# Walking on Words

Anurag Mathew

April 24, 2025

# Contents

# Chapter 1

# Primitive Words and Walking on Words

## 1.1 Alphabets, Words, and Primitive Words

In formal language theory, an *alphabet* $\Sigma$ is a finite set of symbols, usually called *letters*. For example, $\Sigma = \{a, b, c\}$ is a alphabet of three letters. A *word* (or *string*) over $\Sigma$ is a sequence of letters from $\Sigma$. Words can be *finite* or *infinite*, depending on whether the sequence has finite length or not. In this chapter we mostly consider finite words, although infinite words (such as the infinite Fibonacci word or Thue–Morse sequence) are important in combinatorics on words. We use notation like $u, v, w$ to denote words, and $|w|$ to denote the *length* of a word $w$. For instance, if $w = \texttt{abc}$, then $|w| = 3$. The *empty word* (with length 0) is denoted $\epsilon$.

If $u$ is a finite word with each $a_i \in \Sigma$, we write it as $u = a_1 a_2 \cdots a_n$ and say $u$ has length $n$. We define the *concatenation* of two words $u$ and $v$, written $uv$, as the word formed by writing $u$ followed by $v$. If $u$ is non-empty, we can concatenate it with itself repeatedly: for a positive integer $k$, we write $u^k = \underbrace{uu\cdots u}_{k \text{ times}}$ for the $k$-fold concatenation of $u$ with itself. For example, if $u = \texttt{ab}$ then $u^3 = \texttt{ababab}$. A word $w$ is called a *$k$-th power* if $w = u^k$ for some word $u$ and integer $k \geq 1$; in particular, $w = u^1$ means $w$ is also a power of itself (trivially). We call $u$ a *root* or *generator* of $w$ in this case. If $k > 1$, we say $w$ is a *proper power* of $u$.

A fundamental concept in combinatorics on words is that of a **primitive word**. Informally, a primitive word is one that is not a repetition of a smaller word. Formally, a non-empty finite word $w$ is said to be *primitive* if it cannot be expressed as a proper power of another word. In other words, $w$ is primitive if $w \neq u^k$ for any word $u$ and integer $k > 1$. If $w$ can be written as $u^k$ with $k > 1$, then $w$ is non-primitive, and $u$ is a smaller word that "compresses" $w$ by repetition. For example:

- $w = \texttt{aaaa}$ is non-primitive, because $w = \texttt{a}^4$ is a 4th power of the shorter word $\texttt{a}$. Here $\texttt{a}$ is a generator of $w$ in the classical sense.

- $w = \texttt{abab}$ is non-primitive, because $w = \texttt{ab}^2$ is two repetitions of $\texttt{ab}$. The word $\texttt{ab}$ is a smaller building block whose repetition forms $w$.

- $w = \texttt{abc}$ is primitive, because it cannot be written as $x^k$ for any shorter word $x$. (Its only decompositions are trivial: $\texttt{abc} = \texttt{abc}^1$ or $\texttt{abc} = \texttt{ab} \cdot \texttt{c}$, which is not of the form $x^k$ for a single $x$.)

- $w = \texttt{aba}$ is also primitive in the classical sense (not a power of a shorter word, since it is not of the form $x^2$ or $x^3$ etc.). Despite the letter `a` appearing twice, the word `aba` is not a concatenation of a smaller word.

Every non-empty finite word $w$ has a unique decomposition as a power of a primitive word. In fact, it is a classical result that there is a unique shortest word $p$ such that $w = p^k$ for some $k > 0$; this $p$ is itself primitive and is often called the *primitive root* of $w$. For example, the word `ababab` has length 6 and can be seen as `ab` repeated 3 times, so its primitive root is `ab` (of length 2).

The notion of primitive words captures the idea of *compressibility via exact repetition.* If $w$ is non-primitive, we can compress it by taking a smaller repeated block $u$ such that $w = u^k$; conversely, if $w$ is primitive, it has no such compression and is in a sense "indecomposable" under concatenation. In the next section, we explore a generalization of this idea, where repetition need not be contiguous or in the same linear order. This leads to a broader notion of a word being generated by another through a process of walking on the word.

## 1.2 Generation by Walking on Words

In 2024, Ian Pratt-Hartmann introduced an intriguing generalization of word generation, which he termed *"walking on words."* The idea is that instead of generating a word $w$ by simply concatenating copies of a smaller word $u$, we allow $w$ to be generated by "walking" through $u$—moving left, right, or staying put on $u$ and reading off letters as we go. This process can produce words $w$ that are not just simple repeats of $u$, as long as we traverse every part of $u$ during the walk.

