedly remove trailing **01-blocks** from binary(p). Exactly two outcomes are possible:

Case A: Removing all trailing 01-blocks ends on an odd number

Example:

For $p = 10069$,

```
binary(p) = 0b ... 10011101010101
```

Removing all trailing 01-blocks gives

```
...100111
```

which is *odd*. This is the smallest predecessor. Thus:

$$10069 \longmapsto 39.$$

Case B: Removing all trailing 01-blocks ends on an even number

Example:

For $p = 1109$,

```
binary(p) = 0b ... 010001010101
```

Removing trailing 01-blocks yields

```
...0100
```

which is **even**. A predecessor must be **odd**, so we restore the **final** 01-block we removed:

```
...010001
```

which equals $\mathbf{17}$, the smallest predecessor. Thus:

$$1109 \longmapsto 17.$$

The Normalization Procedure

This gives a direct algorithm for recovering the smallest predecessor from *any* predecessor.

Algorithm (k0_normalize).

Input: any odd predecessor p.

Output: smallest predecessor of its successor.

1. Write p in binary.
2. Repeatedly remove trailing 01-blocks while possible.
3. If the resulting number ends in $\mathbf{1}$, return it.
4. If the resulting number ends in $\mathbf{0}$, append a 01-block and return.

This procedure gives p_0, the smallest predecessor, without ever computing the successor.

Why it works

Every predecessor has the closed form

$$p_k = 4^k\, p_0 + \frac{4^k - 1}{3}.$$

In binary this is

$$\text{binary}(p_k) = \text{binary}(p_0)\,\texttt{01}\,\texttt{01}\cdots\texttt{01}.$$

The block `01` is repeated k times.

Thus:

- removing trailing 01-blocks removes the factor 4^k,
- the remainder is exactly p_0, except when the remainder becomes even (meaning we removed one block too many),
- restoring one 01-block fixes that exceptional case.

Therefore **k0_normalize exactly inverts the map** $x \mapsto 4x + 1$ on the immediate predecessor class.

Relation to the Predecessor Formula

Recall the general predecessor parametrization:

$$P_k(n_t) = 4^k\, n_{t+1} + \frac{4^k - 1}{3}.$$

Given any $P_k(n_t)$, $k0\_normalize$ extracts the case $k = 0$, hence the name.

Conclusion

Given any odd predecessor p of an odd integer n_t, the smallest predecessor is obtained simply by inspecting the binary expansion of p. Removing trailing 01-blocks (with one correction in the even case) always recovers

$$p_0 = n_{t+1},$$

the smallest predecessor in the infinite predecessor class. This normalization procedure will play a central role in later chapters, where it becomes the foundation for the **canonical terminal seed** and the **predecessor-class structure**.

A reference implementation of k0_normalize is provided in Appendix D.Sec-3.4.

3.5 Closed-Form Recovery of the Smallest Predecessor

In Section 3.4 we showed that every odd predecessor p_k of an odd integer n_t lies on a unique 01-tower generated from the smallest predecessor p_0 by repeated application of the map $x \mapsto 4x + 1$, so that

$$p_k = 4^k p_0 + \frac{4^k - 1}{3}, \qquad k \geq 0.$$

Moreover, we demonstrated that the smallest predecessor p_0 can be recovered from p_k by inspecting the binary structure of p_k.

In this section we show that this structural recovery admits a **closed-form expression**. Specifically, we prove that p_0 is determined directly by the 2-adic valuation

$$\nu_2(3p_k + 1),$$

yielding an explicit inverse formula that eliminates the need for binary decomposition or iterative descent.

The Informal Algorithm

The binary structure of predecessors makes recovery of the smallest predecessor straightforward.

Given any odd predecessor p_k:

1. Write p_k in binary.
2. Repeatedly delete a trailing 01 block (equivalently, replace p with $(p-1)/4$ whenever $p \equiv 1 \pmod 4$). Let the final result be t.
3. If t is odd, return t.
4. If t is even, return $4t + 1$.

This recovers p_0. We now translate this algorithm into a **closed-form arithmetic formula**.

Step 1: Expressing p_k Through p_0

From (1),

$$p_k = 4^k p_0 + \frac{4^k - 1}{3}.$$

Compute the forward Collatz numerator:

$$3p_k + 1 = 3\left(4^k p_0 + \frac{4^k - 1}{3}\right) + 1 = 4^k(3p_0 + 1) = 2^{2k}(3p_0 + 1). \tag{2}$$

Taking ν_2 gives

$$\nu_2(3p_k + 1) = 2k + \nu_2(3p_0 + 1). \tag{3}$$

From (3), define

$$r = \nu_2(3p_k + 1).$$

Although the informal algorithm strips trailing 01-blocks from the **binary expansion of p_k**, the quantity

$$r = \nu_2(3p_k + 1)$$

provides an **algebraic measure** of how far p_k lies along its predecessor chain.
From (2),

$$3p_k + 1 = 2^{2k}(3p_0 + 1).$$

Thus every factor 2^2 in the numerator reflects one application of the map $x \mapsto 4x + 1$ inside the immediate predecessor class, and it is therefore convenient to define

$$a = \left\lfloor \frac{r}{2} \right\rfloor. \tag{4}$$

In this restricted setting (values of the form (1)), dividing $3p_k + 1$ by 2^{2a} in a single step reproduces exactly the value t that the iterative strip-01-block algorithm would reach.

Step 2: Stripping all 01-blocks

From (2),

$$3p_k + 1 = 2^{2k}(3p_0 + 1).$$

Solving for p_0 gives

$$3p_0 + 1 = \frac{3p_k + 1}{2^{2k}} \quad \Longrightarrow \quad p_0 = \frac{\frac{3p_k + 1}{2^{2k}} - 1}{3}.$$

Define

$$r = \nu_2(3p_k + 1), \qquad a = \left\lfloor \frac{r}{2} \right\rfloor,$$

and set

$$t = \frac{\frac{3p_k + 1}{2^{2a}} - 1}{3}. \tag{5}$$

Within a fixed predecessor class of the form (1), this valuation uniquely determines the number of effective 4-factors applied to p_0. However, t may become **even** if one block too many was removed.

Steps 3 & 4: Restoring One Block if Needed

The smallest predecessor p_0 is:

- $p_0 = t$ if t is odd,
- $p_0 = 4t + 1$ if t is even.

To eliminate the case distinction, use the selector

$$\frac{1 + (-1)^t}{2} = \begin{cases} 0, & t \text{ odd,} \\ 1, & t \text{ even.} \end{cases}$$

Thus the closed-form expression is

$$p_0 = t + \frac{1 + (-1)^t}{2}, (3t + 1). \tag{6}$$

Note that the correction term vanishes when t is odd. Together with (5), this yields a purely arithmetic inversion of the map $x \mapsto 4x + 1$.

Thus (5)–(6) together give an exact closed-form inversion of the predecessor map, even though the parameter a should be understood algebraically—via the valuation $\nu_2(3p_k + 1)$ — and not literally as the count of trailing 01-blocks in the binary expansion of an arbitrary integer.

Final Closed-Form Normalization Formula

For any predecessor p_k of any odd integer n_t:

$$\boxed{\begin{aligned} p_0 &= t + \frac{1 + (-1)^t}{2}(3t + 1), \\ t &= \frac{\frac{3p_k + 1}{2^{2a}} - 1}{3}, \\ a &= \left\lfloor \frac{\nu_2(3p_k + 1)}{2} \right\rfloor. \end{aligned}}$$

This implements the effect of removing trailing 01-blocks **purely in closed form**, with no iteration.

A reference implementation is given in Appendix D.Core.

Conclusion

The map $p \mapsto p_0$ that extracts the smallest predecessor from any predecessor is:

- **combinatorial** in binary (removal of trailing 01-blocks),
- **algebraic** in closed form (via ν_2),
- **structurally exact**, inverting the map $x \mapsto 4x + 1$ on immediate predecessor classes.

This normalization is a key tool for the canonical classification of predecessor chains developed in later chapters.

3.6 Smallest Predecessors as Class Labels

We now use the **smallest predecessor** to label natural equivalence classes of odd integers under the accelerated Collatz map. We begin with two illustrative examples, the odd integers $n = 155$ and $n = 955$. From the predecessor formula (or by direct computation), we obtain the following immediate predecessors.

```
predecessor_list(155) = [103, 413, 1653, 6613, 26453, 105813, 423253, …]

103      x = 1      0b000000000000000000000001100111
413      x = 3      0b000000000000000000000110011101
1653     x = 5      0b000000000000000000011001110101
6613     x = 7      0b000000000000000001100111010101
26453    x = 9      0b000000000000000110011101010101
105813   x = 11     0b000000000000011001110101010101
423253   x = 13     0b000000000001100111010101010101
```

and

```
predecessor_list(955) = [1273, 5093, 20373, 81493, 325973, 1303893, 5215573, …]

1273     x = 2      0b000000000000000000010011111001
5093     x = 4      0b000000000000000001001111100101
20373    x = 6      0b000000000000000100111110010101
81493    x = 8      0b000000000000010011111001010101
325973   x = 10     0b000000000001001111100101010101
1303893  x = 12     0b000000000100111110010101010101
5215573  x = 14     0b000000010011111001010101010101
```

In each table, every listed odd integer p satisfies

$$\frac{3p+1}{2^x} = n$$

for the indicated exponent $x \geq 1$ and the fixed successor $n \in \{155, 955\}$. Equivalently, each listed value is an **immediate predecessor** of the same odd integer n under the accelerated Collatz map T.

For example, for

$$\text{predecessor\_list}(955) = \{1273, 5093, 20373, 81493, 325973, \dots\},$$

every element maps to 955 in a single accelerated step:

$$T(1273) = 955, \quad T(5093) = 955, \quad T(20373) = 955, \ \dots$$

Among these, the **smallest predecessor** of 955 is

$$p_{\min}(955) = \min\{1273, 5093, 20373, 81493, 325973, \dots\} = 1273.$$

Similarly, for $n = 155$, the smallest predecessor is 103.

At this point, two key tools are already available:

- a closed-form formula that recovers the smallest predecessor p_0 of any odd integer n, and
- a normalization procedure (denoted $k0\_normalize$) which, when applied to any member of a predecessor class of the form

$$p_k = 4^k p_0 + \frac{4^k - 1}{3},$$

algebraically undoes the embedded $x \mapsto 4x + 1$ steps and recovers the canonical element m_0.

These observations suggest a natural classification scheme: for each odd integer n, all of its immediate predecessors form a structured infinite predecessor class, and the **smallest predecessor** provides a natural canonical label for that class. We now formalize this idea.

Predecessor classes

Fix an odd integer n. We define its **predecessor class** as

$$\mathcal{P}(n) = \{ p \in \mathbb{Z}_{\text{odd}} : T(p) = n \}.$$

If $\mathcal{P}(n)$ is nonempty, we define its smallest element

$$p_{\min}(n) := \min \mathcal{P}(n),$$

and use this value as the **class label**. Although $\mathcal{P}(n)$ is indexed by its successor n, we will also refer to the same set as $\mathcal{P}_{p_{\min}(n)}$ when emphasizing its role as a predecessor class labeled by its canonical base.

For example:

- For $n = 155$, we have $p_{\min}(155) = 103$, and we write

  $$\mathcal{P}_{103} := \mathcal{P}(155),$$

  meaning "the predecessor class whose base is 103."

- For $n = 955$, we have $p_{\min}(955) = 1273$, and we write

  $$\mathcal{P}_{1273} := \mathcal{P}(955).$$

In later sections, this labeling scheme will be extended from **immediate** predecessor sets to **entire backward chains**, and ultimately to global predecessor families that organize all odd integers into canonical classes.

A reference implementation for obtaining the class labels and members is provided in Appendix D.Sec-3.6.

3.7 The Special Predecessor Class $\mathcal{P}_1$

For the accelerated successor function T, defined by

$$T(p) = \frac{3p + 1}{2^x},$$

the number 1 is exceptional:

> **It is its own successor, and it generates the simplest possible predecessor class.**

Computing:

```
predecessor_list(1) = [1, 5, 21, 85, 341, 1365, 5461, 21845, 87381, 349525, …]
```

These values satisfy the closed form

$$p_k = 4^k \cdot 1 \; + \; \frac{4^k - 1}{3} = \frac{4^{k+1} - 1}{3}, \qquad k = 0, 1, 2, \dots$$

and their binary expansions display a perfect repeated-01 pattern:

p_k	x	binary
1	2	0b…01
5	4	0b…0101
21	6	0b…010101
85	8	0b…01010101
341	10	0b…0101010101
1365	12	0b…010101010101
5461	14	0b…01010101010101
…	…	…

These binary expansions form an infinitely ascending 01-tower, where applying $4x + 1$ appends one additional trailing 01-block. Every one of these satisfies

$$T(p_k) = 1.$$

Thus $\mathcal{P}_1$ is visibly the “purest” predecessor class.

The diagram below shows the beginning of this 01-tower, illustrating the repeated application of 4x+1 and the resulting trailing-01 binary structure.

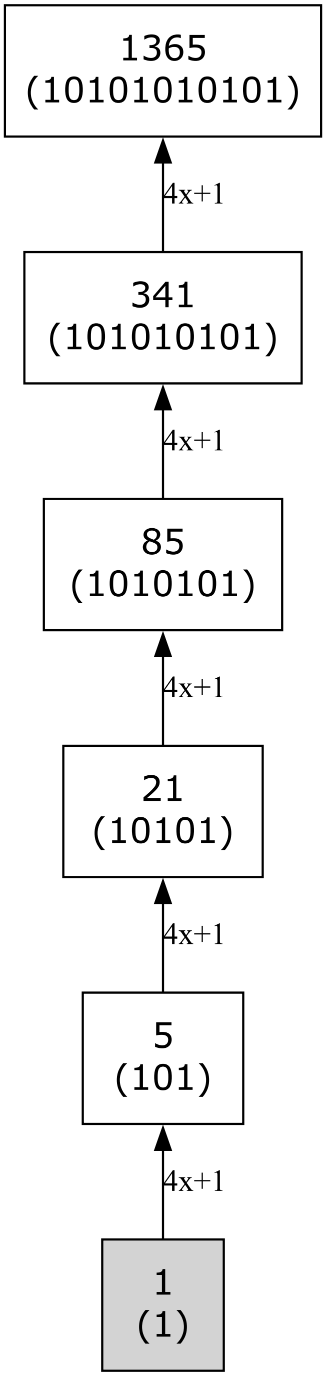

Figure 3.2. *The 01-tower structure of the special predecessor class $\mathcal{P}_1$.* Each element is obtained from the one below by the map $x \mapsto 4x + 1$, appending another trailing 01 in binary. Every number in the tower has accelerated successor $T(p) = 1$.

Why is $\mathcal{P}_1$ Special?

We now prove three structural properties that make $\mathcal{P}_1$ unique among all predecessor classes.

1. $\mathcal{P}_1$ is the unique class whose smallest predecessor is 1.

Recall the predecessor class definition:

$$\mathcal{P}(n) = \{p \in \mathbb{Z}_{\text{odd}} : T(p) = n\}.$$

Define the smallest predecessor

$$p_{\min}(n) := \min \mathcal{P}(n),$$

and use it as the class label:

$$\mathcal{P}_{p_{\min}(n)} := \mathcal{P}(n).$$

Step A: Show 1 is the smallest predecessor of itself. Compute its successor:

$$3 \cdot 1 + 1 = 4, \qquad \nu_2(4) = 2,$$

so

$$T(1) = \frac{4}{2^2} = 1.$$

Thus $1 \in \mathcal{P}(1)$, and clearly no smaller odd integer exists; hence

$$p_{\min}(1) = 1.$$

Step B: Show no other n has 1 as a predecessor. Assume for contradiction that $1 \in \mathcal{P}(n)$ for some $n \neq 1$. Then

$$T(1) = n,$$

but we already know $T(1) = 1$. Thus $n = 1$, contradiction.

Conclusion

$\mathcal{P}_1$ is the *only* predecessor class whose smallest predecessor is 1.

2. $\mathcal{P}_1$ is the unique class whose elements are exactly $\frac{4^{k+1}-1}{3}$.

Start from the predecessor equation for 1:

$$\frac{3p+1}{2^x} = 1 \quad \Leftrightarrow \quad 3p + 1 = 2^x.$$

Solving:

$$p = \frac{2^x - 1}{3}. \tag{1}$$

Integrality condition Since $2 \equiv -1 \pmod 3$,

$$2^x \equiv \begin{cases} 2 \pmod 3, & x \text{ odd}, \\ 1 \pmod 3, & x \text{ even}, \end{cases}$$

so $3 \mid (2^x - 1)$ **iff x is even**.

Write $x = 2(k+1)$:

$$p = \frac{4^{k+1} - 1}{3}.$$

Therefore All predecessors of 1 are exactly

$$\mathcal{P}_1 = \left\{ \frac{4^{k+1} - 1}{3} : k = 0, 1, 2, \dots \right\}.$$

Uniqueness

If any other class had exactly these elements, its smallest member would be

$$\frac{4^1 - 1}{3} = 1,$$

hence it would be $\mathcal{P}_1$ itself. Thus $\mathcal{P}_1$ **is uniquely characterized by this geometric sequence.**

3. $\mathcal{P}_1$ is the only predecessor class whose smallest predecessor is also its own successor.

We solve the fixed-point equation $T(n) = n$, using the definition:

$$\frac{3n+1}{2^{\nu_2(3n+1)}} = n$$

Multiply:

$$3n + 1 = n \cdot 2^r \quad (r := \nu_2(3n+1))$$

Rearrange:

$$1 = (2^r - 3)n \quad \Longrightarrow \quad n = \frac{1}{2^r - 3}.$$

For n to be a **positive odd integer**, we need:

- $2^r - 3 > 0$
- $(2^r - 3) \mid 1$

List values:

- $r = 1$: $2^1 - 3 = -1$
- $r = 2$: $2^2 - 3 = 1$
- $r \geq 3$: denominator > 1 and cannot divide 1

Hence the only possibility is

$$r = 2, \qquad n = \frac{1}{1} = 1.$$

Thus $T(1) = 1$ and **no other odd integer is a fixed point** of the accelerated map.

Conclusion

The predecessor class $\mathcal{P}_1$ is unique in all three senses:

1. It is the only class whose smallest predecessor is 1.
2. It is the only class whose elements are exactly $\frac{4^{k+1}-1}{3}$.
3. It is the only class whose smallest predecessor is also its own successor.

Among all predecessor classes, $\mathcal{P}_1$ is therefore the minimal and structurally simplest class.

3.8 Chapter 3 Summary: Immediate Predecessors

Chapter 3 develops the structure of **immediate predecessors** under the accelerated Collatz map and shows that, although an odd integer may admit infinitely many predecessors, these values fall into a remarkably rigid and computable pattern.

The main conclusions are:

Modular Criterion for Existence

From the predecessor equation

$$n_{t+1} = \frac{2^x n_t - 1}{3},$$

immediate predecessors exist precisely **if and only if**

$$2^x n_t \equiv 1 \pmod 3.$$

This places each odd integer n_t into one of three modular regimes:

- predecessors exist for **all odd exponents** x,
- predecessors exist for **all even exponents** x,
- or **no immediate predecessors exist**.

Thus the existence and structure of immediate predecessors is dictated entirely by a simple mod-3 rule.

Closed-Form Description of All Predecessors

When immediate predecessors exist, there is a **smallest predecessor** n_{t+1}, and *every* other predecessor is obtained by iterating $p \mapsto 4p + 1$. This yields the closed-form class

$$\mathrm{Pred}_k(n_t) = 4^k\, n_{t+1} + \frac{4^k - 1}{3}, \qquad k \geq 0.$$

Thus, whenever predecessors exist, they form a **single infinite algebraic sequence** generated by repeated application of $4x + 1$.

Binary 01-Towers

Because the map $4p + 1$ appends a trailing **01-block** in binary, each predecessor class forms an infinite **01-tower**:

- the smallest predecessor n_{t+1} is a finite binary core,
- higher predecessors are obtained by successively appending 01-blocks.

Along each tower, the exponents $\nu_2(3p + 1)$ increase by exactly two at every step, producing simple arithmetic progressions.

The Exceptional Class $\mathcal{P}_1$

For $n_t = 1$, the predecessor class is

$$p_k = \frac{4^{k+1} - 1}{3}, \qquad k \geq 0,$$

forming the purest possible 01-tower. Since $T(1) = 1$, the integer 1 is the unique odd fixed point of the accelerated Collatz map. This identifies $\mathcal{P}_1$ as the **root predecessor class**, a role it plays in later chapters.

Summary

Chapter 3 shows that immediate predecessors of an odd integer are not chaotic. When they exist, they form a single, completely described **01-tower**, determined by a modular condition and generated by the linear map $x \mapsto 4x + 1$.

This algebraic and binary structure provides the foundation for Chapter 4's global theory of predecessor chains and Chapter 5's symbolic encoding framework.

Chapter 4

Predecessor Chains

4.1 Overview

A **predecessor chain** for an odd integer n is formed by repeatedly applying the *smallest-predecessor map* until the iteration terminates. At each step one of three things could conceivably occur:

1. the next value has **no predecessor**, so the chain returns -1;
2. the values grow without bound;
3. the sequence enters a **cycle**.

In this chapter we will show that only the first case actually occurs: **every predecessor chain of every odd integer $n \geq 3$ eventually reaches -1.**

The predecessor chain rule

Starting from any odd integer $n_0 \geq 3$, we define

$$n_{t+1} = \left\lfloor \frac{\big((n_t - 3) \bmod 6\big)\, n_t - 1}{3} \right\rfloor.$$

This is exactly the formula derived earlier for the **smallest predecessor**. Iterating it produces the predecessor chain:

$$n_0 \longrightarrow n_1 \longrightarrow n_2 \longrightarrow \cdots$$

Because every odd integer can have *infinitely many* predecessors, this chain specifically follows the **smallest** one at each step. Nevertheless, every predecessor—large or small—maps to the same successor under the accelerated Collatz update, so following the smallest predecessor does not lose structural information: it extracts the *canonical representative* of each predecessor class. Although individual predecessor values can grow arbitrarily large within a class, we will show that the class labels themselves cannot do so indefinitely.

Predecessor classes and why chains are "chains of classes"

From earlier sections:

- All immediate predecessors of an odd integer n form an infinite family $\mathcal{P}(n)$.

- The **smallest** member $p_{\min}(n)$ serves as the **class label**.
- Thus we write

$$\mathcal{P}_{p_{\min}(n)} := \mathcal{P}(n).$$

For example, since 1273 is the smallest predecessor of 955,

$$\mathcal{P}_{1273} = \mathcal{P}(955).$$

Each step of the predecessor chain

$$n_t \mapsto n_{t+1}$$

therefore moves from one predecessor class label to the next. A predecessor chain is therefore really a **chain of predecessor classes**.

Examples

```
n = 103
predecessor(103) = 137
predecessor_chain(103) = [137, 91, 121, 161, 107, 71, 47, 31, 41, 27, -1]

n = 5125
predecessor(5125) = 6833
predecessor_chain(5125) = [6833, 4555, 6073, 8097, -1]

n = 15121
predecessor(15121) = 20161
predecessor_chain(15121) = [20161, 26881, 35841, -1]
```

Every example ends in -1. This happens precisely when the last odd integer in the chain has **no predecessor**. The central goal of the next sections is to prove that *this always happens*:

> **Every predecessor chain for every odd integer $n \geq 3$ terminates at -1.**

That is, sooner or later the chain reaches a predecessor class that has **no predecessors of its own**.

4.2 Intuition for Termination at −1 in Odd Predecessor Chains

Our goal in this chapter is to prove the following fundamental fact:

> **Every predecessor chain on odd integers $n \geq 3$ terminates at -1 and admits no cycles.**

Before giving the formal proof, we first develop the intuition for why termination is inevitable.

The update rule

For any odd n_t, the smallest predecessor is

$$n_{t+1} = \left\lfloor \frac{((n_t - 3) \bmod 6)\, n_t - 1}{3} \right\rfloor.$$

Since odd integers satisfy

$$n_t \equiv 1, 3, 5 \pmod 6,$$

the quantity $(n_t - 3) \bmod 6$ equals $4, 0, 2$ respectively. Thus the predecessor recursion splits cleanly into three cases:

$$n_{t+1} = \begin{cases} \dfrac{4n_t - 1}{3}, & n_t \equiv 1 \pmod 6, \\ -1, & n_t \equiv 3 \pmod 6, \\ \dfrac{2n_t - 1}{3}, & n_t \equiv 5 \pmod 6. \end{cases} \tag{$\star$}$$

The floor matters only in the middle case: $n_t \equiv 3 \pmod 6$ immediately forces $n_{t+1} = -1$. This is the backbone of the entire termination argument.

The "playing field": the three odd residue classes mod 6

Every odd integer belongs to exactly one of:

- **Type A:** $n \equiv 1 \pmod 6$
- **Type B:** $n \equiv 3 \pmod 6$
- **Type C:** $n \equiv 5 \pmod 6$

In base-6 language, these are numbers ending with digits **1, 3, 5**. The update rule ($\star$) behaves very differently on each type, and understanding those transitions is the heart of the argument. The big picture we are working toward is:

> Types A and C *cannot avoid* eventually falling into Type B, and Type B maps directly to -1.

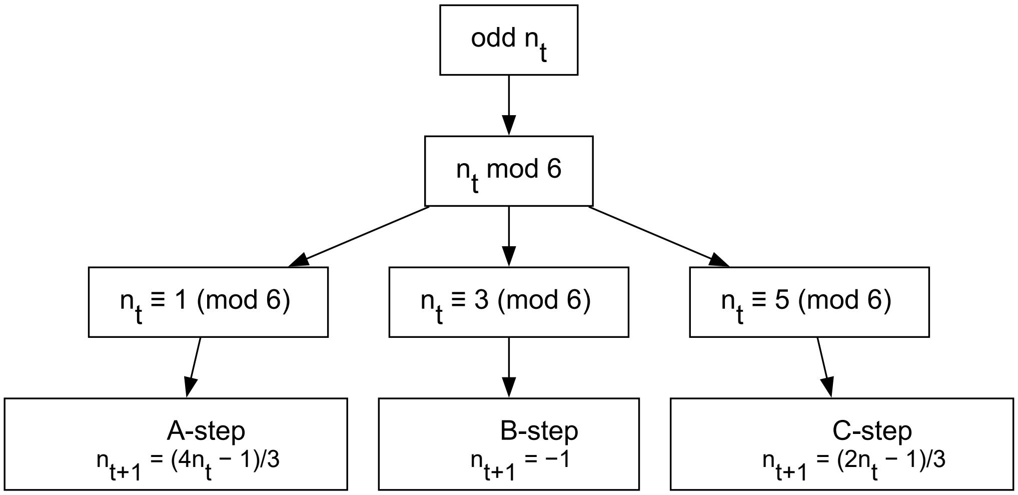

Figure 4.1. *Decision tree for the three predecessor cases.* The residue class of an odd integer n_t modulo 6 determines which inverse rule applies.

Type B is the exit door

If $n_t \equiv 3 \pmod 6$, then $(\star)$ immediately gives

$$n_{t+1} = -1.$$

This is the absorbing terminal value. Thus, to prove termination, we must show that **Types A and C are forced toward Type B after finitely many steps**.

Type C (5 mod 6): always shrinking

Write a Type-C integer as

$$n = 6m + 5.$$

Under $(\star)$,

$$n' = \frac{2n-1}{3} = \frac{12m+9}{3} = 4m + 3.$$

This has two crucial consequences:

1. **n' is always strictly smaller than n** for $n \geq 5$.
2. Whenever n' is again of the form $6m' + 5$ (i.e. we remain in Type C), the new m' is **strictly smaller** than the previous m.

Thus Type-C steps that stay in Type C strictly decrease the "base-6 part" m.

Therefore:

- Type C cannot continue forever — m cannot decrease below 0.

- When Type C finally ends, the next state must be either:
  - directly Type B (which yields -1), or
  - Type A (which we analyze next).

Type A (1 mod 6): growth with a hidden timer

Write a Type-A integer as

$$n = 6m + 1.$$

Then ($\star$) gives

$$n' = \frac{4n - 1}{3} = \frac{24m + 3}{3} = 8m + 1.$$

Rewrite this again in base-6:

$$8m + 1 = 6\left(\frac{4}{3}m\right) + 1.$$

So we remain in Type A **if and only if** $3 \mid m$, in which case

$$m' = \frac{4}{3}m.$$

This looks like growth — and indeed m' may be larger than m — but it has a hidden constraint.

The hidden countdown timer: powers of 3 in m

Let $v_3(m)$ be the 3-adic valuation of m (number of factors of 3 in m). Because 4 contains no factor of 3, replacing m with $\frac{4}{3}m$ removes *exactly one* factor of 3:

$$v_3(m') = v_3(m) - 1.$$

Thus:

- Type-A steps that remain Type A always **reduce $v_3(m)$ by 1**.
- After at most $v_3(m)$ such steps, we reach an m not divisible by 3.
- At that moment we are kicked **out** of Type A, into either Type B or Type C.

Thus, type A cannot persist indefinitely — it is controlled by a finite clock.

Global mechanism: a descent invariant

Across the entire process, the base-6 parameter m (defined by $n = 6m + r$ for $r \in \{1, 3, 5\}$) behaves monotonically in the long run:

1. **Type C:** m *strictly decreases every step* (while we remain in Type C).

2. **Type A:** after finitely many steps, m loses all powers of 3 and exits Type A. The exit either:

   - lands directly in Type B, ending at -1, or
   - moves into Type C with a **strictly smaller** m.

Thus, although n_t may grow or shrink erratically, the hidden parameter m has a monotone descent property:

> **m cannot increase infinitely often; it must eventually drop low enough to force a Type-B hit.**

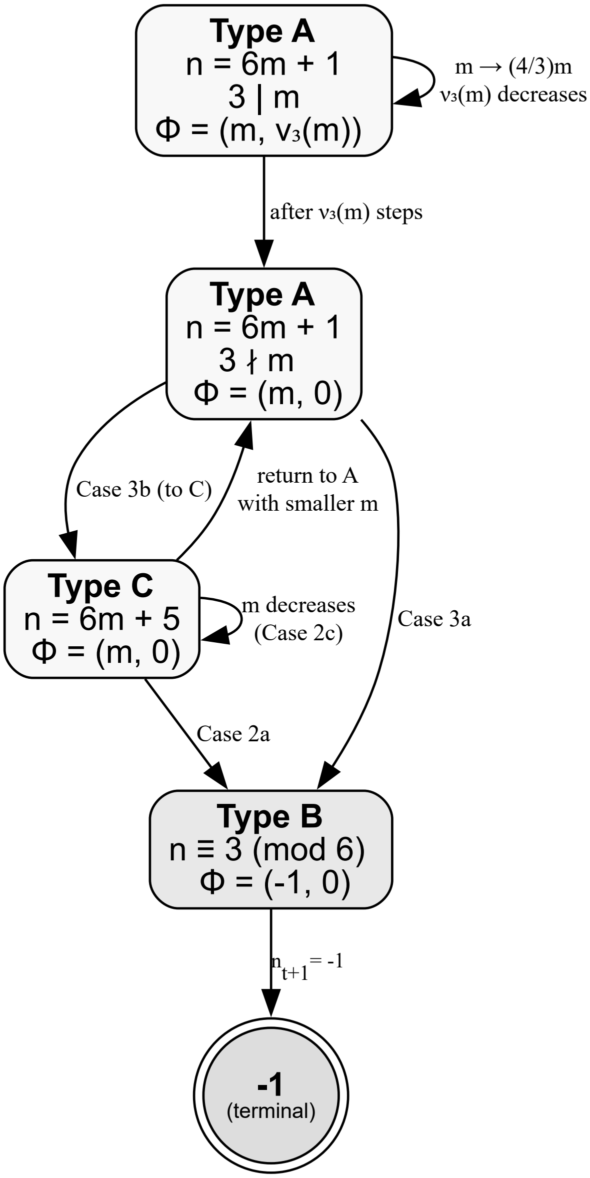

Figure 4.2. *Phase descent invariant for odd predecessor chains.* Although individual predecessor steps may increase or decrease the numerical value of n_t, the underlying base-6 parameter m (defined by $n_t = 6m + r$ with $r \in \{1, 3, 5\}$) exhibits a strictly descending structure.

Type-C transitions decrease m directly, while Type-A transitions are governed by a finite $\nu_3(m)$ timer that must eventually expire, forcing an exit to either Type B or Type C with smaller m. Type B acts as an absorbing terminal state.

This invariant descent in m prevents recurrence and guarantees termination of every odd predecessor chain at -1.

Why cycles are impossible

A cycle would require returning to a previously visited value of m. But:

- Type-C steps strictly decrease m.
- Type-A stretches terminate by exhausting $\nu_3(m)$, after which the next step produces a strictly smaller m.

Thus m never repeats, hence: **No predecessor chain can cycle.**

The big picture

The intuitive termination mechanism is:

1. Partition odd integers into the three mod-6 types A, B, C.
2. **Type B** is an immediate exit to -1.
3. **Type C** strictly decreases the internal parameter m; it cannot persist.
4. **Type A** has a finite internal timer (powers of 3 in m) and must eventually leave.
5. All paths eventually funnel into Type B.

Thus: **Every odd predecessor chain is acyclic and ends at -1.**.

4.3 Proof of Termination at - 1 and Acyclicity of Odd Predecessor Chains

Claim. For any odd $n_t \geq 3$, the iteration

$$n_{t+1} = \left\lfloor \frac{\big((n_t - 3) \bmod 6\big)\, n_t - 1}{3} \right\rfloor$$

eventually reaches -1.

Odd numbers satisfy $n_t \equiv 1, 3, 5 \pmod 6$, so $(n_t - 3) \bmod 6 = 4, 0, 2$ respectively. The **floor only matters in the $n_t \equiv 3 \pmod 6$ branch** (where it turns $-\frac{1}{3}$ into -1); in the other two branches the quotient is already integral. Thus the update is

$$n_{t+1} = \begin{cases} \dfrac{4n_t - 1}{3}, & n_t \equiv 1 \pmod 6, \\ -1, & n_t \equiv 3 \pmod 6, \\ \dfrac{2n_t - 1}{3}, & n_t \equiv 5 \pmod 6. \end{cases} \qquad (\star)$$

In the $r = 1$ case, $4(6m + 1) - 1 = 24m + 3$ is divisible by 3, and in the $r = 5$ case, $2(6m + 5) - 1 = 12m + 9$ is divisible by 3. Write each odd n_t as $n_t = 6m + r$ with $r \in \{1, 3, 5\}$.

Relation to the A/B/C classification

The decomposition $n_t = 6m + r$ with $r \in \{1, 3, 5\}$ corresponds exactly to the residue–class types introduced in §4.2:

- $r = 1$ corresponds to **Type A**,
- $r = 3$ corresponds to **Type B**,
- $r = 5$ corresponds to **Type C**.

In what follows, we analyze the iteration case-by-case according to the value of r. This is the same classification as before, now expressed algebraically through the parameter m.

Case 1: $r = 3$ ($n_t \equiv 3 \pmod 6$)

Immediate termination: by ($\star$) we get $n_{t+1} = -1$.

Case 2: $r = 5$ ($n_t = 6m + 5$)

From ($\star$),

$$n_{t+1} = \frac{2(6m + 5) - 1}{3} = 4m + 3.$$

Classify by m mod 3, and rewrite n_{t+1} uniquely as $6m' + r'$ with $r' \in \{1, 3, 5\}$.

Case 2a: $m \equiv 0 \pmod 3$.

$m = 3k \Rightarrow n_{t+1} = 12k + 3 = 6(2k) + 3 \equiv 3 \pmod 6$, so the next step gives -1.

Case 2b: $m \equiv 1 \pmod 3$.

$m = 3k + 1 \Rightarrow n_{t+1} = 12k + 7 = 6(2k + 1) + 1$, i.e. we move to $r = 1$ with

$$m' = \frac{2m + 1}{3} = 2k + 1 \leq m$$

(since $\frac{2m+1}{3} \leq m$ iff $m \geq 1$), and in fact $m' < m$ whenever $k \geq 1$ (i.e. $m \geq 4$). The small edge case $m = 1$ corresponds to $n_t = 11$ with path $11 \to 7 \to 9 \to -1$.

Case 2c: $m \equiv 2 \pmod 3$.

$m = 3k + 2 \Rightarrow n_{t+1} = 12k + 11 = 6(2k + 1) + 5$, so we **stay** in $r = 5$ with $m' = \frac{2m-1}{3} = 2k + 1 < m$, since $m - m' = k + 1 \geq 1$.

Conclusion of Case 2.

Starting in $r = 5$,

- either we land in $r = 3$ next (and stop),
- or we move to $r = 1$ with $m' \leq m$ and in fact $m' < m$ except for the edge case $m = 1$,
- or we remain in $r = 5$ with a **strictly smaller** m.

Hence repeated visits to $r = 5$ strictly decrease m until we hit $r = 3$ and stop.

Case 3: $r = 1$ $(n_t = 6m + 1)$

From ($\star$),

$$n_{t+1} = \frac{4(6m+1)-1}{3} = 8m + 1.$$

Again split by m mod 3 and rewrite n_{t+1} uniquely as $6m' + r'$ with $r' \in \{1, 3, 5\}$.

Case 3a: $m \equiv 1 \pmod 3$**.**

$m = 3k + 1 \Rightarrow n_{t+1} = 24k + 9 = 6(4k + 1) + 3 \equiv 3 \pmod 6$, so the **next** step is -1.

Case 3b: $m \equiv 2 \pmod 3$**.**

$m = 3k + 2 \Rightarrow n_{t+1} = 24k + 17 = 6(4k + 2) + 5$, so we move to $r = 5$ with

$$m_1 = \frac{4m-2}{3} = 4k + 2.$$

This m_1 can exceed m (indeed $m_1 - m = k \geq 0$), but **within at most one subsequent** $r = 5$ **update, the parameter drops below the original** m. If $m_1 \not\equiv 0 \pmod 3$, the next step is an $r = 5$ update, and Case 2b/2c applies depending on m_1 mod 3. Applying Case 2 once to m_1:

- either
$$m_1 \equiv 0 \pmod 3 \Rightarrow \text{ immediate } r = 3 \Rightarrow -1,$$
- or else a single $r = 5$ update gives
$$m_2 = \begin{cases} \dfrac{2m_1 - 1}{3} = \dfrac{8m-7}{9}, & m_1 \equiv 2 \pmod 3, \\ \dfrac{2m_1 + 1}{3} = \dfrac{8m-1}{9}, & m_1 \equiv 1 \pmod 3, \end{cases}.$$

and in either subcase $m_2 < m$ because $8m - 7 < 9m$ and $8m - 1 < 9m$ for all $m \geq 1$.

(Integrality note: $m_1 \equiv 2 \pmod 3$ *corresponds to* $m \equiv 2 \pmod 9$*, yielding* $9 \mid (8m-7)$*;* $m_1 \equiv 1 \pmod 3$ *corresponds to* $m \equiv 8 \pmod 9$*, yielding* $9 \mid (8m - 1)$*.)*

Case 3c: $m \equiv 0 \pmod 3$**.**

Write $m = 3^e s$ with $3 \nmid s$. Then

$$n_{t+1} = 8m + 1 = 8 \cdot 3^e s + 1 = 6\big(4 \cdot 3^{e-1} s\big) + 1,$$

so we **stay** in $r = 1$ and send $m \mapsto m' = \dfrac{4}{3}m$. This may increase $|m|$, but it **reduces the 3-adic valuation by exactly 1**:

$$\nu_3(m') = \nu_3\left(\frac{4}{3}m\right) = \nu_3(m) - 1.$$

Since $m \equiv 0 \pmod 3$, we have $8m + 1 \equiv 1 \pmod 6$, so the iteration remains in the $r = 1$ branch.

After exactly e such $r = 1$ steps we reach $3 \nmid m$, i.e. (3a) or (3b).

- In (3a) we stop immediately;
- In (3b), within at most two further steps via $r = 5$, we obtain a parameter **strictly smaller** than the m we had when entering this $r = 1$ phase. In (3b), counting steps precisely, it's one step to go $r = 1 \to r = 5$, and then **at most one** additional $r = 5$ update to get $m_2 < m$ (or immediate $r = 3$).

Global termination via a phase-descent invariant

To unify the case analysis, it is useful to view the dynamics in **phases**. An **$r = 1$ phase** is a maximal consecutive block of steps during which the iterate remains in the class $r = 1$ (equivalently, while $3 \mid m$). Such a phase ends either at termination (Case 3a) or when the process exits $r = 1$ into $r = 3$ or $r = 5$. During such a phase:

- If $3 \mid m$, then each $r = 1$ step reduces $\nu_3(m)$ by exactly 1, so only finitely many $r = 1$ steps can occur before we reach a state with $3 \nmid m$.
- Once we reach a state $6m + 1$ with $3 \nmid m$, we are in Case 3a or Case 3b.
  - In **Case 3a** we terminate immediately.
  - In **Case 3b**, we pass through at most two subsequent $r = 5$ steps and then return to $r = 1$ with a parameter **strictly smaller than the m at that Case 3b entry**.

Between these phases, any time we are in $r = 5$ (Case 2), each nonterminal step **strictly decreases** the parameter m. Thus $r = 5$ segments cannot continue indefinitely. Consequently, the process cannot sustain infinitely many transitions:

- every time we enter Case 3b, within at most two steps (the forced $r = 1 \to r = 5$ step and then at most one $r = 5$ update), the process returns to either termination or to an $r = 1$ state with a strictly smaller m than at the Case 3b entry,
- and every move through $r = 5$ also strictly decreases m. Since m is a nonnegative integer, only finitely many such decreases are possible.

Therefore the iteration must eventually reach the class $r = 3$, after which the next value is -1. This completes the proof that every odd predecessor chain is finite and acyclic.

4.4 Phase-Descent Argument and Acyclicity

To package the descent mechanism into a single monotone potential, define a lexicographically ordered invariant $\Phi(n)$ as follows:

$$\Phi(n) = \begin{cases} (m,\, \nu_3(m)) & \text{if } n = 6m + 1, \\ (m,\, 0) & \text{if } n = 6m + 5, \\ (-1,\, 0) & \text{if } n \equiv 3 \pmod 6, \end{cases}$$

ordered lexicographically: $(a, b) < (a', b')$ if either $a < a'$ or $a = a'$ and $b < b'$.

The behavior of Φ under the predecessor iteration is:

- In the case $r = 1$ with $3 \mid m$, one step sends $m \mapsto \frac{4}{3}m$, which may increase m but decreases $\nu_3(m)$ by exactly 1. Thus the second coordinate of Φ strictly decreases.
- In the case $r = 1$ with $3 \nmid m$, within at most two steps the process either terminates or returns to $r = 1$ with a strictly smaller value of m, so the first coordinate strictly decreases.
- In the case $r = 5$, every nonterminal step strictly decreases m, so the first coordinate strictly decreases.
- In the case $r = 3$, the iteration moves immediately to -1, terminating the process.

Therefore, at every nonterminal stage of the iteration, the invariant $\Phi(n)$ strictly decreases in lexicographic order. Since there is no infinite descending sequence in $\mathbb{N} \times \mathbb{N}$ under lexicographic order, the predecessor chain must terminate at -1.

Corollary (No cycles among odd $n \geq 3$): If a nontrivial cycle existed among odd integers $n \geq 3$, traversing one full cycle would strictly decrease Φ at each step, forcing the initial state to return with a strictly smaller value of Φ than itself — an impossibility. Hence the directed dynamics on odd integers $n \geq 3$ are acyclic, with a unique absorbing terminal state at -1. (For completeness: $n = 1$ is a fixed point $1 \mapsto 1$, outside the domain considered here.)