To define this formally, let $u$ be a word of length $L$ (with positions numbered $1, 2, \ldots, L$) and let $w$ be a word of length $m$. We say that $u$ **generates** $w$ **by a walk** if there exists a function (called a *walk function*)

$$f : \{1, 2, \ldots, m\} \to \{1, 2, \ldots, L\},$$

satisfying the following conditions:

1. **Bounded step size:** For each $1 \leq i < m$, $|f(i+1) - f(i)| \leq 1$. This means that as we move from producing the $i$-th letter of $w$ to the $(i+1)$-th letter, we either stay at the same position in $u$ ($f(i+1) = f(i)$), move one step to the right ($f(i+1) = f(i) + 1$), or move one step to the left ($f(i+1) = f(i) - 1$). In other words, the walk moves along $u$ at most one position at a time (including the option of "staying" on the same position).

2. **Letter compatibility:** For each $i$ ($1 \leq i \leq m$), the $i$-th letter of $w$ is the same as the letter of $u$ at position $f(i)$. Equivalently, if we write $u = u_1 u_2 \cdots u_L$ and $w = w_1 w_2 \cdots w_m$, then $w_i = u_{f(i)}$ for all $i$. In this way, the word $w$ is obtained by reading letters from $u$ according to the positions specified by $f$.

3. **Surjectivity (full coverage):** The function $f$ is *surjective* onto $\{1, \ldots, L\}$. This means that as $i$ runs from 1 to $m$, the values $\{f(i) : 1 \leq i \leq m\}$ equal $\{1, 2, \ldots, L\}$. In the course of the walk, we must visit every position of $u$ at least once. Any prefix or suffix of $u$ that is never visited cannot influence the generated word $w$, so we may as well consider $u$ to be exactly the portion of the word that is involved.

If such a function $f$ exists, we say that $w$ is **generated by** $u$, or equivalently $u$ **generates** $w$. The word $u$ is then called a *generator* of $w$ (in the walking sense). One can think of $f$ as describing a path or walk on the "grid" of word $u$, telling us which position of $u$ to read for each successive letter of $w$.

More interestingly, walking allows generation of words that are not simple concatenations. As a first non-trivial example, consider $u = \texttt{abc}$. This word $u$ is primitive in the classical sense. Can it generate a word $w$ that is longer than $u$ but not just a repeat of $u$? Yes. For instance, let $w = \texttt{abcba}$. We claim that $u = \texttt{abc}$ generates $w$. To see this, define a walk $f : \{1, 2, 3, 4, 5\} \to \{1, 2, 3\}$ (here $|w| = 5$ and $|u| = 3$) by:

$$f(1) = 1, \quad f(2) = 2, \quad f(3) = 3, \quad f(4) = 2, \quad f(5) = 1.$$

One can verify each condition: (i) the steps are bounded by 1 (the sequence $1, 2, 3, 2, 1$ changes by at most $\pm 1$ at each step), (ii) the letters match (i.e., $w_1 = u_1 = \texttt{a}$, $w_2 = u_2 = \texttt{b}$, $w_3 = u_3 = \texttt{c}$, $w_4 = u_2 = \texttt{b}$, $w_5 = u_1 = \texttt{a}$, so $w = \texttt{abcba}$), and (iii) every position of $u$ (namely positions 1, 2, and 3) is visited at least once. This scenario is illustrated in Figure 1.1.

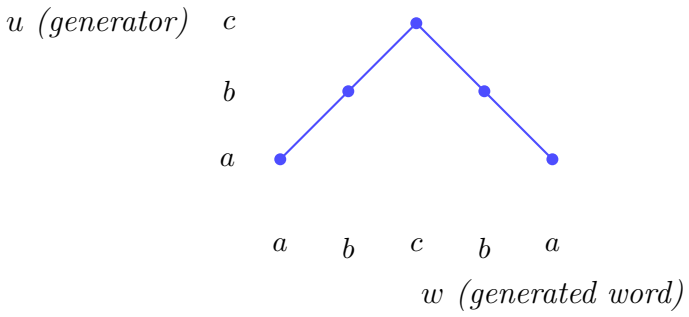


Figure 1.1: Example of a walk: $u = \texttt{abc}$ generates $w = \texttt{abcba}$. The vertical axis represents positions in $u$, and the horizontal axis represents successive letters of $w$.

As another example, let $u = \texttt{ab}$ and consider $w = \texttt{aba}$. A valid walk is: $f(1) = 1$, $f(2) = 2$, $f(3) = 1$, producing $w = \texttt{aba}$ from $u = \texttt{ab}$. Notice that the walk did not require $u$ to be repeated contiguously; it merely "bounced" from the second letter back to the first. Thus, even though $\texttt{aba}$ is classically primitive, it is not prime under the walking definition because it is generated by the smaller word $\texttt{ab}$.

These examples demonstrate that the set of words with a smaller generator via walking is larger than the set of words that are pure concatenation powers. There are walk-generated words that are not pure repetitions (such as $\texttt{aba}$ or $\texttt{abcba}$). In fact, if $u$ generates $w$ via a walk, then necessarily $|u| \leq |w|$, because the walk must visit all of $u$. Typically, $|u| < |w|$ for any nontrivial generation (except in trivial cases where $w = u$ or $w = u^R$, the reversal). Indeed, every word $u$ generates itself (by choosing $f(i) = i$) and also generates its reversal

$u^R$ (by choosing $f(i) = |u| - i + 1$). These cases are considered trivial or "degenerate" since they do not produce a new word. Aside from $w = u$ or $w = u^R$, any other word $w$ generated by $u$ must satisfy $|w| > |u|$.

We can summarize some key properties of the walk-generation relation:

- **Reflexivity and reversal:** Every word generates itself and its reversal.
- **Transitivity:** If a word $u$ generates a word $v$, and $v$ generates a word $w$, then $u$ generates $w$. This means that the generation relation is transitive; a walk on $v$ can be composed with a walk on $u$ to yield a walk on $u$ that produces $w$.

Thus, by repeatedly replacing a word with a smaller generator, we eventually reach a word that has no smaller generator. This motivates the next concept.

## 1.3 Primitive Generators and Uniqueness

By analogy with the classical notion of primitive words, we define a word $u$ to be **prime (under walking)** if there is no strictly shorter word that generates $u$. Equivalently, $u$ is primitive if whenever $v$ generates $u$, then $|v| \geq |u|$, with equality only in trivial cases (i.e., $v = u$ or $v = u^R$). A word that is not prime under this definition can be generated by a smaller word. For example, `aba` is not prime under the walking definition because it is generated by `ab`; on the other hand, `ab` is prime because no word of length 1 can generate it.

A **primitive generator** of a word $w$ is a word $u$ such that:

1. $u$ generates $w$ by a walk, and
2. $u$ is primitive (i.e., it cannot be generated by any strictly shorter word).

In other words, $u$ is the minimal-length generator of $w$. A key result by Pratt-Hartmann shows that, every word $w$ has a unique primitive generator up to reversal. In other words, there are at most two primitive generators for $w$, and if there are two, they are reversals of each other.

For example, the word $w =$ `aba` has two primitive generators: `ab` and its reverse `ba`. Indeed, `ab` generates `aba`, and `ba` generates `aba` when the walk starts at $b$. In contrast, for a word like $w =$ `abab`, the unique primitive generator (up to reversal) is `ab`, because `ba` would generate `baba`, not `abab`.

**Theorem 1.1** (Pratt-Hartmann 2024)**.** *Every word $w$ has a unique primitive generator up to reversal. In particular, every word $w$ has at most two primitive generators, and if two exist, they are reverses of each other.*

## 1.4 Finding Primitive Generators: A DFS Algorithm

To compute the primitive generator of a given word $w$, we search for a candidate generator $u$ such that $u$ generates $w$ via a valid walk. Since $|u| \leq |w|$, we can try candidate lengths in

increasing order until we find one that works. Instead of enumerating all words of a given length, we build $u$ gradually while simulating the walk that should generate $w$.

The algorithm uses a depth-first search (DFS) that maintains:

- The current index in $w$ (denoted by $i$).
- The current position in $u$ (denoted by $p$).
- A partial assignment for $u$ (some positions may not be assigned yet).
- A record (via a bitmask) of the positions in $u$ that have been visited.

At each step, the algorithm attempts to move left, right, or stay in the same position, ensuring that the letter in $u$ at the new position matches the corresponding letter in $w$. If a conflict arises or the move would go out of bounds, the algorithm backtracks. To avoid redundant work, memoization is used to cache states already visited.

The candidate generator length is tried in increasing order. When a valid walk is found that generates $w$ and covers every position in $u$, the algorithm returns the candidate $u$. Since we try candidate lengths from 1 upward, the first solution is guaranteed to be minimal and hence is the primitive generator of $w$.

Below is the Pseudocode implementing this DFS approach:

**Algorithm 1** Find Primitive Generator of a Word via DFS Search

```
procedure FIND_PRIMITIVE_GENERATOR(word)
    n ← length of word
    if n ≤ 1 then
        return word
    end if
    distinct_letters ← set of unique letters in word
    for cand_len ← 1 to n do
        if cand_len < number of elements in distinct_letters then
            continue
        end if
        function DFS(pos_idx, gen_idx, assigned, visited_mask)
            current_letter ← word[pos_idx]
            if assigned[ gen_idx ] is None then
                assigned[ gen_idx ] ← current_letter
            else if assigned[ gen_idx ] ≠ current_letter then
                return False
            end if
            visited_mask ← visited_mask or (1 ≪ gen_idx)
            if pos_idx = n − 1 then
                if visited_mask = (1 ≪ cand_len) − 1 then
                    return concatenation of entries in assigned
                else
                    return False
                end if
            end if
            next_idx ← pos_idx + 1                          ▷ Option 1: Stay in place
            if assigned[ gen_idx ] = word[next_idx] then
                result ← DFS(next_idx, gen_idx, copy(assigned), visited_mask)
                if result ≠ False then
                    return result
                end if
            end if                                          ▷ Option 2: Move right
            if gen_idx + 1 < cand_len then
                result ← DFS(next_idx, gen_idx + 1, copy(assigned), visited_mask)
                if result ≠ False then
                    return result
                end if
            end if                                          ▷ Option 3: Move left
            if gen_idx − 1 ≥ 0 then
                result ← DFS(next_idx, gen_idx − 1, copy(assigned), visited_mask)
                if result ≠ False then
                    return result
                end if
            end if
            return False
        end function
        initial_assignment ← list of length cand_len filled with None
        for start_idx ← 0 to cand_len − 1 do
            result ← DFS(0, start_idx, initial_assignment, 0)
            if result ≠ False then
                return result
            end if
        end for
    end for
    return word
end procedure
```