4.5 Worked Examples

We now illustrate the mechanisms established in §§4.2–4.4. Recall the three congruence classes and their roles:

Congruence	Type	Case	Behavior
$n \equiv 1 \pmod 6$	Type A	Case 3	Growth with a hidden countdown timer $\nu_3(m)$
$n \equiv 3 \pmod 6$	Type B	Case 1	Immediate exit to -1
$n \equiv 5 \pmod 6$	Type C	Case 2	Always shrinking in the base-6 parameter m

In each example below we track:

- the residue type $(1, 3, 5 \text{ mod } 6)$,
- the base-6 parameter m (with $n = 6m + r$),
- the hidden timer $\nu_3(m)$ relevant only in Type A, and
- the movement through Types A, B, C predicted by the proof.

Example 1: A long Type-A stretch with a high 3-adic timer

Start: $n_0 = 4375$

This example shows a prolonged Type-A phase where the hidden timer $\nu_3(m)$ begins large and ticks down step by step before finally exiting to Type B. The predecessor chain is:

$$4375 \to 5833 \to 7777 \to 10369 \to 13825 \to 18433 \to 24577 \to 32769 \to -1.$$

We track the internal parameter $m = \frac{n_t - 1}{6}$, its factorization, and the 3-adic timer $\nu_3(m)$.

$$\begin{array}{lll}
t=0: & 4375 & \text{A},\ m=729,\ \nu_3(m)=6 \\
t=1: & 5833 & \text{A},\ m=972,\ \nu_3(m)=5 \\
t=2: & 7777 & \text{A},\ m=1296,\ \nu_3(m)=4 \\
t=3: & 10369 & \text{A},\ m=1728,\ \nu_3(m)=3 \\
t=4: & 13825 & \text{A},\ m=2304,\ \nu_3(m)=2 \\
t=5: & 18433 & \text{A},\ m=3072,\ \nu_3(m)=1 \\
t=6: & 24577 & \text{A},\ m=4096,\ \nu_3(m)=0 \\
t=7: & 32769 & \text{B} \\
t=8: & -1 &
\end{array}$$

Detailed table

t	n_t	Type	$m = \frac{n_t-1}{6}$	Factorization of m	Timer $\nu_3(m)$	Calculation	n_{t+1}
0	4375	A	729	3^6	6	$\frac{4\cdot 4375-1}{3}$	5833
1	5833	A	972	$2^2 \cdot 3^5$	5	$\frac{4\cdot 5833-1}{3}$	7777
2	7777	A	1296	$2^4 \cdot 3^4$	4	$\frac{4\cdot 7777-1}{3}$	10369
3	10369	A	1728	$2^6 \cdot 3^3$	3	$\frac{4\cdot 10369-1}{3}$	13825
4	13825	A	2304	$2^8 \cdot 3^2$	2	$\frac{4\cdot 13825-1}{3}$	18433
5	18433	A	3072	$2^{10} \cdot 3$	1	$\frac{4\cdot 18433-1}{3}$	24577
6	24577	A	4096	2^{12}	0	$\frac{4\cdot 24577-1}{3}$	32769
7	32769	B	—	—	—	$\frac{0\cdot 32769-1}{3} = -\frac{1}{3}$	−1
8	−1	—	—	—	—	—	—

What this example illustrates

- A single long Type-A phase with large initial m.
- Each Type-A step reduces $\nu_3(m)$ by exactly 1.
- When the timer reaches 0 (at $m = 4096$), the next step must exit Type A.
- It lands directly in Type B, which sends the chain to -1.

This is a clean example of the "hidden-timer" mechanism discussed in previous sections.

Example 2: A short mixed sequence

Start: $n_0 = 37$

This example shows one charged Type-A step (timer $= 1$), followed by an immediate exit from Type A.

$$\begin{aligned}
t = 0: &\quad 37 = 6\cdot 6 + 1 && \text{Type A, } m = 6,\ \nu_3(m) = 1\\
t = 1: &\quad 49 && \text{Type A, } m = 8,\ \nu_3(m) = 0\\
t = 2: &\quad 65 && \text{Type C, } m = 10\\
t = 3: &\quad 43 && \text{Type A, } m = 7,\ \nu_3(m) = 0\\
t = 4: &\quad 57 && \text{Type B}\\
t = 5: &\quad -1
\end{aligned}$$

t	n_t	Type	$m = \frac{n_t-1}{6}$	Factorization of m	Timer $\nu_3(m)$	Calculation	n_{t+1}
0	37	A	6	$2\cdot 3^1$	1	$\frac{4\cdot 37-1}{3}$	49
1	49	A	8	2^3	0	$\frac{4\cdot 49-1}{3}$	65
2	65	C	10	$2\cdot 5$	-	$\frac{2\cdot 65-1}{3}$	43
3	43	A	7	7	0	$\frac{4\cdot 43-1}{3}$	57
4	57	B	-	-	-	$\frac{0\cdot 57-1}{3} = -\frac{1}{3}$	-1
5	-1	-	-	-	-	-	-

Timer behavior:

- At $t = 0$, $m = 6$ has one factor of 3, so the timer is $\mathbf{1}$.
- After one Type-A transformation, the factor of 3 is stripped.
- The next Type-A step must leave Type A (since $3 \nmid m$).
- Type C shrinks until termination via Type B.

Example 3: A longer mixed sequence

Start: $n_0 = 325$

This example shows several A → A, A → C, and C → C transitions before finally hitting Type B.

$$\begin{aligned}
t = 0: &\quad 325 && \text{A, } m = 54,\ \nu_3(m) = 3\\
t = 1: &\quad 433 && \text{C, } m = 72\\
t = 2: &\quad 577 && \text{A, } m = 96,\ \nu_3(m) = 1\\
t = 3: &\quad 769 && \text{A, } m = 128,\ \nu_3(m) = 0\\
t = 4: &\quad 1025 && \text{A, } m = 171,\ \nu_3(m) = 2\\
t = 5: &\quad 683 && \text{C, } m = 113\\
t = 6: &\quad 455 && \text{C, } m = 75\\
t = 7: &\quad 303 && \text{B}\\
t = 8: &\quad -1
\end{aligned}$$

Detailed table

				Factorization of	Timer		
t	n_t	Type	$m = \frac{n_t - 1}{6}$	m	$\nu_3(m)$	Calculation	n_{t+1}
0	325	A	54	$2 \cdot 3^3$	3	$\frac{4\cdot325-1}{3}$	433
1	433	C	72	$2^3 \cdot 3^2$	—	$\frac{2\cdot433-1}{3}$	577
2	577	A	96	$2^5 \cdot 3$	1	$\frac{4\cdot577-1}{3}$	769
3	769	A	128	2^7	0	$\frac{4\cdot769-1}{3}$	1025
4	1025	A	171	$3^2 \cdot 19$	2	$\frac{4\cdot1025-1}{3}$	683
5	683	C	113	113	—	$\frac{2\cdot683-1}{3}$	455
6	455	C	75	$3 \cdot 5^2$	—	$\frac{2\cdot455-1}{3}$	303
7	303	B	—	—	—	$\frac{0\cdot303-1}{3} = -\frac{1}{3}$	−1
8	−1	—	—	—	—	—	—

What this example illustrates

- The first step $325 \to 433$ immediately drops Type A → C.
- Type-C steps reduce m (72 → 113 → 75) even though values bounce.
- Re-entering Type A at 577 and 769 begins new A-phases with new timers.
- At 1025 the timer is nonzero, but the very next step leaves A → C → B.
- As always, Type B sends you to −1 immediately.

This example highlights the core mechanism of §4.4:

- **A-phases**: countdown of $\nu_3(m)$
- **C-phases**: strict descent in m
- **B**: absorbing exit
- **Guaranteed termination**, no cycles.

Summary of the examples

The examples demonstrate explicitly:

1. **Type A** stretches are governed by the hidden timer $\nu_3(m)$; they cannot last longer than the number of 3's dividing m.
2. **Type C** updates strictly decrease m and always force an eventual transition.
3. **Type B** ends the chain immediately with -1.
4. The base-6 parameter m never returns to a previous value, so cycles are impossible.

These worked examples visualize the exact mechanisms behind the formal proofs in §§4.3–4.4.

4.6 Constructing long predecessor chains

If one wishes to construct an odd integer with exactly **L consecutive Type-A steps**, choose

$$m_0 = 3^{L-1}, \quad n_0 = 6m_0 + 1 = 6 \cdot 3^{L-1} + 1.$$

Then:

- $m_t = 4^t \cdot 3^{L-1-t}$
- timer$(t) = v_3(m_t) = L - 1 - t$
- So the timer goes $L - 1$, $L - 2$, …, 2, 1, 0 across **L** Type-A steps $t = 0, 1, \dots, L - 1$.
- At the **next** move (from the last A), you exit A into **Type B** (because after the last 3 is peeled, the remaining factor is $\equiv 1 \pmod 3$).

Worked example for $L = 10$

Say we need a predecessor chain with 10 As. We choose

- $L = 10$
- $m_0 = 3^{L-1} = 3^9 = 19683$
- $n_0 = 6m_0 + 1 = 6 \cdot 3^9 + 1 = 118099$

Then the chain is $118099 \to 157465 \to 209953 \to 279937 \to 373249 \to 497665 \to 663553 \to 884737 \to 1179649 \to 1572865 \to 2097153 \to -1$, with the **first 10 steps all in Type A**, then a final jump to Type B and -1.

Here $n_t \equiv 1 \pmod 6$ for $t = 0 \dots 9$, so each step uses

$$n_{t+1} = \frac{4n_t - 1}{3}, \quad n_t = 6m_t + 1.$$

We track $m_t = \dfrac{n_t - 1}{6}$, its factorization, and the 3-adic timer $v_3(m_t)$.

$$\begin{aligned}
t = 0:&\quad n_0 = 118099 &&\text{A},\ m_0 = 19683,\ v_3(m_0) = 9\\
t = 1:&\quad n_1 = 157465 &&\text{A},\ m_1 = 26244,\ v_3(m_1) = 8\\
t = 2:&\quad n_2 = 209953 &&\text{A},\ m_2 = 34992,\ v_3(m_2) = 7\\
t = 3:&\quad n_3 = 279937 &&\text{A},\ m_3 = 46656,\ v_3(m_3) = 6\\
t = 4:&\quad n_4 = 373249 &&\text{A},\ m_4 = 62208,\ v_3(m_4) = 5\\
t = 5:&\quad n_5 = 497665 &&\text{A},\ m_5 = 82944,\ v_3(m_5) = 4\\
t = 6:&\quad n_6 = 663553 &&\text{A},\ m_6 = 110592,\ v_3(m_6) = 3\\
t = 7:&\quad n_7 = 884737 &&\text{A},\ m_7 = 147456,\ v_3(m_7) = 2\\
t = 8:&\quad n_8 = 1179649 &&\text{A},\ m_8 = 196608,\ v_3(m_8) = 1\\
t = 9:&\quad n_9 = 1572865 &&\text{A},\ m_9 = 262144,\ v_3(m_9) = 0\\
t = 10:&\quad n_{10} = 2097153 &&\text{B}\\
t = 11:&\quad n_{11} = -1
\end{aligned}$$

Detailed table You can see the pattern very cleanly:

- $m_t = 4^t \cdot 3^{9-t}$,
- $v_3(m_t) = 9 - t$ counts down $9, 8, \dots, 1, 0$,
- after the last Type-A step $timer = 0$, we land in Type B and then immediately at -1.

t	n_t	Type	$m_t = \frac{n_t - 1}{6}$	Factorization of m_t	Timer $\nu_3(m_t)$	Calculation	n_{t+1}
0	118099	A	19683	3^9	9	$\frac{4\cdot 118099-1}{3}$	157465
1	157465	A	26244	$2^2 \cdot 3^8$	8	$\frac{4\cdot 157465-1}{3}$	209953
2	209953	A	34992	$2^4 \cdot 3^7$	7	$\frac{4\cdot 209953-1}{3}$	279937
3	279937	A	46656	$2^6 \cdot 3^6$	6	$\frac{4\cdot 279937-1}{3}$	373249
4	373249	A	62208	$2^8 \cdot 3^5$	5	$\frac{4\cdot 373249-1}{3}$	497665
5	497665	A	82944	$2^{10} \cdot 3^4$	4	$\frac{4\cdot 497665-1}{3}$	663553
6	663553	A	110592	$2^{12} \cdot 3^3$	3	$\frac{4\cdot 663553-1}{3}$	884737
7	884737	A	147456	$2^{14} \cdot 3^2$	2	$\frac{4\cdot 884737-1}{3}$	1179649
8	1179649	A	196608	$2^{16} \cdot 3$	1	$\frac{4\cdot 1179649-1}{3}$	1572865
9	1572865	A	262144	2^{18}	0	$\frac{4\cdot 1572865-1}{3}$	2097153
10	2097153	B	—	—	—	$\frac{0\cdot 2097153-1}{3} = -\frac{1}{3}$	(-1)
11	(-1)	—	—	—	—	—	—

Unbounded predecessor depth and arbitrarily long finite trajectories

The construction above shows that predecessor depth is unbounded across the odd integers. For every $L \geq 1$, we can explicitly construct an odd integer $n(L)$ whose predecessor chain begins with L consecutive Type-A steps. Equivalently, the timer $\nu_3(m)$ at the start of the chain can be made arbitrarily large. This does not produce a single odd integer with an infinite predecessor chain. Instead, it produces an infinite sequence of integers whose predecessor chains have strictly increasing (but still finite) Type-A run length. Moreover, because n_{t+1} is defined to be an immediate predecessor of n_t, we have $T(n_{t+1}) = n_t$ at every step of the chain. Therefore, reversing a length-L predecessor chain yields a valid forward accelerated Collatz orbit segment of length L under T. In particular, although every individual predecessor chain terminates at -1, there is no uniform bound (independent of the starting value) on the number of Type-A steps that can occur before termination. Any termination proof must therefore rely on a structural descent mechanism (as in §§4.3–4.4), rather than on bounding the number of steps by a global constant.

A reference implementation for obtaining the detailed analysis tables is provided in Appendix D.Sec-4.6.

4.7 Non-Intersection of Predecessor Trees (Intuition)

Before giving the formal argument in §4.8, we explain the structural reason why **different predecessor trees cannot merge**. The situation is this: for any odd integer n_t, we can form its **minimal predecessor** n_{t+1}, and from that value there is an **infinite family** of additional predecessors

$$P_0, P_1, P_2, \ldots$$

each of which generates its own predecessor chain terminating at -1. The question is:

Can two such chains intersect before reaching -1?

The answer is *no*, and the reason is extremely simple once the right viewpoint is adopted.

The structure we are exploring

Every odd integer $m \neq -1$ has a unique outgoing arrow under the predecessor rule, namely $m \mapsto \text{pred}(m)$.

$$m \longmapsto \text{pred}(m),$$

where "pred" is the rule from §4.3. This rule sends any odd integer to its *unique* smallest predecessor in its predecessor class. In contrast, the *reverse* direction (building all predecessors of a given value) produces a **tree**, since from a single node y we may generate infinitely many values x such that y is the minimal predecessor of x. Thus each n_t gives rise to infinitely many *backward chains*

$$P_j \ \rightarrow \ P_{j,1} \ \rightarrow \ P_{j,2} \ \rightarrow \ \dots \ \rightarrow \ -1,$$

one chain for each predecessor P_j.

Why -1 is the universal endpoint

Section 4.3 established that **every predecessor chain terminates at -1** in finitely many steps. Therefore, if different chains ever *did* intersect, the intersection must occur **strictly before -1**, because afterward every chain collapses to the absorbing endpoint.

The key structural fact

Here is the crucial property: **Every odd integer $y \neq -1$ has at most one forward successor under the predecessor rule.** Equivalently, if we already stand at some integer y, then there is **at most one** integer x such that

$$\text{pred}(x) = y.$$

This is immediate from the construction of the predecessor rule: the forward map is a well-defined function, not a multi-valued relation. Thus in the directed graph of all odd integers:

- every node has **exactly one outgoing arrow** (towards its minimal predecessor), and
- every node except -1 has **at most one incoming arrow** under the predecessor rule.

A directed graph with this property cannot have two "branches" converging into a single node anywhere above the root.

Why merging cannot occur

Suppose two different predecessor trees—say the ones rooted at P_0 and P_1—were to intersect at some value $z \neq -1$. That would imply:

- there exist two distinct integers $x_1 \neq x_2$ such that

$$\text{pred}(x_1) = z = \text{pred}(x_2).$$

But this contradicts the key structural fact: **no integer except -1 can have two distinct parents.** Therefore two chains cannot merge; they must remain disjoint all the way down to the common endpoint -1.

The intuitive picture

Think of the predecessor relations as a forest of rooted trees whose roots all lie at -1. Each node has exactly one arrow pointing downward toward its minimal predecessor, but possibly many arrows branching upward to larger integers.

- **Downward:** the structure behaves like a deterministic function (each node has one child).
- **Upward:** the structure branches infinitely, but branches never reconverge.

Every chain is an independent descending path, and these paths **never collide** except at the unique absorbing terminal node.

4.8 Non-Intersection of Predecessor Trees (Proof)

Set-up

Let's work on the set $\mathcal{O} = \{\text{odd integers}\}$. Let's define the forward map $f : \mathcal{O} \to \mathcal{O} \cup \{-1\}$ by the rule we derived (writing $n \equiv r \pmod{6}$ with $r \in \{1, 3, 5\}$):

$$f(n) = \begin{cases} \dfrac{4n-1}{3}, & n \equiv 1 \pmod{6}, \\ -1, & n \equiv 3 \pmod{6}, \\ \dfrac{2n-1}{3}, & n \equiv 5 \pmod{6}. \end{cases} \tag{$\star$}$$

For a fixed odd n_t, let n_{t+1} be its *canonical* (smallest) immediate predecessor. The formula for **all** immediate predecessors of n_t is

$$P_k \ = \ 4^k n_{t+1} + \frac{4^k - 1}{3} \qquad (k = 0, 1, 2, \dots),$$

with $P_0 = n_{t+1}$. Note that $P_{k+1} - P_k = 4^k(3n_{t+1} + 1) > 0$, so the P_k are **distinct odd** integers. For each k, we define the chain

$$\mathcal{C}_k = \{P_k,\ f(P_k),\ f^2(P_k), \dots\}.$$

From our termination proof, every chain eventually reaches -1, and there are no cycles among odd numbers ≥ 3. We will not re-prove termination; we only use it at the very end to note that -1 is the common sink.

Lemma (in-degree ≤ 1 off -1)

For every odd $y \neq -1$, there is **at most one** odd x with $f(x) = y$.

Proof. If $f(x) = y \neq -1$, then $x \not\equiv 3 \pmod{6}$. There are only two possible branches in $(\star)$:

- If $x \equiv 1 \pmod{6}$, then $y = \dfrac{4x-1}{3}$, so

$$x = \frac{3y+1}{4}. \tag{1}$$

For odd y, the numerator $3y+1$ is divisible by 4 **iff** $y \equiv 1 \pmod 4$ (since $3 \cdot 1 + 1 \equiv 0 \pmod 4$, $3 \cdot 3 + 1 \equiv 2 \pmod 4$). Thus (1) can yield an integer x **only when** $y \equiv 1 \pmod 4$. In that same congruence, $\dfrac{3y+1}{2}$ is even, so it **cannot** be an odd preimage from the other branch below.

- If $x \equiv 5 \pmod 6$, then $y = \dfrac{2x-1}{3}$, so

$$x = \frac{3y+1}{2}. \tag{2}$$

For odd y, the value in (2) is **odd** (hence a valid odd preimage) **iff** $y \equiv 3 \pmod 4$ (because $3 \cdot 3 + 1 \equiv 10 \equiv 2 \pmod 4$ gives an odd after division, while $3 \cdot 1 + 1 \equiv 0 \pmod 4$ gives an even). In that case $\dfrac{3y+1}{4}$ is **not** an integer, so the previous branch cannot apply.

Thus for any fixed odd $y \neq -1$, **at most one** of the two formulas (1) or (2) can produce an admissible odd x. Hence $|f^{-1}(y) \cap \mathcal{O}| \leq 1$. The only exceptional node with many odd preimages is $y = -1$: indeed $f(x) = -1$ for **all** $x \equiv 3 \pmod 6$.

Theorem (disjointness of the P_k-chains)

For $k \neq \ell$,

$$\mathcal{C}_k \cap \mathcal{C}_\ell \subseteq \{-1\}.$$

Equivalently, the chains $\mathcal{C}_k$ and $\mathcal{C}_\ell$ **do not intersect** at any odd number other than the common sink -1.

Proof. Suppose toward a contradiction that there exist distinct $k \neq \ell$ and some $z \in \mathcal{O} \setminus \{-1\}$ with $z \in \mathcal{C}_k \cap \mathcal{C}_\ell$. Choose such a z for which the **total** number of forward steps from P_k and P_ℓ to reach z is minimal. Then there are integers $i, j \geq 1$ with

$$f^i(P_k) = z = f^j(P_\ell),$$

and their respective predecessors

$$u := f^{i-1}(P_k), \qquad v := f^{j-1}(P_\ell)$$

are **odd** and satisfy $f(u) = f(v) = z$. By minimality, $u \neq v$ (otherwise we would have found a smaller meeting point one step earlier). This contradicts the Lemma, which says an odd $z \neq -1$ has **at most one** odd preimage. Therefore no such z exists, and $\mathcal{C}_k \cap \mathcal{C}_\ell \subseteq \{-1\}$.

Corollary (structure of the forest)

- The distinct P_k are distinct roots of **pairwise disjoint** forward chains.
- By our termination result, every chain eventually reaches -1.

Thus the directed graph on odd integers is a **forest of in-arcs** feeding a single sink -1, and no two trees in this forest merge before -1.

This is the whole argument: the **unique-preimage** lemma (off -1) is the engine. Once that is established, disjointness of the chains is forced, because two different chains could only meet by giving the same node two different odd parents—ruled out by the lemma.

4.9 The Terminal Congruence Class $n_t \equiv 3 \pmod 6$

From §4.3 we know that

- every predecessor chain eventually reaches -1, and
- the final odd integer **before** -1 must satisfy $n_t \equiv 3 \pmod 6$.

We now record the precise structure of this congruence class and why it is exactly the set of odd integers with **no forward successor other than the absorbing sink** -1.

Characterization of the terminal class

The condition $n_t \equiv 3 \pmod 6$ is equivalent to $6 \mid (n_t - 3)$. Thus there exists a unique integer $k \in \mathbb{Z}$ with $n_t = 6k + 3$. Since we only consider odd integers $n_t \geq 3$, we may restrict to $k \geq 0$. The full terminal congruence class is therefore

$$\boxed{n_t : n_t = 6k + 3,\ k \geq 0}$$

This is the **entire set** of odd integers that reduce to -1 in one application of the predecessor rule.

Why $6k + 3$ are exactly the nodes with no forward successor

By definition of f, the branch $n \equiv 3 \pmod 6$ is the only one in which the update returns -1. Hence $f(n) = -1$ if and only if $n \equiv 3 \pmod 6$. From the definition of the forward map ($\star$),

$$f(n) = \begin{cases} \dfrac{4n-1}{3}, & n \equiv 1 \pmod 6, \\ -1, & n \equiv 3 \pmod 6, \\ \dfrac{2n-1}{3}, & n \equiv 5 \pmod 6 \end{cases} \tag{$\star$}$$

we immediately see that whenever $n_t \equiv 3 \pmod 6$, $f(n_t) = -1$. Conversely, $n \equiv 3 \pmod 6$ is the only branch of ($\star$) for which the forward iteration terminates, since all other congruence classes map to odd integers ≥ 1. Hence every integer of the form $6k + 3$ is a *terminal node* of the predecessor graph.

$$\boxed{n_t \equiv 3 \pmod 6 \iff f(n_t) = -1 \iff n_t \text{ has no forward successor under } f}$$

Interpretation

Every predecessor chain ends by landing in this congruence class, and the map sends it immediately to -1. Thus the values $6k + 3$ form the **terminal layer** of the predecessor forest—the unique "floor" above the sink. However, not every value of the form $6k + 3$ can actually **appear** as the terminal value of a predecessor chain. Some such integers are genuine terminal seeds, while others possess additional binary structure—extra trailing 01-blocks in their signatures—that makes them *unreachable* as true terminal nodes. Thus the set $6k + 3$ naturally splits into two distinct subclasses:

- **canonical terminal seeds**, which arise from actual predecessor chains, and

- **decorated terminal seeds**, which never occur as true chain endpoints.

This finer structural distinction is developed next in §4.10.

4.10 A Mod-4 Refinement: Canonical vs. Decorated Terminal Seeds

In §4.9 we established that every predecessor chain of an odd integer terminates at a value of the form $n_t = 6k + 3$. This identifies the **coarse** congruence class of all possible terminal nodes. In this section we refine that description by examining the internal 2-adic structure encoded by $\nu_2(3n + 1)$.

A striking pattern emerges:

> **Among the four residue classes k (mod 4), only three correspond to genuine terminal seeds; the fourth consists of structurally inflated (decorated) values. The fourth class consists of *decorated* values carrying extra trailing 01-blocks in their binary signatures; these collapse under normalization to smaller canonical representatives.**

The phenomenon is visible immediately when listing the values $n = 6k + 3$ together with their normalized forms:

```
k=0   n=3    bin(n)=11         k0_normalize(n)=3     bin(k0)=11
k=1   n=9    bin(n)=1001       k0_normalize(n)=9     bin(k0)=1001
k=2   n=15   bin(n)=1111       k0_normalize(n)=15    bin(k0)=1111
k=3   n=21   bin(n)=10101      k0_normalize(n)=1     bin(k0)=1
k=4   n=27   bin(n)=11011      k0_normalize(n)=27    bin(k0)=11011
k=5   n=33   bin(n)=100001     k0_normalize(n)=33    bin(k0)=100001
k=6   n=39   bin(n)=100111     k0_normalize(n)=39    bin(k0)=100111
k=7   n=45   bin(n)=101101     k0_normalize(n)=11    bin(k0)=1011
k=8   n=51   bin(n)=110011     k0_normalize(n)=51    bin(k0)=110011
k=9   n=57   bin(n)=111001     k0_normalize(n)=57    bin(k0)=111001
k=10  n=63   bin(n)=111111     k0_normalize(n)=63    bin(k0)=111111
k=11  n=69   bin(n)=1000101    k0_normalize(n)=17    bin(k0)=10001
k=12  n=75   bin(n)=1001011    k0_normalize(n)=75    bin(k0)=1001011
k=13  n=81   bin(n)=1010001    k0_normalize(n)=81    bin(k0)=1010001
k=14  n=87   bin(n)=1010111    k0_normalize(n)=87    bin(k0)=1010111
k=15  n=93   bin(n)=1011101    k0_normalize(n)=23    bin(k0)=10111
k=16  n=99   bin(n)=1100011    k0_normalize(n)=99    bin(k0)=1100011
k=17  n=105  bin(n)=1101001    k0_normalize(n)=105   bin(k0)=1101001
k=18  n=111  bin(n)=1101111    k0_normalize(n)=111   bin(k0)=1101111
k=19  n=117  bin(n)=1110101    k0_normalize(n)=7     bin(k0)=111
k=20  n=123  bin(n)=1111011    k0_normalize(n)=123   bin(k0)=1111011
```

The decorated cases occur exactly when $k \equiv 3 \pmod 4$.

Trailing 01-Blocks and the Role of $\nu_2(3n + 1)$

For any odd integer m, the standard accelerated Collatz successor is

$$T(m) = \frac{3m + 1}{2^{\nu_2(3m+1)}}.$$

For terminal-class integers $m = 6k + 3$, we compute:

$$3m + 1 = 3(6k + 3) + 1 = 18k + 10 = 2(9k + 5),$$

hence

$$\nu_2(3m + 1) = 1 + \nu_2(9k + 5).$$

Thus the valuation $\nu_2(3m + 1)$ serves as an algebraic measure of the number of reversible predecessor C-steps encoded in m. In particular, the quantity $\lfloor \nu_2(3m+1)/2 \rfloor$ detects precisely when extra '$4x + 1$' layers are present (the decorated case $k \equiv 3 \pmod 4$).

Experimental Observation via Normalization

Using the normalization function

$$m_0 = k0\_normalize(m),$$

which removes all trailing 01-blocks, the behavior for

$$m = 6k + 3, \qquad 0 \leq k < 21,$$

is summarized above. The pattern:

- If $k \equiv 0, 1, 2 \pmod 4$, then

  $$k0\_normalize(m) = m$$

  These are **canonical** terminal values: no excess 01-blocks.

- If $k \equiv 3 \pmod 4$, then

  $$k0\_normalize(m) < m$$

  These are **decorated**: they contain one or more superfluous appended 01-blocks and collapse to a smaller canonical value.

This empirical pattern is exactly explained by the valuation $\nu_2(3m + 1)$.

The Mod-4 Classification

The following valuation computation explains the observed pattern.

Lemma.

Let $n = 6k + 3$. Then

$$\nu_2(3n + 1) = \begin{cases} 1, & k \equiv 0 \pmod 4, \\ 2, & k \equiv 1 \pmod 4, \\ 1, & k \equiv 2 \pmod 4, \\ \geq 3, & k \equiv 3 \pmod 4 \end{cases}.$$

Consequences.

1. For $k \equiv 0, 1, 2 \pmod 4$, the valuation gives at most one trailing 01-block. These values are structurally minimal (canonical).
2. For $k \equiv 3 \pmod 4$, the valuation is ≥ 3, creating multiple trailing 01-blocks. Such values represent **inflated** descendants of smaller canonical cores.

Canonical and Decorated Seeds

Definition.

Let $n = 6k + 3$.

- n is **canonical** if $k \equiv 0, 1, 2 \pmod 4$, equivalently if $k0\_normalize(n) = n$.
- n is **decorated** if $k \equiv 3 \pmod 4$, equivalently if $k0\_normalize(n) < n$.

Examples

- *Canonical seeds* (no collapse):

$$3,\ 9,\ 15,\ 27,\ 33,\ 39,\ 51,\ 57,\ 63,\ \dots$$

- *Decorated seeds* (collapse shown):

$$21 \mapsto 1, \qquad 45 \mapsto 11, \qquad 69 \mapsto 17, \qquad 93 \mapsto 23, \qquad 117 \mapsto 7,\ \dots$$

All decorated values reduce to a smaller canonical core.

Summary of the Refinement

The coarse mod-6 condition $n_t = 6k + 3$ splits naturally into a mod-4 refinement revealing internal binary structure.

$$\boxed{\text{Terminal candidates } n_t = 6k + 3 \text{ fall into four residue classes mod } 4.}$$

Only the classes $k \equiv 0, 1, 2 \pmod 4$ are **canonical**. The class $k \equiv 3 \pmod 4$ consists of **decorated**, nonminimal seeds that collapse under normalization.

A reference implementation for obtaining the analysis table is provided in Appendix D.Sec-4.10.

4.11 Decorated Seeds Are Not Reachable Terminal Values

In §4.10 we refined the coarse classification $n_t = 6k + 3$ into four subclasses according to the residue of $k \pmod 4$. Only the cases $k \equiv 0, 1, 2 \pmod 4$ produce **canonical** terminal seeds, while the case $k \equiv 3 \pmod 4$ produces **decorated** seeds, identifiable by extra trailing 01-blocks in their binary signatures. In this section we prove the central fact:

Decorated seeds never occur as actual terminal values of predecessor chains.

In other words, although numbers of the form $6k+3$ with $k \equiv 3 \pmod 4$ can be written down, *no backward Collatz evolution ever lands on them.* Thus one quarter of the nominal terminal candidates are eliminated, leaving only the canonical subclasses as genuine terminal seeds.

Structural Constraint on Terminal Values

Let n_t be the final odd integer before -1 in a predecessor chain. By definition, n_t has **no odd predecessor** under the inverse accelerated Collatz relation other than those mapping directly to -1. Equivalently, n_t admits **no reversible predecessor step** of type A or C. Recall that reversible predecessor steps correspond exactly to the presence of additional powers of 2 dividing $3n_t + 1$. If

$$v_2(3n_t + 1) \geq 3,$$

then $3n_t + 1$ admits at least one reversible C-predecessor, yielding an odd integer p such that

$$n_t = \frac{3p+1}{2^r}, \qquad r \geq 2.$$

In this case, n_t is **not terminal**, since the predecessor chain can be extended further upward. Therefore, for n_t to be terminal, it must satisfy

$$v_2(3n_t + 1) \leq 2.$$

Equivalently, terminal odd values admit **no reversible predecessor steps**, and hence possess *minimal* 2-adic divisibility compatible with their form.

Decorated Seeds Require Extra 2-Power Divisions

Let

$$n_d = 6k + 3, \qquad k \equiv 3 \pmod 4$$

be a decorated seed. From §4.10 we know:

$$v_2(3n_d + 1) \geq 3.$$

Thus we can write

$$3n_d + 1 = 2^r u, \qquad r \geq 3,\ u \text{ odd}.$$

If n_d were the terminal odd of a predecessor chain, then for some odd p,

$$n_d = \frac{3p+1}{2^r},$$

with no further factor of 2 dividing the numerator. But this is impossible, because:

- **Forward:** $p \mapsto T(p)$ removes *all* powers of 2 from $3p+1$.
- **Backward:** The inverse step $n \mapsto \frac{4n-1}{3}$ does **not** introduce new 2-adic factors in later evaluations of $3n+1$.

Thus any terminal odd must have the *minimum possible* 2-adic divisibility consistent with its form. Decorated seeds have **excess** divisibility and therefore cannot be terminal.

Formal Proof That Decorated Seeds Are Unreachable

Theorem.
Let $n_d = 6k + 3$ with $k \equiv 3 \pmod 4$. Then **no predecessor chain can terminate at** n_d.

Proof.
From §4.10, every decorated seed satisfies

$$\nu_2(3n_d + 1) \geq 3.$$

By the structural constraint established in §4.11.1, any odd integer n_t that is terminal in a predecessor chain must satisfy

$$\nu_2(3n_t + 1) \leq 2,$$

since otherwise it would admit a reversible predecessor step. Thus n_d cannot be terminal. Hence no predecessor chain can terminate at a decorated seed. ■

Consequence: Only Canonical Seeds Are Reachable

Combining the mod-4 classification with the non-reachability theorem, we obtain:

Corollary.
The only odd values that can appear as the last odd before -1 in a predecessor chain are

$$\boxed{6k + 3 \ : \ k \equiv 0, 1, 2 \pmod 4}$$

equivalently,

$$\boxed{n_t \equiv 3,\ 9,\ 15 \pmod{24}.}$$

These are exactly the values for which:

$$\nu_2(3n_t + 1) \in \{1, 2\}, \qquad k0\_normalize(n_t) = n_t.$$

These values are the **canonical terminal seeds**. The case $k \equiv 3 \pmod 4$ corresponds to decorated, non-minimal values that never arise as actual endpoints of predecessor chains.

Summary

We now have the complete refinement:

$$\boxed{\text{Terminal values lie in } 6k + 3} \qquad \boxed{\text{but only } k \equiv 0, 1, 2 \pmod 4 \text{ are reachable.}}$$

Decorated seeds ($k \equiv 3 \pmod 4$) are structurally inflated artifacts of the mod-6 class; normalization collapses them to their smaller canonical cores, and no predecessor chain ever reaches them.

4.12 Parametrization of Odd Integers by Canonical Terminal Seeds

We now combine the predecessor rules and termination results of Chapter 3 and Chapter 4 so far, into a complete parametrization of all odd integers. In Section 4.2, each predecessor step was classified as:

- **A-step**

$$p = \frac{4n-1}{3}, \qquad n \equiv 1 \pmod 6,$$

- **C-step**

$$p = \frac{2n-1}{3}, \qquad n \equiv 5 \pmod 6.$$

Every predecessor chain is a finite sequence of such A/C steps, ending in a single final B-step into -1.

However, as we saw in later sections, **not every odd integer actually lies on an A/C predecessor chain**: certain type-B values inside 01-towers are "non-terminal B's" with no A- or C-predecessor. To obtain a uniform coordinate system based on A/C words, we introduce a simple 01-lift that moves these exceptional B's one step up their 01-tower before tracing predecessors.

In this section we show that A/C steps together with a single **lift bit** give a complete coordinate system for all odd integers.

1. A Unique Canonical Seed for Every Odd Integer

For any positive odd integer n, we associate a terminal "seed" by tracing a predecessor chain down to the final odd value before the terminal B-step into -1; we will show this terminal seed is always canonical.

A minor complication is that some odd integers with $n \equiv 3 \pmod 6$ lie **inside** a 01-tower: these are the *non-terminal* type-B nodes (the pass-through B's) that have no A- or C-predecessor at all. In that case we first apply a single 01-lift (defined in §4.12.2.1) to move one step upward in the tower and enter the A/C structure. Thus we work with the lifted value $L(n)$, which equals n unless n is a non-terminal B.

By the predecessor termination theorem (Chapter 3 / earlier in Chapter 4), the predecessor chain of $L(n)$ is finite and eventually reaches some odd value m with $m \equiv 3 \pmod 6$, at which point the next step is the terminal B-step into -1. The results of §4.10–§4.11 now apply: among the numbers $m = 6k+3$, only the **canonical** residues $m \equiv 3, 9, 15 \pmod{24}$ can occur as actual terminal values of predecessor chains, while the remaining "decorated'' case $k \equiv 3 \pmod 4$ is unreachable. Therefore the terminal odd reached from $L(n)$ is always a **canonical terminal seed**, which we denote by $m_0(n)$.

We now collect all such canonical terminal seeds into a single set:

$$\mathcal{S} := \{\, m_0(n) : n \text{ is a positive odd integer} \,\}.$$

Equivalently, $\mathcal{S}$ is the set of canonical residues $m_0 \equiv 3, 9, 15 \pmod{24}$ that actually arise as terminal seeds.

Theorem (Terminal-seed partition).
Every positive odd integer n determines a unique canonical seed $m_0(n) \in \mathcal{S}$, and the fibers $\{n : m_0(n) = s\}$ for $s \in \mathcal{S}$ form a disjoint partition of the positive odd integers. In other words:

- each odd integer has exactly one canonical seed $s = m_0(n)$,
- the fibers $\{n : m_0(n) = s\}$ for $s \in \mathcal{S}$ are pairwise disjoint and their union is the set of positive odd integers.

Thus the canonical seeds $s \in \mathcal{S}$ induce a **natural partition** of the odd integers.

A/C Words and the 01-Lift

Whenever an odd integer x is not of type B (i.e., $x \not\equiv 3 \pmod 6$), its predecessor step is uniquely determined by $x \pmod 6$:

- if $x \equiv 1 \pmod 6$, the predecessor step is an A-step $p = \frac{4x-1}{3}$,
- if $x \equiv 5 \pmod 6$, the predecessor step is a C-step $p = \frac{2x-1}{3}$.

Thus any odd integer that lies on a genuine predecessor chain above a terminal seed carries a unique finite word $w \in \{A, C\}^*$ recording the successive predecessor steps down to the seed.

The only obstruction is the presence of **non-terminal B nodes** inside 01-towers. These are odd integers $n \equiv 3 \pmod 6$ that are not canonical seeds: equivalently, they satisfy $k0\_normalize(n) \neq n$ and contain at least one trailing 01-block in their binary signature. Such values have no A- or C-predecessor, so they cannot be assigned an A/C word directly.

To include these exceptional B's in the same coordinate system, we apply a single 01-lift. We formalize this lift next, and then use it to define a unique A/C word and canonical seed for every lifted value $L(n)$.

In particular, once $L(n)$ is defined, we attach to every odd integer n a triple (ℓ, w, m_0) by tracing the (unique) A/C predecessor chain of $L(n)$, recording the resulting word w, and taking m_0 as the canonical terminal seed reached at the end.

The 01-Lift

We define a simple auxiliary map which either leaves an odd integer alone, or lifts it once along its 01-tower.

Definition (Lift bit and 01-lift).
For an odd integer n, define:

- the **lift bit** $\ell(n) \in \{0, 1\}$ by

$$\ell(n) = \begin{cases} 1, & \text{if } n \equiv 3 \pmod 6 \text{ and } n \text{ is a non-terminal B in its 01-tower,} \\ 0, & \text{otherwise,} \end{cases}$$

  where "non-terminal B" means $n \equiv 3 \pmod 6$ and $k0\_normalize(n) \neq n$ (equivalently: n lies strictly inside its 01-tower; see §4.10–§4.11).

- the **lifted value**

$$L(n) = \begin{cases} 4n+1, & \text{if } \ell(n) = 1, \\ n, & \text{if } \ell(n) = 0. \end{cases}$$

Thus:

- If n is type A or type C, then $\ell(n) = 0$ and $L(n) = n$.
- If n is a **canonical** type-B seed (a true terminal value of a predecessor chain), then again $\ell(n) = 0$ and $L(n) = n$.
- If n is a **non-terminal B** inside a 01-tower, then $\ell(n) = 1$ and we lift it once to

$$L(n) = 4n + 1,$$

which is an A-type integer lying on a genuine A/C predecessor chain down to some canonical seed.

In all cases, $L(n)$ admits a finite A/C predecessor chain to a canonical seed m_0. The restriction $\ell \in \{0, 1\}$ is structural and will be explained by the ACB cycle in Section 4.13.

Unique A/C Word Representation (with Lift)

For a word $w \in \{A, C\}^*$, we will define $F_w(m_0)$ in a following sub-section as the reconstruction map obtained by reversing the A/C steps. We now extend the basic parametrization to all odds by working with the lifted value $L(n)$.

Theorem (Extended parametrization).
Every odd integer n has a unique triple

$$\boxed{(\ell, w, m_0)}$$

consisting of:

- a lift bit $\ell \in \{0, 1\}$,
- a word $w \in \{A, C\}^*$,
- a canonical seed m_0,

such that

$$\boxed{L(n) = F_w(m_0)}.$$

Equivalently:

- $\ell = 0$ if and only if n already lies on an A/C predecessor chain (type A, type C, or a canonical type-B seed), and then

$$n = F_w(m_0).$$

- $\ell = 1$ if and only if n is a non-terminal B: in that case

$$L(n) = 4n + 1 = F_w(m_0),$$

so n sits exactly one 01-block below the A/C point $F_w(m_0)$ in its tower.

Thus:

- ℓ records whether we had to **lift** n by one 01-block to see that structure.
- w identifies the **A/C predecessor steps** from $L(n)$ to reach m_0.
- m_0 identifies the **canonical terminal seed**

The Extended Signature of an Odd Integer

To encode this structure succinctly, we extend the notion of signature.

Definition (Extended signature).
For any odd integer n, its **extended signature** is the ordered triple

$$\boxed{\Sigma(n) = (\ell, w, m_0)}$$

where:

- $\ell = \ell(n) \in \{0, 1\}$ is the lift bit,
- $L(n)$ is the lifted value defined above,
- (w, m_0) is the A/C word and canonical seed obtained by tracing the predecessor chain of $L(n)$ down to m_0.

Equivalently,

$$L(n) = F_w(m_0), \qquad \Sigma(n) = (\ell, w, m_0).$$

Special cases:

- If n is type A or type C, or a canonical type-B seed, then $\ell = 0$ and $\Sigma(n) = (0, w, m_0)$. In this case the **canonical signature** is just the pair (w, m_0), as in the earlier definition.
- If n is a non-terminal B in a 01-tower, then $\ell = 1$, and $\Sigma(n) = (1, w, m_0)$ encodes the A/C structure seen one 01-block above n.