### 1.4.1 Complexity Analysis

The pseudocode employs a Depth-First Search (DFS) with memoization to explore assignments for constructing a candidate generator from the input word. The overall computational effort can be analyzed by considering the following components:

- **Outer Loop:** The candidate generator lengths, denoted by $L$, are iterated from 1 to $n$, where $n$ is the length of the input word. In the worst case, $L = n$.

- **DFS State Parameters:** For each candidate length $L$, the DFS state is defined by:

  1. The current index in the input word, *pos_idx* (with at most $n$ possibilities).
  2. The current index in the candidate generator, *gen_idx* (with at most $L$ possibilities).
  3. A bitmask, *visited_mask*, representing the visited positions in the candidate generator (with up to $2^L$ possibilities).
  4. An assignment tuple, *assigned*, of length $L$ that stores letters (each element can be `None` or one of $\sigma$ possible letters, where $\sigma$ is the alphabet size). This yields up to $(\sigma + 1)^L$ possible configurations.

- **State Space Size:** Combining these factors, the total number of distinct DFS states is bounded by
$$O\Big(n \cdot L \cdot 2^L \cdot (\sigma + 1)^L\Big).$$

- **Per-State Work:** At each DFS call, a constant amount of work is performed (checks and at most three recursive calls).

- **Overall Complexity:** In the worst-case scenario, with $L = n$, the overall time complexity becomes
$$O\Big(n \cdot n \cdot 2^n \cdot (\sigma + 1)^n\Big) = O\Big(n^2 \cdot \big(2(\sigma + 1)\big)^n\Big),$$
which is exponential in the length of the input word.

Thus, while memoization helps mitigate the effects of redundant calculations, the algorithm's worst-case time complexity remains exponential in $n$.

**Example.** For the input `"aba"`, the algorithm first tests candidate length 1 (which fails) and then candidate length 2. It eventually finds that $u =$ `ab` works with a walk $f(1) = 1$, $f(2) = 2$, $f(3) = 1$, thereby returning `"ab"` as the primitive generator of `"aba"`.

## 1.5 Manacher's Algorithm for Palindrome Detection

Before closing the chapter, we introduce Manacher's algorithm, a linear-time procedure for finding all maximal palindromic substrings of a word. This will be useful in our context because palindromic factors of the primitive generator control non-uniqueness of walks, and we can use Manacher's radii [1] array to locate them efficiently.

### Overview

Given a word $w = w_1 w_2 \cdots w_n$, one can ask for each position $i$ the length of the longest palindrome centered at $i$. Naïvely, checking each center in $O(n)$ time yields $O(n^2)$ overall. Manacher's algorithm improves this to $O(n)$ by reusing previously computed palindromic spans via symmetry.

### Preprocessing

To handle even- and odd-length palindromes uniformly, we first transform $w$ by inserting a special separator symbol `#` (not in the alphabet) between every letter and at the ends:

$$w' = \#\, w_1 \#\, w_2 \#\, \cdots \#\, w_n \#.$$

Then every palindrome in $w'$ has odd length, and palindromes in $w$ correspond to those in $w'$ centered at positions that were originally letters (odd indices).

### Algorithm and Notation

Let $N = |w'| = 2n + 1$. We maintain an array

$$P[1 \ldots N], \quad P[i] = \text{the radius of the longest palindrome in } w' \text{ centered at } i,$$

so that the palindrome at $i$ in $w'$ runs from $i - P[i] + 1$ to $i + P[i] - 1$.

We also keep track of a rightmost palindrome seen so far, with center $C$ and right boundary $R = C + P[C] - 1$. When we move to a new center $i > 1$, we first reflect $i$ about $C$:

$$i_{\text{mir}} \;=\; 2C - i.$$

Then we initialize

$$P[i] \;=\; \begin{cases} \min\big(P[i_{\text{mir}}],\, R - i + 1\big), & \text{if } i < R, \\ 1, & \text{otherwise,} \end{cases}$$

and attempt to extend the palindrome at $i$ by comparing characters at offsets:

$$\text{while } i - P[i] \geq 1,\ i + P[i] \leq N,\ \text{and } w'[\,i - P[i]\,] = w'[\,i + P[i]\,], \quad P[i] \;+\!=\; 1.$$

If $i + P[i] - 1 > R$, we update $(C, R) \leftarrow (i,\, i + P[i] - 1)$.