In this way, **every odd integer** receives uniform coordinates (ℓ, w, m_0).

Computing $F_w(m_0)$

To reconstruct an integer from its A/C word and canonical seed, we invert the predecessor steps. Recall that an A-step is $p = \frac{4n-1}{3}$ and a C-step is $p = \frac{2n-1}{3}$. Solving for n gives the one-step inverse maps $A'(x) = \frac{3x+1}{4}$ and $C'(x) = \frac{3x+1}{2}$.

If $w = w_1 \cdots w_t$ is the A/C word of the predecessor chain $n_0 \to n_1 \to \cdots \to n_t = m_0$, then reconstructing n_0 from m_0 requires undoing the steps in reverse direction along the chain: apply w_t' to m_0, then w_{t-1}', and so on, ending with w_1'. Equivalently, w records the predecessor chain from n down to m_0, while F_w reconstructs upward from m_0 back to n.

Definition.
For a word $w = w_1 w_2 \cdots w_t \in \{A, C\}^*$, define

$$F_w(m_0) = w_1'\big(w_2'(\cdots w_t'(m_0)\cdots)\big),$$

where w_i' denotes A' or C' according to whether $w_i = A$ or $w_i = C$. That is, we apply the inverse operators in the reverse direction along the chain, starting from m_0 with w_t', then w_{t-1}', and so on, ending with w_1'.

Lemma.
Let $L(n) = n_0 \to n_1 \to \cdots \to n_t = m_0$ be the predecessor chain of the lifted value $L(n)$, and let $w = w_1 \cdots w_t$ be the corresponding A/C word (with w_i the step taken from n_{i-1}).
Then

$$F_w(m_0) = n_0 = L(n).$$

Proof sketch.

- Each step w_i sends n_{i-1} to n_i. Equivalently, $n_{i-1} = w_i'(n_i)$ where w_i' is A' or C' accordingly.
- Applying w_i' reverses the step.
- Composing the inverses in reverse order therefore returns the initial value $n_0 = L(n)$. ■

Reconstructing an Odd Integer from its Extended Signature

Given the extended signature

$$\Sigma(n) = (\ell, w, m_0),$$

we can recover the original odd integer n by first reconstructing the lifted value $L(n)$, and then (if necessary) undoing the 01-lift. From a previous subsection, the A/C word w and seed m_0 determine the lifted value uniquely via

$$L(n) \;=\; F_w(m_0).$$

The lift bit ℓ then tells us whether n itself is already on the A/C chain, or sits one 01-block below it.

Reconstruction rule.
Given $\Sigma = (\ell, w, m_0)$:

1. Compute

$$x \coloneqq F_w(m_0).$$

2. Define

$$n = \begin{cases} x, & \text{if } \ell = 0, \\ \dfrac{x-1}{4}, & \text{if } \ell = 1. \end{cases}$$

Then n is exactly the odd integer whose extended signature is $\Sigma(n) = \Sigma$.

Explanation.

- If $\ell = 0$, then by definition of the 01-lift we have $L(n) = n$, so the value reconstructed from (w, m_0) is already the original integer: $n = F_w(m_0)$.
- If $\ell = 1$, then n is a non-terminal B and the lift gives $L(n) = 4n + 1$; since $F_w(m_0) = L(n)$, we recover n by $n = (F_w(m_0) - 1)/4$. In particular, $F_w(m_0) \equiv 1 \pmod 4$, so $(F_w(m_0) - 1)/4$ is an integer.

Thus non-terminal B nodes are exactly those signatures with $\ell = 1$, and they reconstruct via $n = (F_w(m_0) - 1)/4$.

Algorithm for Computing the Extended Signature $\Sigma(n)$

We now state a complete algorithm for determining $\Sigma(n)$.

Input: An odd integer n.
Output: $\Sigma(n) = (\ell, w, m_0)$, the extended signature of n.

1. **Determine the lift.**

   - If $n \equiv 3 \pmod 6$ and $k0\_normalize(n) \neq n$ (non-terminal B), set

   $$\ell \leftarrow 1, \qquad \text{current} \leftarrow 4n + 1.$$

   - Otherwise (types A, C, or canonical B), set

   $$\ell \leftarrow 0, \qquad \text{current} \leftarrow n.$$

2. **Initialize the word**

   $$w \leftarrow \text{empty}.$$

3. **Trace A/C predecessors of the lifted value.**
   While current $\not\equiv 3 \pmod 6$:

   - If current $\equiv 1 \pmod 6$:

   $$w \leftarrow wA, \qquad \text{current} \leftarrow \frac{4 \cdot \text{current} - 1}{3}.$$

   - If current $\equiv 5 \pmod 6$:

   $$w \leftarrow wC, \qquad \text{current} \leftarrow \frac{2 \cdot \text{current} - 1}{3}.$$

   (We append letters on the right, so w is written in the same order as the predecessor steps taken.)

   When the loop stops, set $m_0 \leftarrow$ current.

4. **Return**

$$\boxed{\Sigma(n) = (\ell, w, m_0)}.$$

This algorithm always terminates (by the general predecessor termination theorem), and returns the unique extended signature. For all type-A, type-C, and canonical type-B integers, $\ell = 0$ and the pair (w, m_0) coincides with the original two-component signature $\sigma(n)$ from the earlier definition.

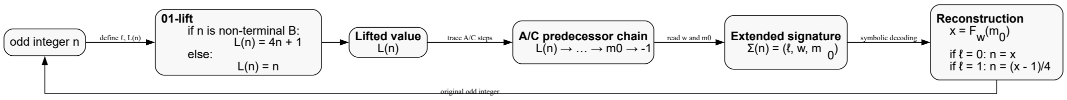

Figure 4.3. *Encoding and decoding of odd integers via extended signatures.* An odd integer n is first lifted (if necessary) to $L(n)$, whose A/C predecessor chain determines the word w and canonical seed m_0. Together with the lift bit ℓ, this yields the extended signature $\Sigma(n) = (\ell, w, m_0)$. The pair (w, m_0) reconstructs $L(n)$ symbolically via $F_w(m_0)$, and ℓ decides whether to undo a single 01-lift to recover n.

Examples of Extended Signatures

We illustrate the definition of the extended signature $\Sigma(n) = (\ell, w, m_0)$ with several explicit computations. The first four examples are type-A or type-C integers, so $\ell = 0$; the final example is a non-terminal B with $\ell = 1$. In each case, we follow the predecessor chain of $L(n) = n$ up to the canonical terminal seed (the last value before -1). In the first four examples we have $\ell = 0$, so $L(n) = n$ and we follow the predecessor chain of n itself. In the final example, $\ell = 1$, and we instead follow the predecessor chain of the lifted value $L(n)$.

Example 1. $n = 37$
Tracing predecessors:

$$37 \xrightarrow{A} 49 \xrightarrow{A} 65 \xrightarrow{C} 43 \xrightarrow{A} 57 \xrightarrow{B} -1.$$

Thus the A/C word is

$$w = \mathrm{AACA},$$

and the canonical seed is

$$m_0 = 57.$$

$$\boxed{\Sigma(37) = (0, \mathrm{AACA}, 57)}.$$

By construction,

$$L(37) = 37 = F_{\mathrm{AACA}}(57).$$

Example 2. $n = 71$
Tracing predecessors:

$$71 \xrightarrow{C} 47 \xrightarrow{C} 31 \xrightarrow{A} 41 \xrightarrow{C} 27 \xrightarrow{B} -1.$$

Thus

$$w = \text{CCAC}, \qquad m_0 = 27.$$

$$\boxed{\Sigma(71) = (0, \text{CCAC}, 27)}.$$

Again,

$$L(71) = 71 = F_{\text{CCAC}}(27).$$

Example 3. A pure A-spine element
Consider $n = 163$. Its predecessor chain is

$$163 \xrightarrow{A} 217 \xrightarrow{A} 289 \xrightarrow{A} 385 \xrightarrow{A} 513 \xrightarrow{B} -1.$$

Thus the word consists only of A-steps:

$$w = \text{AAAA}, \qquad m_0 = 513.$$

$$\boxed{\Sigma(163) = (0, \text{AAAA}, 513)}.$$

Example 4. A mixed A/C branch element
Consider $n = 13$. Its predecessor chain is

$$13 \xrightarrow{A} 17 \xrightarrow{C} 11 \xrightarrow{C} 7 \xrightarrow{A} 9 \xrightarrow{B} -1.$$

Thus the associated word is

$$w = \text{ACCA}, \qquad m_0 = 9.$$

$$\boxed{\Sigma(13) = (0, \text{ACCA}, 9)}.$$

Here both A- and C-steps occur, so 13 lies on a “branch” off the pure A-spine of its class.

Example 5. A non-terminal B inside a 01-tower
Consider $n = 87381$. Since $87381 \equiv 3 \pmod 6$, it is a type-B integer. We check whether it is a **canonical B** (a terminal seed) or a **non-terminal B**. Compute

$$3 \cdot 87381 + 1 = 262144 = 2^{18}, \qquad \nu_2(3n+1) = 18 \geq 3.$$

Thus 87381 is **not** a canonical seed (indeed $k0\_normalize(87381) = 1 \neq 87381$). Hence

$$\ell(87381) = 1, \qquad L(87381) = 4 \cdot 87381 + 1 = 349525.$$

The lifted value $L(87381) = 349525 \equiv 1 \pmod 6$ is type~A and does lie on a valid A/C predecessor chain. Tracing predecessors of $L(87381) = 349525$:

$$349525 \xrightarrow{A} 466033 \xrightarrow{A} 621377 \xrightarrow{C} 414251 \xrightarrow{C} 276167 \xrightarrow{C} 184111 \xrightarrow{A} 245481 \xrightarrow{B} -1.$$

Thus the A/C word is

$$w = \text{AACCCA},$$

and the canonical terminal seed is

$$m_0 = 245481.$$

Therefore the extended signature of 87381 is

$$\boxed{\Sigma(87381) = (1,\ \text{AACCCA},\ 245481)}$$

and indeed

$$L(87381) = 349525 = F_{\text{AACCCA}}(245481).$$

This example shows how a non-terminal B receives a signature:

- we apply one 01-lift to reach a genuine A/C point,
- compute the usual A/C signature of that lifted value,
- and record the lift bit $\ell = 1$.

Summary of the examples

n	ℓ	w	m_0	$\Sigma(n)$
37	0	AACA	57	$(0, \text{AACA}, 57)$
71	0	CCAC	27	$(0, \text{CCAC}, 27)$
163	0	AAAA	513	$(0, \text{AAAA}, 513)$
13	0	ACCA	9	$(0, \text{ACCA}, 9)$
87381	1	AACCCA	245481	$(1, \text{AACCCA}, 245481)$

In each case we have

$$L(n) = F_w(m_0), \qquad \Sigma(n) = (\ell, w, m_0),$$

illustrating the extended parametrization. Non-terminal B's appear here with $\ell = 1$ and $L(n) = 4n + 1$ in place of n.

Terminal Residue Classes of Canonical Seeds

We now record a useful refinement of the canonical seeds and their relationship to the terminal letter of the signature. Recall that canonical seeds satisfy

$$m_0 = 6k + 3, \qquad k \equiv 0, 1, 2 \pmod 4.$$

Lemma (Terminal residue classes).

Let m_0 be a canonical seed. Then:

1. m_0 lies in exactly one of the three residue classes

$$m_0 \equiv 3,\ 9,\ 15 \pmod{24},$$

and

$$\nu_2(3m_0+1) = \begin{cases} 1, & m_0 \equiv 3 \text{ or } 15 \pmod{24}, \\ 2, & m_0 \equiv 9 \pmod{24}. \end{cases}$$

2. Let $\Sigma(n) = (\ell, w, m_0)$ be the extended signature of an odd integer n, with $w \neq \varepsilon$ (nonempty word). Then:

- if w ends with **A**, we necessarily have

$$m_0 \equiv 9 \pmod{24},$$

- if w ends with **C**, we necessarily have

$$m_0 \equiv 3 \pmod{24} \quad \text{or} \quad m_0 \equiv 15 \pmod{24}.$$

In other words, A-terminal signatures always land on a seed in the single class 9 mod 24, while C-terminal signatures land on seeds in the classes 3 mod 24 or 15 mod 24.

Proof.

1. Since $m_0 = 6k+3$ with $k \equiv 0,1,2 \pmod 4$, write $k = 4t + r$ with $r \in \{0,1,2\}$. Then

$$m_0 = 6(4t+r)+3 = 24t + (6r+3),$$

so

$$m_0 \equiv 6r + 3 \pmod{24}.$$

For $r = 0,1,2$ this gives

$$m_0 \equiv 3,\ 9,\ 15 \pmod{24}.$$

Now compute $\nu_2(3m_0+1)$ in each case:

- If $m_0 = 24t+3$, then $3m_0+1 = 72t+10 = 2(36t+5)$ and $36t+5$ is odd, so $\nu_2(3m_0+1) = 1$.
- If $m_0 = 24t+9$, then $3m_0+1 = 72t+28 = 4(18t+7)$ and $18t+7$ is odd, so $\nu_2(3m_0+1) = 2$.

- If $m_0 = 24t + 15$, then $3m_0 + 1 = 72t + 46 = 2(36t + 23)$ and $36t + 23$ is odd, so $v_2(3m_0 + 1) = 1$.

This proves part (1).

2. Let w be the A/C word in the extended signature $\Sigma(n) = (\ell, w, m_0)$, with $w \neq \varepsilon$, and let the last letter of w be $L \in \{A, C\}$. This last letter corresponds to the **last predecessor step** leading into the seed m_0.

- **Case $L = A$.**

The last node before the seed is an A-node, say x, with

$$x \equiv 1 \pmod{6},$$

and the predecessor relation is

$$m_0 = \frac{4x - 1}{3}.$$

Rearranging,

$$4x - 1 = 3m_0 \quad \Rightarrow \quad x = \frac{3m_0 + 1}{4}.$$

Since x is an odd integer, $(3m_0 + 1)/4$ must be an odd integer. Thus $4 \mid (3m_0 + 1)$, and $(3m_0 + 1)/4$ is odd, which forces

$$v_2(3m_0 + 1) = 2.$$

By part (1), this occurs exactly when $m_0 \equiv 9 \pmod{24}$.

- **Case $L = C$.**

The last node before the seed is a C-node, say x, with

$$x \equiv 5 \pmod{6},$$

and the predecessor relation is

$$m_0 = \frac{2x - 1}{3}.$$

Rearranging,

$$2x - 1 = 3m_0 \quad \Rightarrow \quad x = \frac{3m_0 + 1}{2}.$$

Again x is odd, so $(3m_0 + 1)/2$ must be an odd integer. This forces

$$v_2(3m_0 + 1) = 1.$$

By part (1), this occurs exactly when $m_0 \equiv 3 \pmod{24}$ or $m_0 \equiv 15 \pmod{24}$.

This proves part (2). ■

Remark (Optional annotation).
Although the canonical A/C component of the extended signature is the pair (w, m_0), the lemma shows that the terminal letter of w determines the residue of m_0 mod 24:

- If w ends in A, then $m_0 \equiv 9 \pmod{24}$.
- If w ends in C, then $m_0 \equiv 3$ or $15 \pmod{24}$.

When convenient, we may annotate the word w by appending the residue of m_0 mod 24, writing, for example, $AACA09$, $CCAC03$, or $ACCAC15$. This annotation is derived from (w, m_0) and does not change the canonical definition of the signature.

Summary

- Every odd integer has a **unique canonical terminal seed** m_0, and the sets $\{n : m_0(n) = s\}$ with $s \in \mathcal{S}$ form a **partition** of positive odd integers.

- Some odd integers lie directly on finite A/C predecessor chains; others (the non-terminal B's) do not. A single **01-lift** resolves this by sending each such B to an A/C point.

- For every odd integer n, once lifted (if necessary) to $L(n)$, its A/C predecessor chain determines a **unique A/C word** w and a **unique canonical seed** m_0.

- Together with the lift bit ℓ, these data form the **extended signature**

  $$\Sigma(n) = (\ell, w, m_0),$$

  giving **uniform coordinates for all odd integers**, including those inside 01-towers.

- The pair (w, m_0) uniquely reconstructs the lifted value:

  $$L(n) = F_w(m_0),$$

  and the lift bit ℓ determines whether to undo one 01-lift to recover n itself.

- Thus **every odd integer** is represented uniquely by a triple (ℓ, w, m_0), and this parametrization fully captures the predecessor dynamics.

This completes the structural parametrization of the odd integers, including the dynamically "invisible" non-terminal B's.

A reference implementation for concepts discussed in this section is provided in Appendix D.Sec-4.12.

4.13 Type Cycling Along 01-Extension Towers

In Section 3.3 we showed that every odd integer has infinitely many immediate predecessors, generated by iterating the map $x \mapsto 4x+1$, which in binary corresponds to appending a trailing 01-block.

In this section we examine how the **type** A, B, C behaves along these 01-extension towers. We will show that, inside any predecessor class $\mathcal{P}_m$, the types cycle deterministically through $A \to C \to B \to A \to \cdots$. In particular, every predecessor class contains infinitely many elements of each type.

01-Extensions Inside a Predecessor Class

Fix an odd integer n, and let m be its smallest predecessor (in the sense of Section 3.3). Then all immediate predecessors of n are of the form

$$P_k = 4^k m + \frac{4^k - 1}{3}, \qquad k = 0, 1, 2, \ldots$$

and satisfy $P_{k+1} = 4P_k + 1$. Each step appends another trailing 01-block in binary. From Section 3.3 we know that all these P_k are valid predecessors of n. More generally, for any element $x \in \mathcal{P}_m$, the 01-extension tower

$$x,\ 4x+1,\ 4(4x+1)+1,\ 4^3x + \frac{4^3-1}{3}, \ldots$$

stays entirely within the same predecessor class $\mathcal{P}_m$. We now ignore the accelerated successor map T and focus purely on residue classes. Recall from Section 4.2 that for odd integers:

- type A: $x \equiv 1 \pmod 6$,
- type B: $x \equiv 3 \pmod 6$,
- type C: $x \equiv 5 \pmod 6$.

Thus type is completely determined by the residue modulo 6.

Type Evolution Under $x \mapsto 4x + 1$

Let $E(x) = 4x + 1$. Since $4 \equiv 4 \pmod 6$, we have $E(x) \equiv 4x + 1 \pmod 6$. Evaluating on the odd residue classes gives

$$\begin{aligned}
x \equiv 1 \pmod 6 \quad &\Rightarrow \quad E(x) \equiv 4 \cdot 1 + 1 = 5 \pmod 6,\\
x \equiv 5 \pmod 6 \quad &\Rightarrow \quad E(x) \equiv 4 \cdot 5 + 1 = 21 \equiv 3 \pmod 6,\\
x \equiv 3 \pmod 6 \quad &\Rightarrow \quad E(x) \equiv 4 \cdot 3 + 1 = 13 \equiv 1 \pmod 6.
\end{aligned}$$

Translating back into types, we obtain the deterministic transition

$$\boxed{A \longrightarrow C \longrightarrow B \longrightarrow A \longrightarrow \cdots}$$

along any sequence defined by $P_{k+1} = 4P_k + 1$.

Lemma (Type cycle for 01-extensions).
Let P_0 be any odd integer, and define $P_{k+1} = 4P_k + 1$. Then the types of (P_k) follow a 3-cycle:

$$\text{type}(P_{k+1}) = \begin{cases} C, & \text{if type}(P_k) = A, \\ B, & \text{if type}(P_k) = C, \\ A, & \text{if type}(P_k) = B. \end{cases}$$

Thus, starting from any type, repeated 01-extensions cycle rigidly through $A, C, B, A, C, B, \ldots$.

Corollary: Let $x \in \mathcal{P}_m$. Then the 01-extension tower $x, 4x+1, 4^2x + \frac{4^2-1}{3}, \ldots$ remains in $\mathcal{P}_m$ and its types cycle $A \to C \to B \to A \to \cdots$. Hence $\mathcal{P}_m$ contains infinitely many elements of each type.

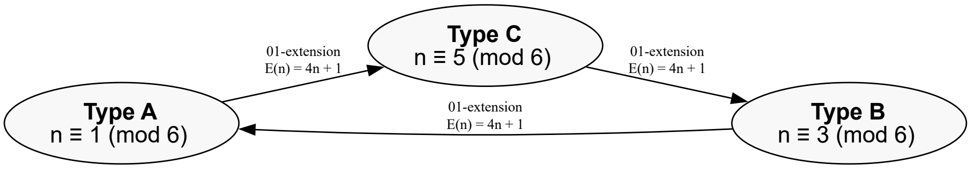

Figure 4.4. *Type cycling under 01-extension.* Prepending a single 01-block (the map $E(n) = 4n+1$) sends Type A ($n \equiv 1 \pmod 6$) to Type C, Type C ($n \equiv 5 \pmod 6$) to Type B, and Type B ($n \equiv 3 \pmod 6$) back to Type A, producing a 3-cycle $A \to C \to B \to A \to \cdots$ along each 01-tower.

Thus type cycling is a purely vertical phenomenon internal to each predecessor class, independent of predecessor-chain dynamics.

Examples

We now illustrate the type cycle within several predecessor classes. The columns below are in the order: *members of $\mathcal{P}_m$; binary; type; the immediate predecessor; signature.*

Example 1 The class $\mathcal{P}_{113}$:

n	binary	type	immediate predecessor	signature
118838613	111000101010101010101010101	B	—	(1, ACCC, 187794351)
29709653	001110001010101010101010101	C	19806435	(0, C, 19806435)
7427413	000011100010101010101010101	A	9903217	(0, AACACAAAA, 24730113)
1856853	000000111000101010101010101	B	—	(1, AACACAAAA, 24730113)
464213	000000001110001010101010101	C	309475	(0, CAAA, 733569)
116053	000000000011100010101010101	A	154737	(0, A, 154737)
29013	000000000000111000101010101	B	—	(1, A, 154737)
7253	000000000000001110001010101	C	4835	(0, CCAAC, 3819)
1813	000000000000000011100010101	A	2417	(0, AC, 1611)
453	000000000000000000111000101	B	—	(1, AC, 1611)
113	000000000000000000001110001	C	75	(0, C, 75)

Example 2 The class $\mathcal{P}_{17}$:

n	binary	type	immediate predecessor	signature
18175317	1000101010101010101010101	B	—	(1, A, 96935025)
4543829	0010001010101010101010101	C	3029219	(0, CCC, 1346319)
1135957	0000100010101010101010101	A	1514609	(0, ACCACA, 797817)
283989	0000001000101010101010101	B	—	(1, ACCACA, 797817)
70997	0000000010001010101010101	C	47331	(0, C, 47331)
17749	0000000000100010101010101	A	23665	(0, AACCAACC, 11079)
4437	0000000000001000101010101	B	—	(1, AACCAACC, 11079)
1109	0000000000000010001010101	C	739	(0, CAACCA, 777)
277	0000000000000000100010101	A	369	(0, A, 369)
69	0000000000000000001000101	B	—	(1, A, 369)
17	0000000000000000000010001	C	11	(0, CCA, 9)

Example 3 The special class $\mathcal{P}_1$:

n	binary	type	immediate predecessor	signature
1398101	101010101010101010101	C	932067	(0, C, 932067)
349525	001010101010101010101	A	466033	(0, AACCCA, 245481)
87381	000010101010101010101	B	—	(1, AACCCA, 245481)
21845	000000101010101010101	C	14563	(0, CAAC, 17259)
5461	000000001010101010101	A	7281	(0, A, 7281)
1365	000000000010101010101	B	—	(1, A, 7281)
341	000000000000101010101	C	227	(0, CCA, 201)
85	000000000000001010101	A	113	(0, AC, 75)
21	000000000000000010101	B	—	(1, AC, 75)
5	000000000000000000101	C	3	(0, C, 3)
1	000000000000000000001	A	1	—

Notice that 1 does not have a signature, since it can never reach the terminal classes $m_0 \equiv 3, 9, 15 \pmod{24}$.

4.14 Structural Interpretation of Predecessor Classes

This section introduces no new results; it synthesizes the preceding sections into a unified structural picture. The results of the preceding sections establish two fundamental properties of predecessor classes:

1. Every predecessor class $\mathcal{P}_n$ (with n an odd integer) contains **infinitely many** odd integers, generated by repeated application of the map $x \mapsto 4x + 1$ which appends a trailing 01 block in binary.

2. Along these 01-extension towers, the residue type cycles rigidly $A \to C \to B \to A \to \cdots$ so that asymptotically one third of the class lies in each of the three types.

In this section we interpret these facts structurally and explain how the backward (predecessor) and forward (successor) dynamics interact within a single predecessor class.

Signature Structure Inside a Predecessor Class

Let $x \in \mathcal{P}_n$. By the results of §4.12, x admits a **unique extended signature**

$$\Sigma(x) = (\ell_x,\ w_x,\ m_0(x)),$$

where:

- $\ell_x \in \{0, 1\}$ is the lift bit, distinguishing values lying directly on an A/C predecessor chain ($\ell_x = 0$) from non-terminal B values inside a 01-tower ($\ell_x = 1$);
- $w_x \in \{A, C\}^*$ is the A/C word describing the predecessor descent of the lifted value $L(x)$;
- $m_0(x)$ is the canonical terminal seed reached by that descent.

The extended signature records the entire predecessor history of $L(x)$ and uniquely identifies the position of x within the class. In particular:

- ℓ_x records whether x itself participates in A/C descent or sits one 01-block below such a point;
- distinct elements of $\mathcal{P}_n$ correspond to distinct extended signatures;
- distinct extended signatures yield distinct predecessor chains terminating at canonical terminal seeds.

Thus each predecessor class decomposes into an infinite family of **pairwise non-intersecting predecessor chains**, each encoded by its own triple $(\ell_x, w_x, m_0(x))$.

As the map $x \mapsto 4x + 1$ generates new non-terminal B values with $\ell = 1$, the class grows unboundedly upward, forming an infinite collection of signature-indexed branches above its minimal predecessor.

Distribution of Types Within a Class

Because the type evolution under 01-extension follows an exact 3-cycle, each predecessor class has a stable asymptotic distribution:

- approximately one third of its elements are type A,
- approximately one third are type C,
- approximately one third are type B.

Only types A and C admit valid predecessor steps; type B values admit no valid predecessor steps and therefore terminate immediately under backward iteration. Consequently, roughly two thirds of the class participate in predecessor descent, while one third consist of terminal nodes.

This imbalance constrains the backward graph to have finite branching width, despite its infinite vertical extent.

Uniqueness of the Genuine Predecessor

Among all elements of a predecessor class $\mathcal{P}_n$, there is exactly one genuine predecessor spine participating in backward descent. This spine is rooted at the smallest predecessor n itself. All other elements of the class are obtained by appending 01-blocks:

$$x, \quad 4x+1, \quad 4(4x+1)+1, \quad \dots$$

These 01-extensions do not arise from valid predecessor steps under the accelerated Collatz dynamics. Accordingly, each predecessor class consists of a single true predecessor spine, together with infinitely many vertically generated values that do not extend backward beyond their initial lift.

Collapse Under the Forward Map

Despite their infinite size, predecessor classes collapse instantly under the forward map. By construction,

$$T(x) = s \qquad \text{for all } x \in \mathcal{P}_n,$$

where s is the successor associated with the class. Thus predecessor classes exhibit a fundamental duality:

$$\boxed{\text{Backward: infinite structure} \quad \longrightarrow \quad \text{Forward: single-step collapse.}}$$

Iterating forward through successive canonical seeds shows that this pattern repeats at every level: infinitely many distinct integers, each with its own signature and predecessor chain, funnel into the same successor in one step.

Summary

Each predecessor class $\mathcal{P}_n$:

- contains infinitely many elements of types A, B, and C in a fixed $1 : 1 : 1$ asymptotic ratio;
- decomposes into infinitely many non-intersecting predecessor chains encoded by unique extended signatures;
- has exactly one element participating in genuine backward descent;
- collapses entirely to a single successor under forward iteration.

Together, these properties yield a complete structural description of predecessor classes as **infinitely extended backward objects that collapse instantaneously under the forward dynamics**.

4.15 Chapter 4 Summary: Predecessor Chains

In this chapter we developed a complete structural theory of **predecessor chains** for odd integers under the accelerated Collatz map. The objective was to understand how backward iteration,

$$n \longmapsto p \quad \text{with} \quad T(p) = n,$$

organizes the odd integers, why such chains are unique and acyclic, how they terminate, and how this backward structure interfaces with the forward Collatz dynamics. The main conclusions are summarized below.

Uniqueness and Acyclicity of Predecessor Chains

Every odd integer admits a **unique predecessor**, determined by exactly one of the two inverse rules:

- **A-step:** $p = \dfrac{4n-1}{3}$, valid when $n \equiv 1 \pmod 6$,
- **C-step:** $p = \dfrac{2n-1}{3}$, valid when $n \equiv 5 \pmod 6$.

As a consequence:

- each odd integer generates a **unique predecessor chain**,
- no integer can reappear within its own chain,
- no two predecessor chains can intersect except at the terminal node -1.

Thus the backward Collatz dynamics form a forest of disjoint trees, all rooted at -1, establishing both uniqueness and acyclicity of predecessor evolution.

Guaranteed Termination and Canonical Terminal Seeds

Every predecessor chain terminates at -1, and the final odd integer before -1 necessarily satisfies

$$n_t \equiv 3 \pmod 6.$$

Among such values, only those with

$$n_t \equiv 3,\, 9,\, 15 \pmod{24}$$

occur dynamically as terminal values. These integers are the **canonical terminal seeds**. The remaining quarter of integers of the form $6k+3$ (with $k \equiv 3 \pmod 4$) are *decorated* values that never arise as terminal points of predecessor chains.

Consequently, every predecessor chain ends at a **unique canonical seed** m_0, and terminal behavior is completely classified.

Partition of the Odds and Extended Signatures

Each canonical seed m_0 determines a **predecessor class**

$$\{\, n : m_0(n) = m_0 \,\},$$

and these classes form a **partition** of the odd integers. Every odd integer n is uniquely described by its **extended signature**

$$\Sigma(n) = (\ell, w, m_0),$$

where:

- $\ell \in \{0, 1\}$ is the lift bit distinguishing true A/C-chain elements from non-terminal B-values,

- $w \in \{A, C\}^*$ is the A/C word encoding the predecessor descent of the lifted value $L(n)$,
- m_0 is the canonical terminal seed.

These triples provide **uniform symbolic coordinates for all odd integers**, including those lying within 01-extension towers, and encode the complete backward structure of each integer.

Internal Structure Within a Predecessor Class

Above each predecessor class rises an infinite tower generated by repeated application of

$$x \mapsto 4x + 1,$$

which appends a trailing binary 01. Along such towers, the type of an integer cycles rigidly as

$$A \to C \to B \to A \to \cdots.$$

As a result:

- A- and C-type values lie on genuine predecessor chains,
- B-type values do not admit predecessors,
- non-terminal B-values are handled by a single 01-lift ($\ell = 1$),
- canonical B-values coincide with terminal seeds.

Each predecessor class is therefore **vertically infinite**, with a fixed asymptotic $1 : 1 : 1$ distribution of types A, C, and B.

Unbounded Predecessor Depth and Arbitrarily Long Forward Trajectories

Although every individual predecessor chain is finite, the constructions of §4.6 show that **predecessor depth is unbounded across the odd integers**. For every integer $L \geq 1$, one can explicitly construct odd integers whose predecessor chains contain at least L consecutive valid Type-A steps before reaching a canonical terminal seed.

Thus there exists no global finite bound on the length or structural depth of predecessor chains. Termination holds for each integer individually, but the complexity of predecessor descent grows without bound across the integers as a whole.

Because predecessor chains are unique and each backward step is invertible, any predecessor chain of length L corresponds, when reversed, to a forward accelerated Collatz trajectory of length at least L before reaching the same canonical seed. Consequently, the accelerated Collatz dynamics admit **arbitrarily long finite forward trajectories**, even though every trajectory ultimately terminates.

In particular, forward convergence cannot be established by any argument that relies on a uniform bound on stopping times or on extrapolating finite computational verification alone. Any global proof of convergence must account for the existence of Collatz orbits of arbitrarily large, yet finite, length.

Forward Funneling Under the Successor Map

Let $\mathcal{P}_b$ denote the predecessor class whose **base** is its smallest element. A fundamental property of such classes is that **all their elements share the same forward image**:

$$\boxed{T(x) = T(b) \qquad \text{for every } x \in \mathcal{P}_b.}$$

Thus an entire infinite predecessor class collapses in a single forward step to the same successor. This yields a striking duality:

$$\boxed{\text{Backward: infinite vertical structure} \quad \longrightarrow \quad \text{Forward: one-step collapse.}}$$

Iterating forward, whole predecessor classes funnel through the forward orbit

$$T(b) = s_1, \qquad T(s_1) = s_2, \qquad T(s_2) = s_3, \qquad \cdots,$$

where each successor s_i is the common forward image of its own predecessor class, whose base is its unique smallest predecessor.

Concluding Picture

The predecessor dynamics of the accelerated Collatz map are now fully characterized:

- every odd integer has a unique predecessor chain terminating at a canonical seed;
- canonical seeds partition the odd integers into infinite predecessor classes;
- each integer has a unique extended signature encoding its class and position;
- predecessor depth is unbounded across the integers;
- within each class, types cycle deterministically along 01-extension towers;
- each infinite class collapses to a single successor under forward iteration.

Together, these results provide a complete structural description of backward Collatz dynamics and their precise relationship to forward iteration.

Chapter 5

Successor Chains and Forward Dynamics

The classical Collatz map in forward time is notoriously irregular. For an odd integer n, the successor

$$T(n) = \frac{3n+1}{2^{\nu_2(3n+1)}}$$

may increase or decrease sharply, oscillate through residue classes, and display erratic binary behavior. Viewed directly, forward chains appear chaotic.

The results of Chapters 3 and 4 fundamentally change this picture. In Chapter 4 we proved that **every odd integer has a unique canonical terminal seed**

$$m_0 \equiv 3, 9, 15 \pmod{24}$$

with $\nu_2(3m_0+1) \in \{1,2\}$, and that every odd integer can be written uniquely in the form

$$n = F_w(m_0)$$

for a **canonical base** m_0 and a finite A/C word w. Thus each odd integer belongs to a unique **predecessor class**, canonically labelled by its terminal seed.

This yields a natural partition

$$\boxed{\mathcal{S} := \{\, m_0(n) : n \text{ a positive odd integer} \,\}}$$

and every odd integer n lies in exactly one of the disjoint sets

$$\boxed{\{\, n : m_0(n) = s \,\}, \qquad s \in \mathcal{S},}$$

which together cover all odd integers with no overlap between distinct seeds.

Key insight.

- This canonical-seed structure is not merely a backward-time artifact.
- It reorganizes the *forward* Collatz map as well.
- Instead of tracking successor chains on raw integers, we may project each value to the base of its predecessor class via normalization.
- In this projected system, forward motion becomes a structured dynamical process on the canonical seeds themselves.

With this perspective, we study the **forward class dynamics**, defined by

$$n_{t-1} = F(n_t) = T\big(k0\_normalize(n_t)\big).$$

Successor chains—when viewed at this class level—exhibit **coherent descent** through predecessor classes and appear to converge toward the root class $\mathcal{P}_1$.

5.1 Definition of the Forward Class Map

For any odd integer n, let $m = k0\_normalize(n)$ be the **canonical representative** of its predecessor class. From Chapter 4, T is constant on predecessor classes, so $T(n) = T(m)$. Thus forward motion may be studied directly on the canonical seeds rather than on the raw integers themselves.

Definition (Forward class map)

For any odd integer n, we define the forward class map by

$$\boxed{F(n) := T\big(k0\_normalize(n)\big).}$$

A single forward-class step therefore consists of two conceptually distinct operations:

1. Class projection

   $$n \;\longmapsto\; m = k0\_normalize(n),$$

   which moves n to the bottom of its 01-tower.

2. Canonical successor step

   $$m \;\longmapsto\; T(m) = \frac{3m+1}{2^{\nu_2(3m+1)}}.$$

Since T is constant on predecessor classes, the combined map F captures exactly how forward Collatz dynamics advance from the class of n via the successor of its canonical seed.

Forward class orbits

A forward class orbit (indexed backward in time) starting from n_0 is the sequence

$$n_0,\ n_{-1},\ n_{-2},\ \ldots$$

generated recursively by

$$n_{t-1} = F(n_t).$$

Because each step applies T only after projecting to the canonical seed, the forward dynamics collapse into a structured motion on the canonical-seed partition developed in Chapter 4. This greatly simplifies the study of forward behavior.

5.2 Type A and Type C Successors Revisited

In the forward class map

$$F(n) = T(k0\_normalize(n)),$$

the actual input to the successor step is not n itself but its **canonical base**

$$m = k0\_normalize(n).$$

From Chapter 4, every canonical base satisfies

$$\nu_2(3m + 1) \in \{1, 2\},$$

so the accelerated successor $T(m)$ always falls into one of two cases. These give the two fundamental successor types in forward class dynamics.

Type A successor $(\nu_2(3m + 1) = 2)$

If $3m + 1$ is divisible by 4 but not 8, then

$$T(m) = \frac{3m + 1}{4}.$$

A Type A step divides by $4 = 2^2$, removing **one entire 01-block's worth** of 2-adic weight from the quantity $3m + 1$. Numerically, Type A successors are typically **smaller** than their inputs.

Type C successor $(\nu_2(3m + 1) = 1)$

If $3m + 1$ is divisible by 2 but not by 4, then

$$T(m) = \frac{3m + 1}{2}.$$

A Type C step removes **only a single factor of 2**, insufficient to eliminate a full 01-block in the predecessor-class sense. Numerically, Type C successors are typically **larger** than their inputs.

Interpretation

Because normalization occurs **before** applying the successor, the exponent in

$$T(m) = \frac{3m + 1}{2^{\nu_2(3m+1)}}$$

is automatically restricted to the values 1 and 2. Thus:

- Every forward class step is unambiguously Type A or Type C.
- Type A steps produce **local descent**.
- Type C steps produce **local ascent**.

This alternating pattern of mild growth (C) and mild contraction (A) is a core feature of forward class dynamics and will play a central role in understanding the behavior of successor chains on canonical bases.

5.3 Successor Types for Canonical Terminal Seeds

In Chapter 4 we identified the canonical terminal seeds as

$$m_0 \equiv 3, 9, 15 \pmod{24}, \qquad \nu_2(3m_0 + 1) \in \{1, 2\}.$$

These three subclasses behave very differently under the *successor* map T. A direct computation shows:

- **If $m_0 \equiv 3 \pmod{24}$**, then
$$\nu_2(3m_0 + 1) = 1,$$
so the first successor is **always Type C**.
- **If $m_0 \equiv 9 \pmod{24}$**, then
$$\nu_2(3m_0 + 1) = 2,$$
so the first successor is **always Type A**.
- **If $m_0 \equiv 15 \pmod{24}$**, then
$$\nu_2(3m_0 + 1) = 1,$$
so the first successor is again **always Type C**.

Thus the three canonical seeds break naturally into two behavioral groups:

$$\text{Type C seeds:} \quad m_0 \equiv 3, 15 \pmod{24},$$
$$\text{Type A seeds:} \quad m_0 \equiv 9 \pmod{24}.$$

This division is crucial for forward-class dynamics:

- **Seeds $\equiv 9 \pmod{24}$** begin with an immediate *Type A* drop.
- **Seeds $\equiv 3$ or $15 \pmod{24}$** begin with a *Type C* expansion.

An implementation demonstrating this is provided in Appendix D.Sec-5.3.

5.4 Forward Orbits from Canonical Terminal Seeds

Backward iteration organizes the odd integers into disjoint predecessor classes

$$\mathcal{P}_{m_0}, \qquad m_0 \equiv 3,\ 9,\ 15 \pmod{24},$$

each with a unique **canonical terminal seed** m_0. Every odd integer belongs to exactly one such class. This structural classification has a striking consequence for forward dynamics:

> **Because every odd integer collapses backward to exactly one canonical terminal seed, and because every canonical terminal seed represents an entire predecessor class, proving that each canonical terminal seed eventually reaches $\mathcal{P}_1$ under forward iteration would suffice to prove the Collatz conjecture within this framework.**

Indeed, if $n \in \mathcal{P}_{m_0}$ then $F(n) = T(k0_n ormalize(n)) = T(m_0)$, so every element of $\mathcal{P}_{m_0}$ follows the same forward class orbit as its canonical terminal seed.

In this sense, **the Collatz problem reduces to understanding the forward behavior of the three canonical terminal seed families**

$$m_0 \equiv 3,\ 9,\ 15 \pmod{24}.$$

Thus it is natural — and essential — to begin forward exploration from these points.

A key structural fact from Chapter 4 is that the three canonical terminal seed families behave differently under the successor map:

- If $m_0 \equiv 3 \pmod{24}$, then $\nu_2(3m_0 + 1) = 1$, so the first step is Type C.
- If $m_0 \equiv 9 \pmod{24}$, then $\nu_2(3m_0 + 1) = 2$, so the first step is Type A.
- If $m_0 \equiv 15 \pmod{24}$, then $\nu_2(3m_0 + 1) = 1$, so the first step is Type C.

Thus only the $9 \pmod{24}$ family begins with an immediate Type A drop, while the 3 and 15 families begin with Type C expansion. This distinction strongly influences the early shape of the forward-class orbit.

Forward Class Map

Let

$$n_0 = m_0, \qquad m_0 \equiv 3, 9, 15 \pmod{24},$$

be any canonical terminal seed, and iterate the forward class map

$$n_{t-1} = F(n_t) = T\big(k0\_normalize(n_t)\big) = \frac{3\,k0\_normalize(n_t) + 1}{2^{\nu_2(3\,k0\_normalize(n_t)+1)}}.$$

This produces the **forward class orbit**

$$n_0 \longrightarrow n_{-1} \longrightarrow n_{-2} \longrightarrow \cdots.$$

Empirical Behavior

Extensive computation shows:

- **Every canonical terminal seed so far** (hundreds of thousands across all three families) eventually reaches the root class $\mathcal{P}_1$.
- Once an orbit reaches a member of $\mathcal{P}_1$, normalization sends it to the base 1, after which forward iteration stabilizes at the fixed point

$$1 \longrightarrow 1.$$

Thus forward class dynamics *appear* to follow a descending path:

$$\mathcal{P}_{m_0} \longrightarrow \mathcal{P}_{m_1} \longrightarrow \mathcal{P}_{m_2} \longrightarrow \cdots \longrightarrow \mathcal{P}_1.$$

Examples

Previously studied canonical terminal seeds illustrate this clearly:

- Starting at 51 (from the $3 \pmod{24}$ family):

  $51 \to_{\text{norm}} 51 \xrightarrow{T} 77 \to_{\text{norm}} 19 \xrightarrow{T} 29 \to_{\text{norm}} 7 \xrightarrow{T} 11 \to_{\text{norm}} 11 \xrightarrow{T} 17 \to_{\text{norm}} 17 \xrightarrow{T} 13 \to_{\text{norm}} 3 \xrightarrow{T} 5 \to_{\text{norm}} 1 \xrightarrow{T} 1.$

- Starting at 99 (also from $3 \pmod{24}$):

  $99 \to_{\text{norm}} 99 \xrightarrow{T} 149 \to_{\text{norm}} 9 \xrightarrow{T} 7 \to_{\text{norm}} 7 \xrightarrow{T} 11 \to_{\text{norm}} 11 \xrightarrow{T} 17 \to_{\text{norm}} 17 \xrightarrow{T} 13 \to_{\text{norm}} 3 \xrightarrow{T} 5 \to_{\text{norm}} 1 \xrightarrow{T} 1.$

These chains pass through several predecessor classes, each time:

1. normalizing to the base of the new class,
2. taking a successor step (Type A or C),
3. and continuing forward.