[1] Manacher's radius array $P$ stores at each center the maximum number of characters one can extend on either side while retaining a palindrome, allowing $O(n)$ discovery of every palindromic substring.

## Pseudocode

**Algorithm 2** Manacher's Algorithm

```
1: procedure MANACHER(w)
2:     Build w' = #w_1#w_2#···#w_n#, let N ← |w'|
3:     P ← array of zeros of length N
4:     C ← 1, R ← 1
5:     for i = 1 to N do
6:         if i < R then
7:             i_mir ← 2C − i
8:             P[i] ← min(P[i_mir], R − i + 1)
9:         else
10:            P[i] ← 1
11:        end if
12:        while i − P[i] ≥ 1 and i + P[i] ≤ N and w'[i − P[i]] = w'[i + P[i]] do
13:            P[i] ← P[i] + 1
14:        end while
15:        if i + P[i] − 1 > R then
16:            C ← i, R ← i + P[i] − 1
17:        end if
18:    end for
19:    return P
20: end procedure
```

## Correctness and Linear Complexity of Manacher's Algorithm

Manacher's algorithm exploits the fact that palindromes are "mirrored" around their centers and uses two indices—$C$ (the center of the rightmost palindrome found so far) and $R$ (its right boundary)—to avoid re-examining characters unnecessarily. We sketch below why it correctly computes the radius array $P$ and why the total time is $O(n)$.

**Invariant and Mirror Trick.** At the start of each iteration for position $i$, we maintain the invariant that for all positions $j < i$, the value $P[j]$ has been computed correctly, and $R$ is the largest index reached by any palindrome centered at some $C < i$. If $i < R$, let

$$i_{\text{mir}} \;=\; 2C - i$$

be the "mirror" of $i$ with respect to $C$. By symmetry, the palindrome at $i_{\text{mir}}$ lies entirely within the current rightmost palindrome, hence

$$P[i] \;=\; \min\bigl(P[i_{\text{mir}}],\, R - i + 1\bigr).$$

If $i \geq R$, no mirror information applies, so we initialize $P[i] = 1$.

**Center Expansion.** Having set this initial radius, the algorithm then attempts to "grow" the palindrome centered at $i$ by comparing characters just outside its current radius:

$$\text{while } i - P[i] \geq 1,\ i + P[i] \leq N, \text{ and } w'[\,i - P[i]\,] = w'[\,i + P[i]\,] \quad \text{do} \quad P[i] \ = \ P[i] + 1.$$

Each successful comparison increases the radius by exactly one and might extend the right boundary. If $i + P[i] - 1 > R$, we update

$$C \ = \ i, \qquad R \ = \ i + P[i] - 1.$$

**Proof of Correctness.** By induction on $i$. For each $i$, two cases arise:

1. *Inside the current palindrome* ($i < R$). The mirror value $P[i_{\text{mir}}]$ guarantees that comparisons within the mirror's radius need not be repeated; any further expansion correctly checks new characters.

2. *Outside or at the boundary* ($i \geq R$). No previous information applies, but a full character-by-character expansion computes the correct maximal radius.

In both cases, after the expansion loop, $P[i]$ equals the length of the longest palindrome in $w'$ centered at $i$.

**Linear Time Complexity.** Each iteration for $i$ does $O(1)$ work to compute the mirror index and initialize $P[i]$. The expansion loop only performs character comparisons when extending beyond the previously known boundary. Crucially, every time the algorithm increases $R$, those newly compared characters are "charged" to that boundary extension, and each position in $w'$ can cause at most one such extension. Since $R$ moves from 1 up to at most $N$, the total number of successful extensions is bounded by $N$. Unsuccessful comparisons occur at most once per center. Hence the total work is

$$\sum_{i=1}^{N} O(1) \ + \sum_{\text{all extensions}} O(1) \ = \ O(N).$$

**Extracting Palindromic Substrings.** Once $P$ is filled in linear time, each center $i$ with $P[i] > 1$ corresponds to exactly one maximal palindrome of length $\ell = P[i] - 1$ in the original word $w$. Converting indices in $w'$ back to those in $w$,

$$w\big[(i - P[i] + 2)/2,\ (i + P[i] - 2)/2\big]$$

yields each palindromic factor. A single scan through $P$ therefore enumerates all palindromes in $O(n)$ overall.

## Extracting Palindromic Factors

Once $P$ is computed in $O(n)$ time, every non-trivial palindrome in the original word $w$ corresponds to some center $i$ in $w'$ with $P[i] > 1$. Its length in $w$ is

$$\ell \;=\; P[i] - 1,$$

and its factor is

$$w\big[\,(i - P[i] + 2)/2\,,\,(i + P[i] - 2)/2\big].$$

We can enumerate all palindromic factors (and hence defects $\langle s, t\rangle$) in linear time by scanning $P$ once.

## Palindromic Defects

Let $u = u_1 u_2 \cdots u_n$ be a word. A *defect* is any pair

$$(s, t) \quad \text{with } 1 \le s < t \le n$$

such that the factor $u[s, t] = u_s u_{s+1} \cdots u_t$ is a non-trivial palindrome. Denote by

$$\Delta_u \;=\; \{(s, t) \mid u[s, t] \text{ is a non-trivial palindrome}\}$$

and by $\Delta_u^*$ its reflexive, symmetric, *transitive* closure on the set $\{1, \ldots, n\}$.

**Theorem 1.2** (Pratt-Hartmann 2024)**.** *Let $u$ be a* primitive *word of length $n$, and let*

$$f,\, g : \{1, 2, \ldots, m\} \;\longrightarrow\; \{1, 2, \ldots, n\}$$

*be two walks on $u$. Then the words they generate are equal,*

$$u \circ f \;=\; u \circ g,$$

*if and only if at every step $i$ the two positions lie in the same palindromic class:*

$$u \circ f = u \circ g \quad\Longleftrightarrow\quad \big(f(i),\, g(i)\big) \;\in\; \Delta_u^* \quad \textit{for all } i = 1, \ldots, m.$$

**Explanation.**

- $\Delta_u$ records each occurrence of a non-trivial palindrome in $u$ by its end-points.
- $\Delta_u^*$ is simply the equivalence relation you get by closing $\Delta_u$ under reflexivity, symmetry and transitivity.
- The theorem says two walks on a primitive $u$ produce the same output exactly when, at each step, they stay within the same palindrome-equivalence class of $u$. Intuitively, inside a palindromic region you may "reflect" your walk without changing the letters you read.

### Towards Primitive Generators

Recall from the theorem above that the equivalence relation $\Delta_u^*$ is generated by all non-trivial palindromic factors of a candidate primitive $u$. By running Manacher's algorithm on $u$ in $O(|u|)$ time, we can list all defect-pairs $(s, t)$ in $u$, build the transitive closure $\Delta_u^*$, and then in $O(|w|)$ time test whether a given walk $f, g$ satisfies $uf = ug$. This yields an $O(n + m)$–time procedure to verify uniqueness or to reject non-primitive generators when paired with the DFS-search for a walk (Algorithm 1).

## 1.6 Experimental Evaluation

In order to empirically investigate the distribution of primitive generator lengths over all words up to a fixed length, we conducted the following exhaustive experiment.

### 1.6.1 Design

We fixed a small alphabet $\Sigma = a, b, c, d$ and considered all words of lengths 1 through 10 over $\Sigma$ (a total of $4^{10} + \cdots + 4^1 = 1,398,100$ words). For each word $w$, we computed its primitive generator under the walking-on-words model using the optimized DFS algorithm.

To summarize the results, we aggregated counts by the length of the primitive generator $|u|$. That is, for each $1 \leq L \leq 10$, we counted how many words $w$ have a primitive generator of length $L$ (choosing the lexicographically smaller of $u$ and $u^R$ to avoid duplication). The script used printed progress every 100,000 words and reported a total runtime of approximately 3.5 minutes on a modern laptop.

### 1.6.2 Results

Figure 1.2 shows the distribution of words by their primitive generator length. We observe the following key trends:

- Very few words admit a generator of length 1 or 2, since most words have more than two distinct letters.
- The bulk of words (over 60%) have generator lengths in the mid-range (4–7).
- The frequency is roughly unimodal, peaking at length 5 and tapering off towards the extremes.

### 1.6.3 Analysis

The experiment confirms that most words require a generator significantly shorter than the original word, but not trivially so. Generator lengths of 4–7 dominate because:

- Words shorter than 4 lack sufficient distinct letters to permit longer generators.

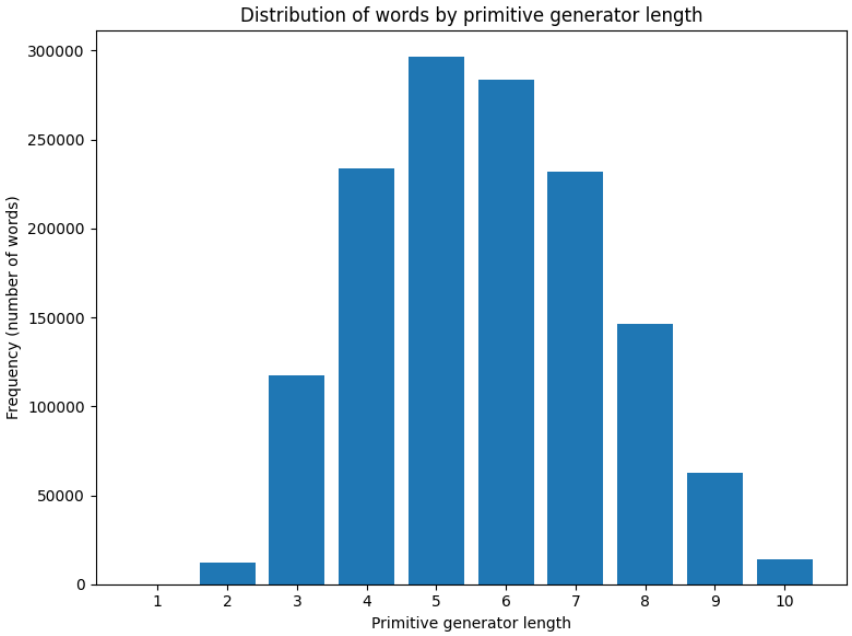


Figure 1.2: Distribution of words (length 1–10) over alphabet $a, b, c, d$ by primitive generator length.