Note that in the example, both 77 and 19 have the successor 29. The $k0\_normalization$ merely gets to the base of the class; the *successor* is the same. In this way, forward dynamics move the orbit "sideways" across classes via successors and "downward" toward smaller canonical terminal seeds via normalization.

> To illustrate forward-class dynamics, we display the three representative canonical terminal seeds $75 \equiv 3 \pmod{24}$, $81 \equiv 9 \pmod{24}$, and $87 \equiv 15 \pmod{24}$, one from each seed family.
>
> The values $75, 81, 87$ are not arbitrary: they are canonical terminal seeds (one from each congruence class $3, 9, 15$ mod 24). Their forward-class orbits model the dynamics of entire predecessor classes.

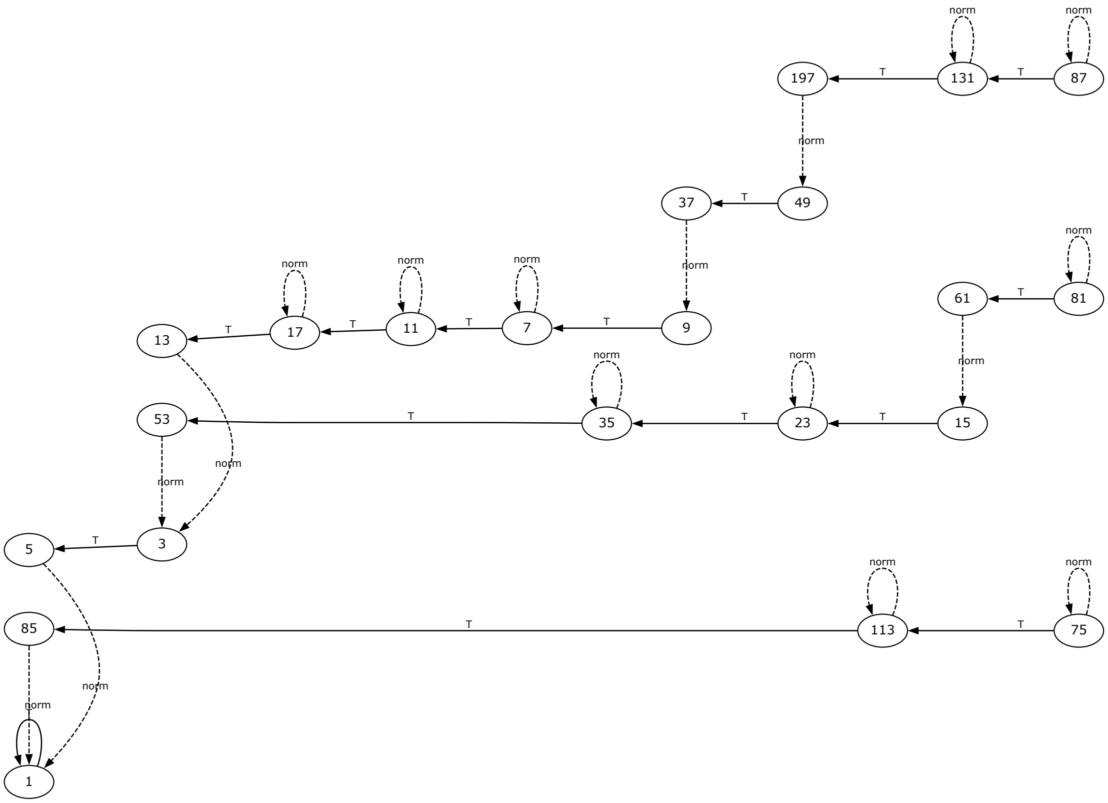

Figure 5.1. *Forward-class orbits for the canonical terminal seeds* 75*,* 81*, and* 87*.* Each of these seeds is a canonical terminal seed, representing an entire predecessor class (from the residue families 3, 9, and $15 \pmod{24}$). Every step consists of a normalization (norm) that sends a value to the base of its predecessor class, followed by a Collatz successor step (T) applied to that base. The figure shows the full forward evolution $n \to \text{norm} \to m \to T \to n'$ for the three seeds. Horizontal T-arrows move between canonical terminal seeds, while vertical norm-arrows collapse values onto their canonical representatives. All three chains ultimately funnel leftward into the class of 1, illustrating the characteristic **"sideways via T, downward via normalization"** structure of forward class orbits.

Reversing the Perspective: $\mathcal{P}_1$ as the Root Class

Although we have written forward orbits in the direction

$$\mathcal{P}_{m_0} \longrightarrow \mathcal{P}_1,$$

it is often conceptually clearer to **reverse the visual orientation** and place $\mathcal{P}_1$ on the **left**, treating it as a *root* from which all other predecessor classes extend to the right. This creates the picture:

$$\mathcal{P}_1 \longleftarrow \mathcal{P}_{m_1} \longleftarrow \mathcal{P}_{m_2} \longleftarrow \cdots \longleftarrow \mathcal{P}_{m_0}.$$

Backward iteration moves **rightward**, building the predecessor tree, while forward iteration moves **leftward**, collapsing toward $\mathcal{P}_1$. For example,

$$1 \xleftarrow{T} 1 \xleftarrow{\text{norm}} 5 \xleftarrow{T} 3 \xleftarrow{\text{norm}} 13 \xleftarrow{T} 17 \xleftarrow{\text{norm}} 17 \xleftarrow{T} 11 \xleftarrow{\text{norm}} 11 \xleftarrow{T} 7 \xleftarrow{\text{norm}} 7 \xleftarrow{T} 9 \xleftarrow{\text{norm}} 149 \xleftarrow{T} 99 \xleftarrow{\text{norm}} 99.$$

An implementation creating the above chain is provided in Appendix D.Sec-5.4.

5.5 Terminal Families, K'-Passage, and Forward Descent

Chapters 3–4 assign every odd integer n a unique **canonical terminal seed** $m_0(n) \equiv 3, 9, 15 \pmod{24}$, defined as the canonical endpoint of its odd predecessor chain. In the previous section we used $m_0(n)$ primarily as a backward classifier; here we reinterpret it dynamically as a partition of the odd integers into **terminal families**, and we refine the notion of a terminal chain τ to reflect an important subtlety:

> Forward accelerated dynamics move within a terminal family via successor steps, but 01-lift structure (the K' mechanism) causes the orbit to *pass through intermediate families* before a fully compressed normalization landing occurs.

Thus τ is not merely "family drops via compressed k_0-normalization." It is the **ordered family sequence actually traversed** by the accelerated orbit, including intermediate canonical families crossed during K'-runs.

Terminal families

Let $m_0 \equiv 3, 9, 15 \pmod{24}$ be a canonical terminal seed. The **terminal family** associated with m_0 is

$$\mathcal{F}(m_0) := \{ n \in \mathbb{Z}_{\mathrm{odd}} : m_0(n) = m_0 \}.$$

Equivalently, $\mathcal{F}(m_0)$ consists of all odd integers whose backward odd-predecessor chain terminates at the same canonical terminal seed m_0.

From the non-intersection of predecessor trees (Chapter 4), it follows that:

- every odd integer belongs to exactly one terminal family, and
- distinct canonical seeds label disjoint families.

Hence the families $\mathcal{F}(m_0)$ form a partition of $\mathbb{Z}_{\mathrm{odd}}$, and m_0 may be viewed as a **family label**.

In forward iteration, successor steps T often preserve family membership for long stretches; changes in canonical identity occur only when the orbit encounters 01-lift structure that forces passage to a different predecessor base.

Equivalently, terminal families define discrete levels in the accelerated Collatz graph: forward dynamics descend by moving between families, while successor steps move within a fixed level.

Family endpoints and the 5 (mod 8) boundary

Each terminal family $\mathcal{F}(m_0)$ contains a distinguished boundary element at which 01-lift structure becomes unavoidable.

We define the **family endpoint** $e(m_0)$ to be the unique element of $\mathcal{F}(m_0)$ that satisfies $e(m_0) \equiv 5 \pmod{8}$ and which appears as the terminal element of the decoded (A/C)-tail associated with m_0 (Appendix A).

The congruence condition $e(m_0) \equiv 5 \pmod{8}$ is decisive: elements in this residue class necessarily trigger nontrivial 01-lift removal. Consequently:

- in **forward dynamics**, $e(m_0)$ marks the first point at which the orbit is forced to engage K' structure and hence cannot remain confined to the same canonical family label indefinitely;
- in **backward decoding**, the same value appears as the entry point of the terminal tail that generates the family above m_0.

Thus $e(m_0)$ serves as the canonical **exit representative** of the family.

Caveat (logical completeness).
The definition presupposes that forward dynamics within $\mathcal{F}(m_0)$ eventually encounter an element congruent to $5 \pmod 8$. No counterexamples have been observed in the verified computational regime used throughout this work; all examples and constructions are restricted to this empirically supported domain.

K'-passage and expanded normalization

A key point is that a single k_0-normalization step generally removes *multiple* 01-blocks at once. Symbolically, this corresponds to collapsing an entire run of K' steps into a single jump. However, each individual K' removal corresponds to removing exactly one trailing 01 block. At the level of bases, this exposes intermediate predecessor bases — and therefore may traverse intermediate canonical families. To expose this structure, we introduce the **one-block normalization map κ**: given an odd integer m, write $3m + 1 = 2^r \cdot s$ with s odd. If $r \geq 2$, define

$$\kappa(m) = m_0\left(\frac{3m+1}{4}\right),$$

and if $r < 2$ set $\kappa(m) = m$. Iterating κ yields the **expanded normalization chain**

$$m \to \kappa(m) \to \kappa^2(m) \to \cdots,$$

which terminates at the same compressed landing one would obtain by removing all available 01 blocks at once (i.e., the k_0-compressed landing).

Interpretation:

- $k0\_normalize(m)$ is a **compressed landing site** after removing all available 01-blocks (up to the canonical parity rule),
- the iterates of κ reveal the **intermediate bases passed through** during that compression.

This is precisely where intermediate canonical families arise.

Terminal chains as traversed family chains

Let n be an odd integer. Consider the **true accelerated forward orbit** under successor steps T:

$$n,\ T(n),\ T^2(n),\ \ldots$$

At each visited odd integer x we evaluate its canonical family label $m_0(x)$, and record the sequence of distinct labels encountered in order. We define the **terminal chain** (or **τ-chain**) of n to be the ordered list of distinct canonical seeds encountered along the accelerated orbit:

$$\tau(n) = [\,m_0(n),\ m_0(T(n)),\ m_0(T^2(n)),\ \ldots,\ 1\,]$$

with repeated consecutive labels suppressed. Equivalently, $\tau(n)$ is the family path actually traversed by forward accelerated dynamics. It ignores repeated motion inside a fixed family label, but it does *not* ignore intermediate families crossed during K'-passage. Two consequences follow.

- (i) **Canonical dependence.** If $m_0(n) = m_0(n')$, then n and n' lie in the same terminal family and $\tau(n) = \tau(n')$.
- (ii) **Structural meaning.** τ is the canonical-level itinerary of the accelerated orbit: a record of the successive *family levels* traversed on the way to the forward-terminal level.

Example: the family chain of 159 and 607

The refined interpretation is visible in the case of $\tau(159)$ and $\tau(607)$. Since $m_0(607) = 159$, both integers belong to the same terminal family $\mathcal{F}(159)$ and therefore share the same τ-chain. Explicit computation yields:

$$\tau(159) = [159, 303, 81, 15, 9, 3, 1].$$

The interpretation is:

- the orbit descends through families $\mathcal{F}(159) \to \mathcal{F}(303) \to \mathcal{F}(81) \to \mathcal{F}(15)$,
- but the subsequent descent involves a nontrivial K' passage,
- and during that passage the orbit traverses the intermediate canonical family $\mathcal{F}(9)$ before entering $\mathcal{F}(3)$ and finally the forward-terminal level 1.

Figures 5.2–5.3 illustrate the same phenomenon visually: distinct numerical trajectories may differ substantially, yet the induced canonical family traversal recorded by τ is invariant across all members of the same terminal family.

```
tau(159) = [159, 303, 81, 15, 9, 3, 1]
```

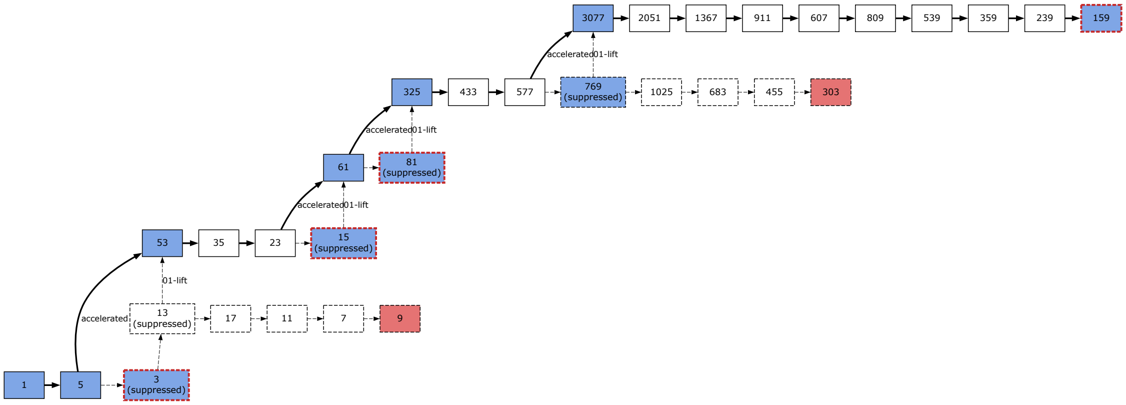

Figure 5.2. *Family-level descent with K' passage along the accelerated forward orbit of the canonical terminal seed* 159. Horizontal arrows correspond to successor steps T taken along the true accelerated orbit. Each visited value is labeled by its canonical family seed, and repeated family labels are suppressed to expose the traversed itinerary. The dashed segment highlights a nontrivial K' passage: a single compressed `k0_normalize` landing would skip intermediate bases, but the accelerated orbit passes through the intermediate family labeled 9 before entering the family labeled 3. Reading the distinct family labels in order yields $\tau(159) = [159, 303, 81, 15, 9, 3, 1]$.

```
tau(607) = [159, 303, 81, 15, 9, 3, 1]
```

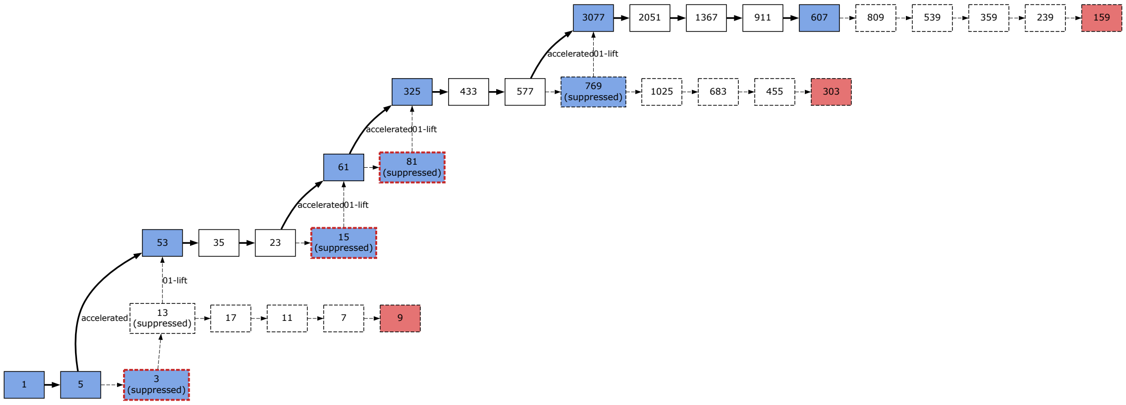

Figure 5.3. *Distinct forward orbits with identical traversed family chains: the case of* 607*.* Although the explicit accelerated orbit of 607 differs from that of 159 at the level of visited values, both integers have the same canonical terminal seed (159) and hence belong to the same terminal family. Consequently, the accelerated orbit traverses the same canonical family itinerary, including the same intermediate family encountered during K' passage, yielding $\tau(607) = \tau(159) = [159, 303, 81, 15, 9, 3, 1]$. This illustrates that τ-chains are family-level invariants: all integers in the same terminal family share the same traversed family sequence, even when their pointwise trajectories differ.

Thus the meaningful invariant is the traversed family chain

$$\mathcal{F}(159) \to \mathcal{F}(303) \to \mathcal{F}(81) \to \mathcal{F}(15) \to \mathcal{F}(9) \to \mathcal{F}(3) \to \mathcal{F}(1),$$

and not merely the compressed landing sequence obtained by applying k_0-normalization in one jump.

A reference implementaion to generate the τ-chains for a given odd integer, is provided in Appendix D.Sec-5.5.

5.6 Reverse τ-Chains and the Terminal Family Tree

Section 5.5 established that accelerated Collatz dynamics may be described as motion through **terminal families**, viewed as discrete canonical levels. Successor steps move an orbit within a fixed family, while K'-passage forces the orbit to traverse successive families until the forward-terminal level is reached. The resulting terminal chain $\tau(n)$ records the **ordered sequence of terminal families actually traversed** by the accelerated orbit of n.

In this section we reverse that perspective. By reversing terminal chains and assembling them across all canonical seeds, we obtain a global hierarchical structure — the **terminal family tree** — that organizes Collatz dynamics at the family level.

Reverse τ-chains

Let $m_0 \equiv 3, 9, 15 \pmod{24}$ be a canonical terminal seed. Recall from Section 5.5 that the terminal chain of any n with $m_0(n) = m_0$ is

$$\tau(m_0) = [\, m_0,\ m_1,\ m_2,\ \dots,\ 1\,],$$

where each m_i is the canonical seed of a terminal family actually traversed by the accelerated orbit.

We define the **reverse τ-chain** associated with m_0 to be the reversed list

$$\tau(m_0)^{\mathrm{rev}} = [\, 1,\ \dots,\ m_2,\ m_1,\ m_0\,].$$

Interpretationally, a reverse τ-chain records the sequence of **ancestor families** encountered when tracing the family-level structure upward from the forward-terminal level. Equivalently, reverse τ-chains encode the **unique family-level path** connecting a given terminal family to the root family $\mathcal{F}(1)$. Two key points follow immediately:

- reverse τ-chains are defined purely at the level of canonical family labels,
- they are independent of individual numerical trajectories within each family.

Thus reverse τ-chains capture **structural ancestry**, not numerical motion.

The terminal family tree

Consider the directed graph defined as follows:

- **Nodes:** terminal families $\mathcal{F}(m_0)$, labeled by canonical seeds m_0,
- **Edges:** a directed edge from $\mathcal{F}(a)$ to $\mathcal{F}(b)$ if b immediately follows a in some reverse τ-chain.

By construction, this graph has a distinguished root corresponding to the forward-terminal family $\mathcal{F}(1)$. Every reverse τ-chain traces a directed path starting at this root and ending at a unique terminal family. No reverse τ-chain revisits the same family, since canonical seeds are unique. This directed graph is therefore a **rooted tree**, which we call the **terminal family tree**. Within this structure:

- each terminal family occupies a unique position in the tree,
- each reverse τ-chain corresponds to a unique root-to-node path,
- shared prefixes of reverse τ-chains correspond to shared canonical ancestry.

The terminal family tree thus provides a global organization of terminal families by **canonical depth**.

Empirical structure of family hierarchies

Computed reverse τ-chains exhibit a rich yet orderly hierarchical structure.

Empirically observed features include:

- **branching:** multiple terminal families share common initial ancestor sequences,
- **variable depth:** different families lie at different canonical heights above the root,
- **recurrent motifs:** certain family subsequences recur across multiple branches.

Within the verified computational range, no cycles have been observed.

All reverse τ-chains form simple paths terminating at the root 1, and every observed terminal family admits a well-defined ancestor chain.

Accordingly, within the verified domain, the terminal family tree is well-defined, acyclic, and globally connected.

Canonical nature of τ-chains

A central consequence of the family-level framework is that τ-chains are **canonical-level invariants**. Specifically:

- every odd integer belongs to exactly one terminal family $\mathcal{F}(m_0)$,
- all members of a given family share the same terminal chain,
- τ-chains therefore depend only on family membership, not on individual orbits.

Forward accelerated dynamics correspond to **descent along a path** in the terminal family tree, while reverse τ-chains encode the **unique ancestral path** back to the root. In this sense, the accelerated Collatz map induces a structured flow on the terminal family tree, with K'-passage providing the mechanism by which orbits move between adjacent family levels.

Structural interpretation

The terminal family tree provides a unified view of forward and backward Collatz structure.

- **Backward:** predecessor trees partition odd integers into terminal families labeled by canonical seeds.
- **Forward:** accelerated dynamics traverse these families in a level-by-level manner.
- **Global:** the resulting family traversals assemble into a rooted hierarchical tree with base 1.

A crucial structural feature of this framework is that a family drop under K' does not correspond to numerical descent, but to an irreversible reduction in symbolic degrees of freedom. While the numerical values encountered along an accelerated orbit may increase during K'-passage, the associated terminal family occupies a strictly lower position in the canonical hierarchy. In this sense, accelerated Collatz dynamics are strictly descending in canonical depth even when they are not numerically monotone. The terminal family tree makes this distinction explicit: numerical motion occurs within families, while structural descent occurs only through normalization-induced family transitions.

Within this framework, universal forward convergence corresponds to the statement that **every terminal family lies on a branch connected to the root**. While this observation does not constitute a proof of convergence, it isolates a precise structural condition whose verification would suffice within the canonical framework developed here. The terminal family tree thus serves not merely as a visualization, but as a meaningful structural object encoding coarse-grained Collatz dynamics.

Demonstration (computed examples)

The following examples list reverse τ-chains associated with selected canonical seeds $n_t \equiv 3, 9, 15 \pmod{24}$. Any odd integer whose canonical seed is n_t shares the same terminal chain.

```
n_t =     3  rev_tau = [1, 3]
n_t =     9  rev_tau = [1, 3, 9]
n_t =    15  rev_tau = [1, 3, 9, 15]
n_t =    27  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27]
n_t =    33  rev_tau = [1, 3, 9, 33]
n_t =    39  rev_tau = [1, 3, 9, 33, 39]
n_t =    51  rev_tau = [1, 3, 9, 33, 51]
n_t =    57  rev_tau = [1, 3, 9, 57]
n_t =    63  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27, 63]
n_t =    75  rev_tau = [1, 75]
n_t =    81  rev_tau = [1, 3, 9, 15, 81]
n_t =    87  rev_tau = [1, 3, 9, 57, 87]
n_t =    99  rev_tau = [1, 3, 9, 57, 99]
n_t =   105  rev_tau = [1, 3, 9, 33, 39, 105]
n_t =   111  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111]
n_t =   123  rev_tau = [1, 3, 9, 33, 39, 123]
n_t =   129  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27, 129]
n_t =   135  rev_tau = [1, 3, 9, 57, 135]
n_t =   147  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27, 129, 147]
n_t =   153  rev_tau = [1, 3, 9, 57, 153]
n_t =   159  rev_tau = [1, 3, 9, 15, 81, 303, 159]
n_t =   171  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27, 171]
n_t =   177  rev_tau = [1, 3, 9, 33, 177]
n_t =   183  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27, 183]
n_t =   195  rev_tau = [1, 3, 9, 15, 81, 303, 159, 111, 27, 129, 195]
n_t =   201  rev_tau = [1, 201]
...
```

Each line corresponds to a unique strictly ordered path in the terminal family tree. For example, since $m_0(607) = 159$, we have

$$\tau(607) = \tau(159) = [159, 303, 81, 15, 9, 3, 1],$$

and hence

$$\tau(159)^{\text{rev}} = [1, 3, 9, 15, 81, 303, 159].$$

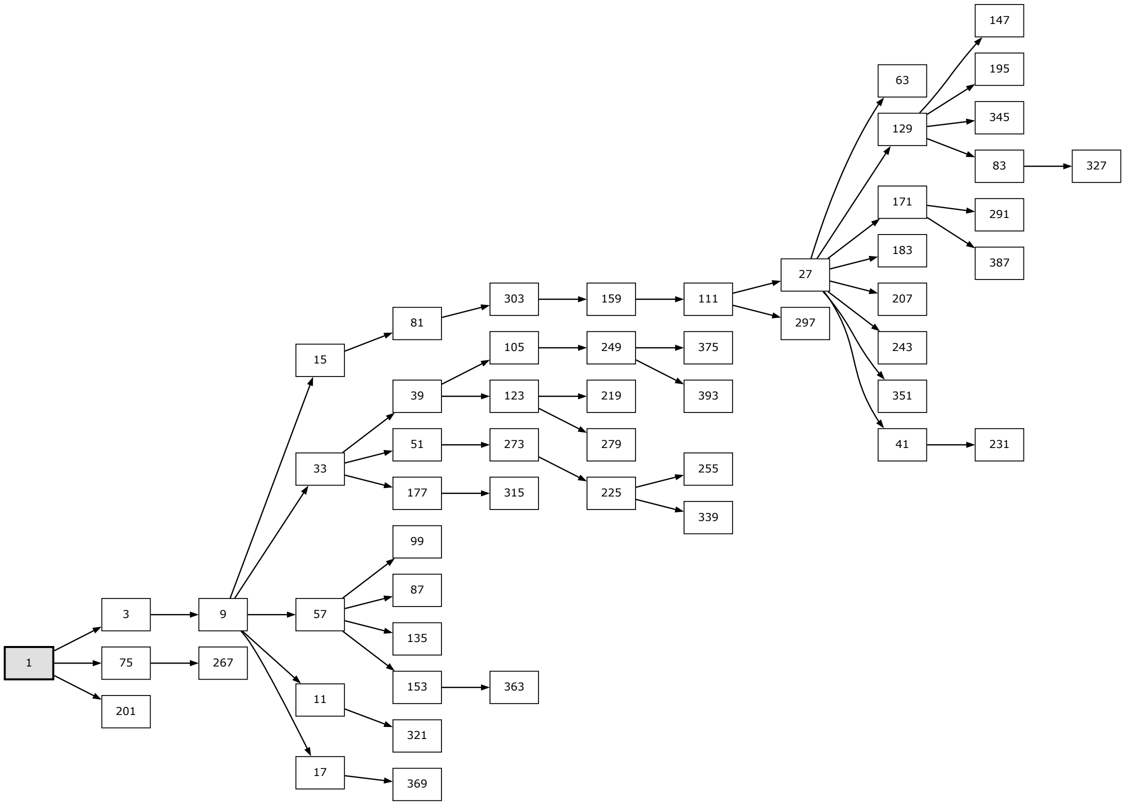

Figure 5.4. *The terminal family tree induced by reverse τ-chains.* Each node represents a terminal family labeled by its canonical seed. Directed edges encode immediate ancestry relations induced by reverse τ-chains. The root corresponds to the forward-terminal family labeled 1. Each path from the root to a node records the reverse τ-chain of that family, while shared prefixes indicate

shared canonical ancestry. The resulting structure is a rooted tree that organizes accelerated Collatz dynamics at the level of terminal families rather than individual orbits.

5.7 Refined Step Types and Signature Convention

The analysis so far has described Collatz dynamics in terms of **predecessor classes**, **canonical terminal seeds**, and **terminal chains**. To move from this structural classification to an explicit encoding of **forward orbits**, we now introduce a refined symbolic language that treats binary 01-lifts as first-class operations. This section establishes the step vocabulary and signature convention that will be used throughout the remainder of the paper.

Backward step types on odd integers

For an odd integer n, define the following backward steps whenever the result is integral:

- **A-step**: $A(n) = (4n - 1)/3$
- **C-step**: $C(n) = (2n - 1)/3$
- **K-step**: $K(n) = 4n + 1$

The A- and C-steps coincide with the predecessor moves developed in Chapter 4. The K-step represents a single binary 01-lift. Repeated application of K generates the familiar infinite 01-towers above a base odd integer. By introducing K as an explicit symbolic step, we separate the combinatorial structure of 01-lifts from the arithmetic behavior of $3n + 1$. This unified step vocabulary allows all backward structure to be recorded symbolically.

Structural signature

In §4.12, the **extended signature** of an odd integer was defined as a triple (ℓ, w, m_0), where the parameter ℓ recorded the number of 01-lifts applied to a canonical base. We now adopt a streamlined but equivalent convention that absorbs all lift information directly into the symbolic word.

Definition (structural signature)

The **structural signature** of an odd integer is the pair (w, m_0), where:

- w is a finite word over the alphabet $\{A, C, K\}$,
- m_0 is the canonical terminal seed obtained by applying the word w (read from right to left) as a backward composition.

Each occurrence of K in the word represents a single binary 01-lift. Thus all information previously carried by the lift parameter ℓ is now recorded explicitly in the word itself. For a non-terminal B integer, the structural signature has the schematic form

$$(w, m_0) = (\mathsf{KAC}\cdots, m_0)$$

for illustration. This convention does not alter any results from Chapter 4. It merely records the same predecessor structure in a uniform symbolic form that is better suited for forward analysis.

Successor steps on odd integers

To prepare for the construction of forward orbit codes, we define successor counterparts to the backward steps above. For an odd integer n, define:

- A'**-step**: $A'(n) = (3n+1)/4$
- C'**-step**: $C'(n) = (3n+1)/2$
- K'**-step**: $K'(n) = (n-1)/4$

Each successor step maps odd integers to odd integers whenever it is defined. The K'-step removes a single trailing binary $\mathtt{01}$ block and is the inverse operation to the K-step. In this way, K and K' encode vertical motion along $\mathtt{01}$-towers, while A' and C' describe horizontal motion between predecessor classes.

In the next section, these successor steps will be assembled into a deterministic successor transform. This will allow the forward orbit of any canonical terminal seed m_0 to be encoded as a symbolic **orbit code**, forming the basis for all subsequent constructions.

A reference implementaion to generate the structural signature for a given odd integer, is provided in Appendix D.Sec-5.7.

5.8 Orbit Codes for Forward Successor Orbits

The refined step vocabulary and structural signature introduced in §5.7 allow the forward dynamics of the accelerated Collatz map to be described by a single deterministic transformation on the odd integers.

In this section we define this transformation explicitly and show how its iterates give rise to a finite symbolic encoding, called the **orbit code**, which records the entire forward successor orbit of an odd integer. This encoding is exact and reversible: from the orbit code alone one can reconstruct the full accelerated predecessor chain from 1 up to the original integer.

The successor transform on odd integers

Let A', C', and K' be the successor steps defined in §5.7.3:

$$A'(n) = (3n+1)/4, \quad C'(n) = (3n+1)/2, \quad K'(n) = (n-1)/4.$$

These steps are assembled into a single successor transform on odd integers.

Definition (successor transform)

Define s on the set of odd integers by

$$s(n) = \begin{cases} A'(n) & n \bmod 8 = 1, \\ C'(n) & n \bmod 8 \in \{3,7\}, \\ K'(n) & n \bmod 8 = 5. \end{cases}$$

Since every odd integer satisfies $n \bmod 8 \in \{1,3,5,7\}$, this definition assigns exactly one successor value to each odd n. Each branch is integral and preserves odd parity, so s defines a deterministic self-map on the odd integers.

In particular, $1 \bmod 8 = 1$ and $A'(1) = 1$, so 1 is a fixed point of s.

Origin of the residue–based successor rule

The definition of the successor transform s via residues modulo 8 is not ad hoc. It arises naturally from the interaction between predecessor classes, 01-tower structure, and the arithmetic of $3n+1$. Every odd integer satisfies $n \bmod 8 \in \{1,3,5,7\}$, and each residue class corresponds to a distinct structural situation.

First, consider integrality constraints. For odd n:

- $(3n+1)/4$ is integral if and only if $n \bmod 8 = 1$,
- $(3n+1)/2$ is integral if and only if $n \bmod 8 \in \{3,7\}$,
- $(n-1)/4$ is integral if and only if $n \bmod 8 = 5$.

Thus the three successor expressions A', C', and K' naturally partition the odd integers by residue modulo 8.

Second, this partition aligns exactly with the predecessor-class structure described earlier. Canonical terminal seeds satisfy

$$m_0 \equiv 3, 9, 15 \pmod{24}.$$

Reducing modulo 8, we obtain:

- $3 \pmod{24}$ and $15 \pmod{24}$ correspond to $3 \pmod 8$ and $7 \pmod 8$, and therefore always produce a C' step,
- $9 \pmod{24}$ corresponds to $1 \pmod 8$ and therefore always produces an A' step.

In contrast, the non-terminal B values satisfy

$$n \equiv 21 \pmod{24},$$

which implies $n \bmod 8 = 5$. These integers lie strictly above a canonical base in a 01-tower and therefore admit a K' step removing a single trailing binary 01 block. Such values cannot be terminal.

Equivalently, the forward-terminal class $\mathcal{P}_1$ consists entirely of integers congruent to $5 \pmod 8$, and every such value must first descend along a 01-tower before any A' or C' successor motion can occur.

Thus the residue-based definition of s reflects the canonical structure of predecessor classes and 01-towers already present in the accelerated Collatz graph, rather than introducing any new case distinctions.

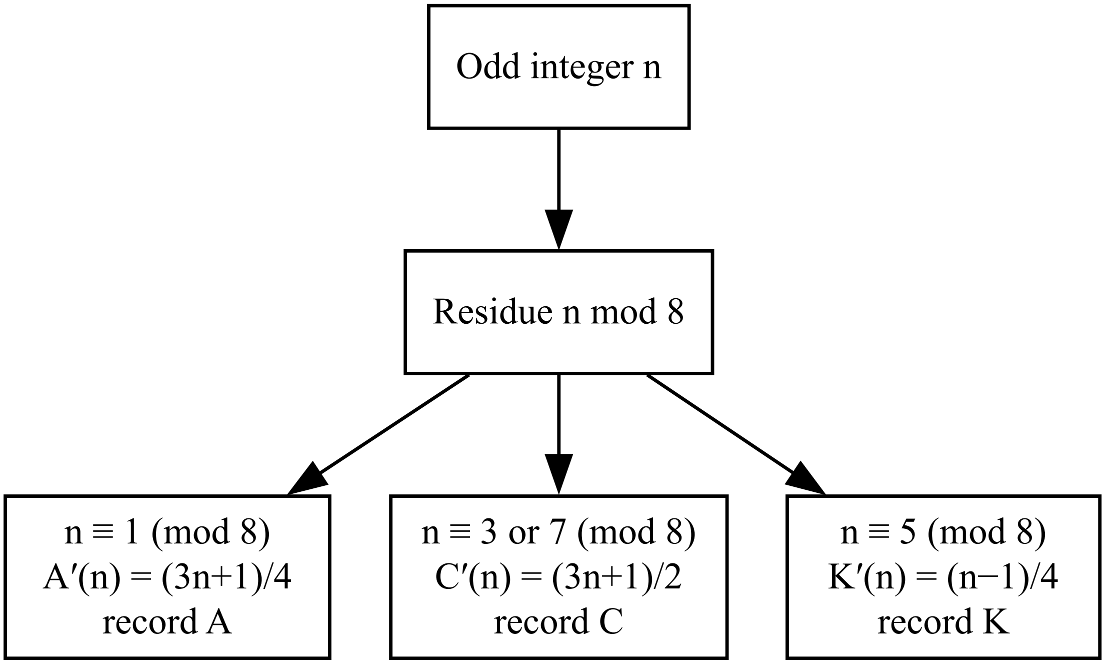

Figure 5.5. *Residue-based routing of the successor transform on odd integers.* Every odd integer satisfies *n mod 8* ∈ *{1,3,5,7}*, and this residue uniquely determines the successor branch applied by the transform s. Values congruent to 1 mod 8 follow the A' branch, values congruent to 3 or 7 mod 8 follow the C' branch, and values congruent to 5 mod 8 follow the K' branch, which removes a trailing binary 01 block. This residue-based partition arises directly from integrality constraints and aligns exactly with the predecessor-class and 01-tower structure of the accelerated Collatz graph.

Remark (Interaction of the three structural mechanisms).

The accelerated Collatz dynamics on odd integers are governed by the interaction of three distinct mechanisms already introduced.

- First, **backward predecessor structure** is determined by residue modulo 6, via

$$f(n) = \begin{cases} \frac{4n-1}{3}, & n \equiv 1 \pmod{6}, \\ \frac{2n-1}{3}, & n \equiv 5 \pmod{6}, \\ \emptyset, & n \equiv 3 \pmod{6}, \end{cases}$$

  yielding the predecessor type classification A, C, and B.

- Second, **forward successor routing** is determined by residue modulo (8), via

$$s(n) = \begin{cases} A'(n), & n \equiv 1 \pmod{8}, \\ C'(n), & n \equiv 3,7 \pmod{8}, \\ K'(n), & n \equiv 5 \pmod{8}, \end{cases} \qquad \begin{aligned} A'(n) &= \frac{3n+1}{4}, \\ C'(n) &= \frac{3n+1}{2}, \\ K'(n) &= \frac{n-1}{4}. \end{aligned}$$

- Third, vertical motion along 01-towers is governed by the 01-extension map $E(n) = 4n + 1$, under which types cycle rigidly as $A \to C \to B \to A$. In particular, whenever both ($n \equiv 5 \bmod 8$) and the corresponding backward obstruction occurs ($n \equiv 3 \bmod 6$), the backward predecessor motion along the same 01-tower is given by the K-step $n \mapsto 4n + 1$.

Structurally, the integer 1 satisfies the A-branch residue conditions in both the successor and predecessor parametrizations; its exclusion from further branching reflects a terminal convention rather than an algebraic obstruction.

These mechanisms act in concert. Backward obstructions arise only at type B values ($n \equiv 3 \bmod 6$). Forward routing then determines whether such an obstruction persists or is removed: if $n \bmod 8 = 5$, a single K' step descends the 01-tower and, by the ACB cycle, converts the value to type A; otherwise no K' descent is available and the value is terminal. Consequently, at most one 01-lift is ever structurally required, explaining why the lift-length parameter ℓ appearing in structural signatures satisfies $\ell \in \{0, 1\}$.

In particular, an integer satisfying $n \equiv 3 \bmod 6$ but not $n \equiv 5 \bmod 8$ is simultaneously of type B (hence admits no backward predecessor) and admits no K ascent along a 01-tower; such values therefore have no backward continuation and are exactly the canonical terminal seeds $n \equiv 3, 9, 15 \bmod 24$.

Viewed in this way, specific residue coincidences such as the characterization of canonical terminal seeds arise as consequences of the compatibility between these three mechanisms, while the remaining residue combinations reflect genuine degrees of freedom rather than unresolved structure.

Remark (Deriving s' from s via inverse branches)

The backward map s' is not an independent construction, but is obtained by algebraically inverting each branch of the forward successor map s.

This derivation plays no essential role in the development of the paper and is included solely to clarify the relationship between forward routing and backward predecessor structure.

- **Inverse of the A'-branch.**
  - From $A'(x) = (3x + 1)/4$, set $n = (3x + 1)/4$, giving $x = (4n - 1)/3$.
  - Integrality requires $4n - 1 \equiv 0 \pmod 3$, hence $n \equiv 1 \pmod 3$.
  - Since n is odd, this is equivalent to $n \equiv 1 \pmod 6$.
  - Thus $A(n) = (4n - 1)/3$ for $n \equiv 1 \pmod 6$.
- **Inverse of the C'-branch.**
  - From $C'(x) = (3x + 1)/2$, set $n = (3x + 1)/2$, giving $x = (2n - 1)/3$.
  - Integrality requires $2n - 1 \equiv 0 \pmod 3$, hence $n \equiv 2 \pmod 3$.
  - Since n is odd, this is equivalent to $n \equiv 5 \pmod 6$.
  - Thus $C(n) = (2n - 1)/3$ for $n \equiv 5 \pmod 6$.
- **Inverse of the K'-branch.**
  - From $K'(x) = (x - 1)/4$, set $n = (x - 1)/4$, giving $x = 4n + 1$.
  - The forward admissibility condition is $x \equiv 5 \pmod 8$, which yields
  - $4n + 1 \equiv 5 \pmod 8$, equivalently $n \equiv 1 \pmod 2$.
  - Thus $K(n) = 4n + 1$ for odd n.

Collecting the inverse branches gives

$$s'(n) = \begin{cases} \dfrac{4n-1}{3}, & n \equiv 1 \pmod{6}, \\ \dfrac{2n-1}{3}, & n \equiv 5 \pmod{6}, \\ 4n+1, & n \equiv 1 \pmod{2}. \end{cases}$$

The parity condition in the K-branch reflects algebraic invertibility of the forward normalization step K', while the residue condition $n \equiv 3 \pmod{6}$ appearing in the predecessor map f expresses a genuine backward obstruction. Imposing termination upon first entry into $n \equiv 3 \pmod{6}$ eliminates the K-branch and recovers exactly the predecessor structure encoded by f. In this sense,

$$f = s' \text{ with termination at the } B\text{-class.}$$

Forward successor orbits

Let n be any odd integer. Starting from n, we iterate the successor transform s to obtain a forward successor orbit

$$n = x_0, \quad x_1 = s(x_0), \quad x_2 = s(x_1), \ \ldots$$

In all verified cases, this orbit reaches 1 in finitely many steps, yielding a finite orbit

$$n \to x_1 \to x_2 \to \cdots \to x_r = 1.$$

When (w, m_0) is the structural signature of n, the successor orbit starting at m_0 is a tail of the orbit starting at n, since both orbits eventually enter the same terminal levels.

Conceptually, the successor dynamics decompose into two types of motion:

- A' and C' steps correspond to local successor motion governed by the arithmetic of $3n+1$,
- K' steps remove a single trailing binary 01 block and correspond to descent along a 01-tower.

Thus iterating s produces a canonical forward orbit on odd integers that is compatible with the canonical terminal tree described in §5.6.

Definition of the orbit code

Each step of the successor orbit corresponds to one of the three successor branches of s. We record this branch information symbolically.

Definition (orbit code)

Let

$$n \to x_1 \to x_2 \to \cdots \to x_r = 1$$

be the forward successor orbit obtained by iterating s from an odd integer n. For each transition $x_i \to x_{i+1}$, record a symbol according to the unique branch of s applied at x_i:

- record A if $x_i \bmod 8 = 1$,
- record C if $x_i \bmod 8 \in \{3, 7\}$,

- record K if $x_i \bmod 8 = 5$.

This produces a finite word

$$u = u_0 u_1 \cdots u_{r-1}$$

over the alphabet $\{A, C, K\}$. The **orbit code** of n is defined to be the reversed word

$$\omega(n) = u_{r-1} \cdots u_1 u_0.$$

The reversal ensures that the code can be decoded naturally from left to right, starting from the distinguished base value 1.

Decoding rule

The orbit code $\omega(n)$ is interpreted as a sequence of backward steps applied to the base value 1. Starting from $m = 1$, read $\omega(n)$ from left to right and apply:

- A as $m \mapsto (4m - 1)/3$,
- C as $m \mapsto (2m - 1)/3$,
- K as $m \mapsto 4m + 1$.

After the final symbol is applied, the resulting value is n. In this way, decoding reconstructs the accelerated predecessor chain from 1 up to n.

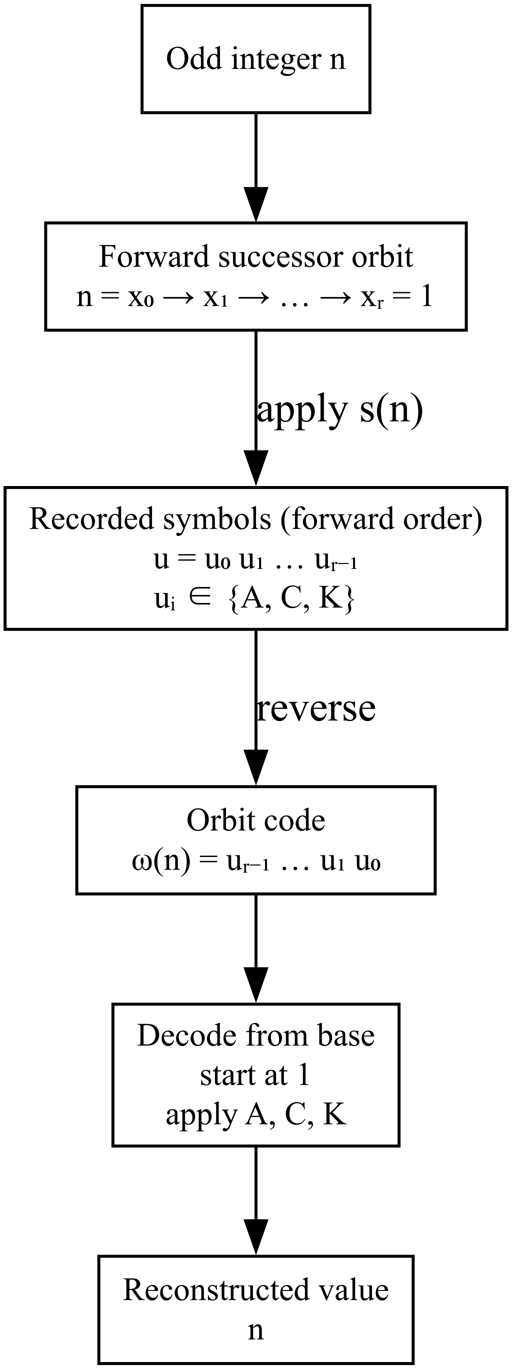

Figure 5.6. *Encoding and decoding of orbit codes via the successor transform.* Starting from an odd integer n, the deterministic successor transform s generates a forward successor orbit $n \to x_1 \to \cdots \to 1$ on the odd integers. At each step, the residue of the current value modulo 8 determines a unique successor branch, recorded symbolically as A, C, or K. The resulting word is reversed to form the orbit code $\omega(n)$, which is decoded from the distinguished base value 1 by applying the corresponding backward steps. This process reconstructs the full accelerated predecessor chain from 1 up to n, showing that orbit codes provide a finite, exact, and reversible symbolic encoding of forward successor dynamics.

Fundamental properties

The orbit code $\omega(n)$ has the following properties:

1. **Finiteness:** The code is finite for every odd integer n in the verified range.
2. **Determinism:** The successor transform s produces a unique orbit code for each n.
3. **Reversibility:** Decoding $\omega(n)$ reconstructs the exact accelerated predecessor chain of n.
4. **Structural completeness:** All predecessor types and all 01-tower motion are recorded explicitly via the symbols A, C, and K.

No normalization procedures, lift counters, or auxiliary parameters are required.

Relation to structural signatures

The structural signature (w, m_0) records how n is constructed from its canonical terminal seed m_0 by backward steps.

The orbit code $\omega(n)$ records the complementary information: the entire forward successor orbit of n terminating at 1 under the deterministic transform s.

Together, the structural signature and the orbit code provide two independent symbolic views of the accelerated Collatz dynamics associated with n: one describing the backward construction of n from m_0, and the other describing the forward successor descent of n to 1.

Examples

We illustrate the construction and interpretation of orbit codes with two explicit examples.

Example 1: Orbit code and decoding

Consider the odd integer $n = 51$. Applying the successor transform s repeatedly yields the forward successor orbit

$$51 \to 77 \to 19 \to 29 \to 7 \to 11 \to 17 \to 13 \to 3 \to 5 \to 1.$$

Recording the successor branches according to the rule of §5.8.3 produces the orbit code

$$\omega(51) = \text{KCKACCKCKC}.$$

Decoding this word from the base value 1 by applying the backward steps

$$A:\ m \mapsto (4m-1)/3, \quad C:\ m \mapsto (2m-1)/3, \quad K:\ m \mapsto 4m+1$$

reconstructs the accelerated predecessor chain

$$1 \to 5 \to 3 \to 13 \to 17 \to 11 \to 7 \to 29 \to 19 \to 77 \to 51.$$

The decoded chain terminates exactly at $n = 51$, demonstrating that $\omega(51)$ is a complete and reversible encoding of the forward successor dynamics.

Example 2: Consistency with structural signatures

Now consider the odd integer $n = 109$. Its structural signature is

$$(w, m_0) = (\text{AAAC}, 171).$$

Starting from the canonical terminal seed $m_0 = 171$ and applying the inverse successor steps A', C', and K' in reverse order according to the word w yields

$$171 \xrightarrow{C'} \cdots \xrightarrow{A'} 109.$$

Thus the structural signature reconstructs n exactly, independent of the orbit code. This example highlights the complementary roles of the two symbolic representations:

- the structural signature (w, m_0) encodes the backward construction of n from its canonical terminal seed,
- the orbit code $\omega(n)$ encodes the forward successor descent of n to 1.

Together, they provide two independent and mutually consistent symbolic descriptions of the accelerated Collatz dynamics associated with n.

A reference implementation to construct and decode orbit codes is provided in Appendix D.Sec-5.8.

5.9 A Database of Terminal Orbit Codes

The results of Chapters 4–5 establish a precise global structure for the accelerated Collatz dynamics on odd integers.

Every odd integer n admits a unique forward successor orbit under the deterministic transform s defined in §5.8.1. This orbit is finite, acyclic, and terminates at 1. Along the way, it passes through a unique **canonical terminal seed** m_0 satisfying

$$m_0 \equiv 3, 9, 15 \pmod{24}.$$

Equivalently, every odd integer lies on exactly one successor orbit that passes through a canonical terminal seed m_0 and terminates at the fixed point 1. By §5.8, this entire forward successor orbit is encoded symbolically by the **orbit code** $\omega(n)$, a finite word over the alphabet $\{A, C, K\}$. In particular, the orbit codes $\omega(m_0)$ of the canonical terminal seeds play a distinguished role.

Terminal orbit codes

For each canonical terminal seed

$$m_0 \equiv 3, 9, 15 \pmod{24},$$

we compute its orbit code $\omega(m_0)$ by iterating the successor transform s from m_0 down to 1 and recording the corresponding successor symbols, as defined in §5.8.3. Because the classical Collatz conjecture has been verified computationally for all integers below 2^{71} (Barina, 2025), every odd integer $n < 2^{71}$ reaches 1 under accelerated iteration. Consequently:

- every such n lies on the successor orbit of a unique canonical terminal seed m_0, and
- every such orbit is fully represented by the terminal orbit code $\omega(m_0)$.

We therefore define the **terminal orbit code database** to be the finite collection

$$\{\, \omega(m_0) \;:\; m_0 \equiv 3, 9, 15 \pmod{24} \,\}.$$

Within the verified range $n < 2^{71}$, this database is complete. We refer to this collection as the **Terminal Orbit Code Atlas**. It serves as a global symbolic reference for the accelerated Collatz dynamics on odd integers, cataloguing one canonical successor track for each terminal congruence class modulo 24. Within the verified range $n < 2^{71}$, the atlas provides a complete and non-redundant description of all forward successor orbits.

Structural interpretation

The terminal orbit code database provides a compact symbolic representation of the global successor structure:

- Each canonical terminal seed m_0 acts as a **hub**.
- Its orbit code $\omega(m_0)$ records the unique successor path from m_0 down to 1.
- Every odd integer n lies **along exactly one such path**, at a well-defined position above m_0.

From this perspective, the accelerated Collatz graph decomposes into disjoint successor "tracks", meaning maximal successor orbits indexed by canonical terminal seeds. This organization mirrors the canonical terminal tree described in §5.6, but is now encoded symbolically rather than graphically.

Relation to structural signatures

Let (w, m_0) be the structural signature of an odd integer n, as defined in §5.7. The two symbolic objects introduced in this chapter play complementary roles:

- the **structural signature** (w, m_0) encodes how n is constructed *above* its canonical terminal seed m_0 by backward steps,
- the **terminal orbit code** $\omega(m_0)$ encodes how m_0 descends *below* to 1 under the successor transform s.

Together, they locate n precisely within the global successor structure:

- m_0 selects the successor track,
- w specifies the position of n above m_0 along that track,
- and $\omega(m_0)$ records the remainder of the descent to 1.

This separation cleanly decouples local predecessor structure from global successor dynamics.

Completeness and compression

The terminal orbit code database has two fundamental properties within the verified range $n < 2^{71}$:

1. **Completeness:** Every odd integer lies on the successor orbit of exactly one canonical terminal seed, and that orbit is represented in the database.

2. **Compression:** Each orbit code $\omega(m_0)$ encodes an entire successor chain of arbitrary length in a finite symbolic word over $\{A, C, K\}$. No auxiliary normalization data or counters are required.

Thus the database provides a highly compressed symbolic summary of all accelerated Collatz successor dynamics below the verification limit.

Representative terminal orbit codes

To illustrate the structure of the terminal orbit code database, we list below a representative sample of terminal orbit codes for canonical seeds below 130. These examples show substantial variation in both length and symbolic complexity, reflecting the diverse shapes of successor orbits.

m_0	terminal orbit code $\omega(m_0)$
3	K C
9	K C K A C C A
15	K C K K C C C
27	K C K K C C C K A K A A A K C C C C K C C C C C A C A C C C K A C C C A C C A C A A C C C C A C
33	K C K A C C K C A A
39	K C K A C C K C A K C A C C
51	K C K A C C K C K C
57	K C K A C C A K A A C A
63	K C K K C C C K A K A A A K C C C C K C C C C C A C A C C C K A C C C A C C A C A K C C C C C
75	K K K A C
81	K C K K C C C K A
87	K C K A C C A K A K C C
105	K C K A C C K C A K C K C C C A
111	K C K K C C C K A K A A A K C C C C K C C C C C A C A C C C
129	K C K K C C C K A K A A A K C C C C K C C C C C A C A C C C K A C C C A C C A C A A C C C C K C C A A A

A complete table of the first fifty terminal orbit codes appears in **Appendix A**.

Role in reconstruction

The terminal orbit code database provides the global backbone for the reconstruction results of the next section. Once $\omega(m_0)$ is known for a canonical terminal seed, and the structural signature (w, m_0) of an odd integer n is given, the full predecessor structure of n can be recovered symbolically. This reconstruction process is made explicit in §5.10.

5.10 Symbolic Reconstruction of Accelerated Orbits

The constructions of §§5.7–5.9 together yield a central result of this work: the accelerated Collatz orbit of any odd integer can be reconstructed **purely symbolically**, without iterating either the Collatz map or the successor transform.

In this section we show how to recover the complete accelerated backward orbit

$$1 \to \cdots \to n$$

of an odd integer n using only:

1. its **structural signature** (w, m_0), and
2. the **terminal orbit code** $\omega(m_0)$ retrieved from the Terminal Orbit Code Atlas.

No numerical iteration of T or s is required. Throughout, we restrict attention to odd integers $n < 2^{71}$, the range for which classical Collatz convergence has been computationally verified (Barina, 2025).

Symbolic inputs

Let n be an odd integer. From §5.7, n has a unique **structural signature** (w, m_0), where:

- $m_0 \equiv 3, 9, 15 \pmod{24}$ is the canonical terminal seed associated with n, and
- $w \in \{A, C, K\}^*$ is a finite word encoding the predecessor steps encountered when descending from n down to m_0.

From §5.9, the **terminal orbit code** $\omega(m_0)$ is stored in the Terminal Orbit Code Atlas and encodes the full symbolic predecessor structure beneath m_0. Decoding $\omega(m_0)$ yields the **structural predecessor chain**

$$1 \to \cdots \to m_0,$$

in which every symbol in $\{A, C, K\}$ is realized as an explicit odd predecessor step, including local 01-lift artifacts.

Structural chains vs. accelerated Collatz orbits

It is essential to distinguish two closely related but different objects:

- the **structural predecessor chain**, obtained by decoding an orbit code, and
- the **accelerated Collatz orbit**, which suppresses intermediate artifacts created by 01-lifts.

Decoding an orbit code applies each symbol as a genuine predecessor transform:

- A produces $m \mapsto (4m-1)/3$,
- C produces $m \mapsto (2m-1)/3$,
- K produces $m \mapsto 4m+1$.

In this refined structural chain, a K step is an explicit move into a 01-tower. In accelerated Collatz dynamics, however, the intermediate tower-entry values are not regarded as separate *odd transitions*. The correct acceleration rule is therefore:

- whenever an A or C symbol is immediately followed by one or more K symbols, the intermediate odd value produced by that A/C step is an internal connector and is suppressed, and
- for each maximal run K^r with $r \geq 1$, all but the final lifted value are suppressed.

Thus the accelerated orbit is obtained from the decoded structural chain by a deterministic symbolic compression that depends only on the placement and run-lengths of the K symbols.

Decomposition of the orbit code

By construction, decoding $\omega(m_0)$ yields the structural predecessor chain from 1 up to m_0. The structural signature word w is constructed **in predecessor symbols**: it records exactly the final predecessor steps encountered when descending from n down to m_0.

Consequently, w corresponds to the terminal segment of the structural code from n down to m_0. Equivalently, n lies at a unique position along the structural chain encoded by $\omega(m_0)$, and its orbit code is obtained by truncation.

Orbit code of a general odd integer

Definition (orbit code of n)

Let (w, m_0) be the structural signature of an odd integer n. The **orbit code of** n, denoted $\omega(n)$, is defined to be the unique prefix of $\omega(m_0)$ whose decoding terminates at n. Equivalently, $\omega(n)$ is obtained from $\omega(m_0)$ by deleting the terminal suffix corresponding to the word w. Under the atlas construction, the symbols encoding the segment from n up to m_0 occur as a terminal suffix of $\omega(m_0)$; this suffix is exactly the structural signature word w.

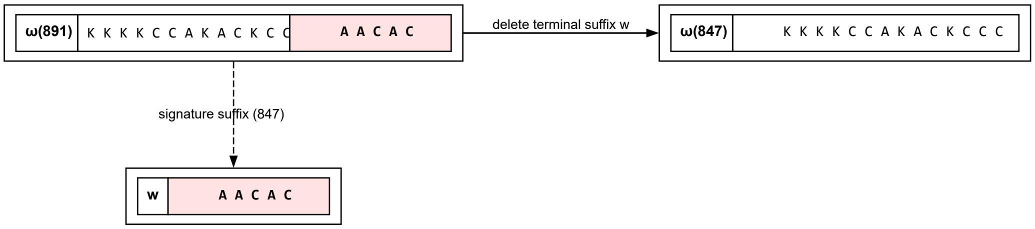

Figure 5.7. *Orbit code truncation from a terminal atlas entry.* The terminal orbit code $\omega(891)$ contains, as a literal terminal suffix, the structural signature word $w = AACAC$ associated with the integer 847. Removing this suffix yields the orbit code $\omega(847)$, which decodes to the full structural predecessor chain from 1 to 847. This illustrates the general reconstruction principle: for any odd integer n with structural signature (w, m_0), the orbit code $\omega(n)$ is obtained by symbolic truncation of the terminal code $\omega(m_0)$, without iteration or numerical computation.

This construction is purely symbolic and requires only:

- lookup of $\omega(m_0)$ in the Terminal Orbit Code Atlas, and
- deletion of a suffix determined by w.

No reversal, concatenation, or numerical iteration is involved.

Decoding and acceleration

To recover the **structural predecessor chain** of n, decode $\omega(n)$ as in §5.8.4, starting from $m = 1$. This yields a refined chain

$$1 \to m_1 \to \cdots \to n$$

in which every symbol in $\omega(n)$ contributes an explicit predecessor step. The **accelerated Collatz orbit** is then obtained by applying the deterministic compression rule of §5.10.2:

- suppress any value produced by an A or C step that is immediately followed by a K symbol,
- and for each maximal run K^r, retain only the final lifted value.

This converts the refined structural chain into the classical accelerated odd trajectory.

Example: $n = 51$

The orbit code of 51 is $\omega(51)$ = KCKACCKCKC. Decoding this word produces the structural predecessor chain

$$1 \to 5 \to 3 \to 13 \to 17 \to 11 \to 7 \to 29 \to 19 \to 77 \to 51.$$

Here 3, 7 and 19 are the values produced by C steps that are immediately followed by K steps, so they are suppressed under acceleration. Applying the compression rule yields the accelerated Collatz orbit

$$1 \to 5 \to 13 \to 17 \to 11 \to 29 \to 77 \to 51,$$

which matches the accelerated dynamics obtained by reversing the forward successor chain.

Algorithm: Symbolic reconstruction of accelerated Collatz orbits

- **Input:** an odd integer $n < 2^{71}$
- **Output:** the accelerated backward Collatz orbit $1 \to \cdots \to n$

1. Compute the structural signature (w, m_0) of n as in §5.7.
2. Retrieve the terminal orbit code $\omega(m_0)$ from the Terminal Orbit Code Atlas.
3. Delete from $\omega(m_0)$ the terminal suffix corresponding to w to obtain $\omega(n)$.
4. Decode $\omega(n)$ symbolically from $m = 1$ using the rules of §5.8.4 to obtain the structural chain.
5. Accelerate the chain by suppressing any A/C output immediately followed by K, and collapsing each maximal run of K symbols to its final value.
6. Record the resulting odd values; the final value is n.

This procedure uses only symbolic operations and table lookup, and performs no numerical Collatz iteration.

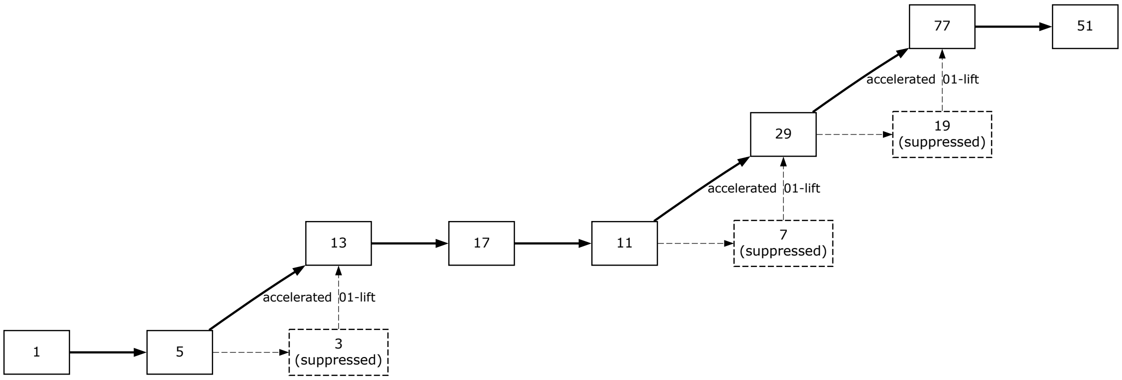

Figure 5.8. *Structural predecessor chain versus accelerated Collatz orbit for* $n = 51$. The orbit code is $\omega(51)$ = KCKACCKCKC. Decoding this word symbol by symbol produces the full *structural* predecessor chain $1 \to 5 \to 3 \to 13 \to 17 \to 11 \to 7 \to 29 \to 19 \to 77 \to 51$, which explicitly includes intermediate nodes created by local 01-lifts. In the accelerated Collatz orbit, any value produced by

an A or C step that is immediately followed by one or more K steps is an internal tower connector and is suppressed; each maximal run K^r contributes only its final lifted value. Accordingly, the suppressed nodes 3, 7, and 19 correspond to C outputs immediately followed by K, and the resulting accelerated orbit is $1 \to 5 \to 13 \to 17 \to 11 \to 29 \to 77 \to 51$.

Odd distance and symbolic depth

Once $\omega(n)$ is known, two natural "odd distances" can be read off immediately: one for the **structural chain** (which records every symbolic predecessor step) and one for the **accelerated Collatz orbit** (which suppresses 01-lift artifacts as in §5.10.2).

Definition (structural odd distance)

For an odd integer $n \geq 1$, define the **structural odd distance** sdist(n) to be the number of symbolic predecessor steps in the structural predecessor chain from 1 to n obtained by decoding $\omega(n)$. Equivalently,

$$\text{sdist}(n) = |\omega(n)|,$$

since decoding applies exactly one predecessor transform per symbol in $\{A, C, K\}$.

Definition (accelerated odd distance)

Define the **accelerated odd distance** odist(n) to be the number of accelerated odd transitions in the accelerated backward Collatz orbit from 1 to n. This can be computed directly from $\omega(n)$ by the symbolic counting rule induced by §5.10.2:

- count an A or C symbol as 1 only when it is **not** immediately followed by a K,
- count each maximal run K^r with $r \geq 1$ as 1.

Equivalently, odist(n) is obtained from $\omega(n)$ by counting the number of "accelerated moves" after applying the same symbolic compression that produces the accelerated orbit. Both sdist(n) and odist(n) are purely combinatorial quantities determined by $\omega(n)$ alone, independent of intermediate even values.

Example (distances for $n = 51$)

For $n = 51$,

$$\omega(51) = \mathsf{KCKACCKCKC}.$$

Hence

- $\text{sdist}(51) = |\omega(51)| = 10$,
- $\text{odist}(51) = 7$,

because $\mathsf{K}, \mathsf{K}, \mathsf{K}$-runs contribute three accelerated steps, while the two C symbols immediately followed by K contribute 0 accelerated steps. This agrees with the accelerated orbit

$$1 \to 5 \to 13 \to 17 \to 11 \to 29 \to 77 \to 51,$$

which has 7 accelerated odd transitions.

Remark (which distance should be called “odd distance”?)

Both distances are meaningful:

- $\mathrm{sdist}(n)$ measures depth in the **structural predecessor graph** (sensitive to explicit 01-lift structure),
- $\mathrm{odist}(n)$ measures depth in the **accelerated Collatz graph** (the classical odd-only dynamics).

In what follows, “odd distance” refers to $\mathrm{odist}(n)$ unless stated otherwise.

Why this matters

§5.10 shows that accelerated Collatz dynamics admit a **fully symbolic coordinate system**. Rather than treating Collatz trajectories as opaque numerical processes, this framework decomposes every verified orbit into two independent symbolic components:

- a **global backbone**, supplied by the Terminal Orbit Code Atlas, and
- a **local displacement**, encoded by the structural signature.

This separation cleanly disentangles global behavior from local variation. Once the atlas is fixed, all remaining complexity is confined to short symbolic words, and the full accelerated orbit of any odd integer can be recovered by symbolic truncation and compression alone.

Several consequences follow immediately:

- Orbits can be reconstructed, compared, and traversed **without iteration**.
- Extremely long trajectories can be **designed symbolically**, without searching or brute force.
- Distance in the odd Collatz graph becomes a **purely combinatorial invariant**, read directly from the orbit code.
- Infinitely many odd integers are organized around a **sparse and structured family of terminal hubs**.

Taken together, these results suggest that much of the apparent complexity of Collatz dynamics arises not from randomness or chaotic growth, but from a rigid symbolic architecture that has so far remained hidden beneath numerical iteration.

Route skeletons and terminal families

Orbit codes $\omega(n)$ provide a complete symbolic description of the structural predecessor chain of n, and §5.10.2 shows how to compress this chain to recover the accelerated Collatz orbit. However, the orbit code does not explicitly display the *phase junctions* (the branch points) of the predecessor structure: the points at which a maximal 01-lift phase ends and the orbit returns to predecessor motion (or vice versa). To make these junctions explicit, we introduce a condensed representation called the **route skeleton**. The route skeleton is also the most natural way to visualize how an entire **terminal family** is reached from 1.

Definition (route skeleton)

Let n be an odd integer and let $\omega(n) \in \{A, C, K\}^*$ be its orbit code. Parse $\omega(n)$ into maximal blocks

$$\omega(n) = B_1 B_2 \cdots B_r,$$

where each B_i is either a maximal run K^{t_i} with $t_i \geq 1$, or a maximal word over $\{A, C\}$ of positive length. Starting from $m = 1$, decode $\omega(n)$ symbol-by-symbol as in §5.8.4. The **route skeleton** of n, denoted $R(n)$, is the list of block endpoints:

$$R(n) = [v_0, v_1, \dots, v_r]$$

where $v_0 = 1$ and v_i is the odd integer reached after decoding $B_1 \cdots B_i$. Equivalently, $R(n)$ records the endpoints of maximal K-runs and maximal (A/C)-runs in $\omega(n)$. It is a projection of the orbit code onto its phase junctions.

Definition (terminal family indexed by m_0)

Let $m_0 \equiv 3, 9, 15 \pmod{24}$ be a canonical terminal seed with terminal orbit code $\omega(m_0)$. Read $\omega(m_0)$ from right to left and take the maximal suffix consisting only of symbols in $\{A, C\}$ before the first occurrence of K from the right. Decoding that suffix by the inverse successor maps A' and C' produces a finite list of odd integers ending at m_0. We denote this finite list by $\mathcal{F}(m_0)$ and call it the **terminal family indexed by** m_0. The explicit construction and justification of $\mathcal{F}(m_0)$ are given in §5.11; here we use only the fact that $\mathcal{F}(m_0)$ is the finite "(A/C)-tail" above m_0 bounded by the first K from the right in $\omega(m_0)$.

Example (route skeleton and family for $m_0 = 57$).

For $m_0 = 57$ we have

$$\omega(57) = \mathsf{KCKACCAKAACA}$$

and the indexed family

$$\mathcal{F}(57) = \{57, 43, 65, 49, 37\}.$$

Decoding $\omega(57)$ yields the structural predecessor chain

$$1 \to 5 \to 3 \to 13 \to 17 \to 11 \to 7 \to 9 \to 37 \to 49 \to 65 \to 43 \to 57.$$

Grouping $\omega(57)$ into maximal blocks gives

$$\mathsf{K\ C\ K\ ACCA\ K\ AACA}$$

and therefore the route skeleton is

$$R(57) = [1, 5, 3, 13, 9, 37, 57].$$

Interpreted as "directions" along the predecessor structure, this reads:

- take K until 5;
- take (A/C) until 3;
- take K until 13;
- take (A/C) until 9;
- take K until 37;
- take (A/C) until 57.

The route skeleton isolates exactly the branching junctions on the path from 1 into the terminal family above 57.

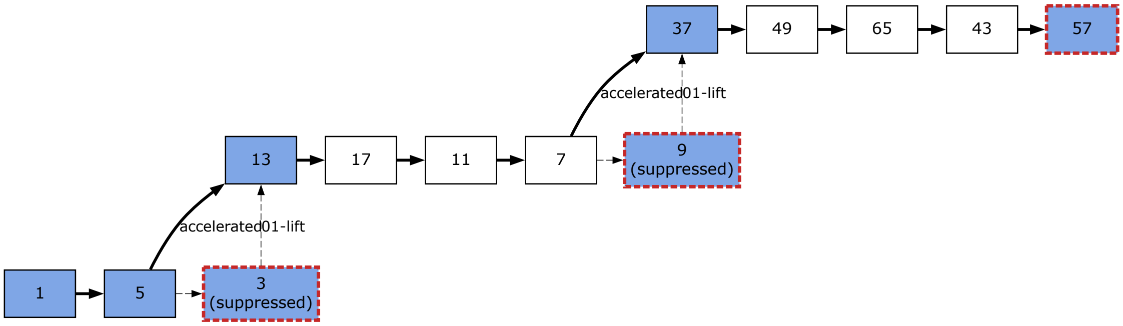

Figure 5.9. *Structural predecessor chain, accelerated Collatz orbit, route skeleton, and terminal family for* $n = 57$. The orbit code is $\omega(57) =$ KCKACCAKAACA. Decoding this word symbol by symbol produces the full *structural* predecessor chain $1 \to 5 \to 3 \to 13 \to 17 \to 11 \to 7 \to 9 \to 37 \to 49 \to 65 \to 43 \to 57$, explicitly including intermediate nodes generated by local 01-lifts. In the accelerated Collatz orbit, any value produced by an A or C step that is immediately followed by a K step is an internal lift connector and is suppressed; each maximal run K^r contributes only its final lifted value. The highlighted nodes form the *route skeleton* $R(57) = [1, 5, 3, 13, 9, 37, 57]$, recording exactly the phase junctions between maximal K-runs and maximal (A/C)-runs. The terminal family indexed by the canonical seed 57, namely $\mathcal{F}(57) = \{57, 43, 65, 49, 37\}$, appears as the final (A/C)-tail above the last K-transition. This illustrates how route skeletons isolate macro-level structure while suppressing local lift artifacts.

Example (the route skeleton of 27).

The odd integer 27 is often cited for the length and apparent complexity of its classical Collatz trajectory. Within the present framework, however, its structure is completely transparent. The orbit code of 27 is

$$\omega(27) = \texttt{KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCAC}.$$

Decoding $\omega(27)$ yields a long structural predecessor chain, but its **route skeleton** consists of only the phase junctions between maximal K-runs and maximal (A/C)-runs. Grouping $\omega(27)$ into maximal blocks produces the route skeleton

$$R(27) = [1,\ 5,\ 3,\ 53,\ 15,\ 61,\ 81,\ 325,\ 769,\ 3077,\ 607,\ 2429,\ 111,\ 445,\ 27].$$

Interpreted as directions along the predecessor structure, this reads:

- take K until 5;
- take (A/C) until 3;
- take K until 53;
- take (A/C) until 15;
- take K until 61;
- take (A/C) until 81;
- take K until 325;
- take (A/C) until 769;
- take K until 3077;

- take (A/C) until 607;
- take K until 2429;
- take (A/C) until 111;
- take K until 445;
- take (A/C) until 27.

Thus, despite the length of its classical orbit, the integer 27 traverses only a **finite and modest number of route junctions**. The apparent complexity of its trajectory arises from extended 01-lift phases within individual blocks, not from branching in the symbolic structure.

This example illustrates that even the most celebrated "pathological" cases occupy a well-defined location within the terminal family framework, and that their global behavior is governed by a compact symbolic route.

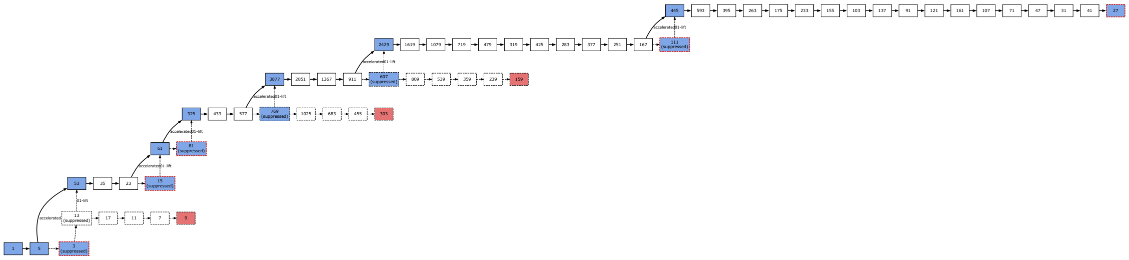

Figure 5.10. *Structural predecessor chain, accelerated Collatz orbit, route skeleton, and reverse τ-chain level indicators for $n = 27$.* The orbit code is $\omega(27)$ = KCKKCCCKAKAAAKCCCCK-CCCCCACACCCKACCCACCACAACCCCAC Decoding this word symbol by symbol produces a long *structural* predecessor chain, explicitly including all intermediate nodes generated by local 01-lifts. In the accelerated Collatz orbit, any value produced by an A or C step that is immediately followed by one or more K steps is an internal lift connector and is suppressed; for each maximal run K^r, all but the final lifted value are suppressed. The highlighted nodes indicate the *route skeleton* $R(27) = [1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 27]$, which records exactly the phase junctions between maximal K-runs and maximal (A/C)-runs. Dashed horizontal extensions depict *reverse τ-chains* emanating from selected suppressed nodes, terminating at their corresponding collapse levels. Nodes with red outlines mark canonical seeds along these chains; all reverse τ-chains ultimately descend to the forward-terminal class $\mathcal{P}_1$ with canonical base 1. Despite the length of its classical trajectory, 27 follows a finite and rigid symbolic route from 1, illustrating that its apparent complexity arises from extended lift phases rather than branching in the symbolic dynamics.

Proposition (route prefix invariance on a terminal family).

Using the notion of a terminal family introduced formally in §5.11, let m_0 be a canonical terminal seed and let $x \in \mathcal{F}(m_0)$. Then the route skeletons agree up to replacement of the final endpoint:

$$R(x) = [v_0, v_1, \ldots, v_{r-1}, x]$$

where $R(m_0) = [v_0, v_1, \ldots, v_{r-1}, m_0]$.

In particular, the "macro-route" from 1 to the terminal family is canonical, and the different family members differ only by where they lie along the final (A/C)-segment.

Sketch of justification.

- The family $\mathcal{F}(m_0)$ is obtained from the maximal rightmost (A/C)-suffix of $\omega(m_0)$.
- Replacing m_0 by any earlier endpoint along that suffix changes only the final decoded endpoint, while leaving the preceding block decomposition unchanged.
- Hence the route skeleton prefix is invariant on $\mathcal{F}(m_0)$.
- Full details are given in §5.11.

■

Remark (relationship to τ-chains).

The route skeleton is not additional dynamical data; it is a condensed view of $\omega(n)$ that highlights phase junctions. The endpoints of maximal K-runs are exactly the moments at which 01-tower collapse completes, which are precisely the locations where terminal transitions occur and are recorded in the τ-chain (see §5.5). Thus route skeletons provide a compact, human-readable summary of how a terminal family is reached from 1.

In Appendix A one may optionally record $R(m_0)$ alongside each terminal atlas entry, providing a condensed atlas view in which the branching junctions are explicit.

A reference implementation to construct the accelerated chain and route skeleton is provided in Appendix D.Sec-5.10.

5.11 Odd Integers Indexed by a Canonical Terminal Seed

The Terminal Orbit Code Atlas introduced in §5.9 associates to each canonical terminal seed m_0 a symbolic orbit code $\omega(m_0)$ encoding the full structural predecessor chain beneath m_0.

In this section we show that the same orbit code also determines, in a purely symbolic way, the finite collection of odd integers whose canonical terminal seed is exactly m_0. In other words, each canonical terminal seed indexes not only a backward orbit, but also a finite family of integers *above* it.

Inverse successor maps and terminal indexing

Recall from §5.2 that the accelerated successor transform partitions odd integers according to residue classes and admits three symbolic inverse branches, denoted symbolically by A', C', and K', corresponding to the predecessor symbols A, C, and K.

Applied to an odd integer x, these inverse successor maps generate candidate predecessors whose forward successor is x. In the symbolic setting of this paper, these inverse maps act directly on the orbit code.

Crucially, application of the inverse K' corresponds to *exiting* the terminal class of a given m_0, while inverse applications of A' and C' preserve the same canonical terminal seed. This observation leads to a sharp symbolic boundary.

Terminal families indexed by a canonical terminal seed

Definition (terminal family).

Let $m_0 \equiv 3, 9, 15 \pmod{24}$ be a canonical terminal seed with terminal orbit code $\omega(m_0)$. Write $\omega(m_0) = vu$, where u is the maximal suffix consisting only of symbols from $\{A, C\}$ (equivalently, u is the block of consecutive A/C symbols immediately to the right of the rightmost occurrence of K in $\omega(m_0)$, or $u = \omega(m_0)$ if no K occurs). Write $u = a_1 a_2 \cdots a_t$ in left-to-right order. Define a sequence of odd integers by

$$x_0 = m_0,$$

$$x_i = a'_{t-i+1}(x_{i-1})$$

for $1 \leq i \leq t$, where A' and C' denote the inverse successor maps corresponding to the predecessor symbols A and C. The terminal family indexed by m_0, denoted $\mathcal{F}(m_0)$, is the finite set

$$\mathcal{F}(m_0) = \{x_0, x_1, \ldots, x_t\}.$$

Equivalently, $\mathcal{F}(m_0)$ consists exactly of those odd integers whose accelerated successor dynamics terminate at m_0 without encountering a K' transition. The first inverse application of K' produces an integer whose canonical seed differs from m_0 and therefore lies outside $\mathcal{F}(m_0)$.

Remark. If the maximal $\{A, C\}$-suffix of $\omega(m_0)$ has length t, then the indexed family satisfies $|\mathcal{F}(m_0)| = t + 1$.

Example: canonical seed $m_0 = 891$

The orbit code of 891 is

$$\omega(891) = \mathtt{KKKKCCAKACKCCCAACAC}.$$

Decoding this code yields the structural predecessor chain

$$1 \to 5 \to 21 \to 85 \to 341 \to \cdots \to 1337 \to 891.$$

Reading $\omega(891)$ from right to left and applying inverse successor maps symbolically up to the first K produces the odd integers

$$1337,\ 1003,\ 1505,\ 1129,\ 847,\ 1271,\ 1907,\ 2861.$$

Each of these integers has canonical terminal seed 891. Together with 891 itself, they form the full terminal family indexed by m_0. The next inverse application would correspond to K', yielding 1073, whose canonical terminal seed differs from 891 and therefore lies outside this indexed family. Thus the terminal family indexed by $m_0 = 891$ is exactly

$$\mathcal{F}(891) = \{891, 1337, 1003, 1505, 1129, 847, 1271, 1907, 2861\}.$$

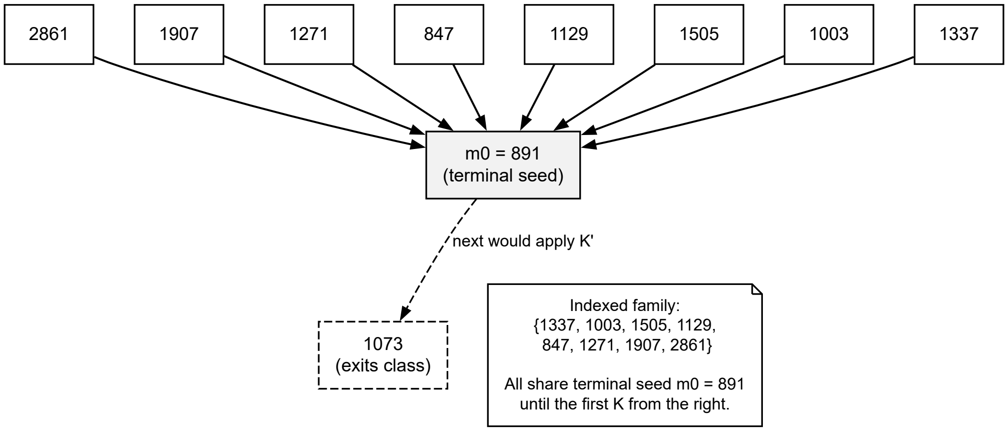

Figure 5.11. *Odd integers indexed by a common canonical terminal seed.* The canonical terminal seed $m_0 = 891$ indexes a finite family of odd integers whose accelerated successor dynamics terminate at the same seed. Each integer in the set $\{1337, 1003, 1505, 1129, 847, 1271, 1907, 2861\}$ maps to m_0 under the accelerated successor transform, and is obtained symbolically by applying inverse successor maps corresponding to the maximal suffix of $\omega(891)$ consisting only of A and C symbols. The boundary of this indexed family is marked by the first K from the right in the orbit code: applying the inverse K' produces 1073, which exits the terminal class of 891.

Structural interpretation

This construction shows that each canonical terminal seed m_0 serves as a hub for two complementary symbolic structures:

- a downward structural predecessor tree encoded by $\omega(m_0)$, and
- a finite upward family of odd integers indexed by the same orbit code.

Both structures are derived from the same symbolic data, and both are determined without numerical iteration. In particular, the boundary of the indexed family is marked symbolically by the first occurrence of K when reading $\omega(m_0)$ from right to left.

Atlas extension and computational remarks

From an implementation perspective, the Terminal Orbit Code Atlas may naturally be extended to store, alongside each orbit code $\omega(m_0)$, the finite list of odd integers indexed by m_0. Such an augmented atlas supports efficient classification and reverse lookup:

- given an odd integer n, its canonical terminal seed m_0 locates the corresponding atlas entry, and
- given a canonical terminal seed m_0, the full family of integers indexed by m_0 is immediately available.

This highlights that the accelerated Collatz graph, within the verified range, admits a compact symbolic representation suitable for direct computation, indexing, and traversal. The significance of this observation lies not in extending the range of verification, but in revealing an additional layer of rigid structure organizing the integers around their canonical seeds.

A reference implementation to list the odd integers indexed by the canonical terminal seed is provided in Appendix D.Sec-5.11.

5.12 The Universal Signature

Sections §5.7–§5.11 assign to each odd integer n a **structural signature**

$$\sigma(n) = (w, m_0),$$

where w is a finite word over $\{A, C, K\}$ encoding symbolic predecessor structure, and m_0 is the canonical terminal seed reached when that structure is removed. This describes the symbolic branching behavior of the accelerated Collatz dynamics on odd integers. To extend the framework to all $N \geq 1$, we must also record the deterministic 2-adic height contributed by even factors.

In this section we introduce the **universal signature $\sigma^*(N)$**, which applies to every positive integer and serves as the natural state descriptor for Collatz dynamics on $\mathbb{Z}_{>0}$.

Deterministic 2-adic motion

Every integer $N \geq 1$ admits a unique factorization

$$N = 2^z n, \qquad n \text{ odd.}$$

The exponent

$$z = \nu_2(N)$$

counts the number of divisions by 2 required to reach the odd component n. We call z the **drop count**. It records the deterministic 2-adic motion of N within the Collatz graph.

Unlike symbolic predecessor steps, divisions by 2 introduce no branching and no new terminal behavior. Their effect is purely vertical, moving an integer within a fixed symbolic corridor determined by (w, m_0). For this reason, the 2-adic component of the dynamics is recorded as a numerical parameter rather than as part of the symbolic word.

Definition of the universal signature

Let $N \geq 1$ and write $N = 2^z n$ with n odd. By §§5.7–§5.11 the odd core has a structural signature

$$\sigma(n) = (w, m_0).$$

Definition.

The **universal signature** of a positive integer N is the triple

$$\boxed{\sigma^*(N) = (z,\ w,\ m_0)}.$$

Equivalently, $\sigma^*(N)$ is obtained by:

1. extracting the odd core n and drop count $z = \nu_2(N)$, and
2. attaching the structural signature (w, m_0) of that odd core.

This definition assigns a unique signature to every integer $N \geq 1$.

Interpretation

The three components of $\sigma^*(N) = (z, w, m_0)$ encode complementary aspects of Collatz structure.

- w encodes symbolic predecessor structure through A/C/K branching.
- z records deterministic 2-adic height above the odd core.
- m_0 identifies the canonical terminal seed governing the terminal basin of the odd core.

Odd integers correspond exactly to the case $z = 0$. Even integers differ from their odd cores only by their 2-adic height, with the same (w, m_0).

The boundary case $n = 1$

One boundary case must be recorded. The odd integer 1 does not terminate at a canonical seed $m_0 \equiv 3, 9, 15 \pmod{24}$. Instead, it generates the trivial fixed-point predecessor class $\mathcal{P}_1$ introduced in §3.7. This case introduces no symbolic branching and no additional structure. It simply marks the unique fixed point of the accelerated Collatz map. Accordingly, whenever the odd core of N is 1, we set

$$m_0 = \mathsf{None}.$$

Examples

The following examples illustrate the definition of the universal signature.