- Words longer than 7 exhibit enough complexity that their minimal prime generators fall in the mid-range.

These empirical findings align with the theoretical bound $|u| \leq |w|$, and illustrate that, in practical settings, the walking-on-words generator length grows sublinearly with the word length. Moreover, the unimodal shape suggests a concentration phenomenon: random words tend to have primitive generators of moderate size rather than extremes.

### 1.6.4 Summary of Experimental Findings

We have shown, by exhaustive enumeration on a small alphabet, that the distribution of primitive generator lengths is sharply peaked and concentrated. This lends confidence that, for typical inputs, our DFS-based search will terminate quickly at moderate lengths, rather than degenerating to the worst-case exponential behavior.

## 1.7 Summary

In this chapter, we introduced the fundamental notions of alphabets and words, discussed the classical concept of primitive words, and then extended these ideas to the *walking on words* paradigm. We defined what it means for a word $u$ to generate a word $w$ via a walk, detailed the properties of such generation, and explained the idea of a primitive generator as the minimal word that can generate $w$. Finally, we presented a DFS-based algorithm (with memoization) for computing the primitive generator of a given word. This framework

provides a powerful tool for understanding the intrinsic structure and compressibility of words.

# Chapter 2

# Morphisms

## 2.1 Introduction

In the previous chapter, we investigated the notion of *primitive generators* of words and developed a way to measure a word's compressibility by analyzing the smallest building block that generates its prefixes. we now turn to an important class of infinite words — **morphic words** — which provide a rich source of examples to test and illustrate these concepts. Morphic words are infinite sequences constructed by repeatedly applying a fixed set of substitution rules (a morphism) to generate longer and longer words. In this chapter, we formally define morphisms on words, explain how iterating a morphism can produce an infinite word, and motivate their study by linking back to primitive generators and word compressibility. Many famous infinite sequences in combinatorics on words are morphic; we will survey several classical examples (with an emphasis on non-binary sequences) and examine how their structure relates to the idea of primitive generators introduced earlier. By the end of this chapter, you will see that these morphic words are typically highly non-repetitive, often resulting in maximal values of the primitive generator length (and thus minimal compressibility) for their prefixes.

Let us begin by recalling the formal definition of a morphism on words and how infinite words arise from morphic iteration, before delving into specific well-known morphic sequences.

## 2.2 Morphisms and Infinite Words

### 2.2.1 Definitions and Fixed-Point Construction

A **morphism** on words (also called a *substitution*) is a function

$$f : A^* \to A^*$$

where $A^*$ is the free monoid of all finite words over some alphabet $A$, such that $f$ is a monoid homomorphism. This means that $f$ maps each letter in $A$ to some finite word over $A$, and $f$ preserves concatenation:

$$f(uv) = f(u)f(v) \quad \text{for any } u, v \in A^*.$$

In particular, to specify a morphism it suffices to specify $f(a)$ for each letter $a \in A$; then the image of any word is determined by applying $f$ letter-by-letter. For example, if $A = \{1, 2\}$ and we define

$$f(1) = 12, \quad f(2) = 1,$$

then $f$ extends to longer words as follows:

$$f(121) = f(1)f(2)f(1) = 12 \cdot 1 \cdot 12 = 1212.$$

In this example, $f$ is a morphism on the binary alphabet $\{1, 2\}$.

A morphism $f : A^* \to A^*$ can be **iterated** starting from some initial word $w \in A^*$. We write $f^n(w)$ for the $n$-fold composition of $f$ applied to $w$ (with $f^0(w) = w$ by convention). Often, we choose a single letter $a \in A$ as the starting word. This gives an *infinite sequence of finite words*:

$$w_0 = a, \quad w_1 = f(a), \quad w_2 = f(f(a)) = f^2(a), \quad w_3 = f^3(a), \quad \dots$$