Example 1 — An odd integer

Let $N = 933$. Since 933 is odd, $z = 0$ and the odd core is $n = 933$. Thus

$$\sigma^*(933) = (0,\ KA,\ 4977),$$

where $(w, m_0) = \sigma(933)$ is the structural signature computed in §5.11.

Example 2 — An even integer

Let $N = 7464$. Then

$$7464 = 2^3 \cdot 933,$$

so $z = 3$ and the odd core is $n = 933$. Therefore

$$\sigma^*(7464) = (3,\ KA,\ 4977),$$

with the same $(w, m_0) = \sigma(933)$ as in the previous example.

Example 3 — A pure power of two

Let $N = 32$. Then

$$32 = 2^5 \cdot 1,$$

so the odd core is 1 and the boundary convention applies. Hence

$$\sigma^*(32) = (5,\ \text{""},\ \mathtt{None}).$$

Closing remark

The universal signature $\sigma^*(N)$ extends the structural signature framework from odd integers to all of $\mathbb{Z}_{>0}$ by adjoining the deterministic 2-adic drop count. In §5.13 we use $\sigma^*(N)$ together with the terminal Atlas to construct canonical universal orbit codes.

A reference implementation to compute the **universal signature** of an integer N is provided in Appendix D.Sec-5.12.

5.13 Universal Orbit Codes

In §§5.8–§5.11 we developed a symbolic description of the accelerated Collatz map for odd integers, culminating in a canonical **orbit code** $\omega(n)$ associated with each odd integer n.

This orbit code encodes the complete accelerated predecessor structure of n using symbolic A/C/K steps and compressed lift-tower structure. It uniquely identifies the symbolic family of n and allows the symbolic predecessor structure to be recovered from the terminal Atlas.

In §5.12 we extended the structural framework to all positive integers by introducing the **universal signature**

$$\sigma^*(N) = (z, w, m_0).$$

which records both the symbolic structure of the odd core and the deterministic 2-adic height of N.

In this section we combine these two constructions. We show that the universal signature determines a canonical **universal orbit code** $\omega^*(N)$ for every integer $N \geq 1$, extending the orbit-code formalism from odd integers to the full Collatz domain.

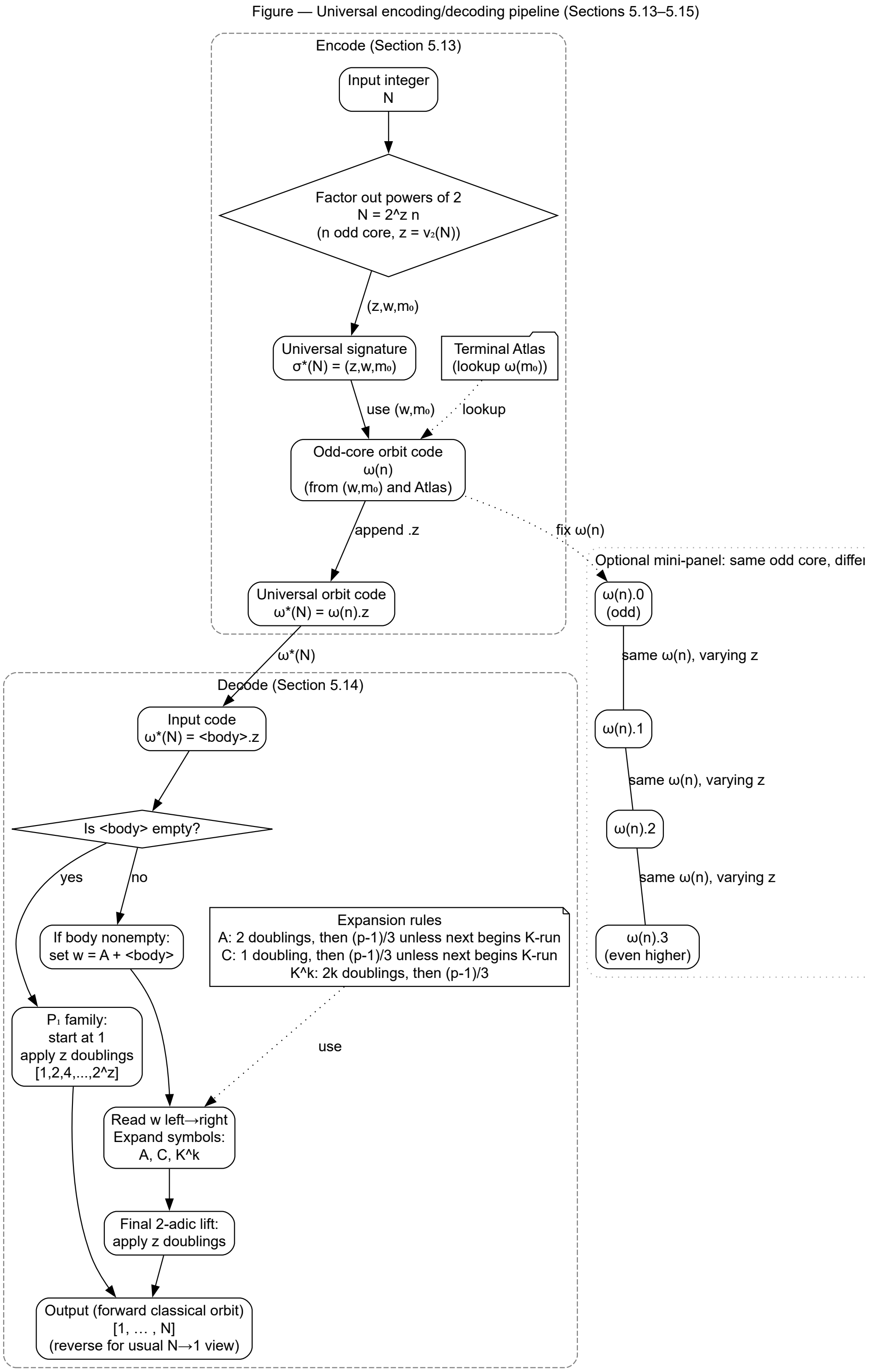
Figure — Universal encoding/decoding pipeline (Sections 5.13–5.15)
Encode (Section 5.13)
Input integer N
Factor out powers of 2 N = 2^z n (n odd core, z = v₂(N))
(z,w,m₀)
Universal signature σ*(N) = (z,w,m₀)
Terminal Atlas (lookup ω(m₀))
use (w,m₀)
lookup
Odd-core orbit code ω(n) (from (w,m₀) and Atlas)
append .z
fix ω(n)
Universal orbit code ω*(N) = ω(n).z
Optional mini-panel: same odd core, differ
ω(n).0 (odd)
same ω(n), varying z
ω(n).1
same ω(n), varying z
ω(n).2
same ω(n), varying z
ω(n).3 (even higher)
ω*(N)
Decode (Section 5.14)
Input code ω*(N) = <body>.z
Is <body> empty?
yes
no
If body nonempty: set w = A + <body>
Expansion rules
A: 2 doublings, then (p-1)/3 unless next begins K-run
C: 1 doubling, then (p-1)/3 unless next begins K-run
K^k: 2k doublings, then (p-1)/3
use
P₁ family: start at 1 apply z doublings [1,2,4,...,2^z]
Read w left→right Expand symbols: A, C, K^k
Final 2-adic lift: apply z doublings
Output (forward classical orbit) [1, … , N] (reverse for usual N→1 view)

Figure 5.12. *Universal encoding and decoding pipeline for Collatz orbits.* An input integer N is first decomposed as $N = 2^z n$, separating its odd core n from its deterministic 2-adic height $z = \nu_2(N)$. The odd core is assigned a structural signature (w, m_0), which is resolved via the terminal atlas to produce the accelerated orbit code $\omega(n)$. Appending the 2-adic height yields the

universal orbit code $\omega^*(N) = \omega(n) \cdot z$.

Decoding proceeds deterministically: an initial A aligns the predecessor-oriented encoding with forward dynamics; symbolic expansions of A, C, and K runs reconstruct all suppressed even and odd transitions; and the final z doublings embed the odd core into its full position in the Collatz graph. The output is the complete classical Collatz orbit $[1, \dots, N]$, whose reversal gives the usual descending orbit from N to 1.

From universal signatures to orbit codes

Let $N \geq 1$ and write

$$N = 2^z n,$$

where n is the odd core of N. From §5.12, the universal signature of N is

$$\sigma^*(N) = (z, w, m_0).$$

From §§5.8–§5.10, the odd core n has a well-defined accelerated orbit code $\omega(n)$, obtained from its structural signature (w, m_0) by referencing the terminal orbit code $\omega(m_0)$ in the Atlas and removing the terminal suffix w.

The orbit code $\omega(n)$ captures all symbolic branching and compressed lift structure associated with the accelerated Collatz dynamics of the odd core. The only remaining information required to locate N uniquely within the full Collatz graph is the deterministic 2-adic height z.

Definition of the universal orbit code

Definition.

The **universal orbit code** of a positive integer N is the symbol string

$$\boxed{\omega^*(N) = \omega(n)\,\texttt{.}\,z},$$

where:

- $\omega(n)$ is the accelerated orbit code of the odd core n,
- `.` is a literal separator symbol, and
- $z = \nu_2(N)$ is the 2-adic drop count, written in decimal.

Thus every integer $N \geq 1$ admits a universal orbit code of the uniform form

$$\texttt{<symbolic-body>.<z>}.$$

When N is odd, $z = 0$ and the code ends in `.0`. The separator . is not an element of the alphabet $\{A, C, K\}$ and therefore cannot collide with symbolic data.

Interpretation

The universal orbit code $\omega^*(N)$ has the following properties.

1. **Odd integers.**

   - If N is odd, then $z = 0$ and

$$\omega^*(N) = \omega(N)\texttt{.0}.$$

The symbolic body is exactly the accelerated orbit code developed in earlier sections.

2. **Even integers.**

   - If $N = 2^z n$ with $z > 0$, then $\omega^*(N)$ preserves the full symbolic structure of the odd core and appends the integer z, recording the precise 2-adic height of N above that core.

3. **Symbolic families.**

   - All integers sharing the same odd core n have universal orbit codes with the same symbolic body $\omega(n)$ and differ only in the appended integer.

4. **Canonical representation.**

   - The universal orbit code merges symbolic predecessor structure and deterministic 2-adic height into a single, canonical symbolic object valid for all positive integers.

Examples

The following examples illustrate the construction of the universal orbit code from the universal signature and the terminal Atlas.

Example 1 — An odd integer

Let $N = 933$. From §5.12, the universal signature of 933 is

$$\sigma^*(933) = (0, \mathsf{KA}, 4977).$$

Since $z = 0$, the odd core is $n = 933$ itself. Using the terminal Atlas, the accelerated orbit code of the odd core is

$$\omega(933) = \mathsf{KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCAK}.$$

By Definition 5.13.2, the universal orbit code is therefore

$$\omega^*(933) = \mathsf{KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCAK}\texttt{.0}.$$

In this case the universal orbit code coincides with the odd-only orbit code, with the explicit suffix `.0` recording that no 2-adic drops are present.

Example 2 — An even integer with the same odd core

Let $N = 7464$. Then

$$7464 = 2^3 \cdot 933,$$

so the odd core is $n = 933$ and the drop count is $z = 3$. Accordingly, the universal signature is

$$\sigma^*(7464) = (3, \mathsf{KA}, 4977).$$

The symbolic structure of the odd core is unchanged, and the accelerated orbit code remains

$$\omega(933) = \mathsf{KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCAK}.$$

Appending the drop count yields the universal orbit code

$$\omega^*(7464) = \mathsf{KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCAK.3}.$$

Thus 933 and 7464 share the same symbolic body and differ only by their 2-adic height, recorded explicitly by the suffix.

Interpretation across a symbolic family

Examples 1 and 2 illustrate a general and persistent principle: all integers sharing the same odd core n have universal orbit codes with the same symbolic body $\omega(n)$ and differ only in the appended integer z.

The universal orbit code therefore separates symbolic predecessor structure from deterministic 2-adic motion in a transparent and canonical way.

Role of the universal orbit code

The universal orbit code $\omega^*(N)$ provides a compact symbolic representation of the position of N within the Collatz graph. The symbolic body $\omega(n)$ encodes the complete accelerated predecessor structure of the odd core via the terminal Atlas, while the appended integer z specifies how that structure is embedded vertically through powers of two. As a result, $\omega^*(N)$ contains all structural information required to **decode** the Collatz behavior of N.

In the next section we give explicit decoding procedures that recover:

- the structural predecessor chain,
- the accelerated Collatz chain, and
- the full classical Collatz orbit

directly from the universal orbit code.

A reference implementation to obtain the **universal orbit code** of an integer N is provided in Appendix D.Sec-5.13.

5.14 Decoding Universal Orbit Codes

In §5.13 we introduced the **universal orbit code**

$$\omega^*(N) = \omega(n)\,\mathtt{.}\,z,$$

which symbolically encodes the position of an integer N within the Collatz graph, where:

- n is the odd core of N,
- $\omega(n)$ is the accelerated orbit code of n, and
- $z = \nu_2(N)$ records the 2-adic height of N above n.

In this section we show that $\omega^*(N)$ admits a **complete symbolic decoding** that reconstructs the **entire classical Collatz orbit of N**, including all intermediate even values, **without applying the Collatz map directly**.

The decoding process is the exact symbolic inverse of the compression rules used to construct orbit codes in §§5.8–§5.10.

Overview of the decoding process

Given a universal orbit code

$$\omega^*(N) = \texttt{<body>.}z,$$

we reconstruct the forward Collatz orbit beginning at 1 and terminating at N.

The decoding proceeds in four conceptual stages:

1. **Initialization at 1.**
   - All decoded orbits begin at the fixed point 1.
2. **Symbolic expansion of the orbit code body.**
   - The symbolic body $\omega(n)$ is expanded from left to right using explicit decoding rules for the symbols A, C, and K.
3. **Reconstruction of odd transitions.**
   - Each symbolic step produces the exact sequence of even and odd integers that were compressed during orbit-code construction.
4. **Final 2-adic expansion.**
   - The suffix $\texttt{.}z$ instructs us to apply z final doublings, embedding the odd core into its full position in the Collatz graph.

The result is a strictly increasing sequence

$$1 = a_0 < a_1 < \cdots < a_r = N,$$

which is **exactly** the classical forward Collatz orbit of N.

Decoding conventions

Let p denote the current value at the end of the partially decoded chain. Let the symbolic word w be defined as follows:

- if the symbolic body of $\omega^*(N)$ is nonempty, prepend an initial A,
- if the symbolic body is empty, then $n = 1$ and the orbit consists solely of powers of 2.

The prepended A aligns the predecessor-oriented encoding of $\omega(n)$ with forward Collatz dynamics beginning at 1.

Symbolic decoding rules

We read the symbolic word from left to right.

Decoding an A symbol

An A symbol expands to:

1. two doublings,
2. followed by $(p-1)/3$ **unless the next symbol is** K, in which case the division is deferred until the end of the maximal K-run.

Thus an A contributes two even steps and conditionally one odd step.

Decoding a C symbol

A C symbol expands to:

1. one doubling,
2. followed by $(p-1)/3$ **unless the next symbol is** K, in which case the division is deferred until the end of the maximal K-run.

Thus a C contributes one even step and conditionally one odd step.

Decoding a K-run

A maximal run K^k represents a compressed 01-lift tower. It expands to:

1. $2k$ successive doublings,
2. followed by a single $(p-1)/3$ step.

This reconstructs exactly the even-only lift structure suppressed during orbit-code construction.

Final 2-adic expansion

After decoding the symbolic body, the trailing integer z instructs us to apply z final doublings:

$$p \leftarrow 2p \quad \text{(repeated z times)}.$$

These steps embed the odd core n into its full value N.

Exceptional case: the $\mathcal{P}_1$ family

If the symbolic body is empty, then the odd core is $n = 1$. In this case, the universal orbit code has the form $\mathbf{.}z$, and the orbit consists solely of powers of 2:

$$1 \rightarrow 2 \rightarrow 4 \rightarrow \cdots \rightarrow 2^z.$$

Example — decoding the orbit of 22

Let $N = 22$. From §5.13, its universal orbit code is

$$\omega^*(22) = \mathtt{KCKAC.1}.$$

We decode as follows:

1. Prepend the initial A, giving the word AKCKAC.
2. Begin at 1 and apply the symbolic decoding rules step by step.
3. After processing the symbolic body, apply the final doubling indicated by .1.

The resulting forward orbit is:

$$[1, 2, 4, 8, 16, 5, 10, 20, 40, 13, 26, 52, 17, 34, 11, 22].$$

Reversing this sequence yields the classical Collatz orbit descending from 22 to 1.

Correctness and reversibility

Theorem (Correctness and Reversibility of Universal Orbit Codes).

For every integer $N \geq 1$, decoding the universal orbit code $\omega^*(N)$ using the rules above reconstructs the **exact** classical forward Collatz orbit of N. The map $N \mapsto \omega^*(N)$ is injective, and decoding provides a left-inverse; hence on the set of valid codes produced by §5.13 it is bijective.

Proof.
Each decoding rule is the exact inverse of the symbolic compression rules used to construct orbit codes in §§5.8–§5.10, while the suffix $.z$ restores the deterministic 2-adic structure suppressed during acceleration. ■

A reference implementation to decode the **universal orbit code** of an integer N is provided in Appendix D.Sec-5.14.

5.15 Symbolic Distance and Collatz Stopping Times

In §5.14 we showed that the universal orbit code

$$\omega^*(N) = \omega(n).z$$

can be decoded symbolically to recover the **entire classical Collatz orbit** of N, including all intermediate even values, without applying the Collatz map directly.

In this section we go one step further. We show that the **length** of that orbit — the classical *Collatz stopping time* of N — can be computed **directly from the universal orbit code itself**, without reconstructing the orbit and without performing any numerical iteration.

This leads to a new invariant of the universal orbit code: the **symbolic distance** of N from 1.

Symbolic distance

Let

$$\omega^*(N) = \mathsf{b.}z$$

be the universal orbit code of N, where b is the symbolic body over $\{A, C, K\}$ and $z = \nu_2(N)$. As in §5.14, decoding proceeds by forming the symbolic word

- $w = \mathsf{Ab}$ if $\mathsf{b} \neq ""$,
- or directly from $\mathsf{.}z$ in the exceptional class $\mathcal{P}_1$.

Decoding w produces a strictly increasing forward chain

$$1 = a_0 < a_1 < \cdots < a_r = N.$$

The integer r is exactly the number of classical Collatz steps required to reach N from 1 (equivalently, from N to 1 in reverse). We define the **symbolic distance** of N to be this integer r.

Local contributions of symbols

Each symbol in the decoding word w contributes a fixed number of steps to the forward Collatz chain. These contributions depend only on the symbol and on whether it initiates a K-run.

Contribution of an A symbol

From §5.14, an A symbol expands to:

- two doublings, and
- an additional odd step $(p-1)/3$ **unless** the next symbol begins a K-run.

Thus each A contributes

$$2 + \mathbf{1}_{\text{next symbol is not } \mathsf{K}}.$$

Contribution of a C symbol

A C symbol expands to:

- one doubling, and
- an additional odd step $(p-1)/3$ **unless** the next symbol begins a K-run.

Thus each C contributes $1 + \mathbf{1}_{\text{next symbol is not } \mathsf{K}}$.

Contribution of a K-run

A maximal run K^k represents a compressed 01-lift tower. From §5.14, it expands to:

- $2k$ successive doublings, and
- exactly one odd step $(p-1)/3$.

Thus a K^k run contributes $2k + 1$
steps.

Contribution of the final $\texttt{.}z$

The trailing integer z contributes exactly z final doublings. Thus its contribution is simply z.

Exceptional case: the $\mathcal{P}_1$ family

If the symbolic body is empty, the universal orbit code has the form $\texttt{.}z$. In this case the decoded orbit is

$$1 \to 2 \to 4 \to \cdots \to 2^z,$$

and the symbolic distance (stopping time) is

$$\mathrm{st}(N) = z.$$

Symbolic stopping time formula

We now combine the local contributions into a single expression.

Theorem (Symbolic stopping time).

Let

$$\omega^*(N) = \mathsf{b}\texttt{.}z$$

be the universal orbit code of N, with b possibly empty. If $\mathsf{b} \neq$ "", let $w = \mathsf{Ab}$. Then the classical Collatz stopping time of N is

$$\boxed{\mathrm{st}(N) = \sum_{\mathsf{A}\in w} \left(2 + \mathbf{1}_{\text{next symbol is not } \mathsf{K}}\right) + \sum_{\mathsf{C}\in w} \left(1 + \mathbf{1}_{\text{next symbol is not } \mathsf{K}}\right) + \sum_{\mathsf{K}^k \subset w} (2k+1) + z.}$$

Here $\mathbf{1}_{\text{next symbol is not } \mathsf{K}}$ denotes the indicator function, equal to 1 if the symbol immediately following the current one does not begin a K-run, and 0 otherwise. If $\mathsf{b} =$ "", then $\mathrm{st}(N) = z$. Thus the stopping time of N is **completely determined by its universal orbit code**, without reconstructing the orbit or applying the Collatz map.

Interpretation

1. The symbolic body b determines the accelerated odd structure of the orbit.

2. The trailing integer z measures the 2-adic height above the odd core.

3. The stopping time decomposes naturally as

$$\mathrm{st}(N) = \mathrm{st}(n) + z,$$

   where n is the odd core of N.

4. Symbolic distance provides a purely combinatorial method for analyzing orbit lengths and stopping times.

This shows that **orbit length is a symbolic invariant**, not merely a numerical byproduct of iteration.

A reference implementation to calculate **stopping time** i.e., the number of **Collatz steps** an integer N takes to reach 1 given the **universal orbit** is provided in Appendix D.Sec-5.15.

5.16 Chapter 5 — Summary

Chapter 5 develops a complete **symbolic framework for the accelerated Collatz map**, culminating in a representation of the full classical Collatz dynamics on $\mathbb{Z}_{>0}$ that is **finite, reversible, and computable without iteration**.

The chapter proceeds in four conceptual phases.

Symbolic structure and family-level dynamics of odd orbits

Sections §§5.1–5.6 develop the symbolic structure of the accelerated Collatz map on odd integers and reinterpret the resulting predecessor theory in forward-dynamical terms. Odd predecessors steps fall into three canonical symbolic types, encoded by the letters A, C, and K, corresponding respectively to Type-A predecessors, Type-C predecessors, and compressed $\mathsf{01}$-lift towers. Each odd integer is assigned:

- a **finite symbolic predecessor word**, and
- a **canonical terminal seed** $m_0 \equiv 3, 9, 15 \pmod{24}$ at which its predecessor chain terminates.

This induces a partition of the odd integers into **terminal families** $\mathcal{F}(m_0)$ indexed by canonical seeds. These terminal families act as discrete dynamical levels: successor steps move within a level, while K'-passage induces descent between levels in the terminal family tree. Forward accelerated dynamics move within a fixed family until a distinguished boundary element $e(m_0)$ is reached, at which point k_0-normalization forces a **family drop** to a new terminal family. These drops define an induced family transition map Φ and give rise to the terminal chain τ, which records the ordered sequence of terminal families actually traversed by accelerated forward dynamics and is invariant across all members of the same family. Reversing these terminal chains yields reverse τ-chains, which assemble into a rooted **terminal family tree** that serves as the canonical phase space for accelerated Collatz dynamics at the family level.

Orbit codes, terminal families, and the Terminal Seed Atlas

Sections §§5.7–5.11 introduce **orbit codes** $\omega(n)$, which compress the entire accelerated predecessor structure of an odd integer into a finite symbolic string over $\{A, C, K\}$. Orbit codes are organized via a **Terminal Seed Atlas**, consisting of canonical terminal seeds m_0 together with their terminal orbit codes $\omega(m_0)$ and associated family data. For a general odd integer n, its orbit code $\omega(n)$ is obtained by **symbolic truncation** of $\omega(m_0)$ according to the structural signature of n. This symbolic representation has two complementary interpretations:

- **downward**, each $\omega(m_0)$ encodes the full structural predecessor tree beneath m_0;
- **upward**, each m_0 indexes a finite **terminal family** $\mathcal{F}(m_0)$ of odd integers sharing the same canonical label and the same terminal chain.

The boundary of each terminal family is determined symbolically by the maximal (A/C)-tail obtained from $\omega(m_0)$ prior to the first K (read from right to left), whose decoded head element is the family endpoint $e(m_0)$. To make global organization explicit, **route skeletons** $R(n)$ are introduced as condensed representations of orbit codes that record only the junctions between $\mathtt{01}$-lift phases and predecessor phases. These skeletons provide canonical macro-routes from 1 into terminal families and remain stable across family members up to the final endpoint. Together, orbit codes, terminal families, and route skeletons yield a finite, navigable symbolic atlas of all accelerated odd dynamics.

Universal signatures and universal orbit codes

Sections §§5.12–5.13 extend the symbolic framework from odd integers to **all positive integers**. Every integer $N \geq 1$ is uniquely written as $N = 2^z n$, where n is odd and $z = \nu_2(N)$. This leads to the **universal signature**

$$\sigma^*(N) = (z, w, m_0),$$

which combines deterministic 2-adic motion with symbolic predecessor structure. From this, a **universal orbit code**

$$\omega^*(N) = \omega(n)\,\mathbf{.}\,z$$

is defined, extending orbit codes to the full Collatz domain. Integers sharing the same odd core have identical symbolic bodies and differ only by their 2-adic height.

Decoding, full orbits, and stopping times

Sections §§5.14–5.15 show that universal orbit codes admit a **complete symbolic decoding**. Explicit expansion rules for A, C, and K^k symbols reconstruct the full classical Collatz orbit

$$1 = a_0 < a_1 < \cdots < a_r = N$$

exactly, including all intermediate even values, without applying the Collatz map. The decoding rules are shown to be the precise inverse of the symbolic compression used to construct orbit codes, establishing full reversibility. Finally, the **symbolic distance** of an integer is defined, allowing the classical Collatz stopping time $\mathrm{st}(N)$ to be computed directly from the universal orbit code, without orbit reconstruction or iteration.

Conceptual outcome

Taken together, Chapter 5 establishes that:

- the Collatz dynamics admit a **finite symbolic description**;
- all branching behavior is isolated to odd dynamics and encoded symbolically;
- odd integers are organized into **terminal families** indexed by canonical seeds;
- family endpoints and K'-passage induce a directed family transition structure recorded by τ;
- reverse τ-chains assemble into a rooted terminal family tree anchored at 1, organizing terminal families by canonical depth;
- route skeletons expose the global branching structure transparently;
- even dynamics are deterministic and recoverable;

- full orbits and stopping times are computable symbolically; and
- the accelerated and classical maps are unified within a single reversible framework.

This completes the symbolic program initiated in earlier chapters and prepares the ground for global synthesis in Chapter 6.

Chapter 6

Paper Summary and Outlook

This paper develops a **complete symbolic framework** for the accelerated Collatz map that unifies backward predecessor structure, forward dynamics, and classical Collatz orbits into a single coherent system. Rather than studying individual trajectories numerically, the work reframes the Collatz problem as a problem of **symbolic organization**: integers are classified, indexed, reconstructed, and analyzed using finite symbolic objects. The main contributions of the paper are summarized below.

6.1 Predecessor structure and canonical organization

Chapters 3 and 4 establish that every odd integer admits a **unique predecessor type** and belongs to a **finite predecessor chain** terminating at a canonical seed. This induces a natural partition of the odd integers into **predecessor classes**, organized by:

- unique predecessor maps,
- canonical terminal seeds,
- finite symbolic chains describing backward growth.

The predecessor structure shows that infinite backward trees collapse into **finite symbolic descriptions**, eliminating ambiguity in backward Collatz dynamics and isolating all branching behavior into a structured symbolic layer.

6.2 Orbit codes and symbolic compression

Sections 5.1–5.10 introduce the **orbit code** $\omega(n)$ as a finite symbolic encoding of the accelerated Collatz dynamics of an odd integer. The orbit code records:

- Type-A and Type-C transitions,
- compressed $\mathsf{01}$-lift towers represented by K-runs,
- the complete symbolic predecessor structure of n.

This encoding compresses arbitrarily long numerical orbits into **short symbolic words** while preserving full structural information. The construction is exact, deterministic, and reversible.

6.3 Terminal seeds, families, and the Atlas perspective

Canonical terminal seeds $m_0 \equiv 3, 9, 15 \pmod{24}$ emerge as **natural indexing objects** for the accelerated Collatz map. Each terminal seed m_0 defines a **finite terminal family** $\mathcal{F}(m_0)$ of odd integers sharing the same canonical structure. These families are generated symbolically from $\omega(m_0)$ and bounded by a distinguished family endpoint $e(m_0)$, determined by the final (A/C)-tail of the orbit code. The resulting **Terminal Seed Atlas** provides:

- a symbolic partition of the odd integers,
- a database of terminal orbit codes $\omega(m_0)$,
- explicit finite terminal families indexed by canonical seeds.

Condensed representations such as **route skeletons** further expose the global organization of the Collatz graph, revealing canonical macro-routes from 1 into each terminal family.

6.4 Family dynamics and the terminal family tree

Forward accelerated Collatz dynamics move within a fixed terminal family until normalization at the family endpoint forces a **family drop**. These drops define an induced family transition map and give rise to **terminal chains** τ that record the sequence of family-to-family transitions encountered during forward descent. Reversing these chains yields reverse τ-chains, and assembling them across canonical seeds produces a rooted **terminal family tree** anchored at the forward-terminal family 1. This tree organizes accelerated Collatz dynamics at the family level and provides a global hierarchical structure in which each terminal family occupies a unique canonical position. In this formulation, the terminal family tree functions as the canonical phase space for accelerated Collatz dynamics at the family level: forward iteration corresponds to descent along a unique branch, while reverse τ-chains encode canonical ancestry.

6.5 Universal orbit codes and full reconstruction

Sections 5.12–5.14 extend orbit codes to **universal orbit codes**

$$\omega^*(N) = \omega(n) \,.\, z,$$

which encode the complete position of any integer N in the Collatz graph. Universal orbit codes admit a **complete symbolic decoding** that reconstructs the entire classical Collatz orbit of N, including all intermediate even and odd values, **without applying the Collatz map directly**. Thus classical Collatz dynamics are shown to be a deterministic unfolding of symbolic data.

6.6 Symbolic distance and stopping times

Section 5.15 introduces **symbolic distance**, a new invariant derived directly from $\omega^*(N)$. The classical Collatz stopping time $\mathrm{st}(N)$ is computed exactly as a weighted sum of symbolic contributions arising from A-, C-, and K-runs together with the 2-adic height z. Stopping times thereby become **symbolic quantities**, computable without numerical iteration or orbit reconstruction.

6.7 Structural reformulation of the Collatz conjecture

While this work does not resolve the Collatz conjecture, it establishes a framework in which questions of global convergence can be posed with explicit, computable structural constraints. Every positive integer admits a unique symbolic reduction to an odd core n and a unique canonical terminal seed $m_0 \equiv 3, 9, 15 \pmod{24}$. The universal orbit code records this reduction bijectively. Consequently, global convergence of the Collatz map is equivalent to the assertion that **every canonical terminal seed** lies on a branch of the terminal family tree that connects to the root $\mathcal{F}(1)$ under the accelerated dynamics. In particular:

- any counterexample to convergence must be witnessed by at least one canonical terminal seed whose accelerated forward orbit avoids $\mathcal{P}_1$ indefinitely;
- conversely, a proof that all canonical terminal seeds enter $\mathcal{P}_1$ would imply convergence for all positive integers.

This does not resolve the conjecture, but it localizes it to a **discrete, symbolically indexed set with explicit structure**, clarifying precisely where any contradiction would have to arise and which structural features it would have to violate.

6.8 Outlook

The symbolic framework developed here suggests several directions for future work, including:

- refinement and further compression of orbit codes,
- asymptotic analysis of symbolic distance,
- probabilistic models over terminal families,
- structural constraints on possible infinite trajectories.

More broadly, the methods introduced in this paper demonstrate that problems defined by simple numerical rules may admit **unexpected symbolic structure** when viewed from an appropriate organizational perspective. *The Collatz problem, long approached as a numerical curiosity, emerges instead as a problem of symbolic geometry on the integers, with a discrete canonical phase space and computable structural invariants.*

Appendix A — Terminal Seed Atlas (excerpt)

This appendix presents an excerpt of the **Terminal Seed Atlas** for the accelerated Collatz map. Each entry is indexed by a **canonical terminal seed** $m_0 \equiv 3, 9, 15 \pmod{24}$ and records:

- the **terminal orbit code** $\omega(m_0)$, encoding the full symbolic predecessor structure from 1 to m_0; and
- the corresponding terminal family $\mathcal{F}(m_0)$, consisting of all odd integers n whose backward odd-predecessor chain terminates at the same canonical seed m_0.

Together, these data make explicit the partition of odd integers into **terminal-indexed symbolic families**, as developed in §§5.8–5.11 and used throughout Chapter 5. The terminal orbit codes listed here form the computational backbone of the Atlas referenced in §§5.9–5.14 and support:

- symbolic reconstruction of accelerated and classical Collatz orbits,
- classification of odd integers by terminal class, and
- computation of stopping times via symbolic distance.

Each table entry has the following form:

- **first column**: the canonical terminal seed m_0,
- **second column**: its terminal orbit code $\omega(m_0)$,
- **third column**: the terminal family $\mathcal{F}(m_0)$ (including m_0 itself), listed in algorithmic generation order,
- **fourth column**: the **entry point** $e(m_0)$ of the family (defined below), and
- **fifth column**: the route skeleton $R(m_0)$ (optional condensed macro-route from 1).

Definition (entry / exit point of a terminal family).

Let m_0 be a canonical terminal seed and let $\omega(m_0)$ be its terminal orbit code. Scan $\omega(m_0)$ from right to left and extract the maximal suffix over $\{A, C\}$ preceding the first occurrence of K. Decoding this suffix using the inverse successor maps A' and C' produces a finite list of elements of the terminal family $\mathcal{F}(m_0)$ above m_0, as described in §5.11. We define the entry point $e(m_0)$ to be the first element of this decoded list (equivalently: the farthest member of $\mathcal{F}(m_0)$ from m_0 along the final (A/C)-tail). If the suffix is empty, then $\mathcal{F}(m_0) = \{m_0\}$ and $e(m_0) = m_0$. In backward decoding, $e(m_0)$ is the natural entry point of the terminal family. In forward dynamics, the same value serves as the exit point at which a k_0-normalization step is forced, inducing a transition to a new terminal family as described in §5.5.

Terminal Seed Atlas (excerpt)

Terminal seed m_0	Orbit code $\omega(m_0)$	Terminal family $\mathcal{F}(m_0)$	Entry point $e(m_0)$	Route skeleton $R(m_0)$
3	KC	[3, 5]	5	[1, 5, 3]
9	KCKACCA	[9, 7, 11, 17, 13]	7	[1, 5, 3, 13, 9]
15	KCKKCCC	[15, 23, 35, 53]	23	[1, 5, 3, 53, 15]
27	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCAC	[27, 41, 31, 47, 71, 107, 161, 121, 91, 137, 103, 155, 233, 175, 263, 395, 593, 445]	41	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 27]
33	KCKACCKCAA	[33, 25, 19, 29]	25	[1, 5, 3, 13, 7, 29, 33]
39	KCKACCKCAKCACC	[39, 59, 89, 67, 101]	59	[1, 5, 3, 13, 7, 29, 25, 101, 39]
51	KCKACCKCKC	[51, 77]	77	[1, 5, 3, 13, 7, 29, 19, 77, 51]
57	KCKACCAKAACA	[57, 43, 65, 49, 37]	43	[1, 5, 3, 13, 9, 37, 57]
63	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAKCCCCC	[63, 95, 143, 215, 323, 485]	95	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 121, 485, 63]
75	KKKAC	[75, 113, 85]	113	[1, 85, 75]
81	KCKKCCCKA	[81, 61]	61	[1, 5, 3, 53, 15, 61, 81]
87	KCKACCAKAKCC	[87, 131, 197]	131	[1, 5, 3, 13, 9, 37, 49, 197, 87]
99	KCKACCAKKC	[99, 149]	149	[1, 5, 3, 13, 9, 149, 99]
105	KCKACCKCAKCKCCCA	[105, 79, 119, 179, 269]	79	[1, 5, 3, 13, 7, 29, 25, 101, 67, 269, 105]
111	KCKKCCCKAKAAAKCCCCKCCCCCACACCC	[111, 167, 251, 377, 283, 425, 319, 479, 719, 1079, 1619, 2429]	167	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111]
123	KCKACCKCAKCACCKACAC	[123, 185, 139, 209, 157]	185	[1, 5, 3, 13, 7, 29, 25, 101, 39, 157, 123]
129	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCKCCAAA	[129, 97, 73, 55, 83, 125]	97	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 31, 125, 129]
135	KCKACCAKAACAKACC	[135, 203, 305, 229]	203	[1, 5, 3, 13, 9, 37, 57, 229, 135]
147	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCKCCKC	[147, 221]	221	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 31, 125, 55, 221, 147]

Terminal seed m_0	Orbit code $\omega(m_0)$	Terminal family $\mathcal{F}(m_0)$	Entry point $e(m_0)$	Route skeleton $R(m_0)$
153	KCKACCAKAACKCA	[153, 115, 173]	115	[1, 5, 3, 13, 9, 37, 43, 173, 153]
159	KCKKCCCKAKAAAKCCCCACCCC	[159, 239, 359, 539, 809, 607, 911, 1367, 2051, 3077]	239	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 159]
171	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCACKAAAC	[171, 257, 193, 145, 109]	257	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 27, 109, 171]
177	KCKACCKCAAKA	[177, 133]	133	[1, 5, 3, 13, 7, 29, 33, 133, 177]
183	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCKCC	[183, 275, 413]	275	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 103, 413, 183]
195	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCKCCAKC	[195, 293]	293	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 31, 125, 73, 293, 195]
201	KKKKCCA	[201, 151, 227, 341]	151	[1, 341, 201]
207	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCKCCC	[207, 311, 467, 701]	311	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 175, 701, 207]
219	KCKACCKCAKCACCKACKCCAC	[219, 329, 247, 371, 557]	329	[1, 5, 3, 13, 7, 29, 25, 101, 39, 157, 139, 557, 219]
225	KCKACCKCKCKAKACCCCCCAA	[225, 169, 127, 191, 287, 431, 647, 971, 1457, 1093]	169	[1, 5, 3, 13, 7, 29, 19, 77, 51, 205, 273, 1093, 225]
231	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCAKKACCACC	[231, 347, 521, 391, 587, 881, 661]	347	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 41, 661, 231]
243	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACKC	[243, 365]	365	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 91, 365, 243]
249	KCKACCKCAKCKCCCKCACA	[249, 187, 281, 211, 317]	187	[1, 5, 3, 13, 7, 29, 25, 101, 67, 269, 79, 317, 249]
255	KCKACCKCKCKAKKCCCCCCC	[255, 383, 575, 863, 1295, 1943, 2915, 4373]	383	[1, 5, 3, 13, 7, 29, 19, 77, 51, 205, 273, 4373, 255]

Terminal seed m_0	Orbit code $\omega(m_0)$	Terminal family $\mathcal{F}(m_0)$	Entry point $e(m_0)$	Route skeleton $R(m_0)$
267	KKKACKAC	[267, 401, 301]	401	[1, 85, 75, 301, 267]
273	KCKACCKCKCKA	[273, 205]	205	[1, 5, 3, 13, 7, 29, 19, 77, 51, 205, 273]
279	KCKACCKCAKCACCKKCC	[279, 419, 629]	419	[1, 5, 3, 13, 7, 29, 25, 101, 39, 629, 279]
291	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCACKKC	[291, 437]	437	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 27, 437, 291]
297	KCKKCCCKAKAAAKCCCCKCCCCCACKCCCCA	[297, 223, 335, 503, 755, 1133]	223	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 283, 1133, 297]
303	KCKKCCCKAKAAAACCC	[303, 455, 683, 1025, 769, 577, 433, 325]	455	[1, 5, 3, 53, 15, 61, 81, 325, 303]
315	KCKACCKCAAKKCAC	[315, 473, 355, 533]	473	[1, 5, 3, 13, 7, 29, 33, 533, 315]
321	KCKACKKAA	[321, 241, 181]	241	[1, 5, 3, 13, 11, 181, 321]
327	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCKCKKAACACCCCAACC	[327, 491, 737, 553, 415, 623, 935, 1403, 2105, 1579, 2369, 1777, 1333]	491	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 31, 125, 83, 1333, 327]
339	KCKACCKCKCKAKACCCCCCKC	[339, 509]	509	[1, 5, 3, 13, 7, 29, 19, 77, 51, 205, 273, 1093, 127, 509, 339]
345	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCKCCAAKCA	[345, 259, 389]	259	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 31, 125, 97, 389, 345]
351	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKKCCCC	[351, 527, 791, 1187, 1781]	527	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 1781, 351]
363	KCKACCAKAACKCKCAAC	[363, 545, 409, 307, 461]	545	[1, 5, 3, 13, 9, 37, 43, 173, 115, 461, 363]
369	KCKAKKA	[369, 277]	277	[1, 5, 3, 13, 17, 277, 369]
375	KCKACCKCAKCKCCCKCKCC	[375, 563, 845]	563	[1, 5, 3, 13, 7, 29, 25, 101, 67, 269, 79, 317, 211, 845, 375]

Terminal seed m_0	Orbit code $\omega(m_0)$	Terminal family $\mathcal{F}(m_0)$	Entry point $e(m_0)$	Route skeleton $R(m_0)$
387	KCKKCCCKAKAAAKCCCCKCCCCCACACCCKACCCACCACAACCCCACKAKC	[387, 581]	581	[1, 5, 3, 53, 15, 61, 81, 325, 769, 3077, 607, 2429, 111, 445, 27, 109, 145, 581, 387]
393	KCKACCKCAKCKCCCKCACKCACCA	[393, 295, 443, 665, 499, 749]	295	[1, 5, 3, 13, 7, 29, 25, 101, 67, 269, 79, 317, 187, 749, 393]

(table continues algorithmically; the Terminal Seed Atlas is defined by the procedures given below, and a computed prefix is shown here for illustration).

Code to construct the Atlas

Below is reference code that generates terminal orbit codes $\omega(m_0)$ for canonical terminal seeds and computes the associated terminal families, entry points, and route skeletons. This code is provided purely for illustration; no results in this paper depend on its correctness.

```
def iter_mod24(residue=None, start=1):
    ”””
    Yield odd integers n >= start whose residue class modulo 24 matches a
    selected canonical terminal-seed class.

    If residue is None, yield n such that (n % 24) is in {3, 9, 15}.
    If residue is one of {3, 9, 15}, yield n such that (n % 24) == residue.

    This is a generator (streams values) so you can build an ATLAS without
    storing the whole list first.
    ”””
    valid = {3, 9, 15}
    if residue is not None and residue not in valid:
        raise ValueError(”residue must be one of {3, 9, 15} or None”)

    # Ensure we start on an odd n
    n = start if start % 2 == 1 else start + 1

    while True:
        r = n % 24
        if residue is None:
            if r in valid:
                yield n
        else:
            if r == residue:
                yield n
        n += 2


def collect_mod24(limit, residue=None, start=1):
    ”””Return a list of the first `limit` values from iter_mod24(...).”””
    if limit is None or limit < 1:
        raise ValueError(”limit must be a positive integer”)
    out = []
    gen = iter_mod24(residue=residue, start=start)
    for _ in range(limit):
        out.append(next(gen))
    return out


def indexed_integers_by_terminal_seed(m0, ORBITS, max_steps=100000):
    ”””
    Return the odd integers that share terminal seed m0, generated symbolically
    from the rightmost A/C suffix of the terminal orbit code of m0.

    Method:
      - Let code = terminal_orbit_code(m0).
      - Scan code from right to left until (but not including) the first 'K'.
      - Starting at x = m0, apply Aprime/Cprime for each scanned symbol
        (in that right-to-left order), collecting each new x.
      - Stop before 'K' because applying Kprime would exit the terminal class.

    Example: for m0 = 891, this yields
      [1337, 1003, 1505, 1129, 847, 1271, 1907, 2861]
    ”””
    if m0 % 2 == 0:
        raise ValueError(”m0 must be odd”)

    code = terminal_orbit_code(m0, ORBITS, max_steps=max_steps)

    # Collect maximal right suffix over {A,C} before the first K from the right.
    suffix = []
    i = len(code) - 1
    while i >= 0 and code[i] != ”K”:
        ch = code[i]
        if ch != ”A” and ch != ”C”:
            raise ValueError(”ω(m0) contains an unexpected symbol: ” + str(ch))
        suffix.append(ch)  # this is in right-to-left application order
        i -= 1

    # Apply primes in the same order we collected (right-to-left).
    x = m0
    out = []
    for ch in suffix:
        if ch == ”A”:
            x = Aprime(x)
        else:  # ch == ”C”
            x = Cprime(x)
        out.append(x)

    return out


def terminal_family_and_entry_point(m0: int, ORBITS: dict, max_steps: int = 100000):
    ”””
    Return (family, entry_point) for the terminal family indexed by m0.

    - family is returned as [m0, ...above...], where ...above... is the list
      produced by indexed_integers_by_terminal_seed(m0) (farthest-first).
    - entry_point is the farthest element above m0, or m0 if the family
      contains only m0.
    ”””
    above = indexed_integers_by_terminal_seed(m0, ORBITS, max_steps=max_steps)  #
 ↪  farthest-first
    family = [m0] + list(above)

    entry_point = above[0] if len(above) > 0 else m0
    return family, entry_point


def build_terminal_orbit_atlas(limit=100, residue=None, start=1, max_steps=100000):
    ”””
    Build a mapping from canonical terminal seeds to their orbit codes.

    Returns a dict ATLAS where:
      ATLAS[m0] = orbit_code(m0)

    Parameters:
      limit      number of canonical seeds to include
      residue    optional residue class filter in {3, 9, 15}
      start      minimum value to start scanning from
      max_steps  safety cap passed to orbit_code(...)
    ”””
    ATLAS = {}
    for m0 in collect_mod24(limit=limit, residue=residue, start=start):
        ATLAS[m0] = orbit_code(m0, max_steps=max_steps)
    return ATLAS


def highlight_entry(family, entry):
    ”””
    Return a string representation of a family list with the entry element
 ↪  highlighted.

    The entry value is wrapped in asterisks (e.g. *x*). All other elements are
    rendered normally. If family is empty, it is returned unchanged.
    ”””
    if not family:
        return family
    out = []
    for x in family:
        out.append(f”*{x}*” if x == entry else str(x))
    return ”[” + ”, ”.join(out) + ”]”


def print_atlas_rows(limit=100, residue=None, start=1, max_steps=100000):
    ”””
    Print a tabular summary of terminal atlas entries.

    For each canonical terminal seed m0, this prints:
      m0, orbit code, terminal family, entry point, and route skeleton.

    Intended for inspection/debugging, not as a data API.
    ”””
    ATLAS = build_terminal_orbit_atlas(limit=limit, residue=residue, start=start,
 ↪  max_steps=max_steps)

    keys = sorted(ATLAS.keys())

    for m0 in keys:
        code = ATLAS[m0]
        family, entry = terminal_family_and_entry_point(m0, ATLAS, max_steps=max_steps)
        route = route_skeleton(m0, ATLAS)

        print(f”{m0:5d}\t{code}\t{family}\t{entry}\t{route}”)
```

Appendix B — Atlas Construction and Reference Implementation

This appendix provides a **reference implementation** for constructing the Terminal Seed Atlas used throughout Chapter 5. The purpose of this code is **not** to introduce new mathematical ideas, but to:

- make the Atlas used in §§5.9–5.15 **fully reproducible**,
- provide a practical artifact for researchers who wish to explore or extend the framework computationally, and
- support independent verification of orbit codes, terminal families, route skeletons, and symbolic stopping-time calculations.

The implementation constructs a finite Atlas consisting of:

- canonical terminal seeds $m_0 \equiv 3, 9, 15 \pmod{24}$,
- their terminal orbit codes $\omega(m_0)$,
- the indexed terminal family $\mathcal{F}(m_0)$ derived from $\omega(m_0)$, and
- the optional route skeleton $R(m_0)$, a condensed macro-route from 1.

Because the set of canonical terminal seeds is infinite, **no finite Atlas can be complete**. Any stored database (10K, 100K, 1M terminals, etc.) is therefore a finite excerpt intended for reference and experimentation. The Atlas continues algorithmically beyond any fixed bound. The code below produces portable Atlas artifacts in:

- compressed JSON format, and
- SQLite relational format,

both of which may be loaded directly into standard Python workflows.

B.1 Canonical terminal enumeration

```
import json
import gzip
import sqlite3
from typing import Dict, Any, Iterator, List, Optional


def iter_canonical_terminals(start: int = 1) -> Iterator[int]:
    ”””
    Yield odd integers m >= start such that m % 24 is in {3, 9, 15}.

    This is an infinite generator; oddness is enforced internally.
    ”””
    m = start | 1  # force odd
    while True:
        if m % 24 in (3, 9, 15):
            yield m
        m += 2
```

B.2 Building the Atlas

Important note. The function **indexed_integers_by_terminal_seed(m0, ORBITS, ...)** (as defined in §5.11) expects an orbit-code lookup dictionary **ORBITS** so it can retrieve $\omega(m_0)$. Here we build **ORBITS** directly as we go.

Also, the indexed family returned by **indexed_integers_by_terminal_seed(...)** is typically the “above m_0” list; the family stored in the Atlas is defined as $\mathcal{F}(m_0) = [m_0] +$ (above-list).

If you additionally want to record the **entry point** of the family (the first element above m_0 when it exists), we store:

- **entry_point** = **family**[**1**] when the family has length > 1,
- otherwise **entry_point** = **None**.

```
def build_atlas(
    num_terminals: int,
    family_limit: int = 50,
    max_steps: int = 100000,
    start: int = 1,
) -> Dict[int, Dict[str, Any]]:
    ”””
    Build a finite terminal-seed atlas.

    Returns a dict mapping each terminal seed m0 to a record with:
      - ”omega”: orbit code of m0
      - ”family”: list of odd integers in the terminal family of m0,
                  truncated to family_limit
      - ”entry_point”: first element above m0 in the family, or None
      - ”route_skeleton”: condensed macro-route for m0

    The atlas is built incrementally so downstream functions can resolve
    orbit codes via the shared ORBITS mapping.
    ”””
    if num_terminals < 1:
        raise ValueError(”num_terminals must be a positive integer”)
    if family_limit < 1:
        raise ValueError(”family_limit must be a positive integer”)

    ATLAS: Dict[int, Dict[str, Any]] = {}
    ORBITS: Dict[int, str] = {}  # m0 -> omega(m0)

    gen = iter_canonical_terminals(start=start)

    for _ in range(num_terminals):
        m0 = next(gen)
        # Store omega first so downstream functions can resolve ORBITS[m0].
        omega = orbit_code(m0, max_steps=max_steps)
        ORBITS[m0] = omega

        # Terminal family: [m0] + above-list.
        above = indexed_integers_by_terminal_seed(m0, ORBITS, max_steps=max_steps)
        family = [m0] + list(above)

        # Optional: entry point into the family (first element above m0).
        entry_point = family[1] if len(family) > 1 else None

        # Optional condensed macro-route.
        route = route_skeleton(m0, ORBITS)

        ATLAS[m0] = {
            ”omega”: omega,
            ”family”: family[:family_limit],
            ”entry_point”: entry_point,
            ”route_skeleton”: route,
        }

    return ATLAS
```

B.3 Saving and loading the Atlas (JSON)

```
def save_atlas_json_gz(ATLAS: Dict[int, Dict[str, Any]], path: str) -> None:
    ”””
    Save the atlas to a gzip-compressed JSON file.

    Note: JSON object keys must be strings, so terminal seeds (m0) are
    converted to strings before serialization.
    ”””
    with gzip.open(path, ”wt”, encoding=”utf-8”) as f:
        json.dump({str(k): v for k, v in ATLAS.items()}, f, ensure_ascii=False)


def load_atlas_json_gz(path: str) -> Dict[int, Dict[str, Any]]:
    ”””
    Load a gzip-compressed JSON atlas.

    Terminal seed keys are converted back from strings to integers.
    ”””
    with gzip.open(path, ”rt”, encoding=”utf-8”) as f:
        data = json.load(f)
    return {int(k): v for k, v in data.items()}
```

B.4 Saving and loading the Atlas (SQLite)

SQLite is convenient when the Atlas becomes large and you want fast lookup by m_0 without loading a full JSON blob into memory.

```
def save_atlas_sqlite(ATLAS: Dict[int, Dict[str, Any]], db_path: str) -> None:
    ”””
    Save the atlas to a SQLite database.

    Creates two tables:
      - terminal_seed: stores m0, orbit code, entry point, and route skeleton
      - terminal_family: stores the terminal family list for each m0

    Existing tables with the same names are dropped and recreated.
    ”””
    con = sqlite3.connect(db_path)
    cur = con.cursor()

    cur.execute(”DROP TABLE IF EXISTS terminal_seed”)
    cur.execute(”DROP TABLE IF EXISTS terminal_family”)

    cur.execute(”””
        CREATE TABLE terminal_seed (
            m0 INTEGER PRIMARY KEY,
            omega TEXT NOT NULL,
            entry_point INTEGER,
            route_skeleton_json TEXT
        )
    ”””)

    cur.execute(”””
        CREATE TABLE terminal_family (
            m0 INTEGER PRIMARY KEY,
            family_json TEXT NOT NULL,
            FOREIGN KEY (m0) REFERENCES terminal_seed(m0)
        )
    ”””)

    for m0, rec in ATLAS.items():
        cur.execute(
            ”INSERT INTO terminal_seed VALUES (?, ?, ?, ?)”,
            (
                m0,
                rec[”omega”],
                rec.get(”entry_point”, None),
                json.dumps(rec.get(”route_skeleton”, [])),
            ),
        )
        cur.execute(
            ”INSERT INTO terminal_family VALUES (?, ?)”,
            (m0, json.dumps(rec[”family”])),
        )

    con.commit()
    con.close()


def load_atlas_sqlite(db_path: str) -> Dict[int, Dict[str, Any]]:
    ”””
    Load a terminal-seed atlas from a SQLite database.

    Reconstructs the same dictionary structure produced by save_atlas_sqlite().
    ”””
    con = sqlite3.connect(db_path)
    cur = con.cursor()

    cur.execute(”””
        SELECT s.m0, s.omega, s.entry_point, s.route_skeleton_json, f.family_json
        FROM terminal_seed s
        JOIN terminal_family f ON s.m0 = f.m0
        ORDER BY s.m0
    ”””)

    ATLAS: Dict[int, Dict[str, Any]] = {}
    for m0, omega, entry_point, route_json, fam_json in cur.fetchall():
        ATLAS[m0] = {
            ”omega”: omega,
            ”entry_point”: entry_point,
            ”route_skeleton”: json.loads(route_json) if route_json else [],
            ”family”: json.loads(fam_json),
        }

    con.close()
    return ATLAS
```

B.5 Example usage

```
# Build a 10K-terminal excerpt (with a short family prefix per seed).
ATLAS = build_atlas(num_terminals=10000, family_limit=50, max_steps=100000)

# Save
save_atlas_json_gz(ATLAS, ”output/atlas_10000.json.gz”)
save_atlas_sqlite(ATLAS, ”output/atlas_10000.sqlite”)

# Load
A1 = load_atlas_json_gz(”output/atlas_10000.json.gz”)
A2 = load_atlas_sqlite(”output/atlas_10000.sqlite”)
```

Appendix C — Programmatic construction of route skeleton diagrams

The route skeleton introduced in §5.10.9 is defined purely in symbolic terms, as a projection of the orbit code $\omega(n)$ onto the endpoints of its maximal symbolic phases. Nevertheless, because all constructions in Chapters 4–5 are deterministic and finite, the route skeleton admits a direct algorithmic realization. This appendix records a **reference procedure** used to generate the route skeleton diagrams appearing in Chapter 5.

C.1 Input data

The procedure takes as input:

- an odd integer n,
- its orbit code $\omega(n) \in \{A, C, K\}^*$,
- the **fully decoded structural predecessor chain**

  $1 = v_0 \to v_1 \to \cdots \to v_\ell = n$,

  obtained by decoding $\omega(n)$ as described in §5.8–§5.10,
- the corresponding accelerated Collatz orbit.

These objects are defined symbolically in §5.8–§5.10 and require no numerical iteration of the Collatz map beyond symbolic decoding.

C.2 Route skeleton extraction

The route skeleton $R(n)$ is computed by parsing $\omega(n)$ into maximal blocks of the form K^t or $(A/C)^+$, and recording the structural chain nodes reached at the completion of each block. Formally, if

$\omega(n) = B_1 B_2 \cdots B_r$,

then

$R(n) = [v_0, v_1, \dots, v_r]$,

where $v_0 = 1$ and v_i is the odd integer obtained after decoding the prefix $B_1 \cdots B_i$. This construction exactly matches Definition 5.10.9 and is independent of any visualization choices.

C.3 Diagram construction

To produce the diagrams in Chapter 5, the following graphical conventions are applied:

- **Nodes** correspond to odd integers in the structural predecessor chain.
- **Route skeleton nodes** are highlighted with a filled background.
- **Suppressed nodes**—those appearing in the structural chain but omitted from the accelerated orbit—are rendered with dashed borders.
- **Solid edges** represent accelerated predecessor transitions.
- **Dashed edges** represent internal 01-lift connectors collapsed by the accelerated map, with the final connector in each maximal lift phase labeled “01-lift”.
- **Level indicator extensions** (cf. §5.6) are drawn horizontally from selected nodes to visualize the terminal collapse level of their reverse τ-chains.

Nodes are arranged in horizontal lanes corresponding to successive symbolic phases, so that each lane visually represents a maximal block of $\omega(n)$. The resulting diagram makes explicit the phase junctions of the predecessor structure and visually isolates the macro-route from 1 into the terminal family indexed by n.

C.4 Reference implementation

A reference implementation of this construction was written in Python and emits Graphviz DOT code, which is subsequently rendered as an SVG diagram. The implementation follows the definitions in §5.8–§5.11 exactly and performs no heuristic simplifications or numerical exploration. Its role is purely illustrative: all mathematical statements in the paper are established independently of any computational output. In particular, no diagrammatic property is used in any proof. The complete source code used to generate the figures in Chapter 5 is provided below.

```
from __future__ import annotations

from typing import List, Dict, Callable, Optional, Set
import subprocess
import shutil


def parse_blocks(code: str) -> List[str]:
    ”””
    Split an orbit code into maximal contiguous blocks.

    Each block is either a run of 'K' symbols or a run of 'A'/'C' symbols.
    ”””
    blocks: List[str] = []
    i = 0
    while i < len(code):
        ch = code[i]
        if ch == ”K”:
            j = i
            while j < len(code) and code[j] == ”K”:
                j += 1
            blocks.append(code[i:j])
            i = j
        else:  # A or C
            j = i
            while j < len(code) and code[j] in (”A”, ”C”):
                j += 1
            blocks.append(code[i:j])
            i = j
    return blocks


def route_skeleton_from_code(code: str, structural_chain: List[int]) -> List[int]:
    ”””
    Compute the route skeleton from an orbit code and its structural chain.

    The structural_chain must have length len(code) + 1. The returned list
    contains the endpoints after each maximal block in the code.
    ”””
    if len(structural_chain) != len(code) + 1:
        raise ValueError(”Expected len(structural_chain) == len(ω(n)) + 1”)

    blocks = parse_blocks(code)
    endpoints = [structural_chain[0]]
    idx = 0
    for b in blocks:
        idx += len(b)
        endpoints.append(structural_chain[idx])
    return endpoints


def make_route_dot(
    n: int,
    code: str,
    structural_chain: List[int],
    accelerated_chain: List[int],
    *,
    graph_name: str | None = None,
    fillcolor_route: str = ”#7FA6E6”,
    nodesep: float = 0.15,
    ranksep: float = 0.65,
    font_node: str = ”Helvetica”,
    font_edge: str = ”Helvetica”,
    fontsize_node: int = 12,
    fontsize_edge: int = 11,
    penwidth_solid: float = 2.2,
    penwidth_dashed: float = 0.55,

    # Level-indicator extension
    predecessor_chain_fn: Optional[Callable[[int], List[int]]] = None,
    add_level_indicators: bool = False,

    # Styling for ”true terminal” of a level chain when it’s an extension-only node
    level_terminal_color: str = ”#E57373”,  # soft red

    # NEW: also treat any of these as ”indicator seeds”, even if not suppressed
    extra_indicator_seeds: Optional[List[int]] = None,
) -> str:
    ”””
    Return a Graphviz DOT string for a Collatz route diagram.

    Inputs:
      - n: target integer shown at the top of the diagram
      - code: orbit code for n (string over {A, C, K})
      - structural_chain: full structural predecessor chain (len = len(code) + 1)
      - accelerated_chain: accelerated predecessor chain (subset of structural_chain)

    Features:
      - Highlights the route skeleton induced by maximal K-runs and (A/C)-runs.
      - Draws the accelerated backbone as solid edges and 01-lift bridges as dashed.
      - Optionally adds ”level-indicator” side chains using predecessor_chain_fn.
    ”””
    gname = graph_name or f”Collatz{n}”

    route_nodes = set(route_skeleton_from_code(code, structural_chain))

    structural_set: Set[int] = set(structural_chain)
    accel_set: Set[int] = set(accelerated_chain)

    suppressed: Set[int] = {x for x in structural_set if x not in accel_set}

    # Node ids
    def nid(x: int) -> str:
        return f”n_{x}”

    def lid(seed: int, x: int) -> str:
        return f”lvl_{seed}_{x}”

    # ---------
    # Helper: build the FULL ”level chain” by repeatedly chaining
      ↪ predecessor_chain(last)
    # until we hit the diagram’s target n (e.g., 27), or we can’t progress.
    # Returns a list of values to place to the RIGHT of the seed.
    # ---------
    def transitive_level_chain(seed: int) -> List[int]:
        if predecessor_chain_fn is None:
            return []

        seen: Set[int] = set()
        out: List[int] = []

        cur = seed
        while True:
            if cur in seen:
                break
            seen.add(cur)

            chain = predecessor_chain_fn(cur) or []
            # drop -1 if present (your predecessor_chain ends with -1)
            chain = [x for x in chain if x != -1]

            if not chain:
                break

            # If your predecessor_chain(cur) returns a list that ends in cur itself,
            # drop that self-repeat so we only draw nodes to the right.
            if chain and chain[-1] == cur:
                chain = chain[:-1]
                if not chain:
                    break

            # Append, but stop early if we reach the target n
            for x in chain:
```

```
            out.append(x)
            if x == n:
                return out

        cur = out[-1]

    return out

# ---------
# Build level chains for:
#   - all suppressed nodes (as before)
#   - plus any extra indicator seeds (e.g., [137]) you want
# ---------
level_chains: Dict[int, List[int]] = {}
indicator_seeds: Set[int] = set(suppressed)
if extra_indicator_seeds:
    indicator_seeds |= set(extra_indicator_seeds)

if add_level_indicators:
    if predecessor_chain_fn is None:
        raise ValueError(”add_level_indicators=True requires
          ↪ predecessor_chain_fn.”)

    for s in indicator_seeds:
        chain = transitive_level_chain(s)
        level_chains[s] = chain

# Track ”true terminal endpoints” of level chains (e.g., 27)
level_terminals: Set[int] = set()
for s, ch in level_chains.items():
    if ch:
        level_terminals.add(ch[-1])

# ---------
# Main nodes
# ---------
node_lines: List[str] = []
for x in structural_chain:
    styles: List[str] = []
    label = str(x)

    is_supp = x in suppressed
    is_route = x in route_nodes
    # is_indicator_seed = add_level_indicators and (x in indicator_seeds)
    # is_true_terminal = add_level_indicators and (x in level_terminals)

    # Terminals are the *endpoints* of the transitive level chains,
    # plus any indicator seed whose chain is empty (e.g., 3).
    terminal_indicators: Set[int] = set(level_terminals)
    if add_level_indicators:
        for s in indicator_seeds:
            if not level_chains.get(s, []):
                terminal_indicators.add(s)

    is_terminal_indicator = add_level_indicators and (x in terminal_indicators)


    extra_attrs_parts: List[str] = []
```

```
        if is_supp:
            label = f”{x}\\n(suppressed)”
            styles.append(”dashed”)

        if is_route:
            styles.append(”filled”)

        # Red dashed outline ONLY for terminal indicators (123,39,33,9,3 in your
          ↪  example)
        if is_terminal_indicator:
            extra_attrs_parts.append('color=”#C62828”')
            extra_attrs_parts.append(”penwidth=2.6”)
            if ”dashed” not in styles:
                styles.append(”dashed”)

        extra_attrs = ””
        if extra_attrs_parts:
            extra_attrs = ”, ” + ”, ”.join(extra_attrs_parts)

        if styles:
            style_str = ”,”.join(styles)
            if ”filled” in styles:
                node_lines.append(
                    f'  {nid(x)} [label=”{label}”, style=”{style_str}”,
                      ↪  fillcolor=”{fillcolor_route}”{extra_attrs}];'
                )
            else:
                node_lines.append(
                    f'  {nid(x)} [label=”{label}”, style=”{style_str}”{extra_attrs}];'
                )
        else:
            node_lines.append(f'  {nid(x)} [label=”{label}”{extra_attrs}];')

    # ---------
    # Rank lanes (your existing logic)
    # ---------
    pos: Dict[int, int] = {v: i for i, v in enumerate(structural_chain)}
    rs = route_skeleton_from_code(code, structural_chain)
    accel_index = {v: i for i, v in enumerate(accelerated_chain)}

    def is_skip_target(b: int) -> bool:
        j = accel_index.get(b)
        if j is None or j == 0:
            return False
        prev = accelerated_chain[j - 1]
        return abs(pos[prev] - pos[b]) > 1

    rank_lines: List[str] = []
    for a, b in zip(rs[:-1], rs[1:]):
        if is_skip_target(b):
            continue
        i, j = pos[a], pos[b]
        lane = structural_chain[i : j + 1] if i <= j else structural_chain[j : i + 1]
        rank_lines.append(”  { rank=same; ” + ”; ”.join(nid(x) for x in lane) + ” }”)

    # ---------
    # Level-indicator extension nodes/edges/ranks
    #   IMPORTANT: reuse existing main nodes when possible (avoid duplicates)
    # ---------
    level_node_lines: List[str] = []
    level_edge_lines: List[str] = []
    level_rank_lines: List[str] = []

    def ref_for(seed: int, x: int) -> str:
        return nid(x) if x in structural_set else lid(seed, x)

    if add_level_indicators:
        for s, chain in level_chains.items():
            if not chain:
                continue

            # Create extension-only nodes (only those NOT in main structural chain)
            for i, x in enumerate(chain):
                if x in structural_set:
                    continue  # reuse existing node
                is_terminal = (i == len(chain) - 1)
                if is_terminal:
                    level_node_lines.append(
                        f'  {lid(s, x)} [label=”{x}”, shape=box, '
                        f'style=”dashed,filled”, fillcolor=”{level_terminal_color}”, '
                        f'fontname=”{font_node}”, fontsize={fontsize_node},
                          ↪  margin=”0.12,0.08”];'
                    )
                else:
                    level_node_lines.append(
                        f'  {lid(s, x)} [label=”{x}”, shape=box, style=”dashed”, '
                        f'fontname=”{font_node}”, fontsize={fontsize_node},
                          ↪  margin=”0.12,0.08”];'
                    )

            # dashed edges: s -> chain[0] -> chain[1] -> ...
            level_edge_lines.append(f'  edge [penwidth={penwidth_dashed},
↪   style=”dashed”];')
            level_edge_lines.append(f”  {nid(s)} -> {ref_for(s, chain[0])};”)
            for a, b in zip(chain[:-1], chain[1:]):
                level_edge_lines.append(f”  {ref_for(s, a)} -> {ref_for(s, b)};”)

            # same rank: s and all chain nodes
            level_rank_lines.append(
                ”  { rank=same; ” + ”; ”.join([nid(s)] + [ref_for(s, x) for x in
                  ↪  chain]) + ” }”
            )

    # ---------
    # Accelerated backbone + 01-lift bridges (unchanged)
    # ---------
    edge_solid: List[str] = []
    edge_dashed: List[str] = []

    def structural_path(u: int, v: int) -> List[int]:
        iu, iv = pos[u], pos[v]
        if iu <= iv:
            return structural_chain[iu : iv + 1]
        return structural_chain[iv : iu + 1]

    edge_solid.append(f'  edge [penwidth={penwidth_solid}, style=”solid”];')
    for u, v in zip(accelerated_chain[:-1], accelerated_chain[1:]):
        path = structural_path(u, v)
        if len(path) == 2:
            edge_solid.append(f”  {nid(u)} -> {nid(v)};”)
        else:
            edge_solid.append(f'  {nid(u)} -> {nid(v)} [label=”accelerated”];')
            edge_dashed.append(f'  edge [penwidth={penwidth_dashed},
 ↪ style=”dashed”];')
            for a, b in zip(path[:-1], path[1:]):
                if b == path[-1]:
                    edge_dashed.append(
                        f'  {nid(a)} -> {nid(b)} [label=” 01-lift”, style=”dashed”,
                          ↪ weight=5];'
                    )
                else:
                    edge_dashed.append(f”  {nid(a)} -> {nid(b)};”)

    # ---------
    # Assemble DOT
    # ---------
    dot: List[str] = []
    dot.append(f”digraph {gname} {{”)
    dot.append(”  rankdir=BT;”)
    dot.append(”  splines=true;”)
    dot.append(f”  nodesep={nodesep};”)
    dot.append(f”  ranksep={ranksep};”)
    dot.append(””)
    dot.append(
        f'  node [shape=box, fontname=”{font_node}”, fontsize={fontsize_node},
          ↪ margin=”0.12,0.08”];'
    )
    dot.append(f'  edge [fontname=”{font_edge}”, fontsize={fontsize_edge},
 ↪ arrowsize=0.7];')
    dot.append(””)

    dot.append(”  // Nodes”)
    dot.extend(node_lines)

    if level_node_lines:
        dot.append(””)
        dot.append(”  // Level-indicator extension nodes”)
        dot.extend(level_node_lines)

    dot.append(””)
    dot.append(”  // Rank lanes (optional)”)
    dot.extend(rank_lines)

    if level_rank_lines:
        dot.append(””)
        dot.append(”  // Level-indicator rank lanes”)
        dot.extend(level_rank_lines)

    dot.append(””)
    dot.append(”  // Accelerated backbone + dashed 01-lift bridges”)
    dot.extend(edge_solid)
    if edge_dashed:
        dot.append(””)
        dot.extend(edge_dashed)

    if level_edge_lines:
        dot.append(””)
```

```
        dot.append(”  // Level-indicator dashed edges”)
        dot.extend(level_edge_lines)

    dot.append(””)
    dot.append('  edge [penwidth=1.0, style=”solid”];')
    dot.append(”}”)
    return ”\n”.join(dot)


def render_svg(dot_text: str, svg_path: str) -> None:
    ”””
    Render a Graphviz DOT string to an SVG file.

    Requires the `dot` executable from Graphviz to be available on PATH.
    The DOT source is passed via stdin and the rendered SVG is written to svg_path.
    ”””
    if shutil.which(”dot”) is None:
        raise RuntimeError(”Graphviz 'dot' not found. Install Graphviz to render
          ↪  SVG.”)
    proc = subprocess.run(
        [”dot”, ”-Tsvg”],
        input=dot_text.encode(”utf-8”),
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        check=True,
    )
    with open(svg_path, ”wb”) as f:
        f.write(proc.stdout)
```

The following example illustrates how route skeleton diagrams may be generated programmatically from orbit codes and predecessor chains. Given an odd integer n, we compute its orbit code $\omega(n)$, full structural predecessor chain, and accelerated predecessor chain, and then emit a **Graphviz DOT** description encoding both the structural and accelerated views, with route skeleton nodes highlighted.

```
n = 57

dot = make_route_dot(
    n=n,
    code=orbit_code(n),
    structural_chain=structural_predecessor_chain(n, ORBITS),
    accelerated_chain=accelerated_predecessor_chain(n, ORBITS),
    graph_name=”Collatz_StructuralVsAccelerated_level”,
    predecessor_chain_fn=predecessor_chain,
    add_level_indicators=True,
    extra_indicator_seeds=[n],
)

with open(f”output/Collatz_StructuralVsAccelerated_{n}.dot”, ”w”, encoding=”utf-8”)
 ↪  as f:
    f.write(dot)

# Optional SVG render (requires Graphviz installed)
render_svg(dot, ”output/Collatz_StructuralVsAccelerated.svg”)
```

The generated file Collatz_general_StructuralVsAccelerated_n.dot is a plain-text Graphviz description and may be rendered in several ways:

- using a local Graphviz installation via dot -Tsvg file.dot -o file.svg, or
- by pasting the DOT source into an online Graphviz renderer such as `https://dreampuf.github.io/GraphvizOnline`.

This approach separates **symbolic computation** from **visualization**, allowing diagrams to be regenerated, styled, or extended without modifying the underlying mathematical framework.

Appendix D — Reference Implementation (Auxiliary Code)

Appendix D collects **reference implementations** of the constructions developed in Chapters 4 and 5. The emphasis is on **clarity and fidelity to the definitions**, rather than micro-optimization. The appendix is organized progressively: later subsections build on earlier helpers, broadly mirroring the logical development of the theory. Readers who want a specific construction may jump directly to the relevant subsection and trace dependencies upward as needed.

- **Local runnability.** A small set of helpers (e.g., `nu2`, `predecessor`, and the branch maps `Aprime/Cprime/Kprime`) may appear in more than one subsection so that code blocks can be executed independently. When duplicated, these helpers are intended to be **identical**.
- **Name collisions (single-file use).** If Appendix D is executed as a single Python module, some function names may be redefined in later subsections (e.g., alternative `successor` implementations used for symbolic analysis). In that case, treat each subsection as a self-contained reference block, or rename functions when consolidating into a single script.

D.Core

This subsection defines the **core arithmetic primitives** used throughout the appendix:

- valuations `nu2` and `nu3`,
- the accelerated Collatz successor `successor(n)`,
- the canonical inverse rule `predecessor(n)` (smallest accelerated predecessor, with a sentinel for type-B),
- and the normalization map `k0_normalize(n)` that strips trailing `01` lifts to reach the base of the associated 01-tower.

All subsequent routines either call these directly or reproduce them verbatim when a subsection is designed to run standalone.

```
from math import isqrt

def nu2(x: int) -> int:
    ”””
    Return the exponent of 2 in the prime factorization of x (x > 0).

    This is the largest integer k such that 2**k divides x.
    ”””
    return (x & -x).bit_length() - 1

def nu3(x: int) -> int:
    ”””
    Return the 3-adic valuation of x.

    This is the largest integer k such that 3**k divides x.
    By convention in this codebase, nu3(0) returns 0
    (x = 0 does not occur in normal use).
    ”””
    if x == 0:
        return 0
    k = 0
    while x % 3 == 0:
        x //= 3
        k += 1
    return k

def successor(n: int) -> int:
    ”””
    Return the accelerated Collatz successor of an odd integer n.

    Computes s = 3*n + 1 and removes all factors of 2 by dividing by 2**nu2(s),
    returning the next odd value in the Collatz sequence.
    ”””
    s = 3 * n + 1
    return s » nu2(s)   # divide by 2**nu2(s)

def predecessor(n: int) -> int:
    ”””
    Return the canonical (smallest) odd predecessor of an odd integer n in the
    accelerated Collatz graph.

    The value is computed by the closed-form inverse rule determined by n mod 6.
    For type-B inputs (n % 6 == 3), the sentinel value -1 is returned.
    ”””
    return (((n - 3) % 6) * n - 1) // 3

def k0_normalize(n: int) -> int:
    ”””
    Normalize n to the odd base of its 01-tower.

    This uses the 2-adic valuation of (3*n + 1) to remove any appended 01-lifts
    in one shot, then applies a parity fix to ensure the result is odd.
    ”””
    s = 3*n + 1
    r = nu2(s)             # ν₂(3n + 1)
    a = r // 2             # number of appended 01-blocks
    t = ((s » (2*a)) - 1) // 3

    # Parity fix: ensure the base is odd; if stripping 01 blocks lands on an even t,
     ↪  lift once)
    k0 = t + ((1 + (-1)**t) // 2) * (3*t + 1) # or k0 = t if (t & 1) else (4*t + 1)

    return k0
```

D.Sec-2.1

```
def successor_loop(n: int) -> int:
    ”””
    Return the accelerated Collatz successor of an odd integer n.

    Computes 3*n + 1 and removes all factors of 2, returning the next odd value.
    ”””
    succ = 3 * n + 1
    while succ % 2 == 0:
        succ //= 2
    return succ


def successor_chain(n: int) -> list[int]:
    ”””
    Return the odd-only accelerated Collatz chain starting from n and ending at 1.

    Each step applies successor(...) until 1 is reached.
    ”””
    chain = [n]
    while n != 1:
        n = successor(n)
        chain.append(n)
    return chain
```

D.Sec-2.2

```
def accelerated_preimages(n: int, length: int, a_max: int = 2000) -> list[int]:
    ”””
    Return up to `length` odd Collatz predecessors of n.

    The function searches exponents a >= 1 such that (2**a * n - 1) is divisible
    by 3 and yields a positive odd integer. Search stops once `length`
    predecessors are found or a exceeds `a_max`.

    If no valid predecessors exist within the search range, an empty list
    is returned.
    ”””
    predecessors = []
    a = 1

    while len(predecessors) < length and a <= a_max:
        numerator = (1 « a) * n - 1  # 2**a * n - 1
        if numerator % 3 == 0:
            candidate = numerator // 3
            if candidate > 0 and candidate % 2 == 1:
                predecessors.append(candidate)
        a += 1

    return predecessors
```

D.Sec-3.1

```
def scan_accelerated_preimages(n: int, max_x: int = 11, bits: int = 40) -> None:
    ”””
    Print a diagnostic scan of potential predecessors of an odd integer n.

    For each exponent x = 1..max_x, the function computes
        v = (2**x)*n - 1
    and reports whether v is divisible by 3.

    If divisible, it prints the candidate predecessor q = v // 3
    together with its binary representation, zero-padded to the
    specified bit width.

    This routine is intended for inspection and debugging of the
    backward Collatz condition (2**x * n - 1) divisible by 3.
    ”””
    for x in range(1, max_x + 1):
        v = (1 « x) * n - 1
        r = v % 3
        line = f”x= {x:<2}; \t (2^{x})*{n}-1= {v}; \t {v}%3= {r}”
        if r == 0:
            q = v // 3
            bin_q = ”0b” + format(q, f”0{bits}b”)
            line += f”; \t {v}/3= {q} \t bin({q})= {bin_q}”
        else:
            line += f”\t {v} is not divisible by 3;”
        print(line)
    print()  # blank line between n blocks
```

D.Sec-3.3

```
def predecessor_tower(n_t: int, count: int) -> list[int]:
    ”””
    Generate the first `count` elements of the predecessor 01-tower above n_t.

    The tower is rooted at the smallest accelerated predecessor of n_t and is
    obtained by repeated application of the 01-lift:
        x -> 4*x + 1.

    All returned values lie in the same accelerated predecessor class.
    ”””
    preds = []
    n_t_plus_1 = predecessor(n_t)


    if n_t_plus_1 == -1:
        # n_t is a B-node (no predecessor in the canonical inverse map)
        return []   # or raise ValueError if you'd rather fail loudly


    preds.append(n_t_plus_1)
    for _ in range(1, count):
        n_t_plus_1 = 4 * n_t_plus_1 + 1  # Alternatively, using bitwise operations: n_t_plus_1 = (n_t_plus_1 « 2) | 1
        preds.append(n_t_plus_1)

    return preds

def predecessor_tower_kth(n_t: int, k: int) -> int:
    ”””
    Return the k-th element of the predecessor 01-tower above n_t.

    Let p0 = predecessor(n_t). This returns K^k(p0), where the 01-lift
    is K(x) = 4*x + 1. Implemented via the closed form:
        K^k(x) = 4^k * x + (4^k - 1) / 3.
    ”””
    n_t_plus_1 = predecessor(n_t)

    if n_t_plus_1 == -1:
        raise ValueError(f”n_t={n_t} is a B-node; no canonical predecessor exists.”)

    pow4k = 1 « (2 * k)  # 4**k

    return pow4k * n_t_plus_1 + (pow4k - 1) // 3
```

D.Sec-3.4

```
def k0_normalize_iter(n: int) -> int:
    ”””
    Normalize n by repeatedly removing trailing binary '01' blocks.

    This maps n to the base of its 01-tower by undoing repeated x -> 4*x + 1 lifts.
    If the result is even, a final lift is applied to return an odd value.
    ”””
    while (n & 3) == 1:           # n ≡ 1 (mod 4)
        n »= 2                   # n //= 4
    return n if (n & 1) else ((n « 2) | 1)
```

D.Sec-3.6

```
def class_base(n: int) -> int:
    ”””
    Return the normalized 01-tower base associated with n.

    This is the k0_normalize(n) value used as a canonical representative.
    ”””
    return k0_normalize(n)
```

```
def class_elements(n: int, count: int):
    ”””
    Generate `count` elements from the same 01-tower as n.

    Steps:
      1) Compute core = k0_normalize(n).
      2) Form values by appending k copies of the bit-pattern ”01” to core,
         for k = 0..count-1.

    Returns a list [core, core01, core0101, ...] as integers.

    Example (illustrative):
        generate_class(117, 7)
        -> [7, 29, 117, 469, 1877, 7509, 30037]
    ”””
    if n % 2 == 0:
        raise ValueError(”Input must be odd.”)

    # 1. Get the core (already stripped of 01-blocks by your k0_normalize)
    core = class_base(n)

    # 2. Binary representation of the core (as string, no '0b' prefix)
    core_bits = bin(core)[2:]

    # 3. Build class elements by appending 0, 1, 2, ... copies of '01'
    elements = []
    for k in range(count):
        bits = core_bits + ”01” * k
        elements.append(int(bits, 2))

    return elements
```

D.Sec-4.6

```
def factorize(n: int) -> str:
    ”””
    Naive integer factorization for display purposes.

    Returns a string of the form:
        '2^3 * 3^2 * 5'

    Intended for small values and explanatory output, not performance.
    ”””
    if n <= 1:
        return str(n)
    factors = []
    # factor 2
    cnt = 0
    while n % 2 == 0:
        n //= 2
        cnt += 1
    if cnt > 0:
        factors.append(f”2^{cnt}” if cnt > 1 else ”2”)
    # odd factors
    p = 3
    while p * p <= n:
        cnt = 0
        while n % p == 0:
            n //= p
            cnt += 1
        if cnt > 0:
            factors.append(f”{p}^{cnt}” if cnt > 1 else f”{p}”)
        p += 2
    # leftover prime
    if n > 1:
        factors.append(str(n))
    return ” * ”.join(factors)


def predecessor_class(n: int) -> str:
    ”””
    Classify an odd integer n by predecessor type based on n mod 6.

    Returns:
      'A' if n % 6 == 1
      'B' if n % 6 == 3
      'C' if n % 6 == 5
      '-' for the sentinel value n == -1

    Assumes n is an odd integer >= 3, or the sentinel -1.
    ”””
    if n == -1:
        return ”-”
    r = n % 6
    if r == 1:
        return ”A”
    elif r == 3:
        return ”B”
    elif r == 5:
        return ”C”
    else:
        return ”?”  # shouldn't happen for odd n >= 3


def predecessor_step(n: int) -> int:
    ”””
    Perform one step of the smallest-predecessor map.

    Rules:
      - If n % 6 == 1: return (4*n - 1) // 3
      - If n % 6 == 3: return -1   (no predecessor)
      - If n % 6 == 5: return (2*n - 1) // 3

    The sentinel value -1 is absorbing and returns -1.
    ”””
    if n == -1:
        return -1

    if n % 2 == 0 or n < 3:
        raise ValueError(”n must be an odd integer >= 3 (or -1).”)

    r = n % 6
    if r == 1:
        return (4*n - 1) // 3
    elif r == 3:
        return -1
    elif r == 5:
        return (2*n - 1) // 3
    else:
        raise ValueError(f”Unexpected residue n % 6 = {r} for n={n}.”)


def predecessor_chain_rows(n0: int, max_steps: int = 100):
    ”””
    Generate table-friendly data for the smallest-predecessor chain starting at n0.

    Returns a list of dict rows with keys:
      - t      : step index (int)
      - n      : current value n_t (int, or -1 sentinel)
      - type   : predecessor type 'A', 'B', 'C', or '-' for -1
      - m      : (n-1)//6 for type A, (n-5)//6 for type C, otherwise '–'
      - factor : factorization of m as a string, otherwise '–'
      - timer  : nu3(m) for type A, otherwise '–'
      - calc   : human-readable string describing the update used
      - n_next : next value n_{t+1} (int or '–' on the final -1 row)

    Notes:
      - Type B rows terminate the chain by setting n_next = -1.
      - A final sentinel row for n = -1 is included and then iteration stops.
    ”””
    if n0 % 2 == 0 or n0 < 3:
        raise ValueError(”n0 must be an odd integer >= 3.”)

    rows = []
    n = n0
    t = 0

    while t < max_steps:
        t_type = predecessor_class(n)

        if n == -1:
            # final state
            row = {
                ”t”: t,
                ”n”: n,
                ”type”: ”-”,
                ”m”: ”–”,
                ”factor”: ”–”,
                ”timer”: ”–”,
                ”calc”: ”–”,
                ”n_next”: ”–”,
            }
            rows.append(row)
            break

        if t_type == ”A”:
            # n = 6m + 1
            m = (n - 1) // 6
            timer = nu3(m)
            calc = f”(4*{n} - 1)//3”
            n_next = predecessor_step(n)
            factor = factorize(m)
        elif t_type == ”C”:
            # n = 6m + 5
            m = (n - 5) // 6
            timer = ”–”
            calc = f”(2*{n} - 1)//3”
            n_next = predecessor_step(n)
            factor = factorize(m)
        elif t_type == ”B”:
            # n = 6m + 3, but we don't use m in the theory tables
            m = ”–”
            factor = ”–”
            timer = ”–”
            calc = f”(0*{n} - 1)//3 = -1/3”
            n_next = -1
        else:
            # Should not occur for odd n >= 3
            m = ”?”
            factor = ”?”
            timer = ”?”
            calc = ”?”
            n_next = ”?”

        row = {
            ”t”: t,
            ”n”: n,
            ”type”: t_type,
            ”m”: m,
            ”factor”: factor,
            ”timer”: timer,
            ”calc”: calc,
            ”n_next”: n_next,
        }
        rows.append(row)

        if n_next == -1:
            # Add final -1 row and stop on next iteration
            t += 1
            n = -1
            continue

        n = n_next
        t += 1

    return rows


def print_predecessor_chain_table(n0: int, max_steps: int = 100):
    ”””
    Print the smallest-predecessor chain as a Markdown table.

    The output format matches the tables used in the paper and is intended
    for inspection or direct inclusion in Markdown documents.
    ”””
    rows = predecessor_chain_rows(n0, max_steps=max_steps)

    # Header
    print(”| t | n_t | Type | m | Factorization of m | Timer ν₃(m) | Calculation |
     ↪  n_{t+1} |”)
    print(⌋
     ↪ ”|---|-----|------|---|-------------------|------------|------------|---------|”⌋
     ↪  )
    for row in rows:
        t = row[”t”]
        n = row[”n”]
        t_type = row[”type”]
        m = row[”m”]
        factor = row[”factor”]
        timer = row[”timer”]
        calc = row[”calc”]
        n_next = row[”n_next”]

        print(
            f”| {t} | {n} | {t_type} | {m} | {factor} | {timer} | ”
            f”`{calc}` | {n_next} |”
        )
```

D.Sec-4.10

```
for k in range(0, 21):
    n = 6 * k + 3
    kn = k0_normalize(n)

    print(
        f”k={k:<2}  ”
        f”n={n:<4}  bin(n)={n:<10b}  ”
        f”k0_normalize(n)={kn:<4}  bin(k0)={kn:<10b}”
    )
```

D.Sec-4.12

```
def extended_signature(n0: int):
    ”””
    Compute the extended signature (ell, w, m0) for an odd integer n0.

    Returns:
      - ell: 0 or 1. If n0 is a non-terminal B inside a 01-tower, set ell = 1
             and start from the lifted value 4*n0 + 1. Otherwise ell = 0 and
             start from n0.
      - w:   a word over {'A','C'} (possibly empty) obtained by repeatedly
             taking smallest-predecessor steps until the first 'B' node is reached.
      - m0:  the first 'B' node encountered (the terminal seed for this encoding).
             For n0 == 1, returns (0, ””, None).
    ”””
    if n0 % 2 == 0:
        raise ValueError(”Input must be an odd number”)

    # Special case: 1 does not have a meaningful predecessor signature
    if n0 == 1:
        return 0, ””, None

    # 1. Determine the lift bit and lifted value L(n)
    if n0 % 6 == 3 and k0_normalize(n0) != n0:
        # non-terminal B inside a 01-tower
        ell = 1
        current = 4 * n0 + 1     # L(n)
    else:
        # A, C, or canonical B
        ell = 0
        current = n0             # L(n) = n

    # 2. Trace A/C predecessors of the lifted value
    letters = []

    while True:
        t = predecessor_class(current)

        # If we're at a B-node, we stop: current is the canonical seed m0
        if t == ”B”:
            m0 = current
            return ell, ””.join(letters), m0

        # Otherwise record the step type and move to its predecessor
        letters.append(t)
        nxt = predecessor(current)

        # Safety: if for some reason we hit -1 directly, treat current as m0
        if nxt == -1:
            m0 = current
            return ell, ””.join(letters), m0

        current = nxt


def canonical_signature(n0: int):
    ”””
    Return the canonical two-component signature (w, m0).

    This is only defined when the lift bit ell == 0, i.e. for
    type A, type C, or canonical B values.

    For non-terminal B values (ell == 1), this function raises
    a ValueError; use extended_signature instead.
    ”””
    ell, w, m0 = extended_signature(n0)
    if ell == 1:
        raise ValueError(
            ”canonical_signature is not defined for non-terminal B values; ”
            ”use extended_signature instead.”
        )
    return w, m0


def F_w(m0: int, w: str) -> int:
    ”””
    Compute the lifted value F_w(m0) from a canonical seed m0 and an A/C word w.

    Starting from x = m0, apply the inverse successor maps corresponding to
    the letters of w in right-to-left order:

      - A-step: x -> (3*x + 1) // 4
      - C-step: x -> (3*x + 1) // 2

    Raises a ValueError if an intermediate step is not integral.
    ”””
    x = m0
    for letter in reversed(w):
        if letter == ”A”:
            num = 3 * x + 1
            if num % 4 != 0:
                raise ValueError(f”A*-step not integral at x={x}”)
            x = num // 4
        elif letter == ”C”:
            num = 3 * x + 1
            if num % 2 != 0:
                raise ValueError(f”C*-step not integral at x={x}”)
            x = num // 2
        else:
            raise ValueError(f”Unexpected letter in word: {letter!r}”)
    return x


def reconstruct_from_extended_signature(ell: int, w: str, m0: int) -> int:
    ”””
    Reconstruct an odd integer n from an extended signature (ell, w, m0).

    Steps:
      1) Compute the lifted value L = F_w(m0).
      2) If ell == 0, then n = L.
         If ell == 1, then n = (L - 1) // 4.

    The special case (ell=0, w=””, m0=None) reconstructs n = 1.
    ”””
    if ell not in (0, 1):
        raise ValueError(”ell must be 0 or 1”)

    # Handle the trivial 1-case if you ever pass in (0, '', None)
    if m0 is None:
        if ell == 0 and w == ””:
            return 1
        raise ValueError(”m0=None only makes sense for n = 1 (w='' and ell=0).”)

    lifted = F_w(m0, w)

    if ell == 0:
        # type A / C / canonical B
        return lifted
    else:
        # non-terminal B: L(n) = 4n + 1
        if (lifted - 1) % 4 != 0:
            raise ValueError(f”Lifted value {lifted} is not ≡ 1 (mod 4); ”
                             ”invalid extended signature?”)
        return (lifted - 1) // 4


def reconstruct_from_n(n: int) -> int:
    ”””
    Reconstruct n from its own extended signature.

    This is a sanity check that:
      reconstruct_from_extended_signature(*extended_signature(n)) == n
    ”””
    ell, w, m0 = extended_signature(n)
    return reconstruct_from_extended_signature(ell, w, m0)
```

D.Sec-5.3

```
def generate_congruence_examples(k=10) -> dict[int, list[int]]:
    ”””
    Generate the first k positive integers in each congruence class
    modulo 24: 3, 9, and 15.

    Returns a dict mapping each residue to a list of k integers.
    ”””
    classes = {3: [], 9: [], 15: []}

    for r in classes:
        classes[r] = [r + 24*i for i in range(k)]

    return classes


def classify_type(n: int) -> str:
    ”””
    Classify an odd integer n by its residue modulo 6.

    Returns:
      'A' if n % 6 == 1
      'C' if n % 6 == 5
      otherwise a descriptive string for other residues
    ”””
    r = n % 6
    if r == 1:
        return ”A”
    elif r == 5:
        return ”C”
    else:
        return f”other (mod 6 = {r})”
```

D.Sec-5.4

```
def forward_chain_pairs(seed: int, max_steps: int = 100):
    ”””
    Generate the normalized-forward pairs along the accelerated Collatz orbit.
    Each step produces a pair (m, s) where:
      - m = k0_normalize(n) is the canonical predecessor-class representative
      - s = successor(m) is its accelerated Collatz successor

    The iteration follows:
        n -> normalize -> m -> successor -> s
    with the next input n set to s.

    The process stops early if the fixed point (1, 1) is reached,
    or after max_steps iterations.
    ”””
    pairs = []
    n = seed
    for _ in range(max_steps):
        m = k0_normalize(n)   # normalize FIRST
        s = successor(m)    # then apply T

        pairs.append((m, s))

        # Stop at the fixed point 1 -> 1
        if m == 1 and s == 1:
            break

        n = s  # next n_t is the successor

    return pairs


def forward_chain_latex(seed: int, max_steps: int = 100) -> str:
    ”””
    Return a LaTeX string displaying the normalized forward Collatz chain.

    The output alternates normalization and successor steps in the form:
        n ->_{norm} m ->^{T} s ->_{norm} m' ->^{T} s' -> ...

    The initial seed is shown explicitly. Thereafter, values are not duplicated:
    each successor is immediately followed by its normalization and next T-step.

    The chain is built from forward_chain_pairs(seed, max_steps) and stops
    either at the fixed point (1 -> 1) or after max_steps iterations.
    ”””
    pairs = forward_chain_pairs(seed, max_steps=max_steps)
    if not pairs:
        return f”${seed}.$”

    pieces = []

    # First step: show the seed explicitly
    m0, s0 = pairs[0]
    pieces.append(f”{seed} \\to_{{\\mathrm{{norm}}}} {m0}”)
    pieces.append(f”\\xrightarrow{{T}} {s0}”)

    # Subsequent steps: only show norm + T from the last successor
    for (m, s) in pairs[1:]:
        pieces.append(”\\to_{\\mathrm{norm}}”)
        pieces.append(f”{m} \\xrightarrow{{T}} {s}”)

    return ”$” + ” \\, ”.join(pieces) + ”.$”
```

```
def forward_chain_latex_reversed(seed: int, max_steps: int = 100) -> str:
    ”””
    Return a LaTeX string for the reversed normalized forward Collatz chain.

    The chain is displayed left-to-right as a reversed version of the forward
    normalize–successor sequence, using left-pointing arrows:

        1 <-^{T} 1 <-_{norm} 5 <-^{T} 3 <-_{norm} 13 <-^{T} 17 <- ... <-_{norm} seed

    Internally, the function constructs the forward chain
        n ->_{norm} m ->^{T} s -> ...
    and then reverses both the order of steps and the arrow directions.

    This representation places the forward-terminal class (P_1) on the left
    and larger classes toward the right, matching the visual convention used
    in the paper’s diagrams.
    ”””
    pairs = forward_chain_pairs(seed, max_steps=max_steps)
    if not pairs:
        return f”${seed}.$”

    # Build the full list of edges in forward direction.
    # Nodes: seed, m0, s0, m1, s1, ..., m_k, s_k
    # Edges (in order):
    #   seed --norm--> m0
    #   m0   --T-->    s0
    #   s0   --norm--> m1
    #   m1   --T-->    s1
    #   ...
    edges = []
    n = seed
    for (m, s) in pairs:
        # normalization: n -> m
        edges.append((n, ”norm”, m))
        # successor: m -> s
        edges.append((m, ”T”, s))
        n = s  # next n_t is the successor

    # Helper to render a reversed arrow with label
    def rev_arrow(label: str) -> str:
        if label == ”T”:
            return ”\\xleftarrow{T}”
        elif label == ”norm”:
            return ”\\xleftarrow{\\mathrm{norm}}”
        else:
            return ”\\xleftarrow{” + label + ”}”

    # Reverse the edges: for each (src, label, dst),
    # we will display dst <-label src in sequence.
    edges_rev = list(reversed(edges))

    # Start with the leftmost node: dst of the first reversed edge
    src0, label0, dst0 = edges_rev[0]
    pieces = [str(dst0), rev_arrow(label0), str(src0)]

    # Then consume the remaining reversed edges
    for (src, label, dst) in edges_rev[1:]:
        pieces.append(rev_arrow(label))
        pieces.append(str(src))

    return ”$” + ” \\, ”.join(pieces) + ”.$”
```

D.Sec-5.5

```
def k_once_normalize(mk: int) -> int:
    ”””
    Single-step predecessor normalization.

    Removes exactly one trailing '01' binary block from the predecessor
    representation of an odd integer mk, if such a block exists.
    This corresponds to dividing (3*mk + 1) by 4 once, then applying
    the standard parity correction to ensure the result is odd.

    If 3*mk + 1 has fewer than two factors of 2, no normalization is
    possible and mk is returned unchanged.
    ”””
    s = 3 * mk + 1
    if nu2(s) < 2:
        return mk  # no 01 block to remove

    t = ((s » 2) - 1) // 3  # remove one 01 block (divide by 4)
    return t if (t % 2 == 1) else (4 * t + 1)


def is_P1(m: int) -> bool:
    ”””
    Return True if m lies in the forward-terminal class P1.

    By convention, this holds if m = 1 or if predecessor-normalization
    collapses m to the canonical base 1.
    ”””
    return m == 1 or k0_normalize(m) == 1


def signature(n0: int):
    ”””
    Compute the predecessor-chain signature of an odd integer n0.

    Returns:
        (word, terminal)

    where:
        - word is the sequence of predecessor types ('A'/'B'/'C') encountered
          while stepping backward via the smallest-predecessor map, stopping
          before the terminal condition triggers.
        - terminal is:
            * 1 if the chain reaches the forward-terminal class P1
              (as determined by is_P1), otherwise
            * the first B-node at which the predecessor walk terminates.

    Notes:
        - n0 must be odd; n0 == 1 returns (””, 1).
        - The walk stops either when is_P1(current) is true, or when the
          current node is type 'B' (or when predecessor(...) returns -1,
          which is treated as termination at the current node).
    ”””
    if n0 % 2 == 0:
        raise ValueError(”Input must be odd.”)
    if n0 == 1:
        return ””, 1

    letters = []
    current = n0

    while True:
        if is_P1(current):
            return ””.join(letters), 1

        t = predecessor_class(current)

        if t == ”B”:
            return ””.join(letters), current

        letters.append(t)
        nxt = predecessor(current)
        if nxt == -1:
            return ””.join(letters), current
        current = nxt


def m0_canonical(n: int) -> int:
    ”””
    Canonical family label m0(n):
      - compute terminal B-node via signature
      - canonicalize that B-node via k0_normalize
      - return 1 for P1
    ”””
    if n == 1:
        return 1
    _, terminal = signature(n)
    if terminal == 1:
        return 1
    return k0_normalize(terminal)


def forward_chain_expanded(seed: int, max_steps: int = 10000):
    ”””
    Generate the expanded normalized forward Collatz chain starting from `seed`.

    This follows the forward accelerated Collatz map, but after each successor
    step applies repeated one-step normalization (k_once_normalize) to collapse
    all removable trailing 01-lifts, exactly as described in §5.5.

    The resulting list records every successor value together with all
    intermediate normalized values. It is the chain used to define τ-chains
    and to track transitions between canonical terminal seeds.

    Parameters
    ----------
    seed : int
```

```
        Odd starting integer (seed >= 1).
    max_steps : int
        Safety bound on the number of successor phases.

    Returns
    -------
    list[int]
        The expanded forward chain beginning at `seed` and terminating at 1
        (if reached within the step limit).
    ”””
    if seed % 2 == 0 or seed < 1:
        raise ValueError(”Seed must be an odd integer >= 1.”)

    chain = [seed]
    base = seed

    for _ in range(max_steps):
        if base == 1:
            break

        s = successor(base)
        chain.append(s)
        if s == 1:
            break

        mk = s
        while True:
            mk2 = k_once_normalize(mk)
            if mk2 == mk:
                break
            chain.append(mk2)
            mk = mk2
            if mk == 1:
                break

        base = mk

    return chain


def tau_chain(seed: int, max_steps: int = 10000):
    ”””
    Compute the tau-chain of an integer.

    The tau-chain is the ordered list of distinct canonical terminal seeds
    encountered while following the expanded forward Collatz chain starting
    from `seed`, stopping when the forward-terminal class (m0 = 1) is reached.

    Each value v in the expanded forward chain is mapped to its canonical
    terminal seed m0 = m0_canonical(v). Repeated m0 values are suppressed,
    preserving first occurrence order.

    Parameters:
        seed : int
            Starting integer.
        max_steps : int
            Safety limit on the forward chain expansion.

    Returns:
        List[int]
            Ordered list of canonical terminal seeds, ending with 1.
    ”””
    values = forward_chain_expanded(seed, max_steps=max_steps)

    seen = set()
    tau = []

    for v in values:
        m0 = m0_canonical(v)
        if m0 not in seen:
            seen.add(m0)
            tau.append(m0)
        if m0 == 1:
            break

    return tau


def print_tau_chain(seed: int, max_steps: int = 10000):
    ”””
    Print the tau-chain of `seed` in a compact readable form.
    ”””
    print(f”tau({seed}) = {tau_chain(seed, max_steps=max_steps)}”)
```

D.Sec-5.7

```
def is_canonical_B(n: int) -> bool:
    ”””
    Return True iff n is a canonical B-node.

    A B-node is canonical exactly when:
        - n mod 6 == 3
        - n mod 24 is one of {3, 9, 15}

    All B-nodes satisfy n mod 24 in {3, 9, 15, 21}.
    The residue 21 corresponds to non-terminal B-nodes.
    ”””
    if n % 2 == 0:
        raise ValueError(”Input must be an odd integer”)
    if predecessor_class(n) != ”B”:
        return False
    return (n % 24) in (3, 9, 15)


def is_nonterminal_B(n: int) -> bool:
    ”””
    Return True iff n is a non-terminal B-node.

    Non-terminal B-nodes are exactly the B-nodes inside
    trailing '01' towers, characterized by:
        n mod 6 == 3 and n mod 24 == 21.
    ”””
    if n % 2 == 0:
        raise ValueError(”Input must be an odd integer”)
    return (predecessor_class(n) == ”B”) and ((n % 24) == 21)


def A_step(n: int) -> int:
    ”””
    Backward A-step.

    Computes A(n) = (4n - 1) / 3.
    This is integral exactly when n ≡ 1 (mod 6).
    ”””
    return (4 * n - 1) // 3


def C_step(n: int) -> int:
    ”””
    Backward C-step.

    Computes C(n) = (2n - 1) / 3.
    This is integral exactly when n ≡ 5 (mod 6).
    ”””
    return (2 * n - 1) // 3


def K_step(n: int) -> int:
    ”””
    Backward K-step (01-lift).

    Computes K(n) = 4n + 1, corresponding to appending
    a single trailing '01' block in binary.
    ”””
    return 4 * n + 1


def structural_signature(n0: int):
    ”””
    Return the 2-part structural signature (w, m0) for an odd integer n0.

    Outputs
    -------
    w  : str
        A word over {'A','C','K'} (possibly empty).
    m0 : int or None
        The terminal B-node reached by tracing A/C predecessors.
        For n0 == 1, returns (””, None) by convention.

    Conventions
    -----------
    - If n0 is a nonterminal B-node (i.e., n0 % 24 == 21), prepend 'K' once
      and start from K_step(n0) = 4*n0 + 1.
    - Otherwise start from n0.

    The loop then repeats:
      - if predecessor_class(current) == 'B', stop and return (w, current)
      - else append 'A' or 'C' and set current = predecessor(current)
    ”””
    if n0 % 2 == 0:
        raise ValueError(”Input must be an odd integer”)
    if n0 == 1:
        return ””, None

    letters = []

    # Fold the old lift-bit into the word via a leading 'K'
    if is_nonterminal_B(n0):
        letters.append(”K”)
        current = K_step(n0)   # 4n0 + 1
    else:
        current = n0

    # Trace A/C predecessors until we hit the canonical B seed
    while True:
        t = predecessor_class(current)

        if t == ”B”:
            # Optional: enforce canonical terminal seed condition
            # if m0 % 24 not in (3, 9, 15):
            #     raise ValueError(f”Reached non-canonical B-node m0={m0}.”)
            m0 = current
            return ””.join(letters), m0

        # Record step type and move to predecessor
        # Note: here t is 'A' or 'C' by definition
        letters.append(t)
        nxt = predecessor(current)

        # predecessor() returns -1 exactly when current is a B-node.
        # We should not reach that here because we stop when t == ”B”,
        # but keep as a defensive check.
        if nxt == -1:
            m0 = current
            return ””.join(letters), m0

        current = nxt


def canonical_signature(n0: int):
    ”””
    Return the canonical 2-part signature (w, m0) for odd n0 that do NOT
    require a leading K.

    Defined for:
      - Type A or Type C inputs, and
      - canonical B-nodes (n0 % 24 in {3, 9, 15}).

    Raises:
      ValueError for nonterminal B-nodes (n0 % 24 == 21), since those require
      a leading 'K' and should use structural_signature(...) instead.
    ”””
    if is_nonterminal_B(n0):
        raise ValueError(
            ”canonical_signature is not defined for non-terminal B values; ”
            ”use structural_signature instead.”
        )
    return structural_signature(n0)
```

```
def Aprime(n: int) -> int:
    ”””
    Successor A-step.

    A'(n) = (3n + 1) / 4

    Defined for odd n with n mod 8 == 1.
    ”””
    return (3 * n + 1) // 4


def Cprime(n: int) -> int:
    ”””
    Successor C-step.

    C'(n) = (3n + 1) / 2

    Defined for odd n with n mod 8 in {3, 7}.
    ”””
    return (3 * n + 1) // 2


def Kprime(n: int) -> int:
    ”””
    Successor K-step (inverse 01-lift).

    K'(n) = (n - 1) / 4

    Defined for odd n with n mod 8 == 5.
    ”””
    return (n - 1) // 4
```

D.Sec-5.8

```
def successor_symbol(n: int) -> str:
    ”””
    Successor branch symbol for odd n.

    Classification by n mod 8:
        A  <->  1
        C  <->  3, 7
        K  <->  5
    ”””
    if n % 2 == 0:
        raise ValueError(”Input must be an odd integer”)

    r = n % 8

    if r == 1:
        return ”A”
    if r in (3, 7):
        return ”C”
    if r == 5:
        return ”K”
    raise ValueError(”Unexpected residue mod 8 for an odd integer.”)


def successor(n: int) -> int:
    ”””
    Accelerated Collatz successor of an odd integer n (symbolic form).

    This function implements the standard accelerated Collatz map

        T(n) = (3n + 1) / 2^{nu2(3n + 1)}

    but computes it *symbolically* by first classifying n modulo 8 and then
    applying the corresponding closed-form branch:

        - A : Aprime(n) = (3n + 1) / 4        if n ≡ 1 (mod 8)
        - C : Cprime(n) = (3n + 1) / 2        if n ≡ 3, 7 (mod 8)
        - K : Kprime(n) = (n − 1) / 4         if n ≡ 5 (mod 8)

    The output is identical to the usual bit-based accelerated successor
    obtained by dividing 3n + 1 by all factors of 2; this formulation makes
    the branch structure explicit and is used for orbit codes and symbolic
    analysis.
    ”””
    sym = successor_symbol(n)

    if sym == ”A”:
        return Aprime(n)
    if sym == ”C”:
        return Cprime(n)
    return Kprime(n)  # sym == ”K”


def orbit_code_from_n(n: int, max_steps: int = 100000) -> str:
    ”””
    Construct the orbit code from n.

    Iterate s from n to 1, record symbols in {A,C,K},
    then reverse the sequence.
    ”””
    if n % 2 == 0:
        raise ValueError(”n must be an odd integer”)

    if n == 1:
        return ””

    symbols = []
    current = n

    for _ in range(max_steps):
        if current == 1:
            break
        symbols.append(successor_symbol(current))
        current = successor(current)
    else:
        raise RuntimeError(”Max steps exceeded; orbit did not reach 1.”)

    symbols.reverse()
    return ””.join(symbols)
```

```
def orbit_code(n: int, max_steps: int = 100000) -> str:
    ”””
    Orbit code of n.

    Wrapper for orbit_code_from_n.
    ”””
    return orbit_code_from_n(n, max_steps=max_steps)


def decode_orbit_code(code: str):
    ”””
    Decode an orbit code starting from 1.

    Backward steps:
        A : (4m - 1) / 3
        C : (2m - 1) / 3
        K : 4m + 1
    ”””
    m = 1
    orbit = [m]

    for symbol in code:
        if symbol == ”A”:
            num = 4 * m - 1
            if num % 3 != 0:
                raise ValueError(f”Non-integral A-step at m={m}”)
            m = num // 3
            orbit.append(m)

        elif symbol == ”C”:
            num = 2 * m - 1
            if num % 3 != 0:
                raise ValueError(f”Non-integral C-step at m={m}”)
            m = num // 3
            orbit.append(m)

        elif symbol == ”K”:
            m = 4 * m + 1
            orbit.append(m)

        else:
            raise ValueError(f”Invalid symbol in orbit code: {symbol}”)

    return orbit


def reconstruct_from_signature(w: str, m0: int) -> int:
    ”””
    Reconstruct n from a signature (w, m0).

    Starting from m0, read w in reverse and apply:
        A -> Aprime
        C -> Cprime
        K -> Kprime
    ”””
    if m0 % 2 == 0:
        raise ValueError(”m0 must be an odd integer”)
    x = m0
    for ch in reversed(w):
        if ch == ”A”:
            x = Aprime(x)
        elif ch == ”C”:
            x = Cprime(x)
        elif ch == ”K”:
            x = Kprime(x)
        else:
            raise ValueError(f”Invalid symbol in signature word: {ch}”)

    return x
```

D.Sec-5.10

```
def iter_mod24(residue=None, start=1):
    ”””
    Yield odd integers n >= start whose residue class modulo 24 matches a
    selected canonical terminal-seed class.

    If residue is None, yield n such that (n % 24) is in {3, 9, 15}.
    If residue is one of {3, 9, 15}, yield n such that (n % 24) == residue.

    This is a generator (streams values) so you can build an ATLAS without
    storing the whole list first.
    ”””
    valid = {3, 9, 15}
    if residue is not None and residue not in valid:
        raise ValueError(”residue must be one of {3, 9, 15} or None”)

    # Ensure we start on an odd n
    n = start if start % 2 == 1 else start + 1

    while True:
        r = n % 24
        if residue is None:
            if r in valid:
                yield n
        else:
            if r == residue:
                yield n
        n += 2

def collect_mod24(limit, residue=None, start=1):
    ”””Return a list of the first `limit` values from iter_mod24(...).”””
    if limit is None or limit < 1:
        raise ValueError(”limit must be a positive integer”)

    out = []
    gen = iter_mod24(residue=residue, start=start)
    for _ in range(limit):
        out.append(next(gen))
    return out
```

```
def build_terminal_orbit_database(limit, max_steps=100000):
    ”””
    Build a terminal-orbit-code database for canonical seeds m0 <= limit.

    Seeds are the canonical terminal seeds with m0 mod 24 in {3, 9, 15}.
    Stores omega(m0) = orbit_code(m0) in a dict keyed by m0.
    ”””
    ORBITS = {}
    for m0 in collect_mod24(limit):
        ORBITS[m0] = orbit_code(m0, max_steps=max_steps)
    return ORBITS


def terminal_orbit_code(m0, ORBITS, max_steps=100000):
    ”””
    Return the terminal orbit code for a canonical terminal seed m0.

    If m0 is present in ORBITS, reuse it; otherwise compute orbit_code(m0)
    and cache the result.
    ”””
    if m0 % 2 == 0:
        raise ValueError(”m0 must be odd”)
    code = ORBITS.get(m0)
    if code is None:
        code = orbit_code(m0, max_steps=max_steps)  # computed via orbit_code
        ORBITS[m0] = code
    return code


def orbit_code_from_signature(n, ORBITS, max_steps=100000):
    ”””
    Compute omega(n) using only symbolic data.

    If (w, m0) is the structural signature of n, then omega(n) is obtained by
    removing the terminal suffix w from omega(m0).
    ”””
    if n % 2 == 0:
        raise ValueError(”n must be odd”)
    if n == 1:
        return ””

    w, m0 = structural_signature(n)
    omega_m0 = terminal_orbit_code(m0, ORBITS, max_steps=max_steps)

    if w:
        if not omega_m0.endswith(w):
            raise ValueError(”Inconsistent data: omega(m0) does not end with w”)
        return omega_m0[: len(omega_m0) - len(w)]

    return omega_m0


def structural_predecessor_chain(n, ORBITS, max_steps=100000):
    ”””
    Return the full structural predecessor chain for n.

    Decodes omega(n) into the explicit step-by-step chain (A/C/K steps),
    returning a list [1, ..., n].
    ”””
    code = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)
    chain = decode_orbit_code(code)

    if chain[-1] != n:
        raise ValueError(”Structural chain does not terminate at n”)

    return chain


def accelerated_predecessor_chain(n, ORBITS, max_steps=100000):
    ”””
    Return the canonical accelerated predecessor chain for n as [1, ..., n].

    Compression:
      - Suppress values produced by an A or C step if the next symbol is K.
      - Collapse each maximal run of K symbols into a single recorded value.
    ”””
    if n % 2 == 0:
        raise ValueError(”n must be odd”)

    code = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)
    m = 1
    chain = [m]

    i = 0
    L = len(code)

    while i < L:
        ch = code[i]

        if ch == ”A”:
            m = (4 * m - 1) // 3
            if i + 1 >= L or code[i + 1] != ”K”:
                chain.append(m)
            i += 1
            continue

        if ch == ”C”:
            m = (2 * m - 1) // 3
            if i + 1 >= L or code[i + 1] != ”K”:
                chain.append(m)
            i += 1
            continue

        if ch == ”K”:
            while i < L and code[i] == ”K”:
                m = 4 * m + 1
                i += 1
            chain.append(m)
            continue

        raise ValueError(f”Invalid symbol '{ch}' in omega(n)”)

    if chain[-1] != n:
        raise ValueError(”Accelerated chain does not terminate at n”)

    return chain
```

```
def structural_odd_distance_from_code(n, ORBITS, max_steps=100000):
    ”””
    Structural odd distance for n.

    Equal to the number of A/C/K predecessor steps, i.e. the length of omega(n).
    ”””
    return len(orbit_code_from_signature(n, ORBITS, max_steps=max_steps))


def accelerated_odd_distance_from_code(n, ORBITS, max_steps=100000):
    ”””
    Accelerated odd distance computed directly from the orbit code.

    Counting rules:
      - Count an A or C only if it is not immediately followed by K.
      - Count each maximal run of K symbols as one step.
    ”””
    code = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)
    L = len(code)
    if L == 0:
        return 0

    dist = 0
    i = 0

    while i < L:
        ch = code[i]

        if ch == ”A” or ch == ”C”:
            # Count only if next symbol is not K
            if i + 1 >= L or code[i + 1] != ”K”:
                dist += 1
            i += 1
            continue

        if ch == ”K”:
            # One accelerated step for the entire K-run
            dist += 1
            while i < L and code[i] == ”K”:
                i += 1
            continue

        raise ValueError(f”Invalid symbol '{ch}' in omega(n)”)

    return dist


# ==============================================================
# Route skeletons and phase directions (helper routines)
# ==============================================================

def route_from_orbit_code(code):
    ”””
    Return the route skeleton from an orbit code over {A,C,K}.

    The route records phase endpoints: endpoints of maximal K-runs and maximal
    A/C-runs. This is a boundary/junction-node route, not the accelerated chain.

    Starts at m = 1, so the first entry is 1.
    ”””
    m = 1
    route = [m]

    i = 0
    L = len(code)

    while i < L:
        # Maximal K-run
        if code[i] == ”K”:
            while i < L and code[i] == ”K”:
                m = 4 * m + 1
                i += 1
            route.append(m)
            continue

        # Maximal A/C-run
        if code[i] in (”A”, ”C”):
            while i < L and code[i] in (”A”, ”C”):
                ch = code[i]
                if ch == ”A”:
                    m = (4 * m - 1) // 3
                else:  # ch == ”C”
                    m = (2 * m - 1) // 3
                i += 1
            route.append(m)
            continue

        raise ValueError(f”Invalid symbol '{code[i]}' in orbit code”)

    return route


def route_skeleton(n, ORBITS, max_steps=100000):
    ”””
    Return the route skeleton for n computed from omega(n).

    The route alternates endpoints of maximal K-runs and maximal A/C-runs,
    starting at 1 and ending at n.

    Example:
      omega(57) = KCKACCAKAACA  ->  route = [1, 5, 3, 13, 9, 37, 57]
    ”””
    if n % 2 == 0:
        raise ValueError(”n must be odd”)
    if n == 1:
        return [1]

    code = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)
    route = route_from_orbit_code(code)

    if route[-1] != n:
        raise ValueError(”Route skeleton does not terminate at n (check omega(n)
          ↪  decoding rules)”)

    return route


def route_segments(n, ORBITS, max_steps=100000):
    ”””
    Return (route, segments) for n, computed from omega(n).

    route is the route skeleton [v0, v1, ..., vk] with v0 = 1 and vk = n.
    segments records the phase type between consecutive route nodes as triples:
        (”K”,  vi, vi+1) for a maximal K-run
        (”AC”, vi, vi+1) for a maximal A/C-run
    ”””
    if n % 2 == 0:
        raise ValueError(”n must be odd”)
    if n == 1:
        return [1], []

    code = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)

    m = 1
    route = [m]
    segs = []

    i = 0
    L = len(code)

    while i < L:
        start = m

        # Maximal K-run
        if code[i] == ”K”:
            while i < L and code[i] == ”K”:
                m = 4 * m + 1
                i += 1
            route.append(m)
            segs.append((”K”, start, m))
            continue

        # Maximal A/C-run
        if code[i] in (”A”, ”C”):
            while i < L and code[i] in (”A”, ”C”):
                ch = code[i]
                if ch == ”A”:
                    m = (4 * m - 1) // 3
                else:  # ch == ”C”
                    m = (2 * m - 1) // 3
                i += 1
            route.append(m)
            segs.append((”AC”, start, m))
            continue

        raise ValueError(f”Invalid symbol '{code[i]}' in orbit code”)

    if route[-1] != n:
        raise ValueError(”Route does not terminate at n”)

    return route, segs


def print_route(n, ORBITS, max_steps=100000, show_chains=True, show_directions=False):
    ”””
    Pretty-print orbit data for n.
    Prints omega(n) and the route skeleton. Optionally prints the structural
    and accelerated predecessor chains, and optional human-readable
    route segment directions.
    ”””
    code = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)
    route = route_skeleton(n, ORBITS, max_steps=max_steps)

    print(f”omega(n) = {code}”)
    print(f”route skeleton R(n)= {route}”)

    if show_chains:
        structural = structural_predecessor_chain(n, ORBITS, max_steps=max_steps)
        accelerated = accelerated_predecessor_chain(n, ORBITS, max_steps=max_steps)
        print(f”structural chain   = {structural}”)
        print(f”accelerated chain  = {accelerated}”)

    print(f”route skeleton R(n)= {route}”)

    if show_directions:
        _, segs = route_segments(n, ORBITS, max_steps=max_steps)
        for typ, a, b in segs:
            if typ == ”K”:
                print(f”- take K until {b} (from {a})”)
            else:
                print(f”- take A/C until {b} (from {a})”)


ORBITS = build_terminal_orbit_database(1000)  # small demo atlas

if __name__ == ”__main__”:
    n=51
    print(f”orbit_code_from_signature({n}):\t”, orbit_code_from_signature(n, ORBITS))
    print(f”structural_predecessor_chain({n}):\t”, structural_predecessor_chain(n,
      ↪ ORBITS))
    print(f”successor_chain({n}):\t”, successor_chain(n)[::-1])
    print(f”accelerated_predecessor_chain({n})\t”, accelerated_predecessor_chain(n,
      ↪ ORBITS))

    print_route(27, ORBITS, show_directions=True)
```

D.Sec-5.11

```
def terminal_orbit_code(m0, ORBITS, max_steps=100000):
    ”””
    Return the terminal orbit code for a canonical terminal seed m0.

    If m0 is present in ORBITS, reuse it; otherwise compute orbit_code(m0)
    and cache the result.
    ”””
    if m0 % 2 == 0:
        raise ValueError(”m0 must be odd”)
    code = ORBITS.get(m0)
    if code is None:
        code = orbit_code(m0, max_steps=max_steps)  # computed via orbit_code
        ORBITS[m0] = code
    return code


def indexed_integers_by_terminal_seed(m0, ORBITS, max_steps=100000):
    ”””
    Return the finite list of odd integers indexed by terminal seed m0.

    The list is generated symbolically from the terminal orbit code of m0
    by applying Aprime and Cprime along the maximal rightmost A/C suffix
    (stopping before the first K from the right).
    ”””
    if m0 % 2 == 0:
        raise ValueError(”m0 must be odd”)

    code = terminal_orbit_code(m0, ORBITS, max_steps=max_steps)

    # Collect maximal right suffix over {A,C} before the first K from the right.
    suffix = []
    i = len(code) - 1
    while i >= 0 and code[i] != ”K”:
        ch = code[i]
        if ch != ”A” and ch != ”C”:
            raise ValueError(”omega(m0) contains an unexpected symbol: ” + str(ch))
        suffix.append(ch)  # this is in right-to-left application order
        i -= 1

    # Apply primes in the same order we collected (right-to-left).
    x = m0
    out = []
    for ch in suffix:
        if ch == ”A”:
            x = Aprime(x)
        else:  # ch == ”C”
            x = Cprime(x)
        out.append(x)

    return out

def family(m0, ORBITS, max_steps=100000):
    ”””
    Return the family of m0.

    The family consists of m0 together with all integers indexed by
    the same terminal seed.
    ”””
    return [m0] + indexed_integers_by_terminal_seed(m0, ORBITS, max_steps=max_steps)
```

D.Sec-5.12

```
def odd_core_and_z(N: int):
    ”””
    Decompose N as N = 2^z * n with n odd.
    Returns (z, n).
    ”””
    if N < 1:
        raise ValueError(”N must be a positive integer.”)
    z = nu2(N)
    n = N // (2 ** z)
    return z, n


def universal_signature(N: int):
    ”””
    Return the universal signature (z, w, m0) of N.

    Here N = 2^z * n with n odd, and (w, m0) is the structural
    signature of the odd core n.
    ”””
    z, n = odd_core_and_z(N)

    # structural_signature is assumed from §5.7:
    # returns (w, m0) for odd n, with (”” , None) for n == 1.
    w, m0 = structural_signature(n)

    # Boundary sanity
    if n == 1:
        if w != ”” or m0 is not None:
            raise ValueError(”Expected structural_signature(1) = ('', None).”)
    else:
        if m0 is None:
            raise ValueError(f”Unexpected m0=None for odd core n={n}.”)

    return (z, w, m0)
```

D.Sec-5.13

```
def universal_orbit_code(N: int, ORBITS, max_steps=100000):
    ”””
    Return the universal orbit code of N.

    If N = 2^z * n with n odd, the code is formed by appending z
    to the orbit code of n.
    ”””
    z, n = odd_core_and_z(N)

    # Reuse your §5.10 routine (odd-only):
    omega_n = orbit_code_from_signature(n, ORBITS, max_steps=max_steps)

    return omega_n + ”.” + str(z)
```

D.Sec-5.14

```
def decode_universal_orbit_code(code: str):
    ”””
    Decode a universal orbit code of the form ”<body>.<z>” into a forward
    classical Collatz chain [1, ..., N].

    The body is a word over {A,C,K} and is expanded starting from p = 1 using:
      - A : apply 2,2; then apply (p-1)/3 unless the next symbol is K
      - C : apply 2;   then apply (p-1)/3 unless the next symbol is K
      - K-run : apply (2,2) for each K in the run; then apply (p-1)/3

    After expanding the body, apply z final doublings. If body is empty, the
    chain is [1, 2, 4, ..., 2^z].
    ”””
    if ”.” not in code:
        raise ValueError(”Universal orbit code must contain a '.' separator.”)

    body, z_str = code.split(”.”, 1)
    if z_str == ””:
        raise ValueError(”Missing z after '.' in universal orbit code.”)

    z = int(z_str)
    if z < 0:
        raise ValueError(”z must be nonnegative.”)

    # P1-family: ”.z”
    if body == ””:
        p = 1
        chain = [1]
        for _ in range(z):
            p *= 2
            chain.append(p)
        return chain

    # General case: prepend initial A
    word = ”A” + body

    p = 1
    chain = [1]
    i = 0
    L = len(word)

    while i < L:
        sym = word[i]
        nxt = word[i + 1] if i + 1 < L else None

        if sym == ”A”:
            # two doublings
            p *= 2
            chain.append(p)
            p *= 2
            chain.append(p)

            # only do (p-1)/3 if we are NOT entering a K-run
            if nxt != ”K”:
                if (p - 1) % 3 != 0:
                    raise ValueError(f”Non-integral (p-1)/3 after A at p={p}.”)
                p = (p - 1) // 3
                chain.append(p)

            i += 1
            continue

        if sym == ”C”:
            # one doubling
            p *= 2
            chain.append(p)

            # only do (p-1)/3 if we are NOT entering a K-run
            if nxt != ”K”:
                if (p - 1) % 3 != 0:
                    raise ValueError(f”Non-integral (p-1)/3 after C at p={p}.”)
                p = (p - 1) // 3
                chain.append(p)

            i += 1
            continue

        if sym == ”K”:
            # read maximal run K^k
            j = i
            while j < L and word[j] == ”K”:
                j += 1
            k = j - i

            # each K contributes two doublings
            for _ in range(k):
                p *= 2
                chain.append(p)
                p *= 2
                chain.append(p)

            # K^k always ends with (p-1)/3
            if (p - 1) % 3 != 0:
                raise ValueError(f”Non-integral (p-1)/3 after K-run at p={p}.”)
            p = (p - 1) // 3
            chain.append(p)

            i = j
            continue

        raise ValueError(f”Invalid symbol '{sym}' in orbit code body.”)

    # Final .z doublings
    for _ in range(z):
        p *= 2
        chain.append(p)

    return chain
```

D.Sec-5.15

```
def stopping_time_from_universal_orbit_code(code: str) -> int:
    ”””
    Compute the classical Collatz stopping time directly from a universal
    orbit code of the form ”<body>.<z>”, without reconstructing the orbit.

    The body (over {A,C,K}) is evaluated symbolically, and z contributes
    the final number of doublings. If body is empty, the stopping time is z.
    ”””
    if ”.” not in code:
        raise ValueError(”Universal orbit code must contain a '.' separator.”)

    body, z_str = code.split(”.”, 1)
    if z_str == ””:
        raise ValueError(”Missing z after '.' in universal orbit code.”)

    z = int(z_str)
    if z < 0:
        raise ValueError(”z must be nonnegative.”)

    # P1-family: ”.z”  -> orbit is 1 -> 2 -> ... -> 2^z, so stopping time is z
    if body == ””:
        return z

    word = ”A” + body
    st = 0

    i = 0
    L = len(word)
    while i < L:
        sym = word[i]
        nxt = word[i + 1] if i + 1 < L else None

        if sym == ”A”:
            st += 2
            if nxt != ”K”:
                st += 1
            i += 1
            continue

        if sym == ”C”:
            st += 1
            if nxt != ”K”:
                st += 1
            i += 1
            continue

        if sym == ”K”:
            j = i
            while j < L and word[j] == ”K”:
                j += 1
            k = j - i
            st += (2 * k + 1)
            i = j
            continue

        raise ValueError(f”Invalid symbol '{sym}' in orbit code body.”)
    st += z
    return st

# ---------------- verification helpers ----------------

def collatz_stopping_time(N: int) -> int:
    ”””
    Compute the classical Collatz stopping time of N by direct iteration.

    Returns the number of steps required to reach 1.
    ”””
    if N < 1:
        raise ValueError(”N must be a positive integer.”)
    steps = 0
    x = N
    while x != 1:
        if x % 2 == 0:
            x //= 2
        else:
            x = 3 * x + 1
        steps += 1
    return steps


def verify_stopping_time(N: int, ORBITS) -> None:
    ”””
    Verify symbolic vs direct Collatz stopping time for N.

    Prints the universal orbit code, decoded orbit, symbolic stopping time,
    and direct stopping time, and checks whether they agree.
    ”””
    uoc = universal_orbit_code(N, ORBITS)
    st_sym = stopping_time_from_universal_orbit_code(uoc)
    st_act = collatz_stopping_time(N)
    print(”N =”, N)
    print(”uoc =”, uoc)
    print(”collatz orbit =”, decode_universal_orbit_code(uoc))
    print(”st(symbolic) =”, st_sym)
    print(”st(actual)   =”, st_act)
    print(”match =”, (st_sym == st_act))
```

References

Barina, D. (2025). Improved verification limit for the convergence of the Collatz conjecture. Journal of Supercomputing, 81, Article 810. https://doi.org/10.1007/s11227-025-07337-0