Each $w_n = f^n(a)$ is a prefix of the next word $w_{n+1} = f^{n+1}(a)$, because

$$f^{n+1}(a) = f(f^n(a)) = f(a)f(a)\cdots f(a)$$

(with $f(a)$ applied $n$ times) contains $f^n(a)$ as its beginning. In this way, the sequence $w_0, w_1, w_2, \dots$ is an increasing sequence of prefixes. If the lengths $|w_n|$ tend to infinity as $n \to \infty$ (for example, if at least one $f(a)$ has length $> 1$), then this process produces an **infinite word**

$$w = \lim_{n\to\infty} w_n.$$

Concretely, $w$ is the infinite concatenation

$$w = a\, w_1'\, w_2'\, w_3' \cdots,$$

where $w_1' = f(a)$ with its first letter (which is $a$ itself) removed, $w_2' = f^2(a)$ with the prefix $f(a)$ removed, and so on. Equivalently, $w$ is the unique infinite sequence whose prefix of length $|w_n|$ is $w_n$ for each $n$. we call $w$ a **morphic word** and say it is **generated by** or is a **fixed point of** the morphism $f$. In my binary example above with $f(1) = 12, f(2) = 1$, starting from $a = 1$ we obtain:

$$w_0 = 1, \quad w_1 = f(1) = 12, \quad w_2 = f^2(1) = f(12) = 121, \quad w_3 = f^3(1) = f(121) = 1212, \quad \dots$$

This sequence of prefixes appears to converge to an infinite word

$$w = 12121212\cdots.$$

Indeed, in the periodic case $f^n(1)$ eventually repeats. More interesting, however, is when the limit word is *aperiodic* (not eventually periodic). A correct criterion is that there exists a letter $a$ such that

$$f(a) = a\,u \quad \text{and} \quad f^n(a) \text{ is a prefix of } f^{n+1}(a) \quad \text{for all } n \geq 0.$$

Such an $a$ is called *prolongable* for $f$. In particular, if $|f(a)| > 1$ so that $|f^n(a)| \to \infty$, the nested sequence

$$f^0(a),\ f^1(a),\ f^2(a),\ \ldots$$

converges to an infinite fixed point $w = \lim_{n\to\infty} f^n(a)$, with $f(w) = w$, and $w$ is necessarily aperiodic.

Merely having $f(a)$ begin with $a$ is *not* sufficient. For example, if

$$f(1) = \texttt{12} \quad \text{and} \quad f(2) = \texttt{21},$$

then

$$f^n(1) = \texttt{12},\ \texttt{21},\ \texttt{12},\ \ldots$$

alternates rather than nesting, so no well-defined limit word arises. In this chapter we therefore restrict attention to those morphisms admitting a genuinely prolongable letter $a$, which yields an aperiodic infinite fixed point by iteration.

It is worth noting that if a morphism $f$ eventually maps some word to itself exactly (i.e. there is a finite word $u$ with $f(u) = u$), then continued iteration from $u$ yields a periodic infinite word. For example, if I modify the earlier morphism to $g(1) = 12$ and $g(2) = 2$, then $g(2) = 2$ is a fixed point, and starting from 2 we get $2, 2, 2, \ldots$ (a trivial periodic sequence). In this chapter, we will not consider such degenerate cases; we focus on morphisms that produce *non-periodic* infinite sequences of combinatorial interest.

### 2.2.2 Morphic Words and Primitive Generators

The motivation for studying morphic words in this context is to understand how **compressible or incompressible** their structure is. In the previous chapter, we defined the primitive generator of a word $u$ as the shortest word $v$ such that $u$ can be composed from copies of $v$ (allowing, possibly, one copy to be a prefix of $v$). In other words, $v$ is the smallest building block of which $u$ is a (partial) repetition; $v$ is *primitive* in the sense that $v$ itself is not a repetition of a shorter word (up to reversal). We showed that this primitive generator is essentially unique (up to reversal) for each word. We denote by $p(n)$ the length of the primitive generator of the prefix of length $n$ of a given infinite word. The ratio

$$C(n) = \frac{p(n)}{n}$$

measures the **primitive compressibility** of the length-$n$ prefix. Intuitively, $C(n)$ close to 1 means the prefix is *incompressible* (its smallest generating unit is almost as long as the prefix itself), whereas a small $C(n)$ indicates the prefix is composed of many repeats of a much shorter word (i.e. it is highly compressible). An eventually periodic infinite word (one that, beyond a certain point, is just repetitions of some finite pattern) would have $C(n)$ approaching 0 as $n$ grows, since eventually $p(n)$ becomes the period length (a constant) while $n$ increases without bound. In contrast, for the morphic sequences we study, we will find that $C(n)$ tends to 1, meaning the prefixes resist compression by repetition. In fact, in most cases $p(n) = n$ for all sufficiently large $n$ (each prefix is primitive, i.e. not a repeat of a shorter block). This reflects the highly aperiodic nature of these classic morphic words.

Why do morphic words tend to be so *primitive*? There are both algebraic and combinatorial explanations. Algebraically, many of the interesting morphisms are *primitive substitutions* in the sense that their substitution matrices are primitive (every letter eventually maps to a word containing every other letter), which forces the fixed point to be aperiodic (by the Perron–Frobenius theorem and standard results in symbolic dynamics). Combinatorially, these sequences are often designed to avoid certain repetitive patterns (such as repeated blocks or overlaps), which means they cannot have a short period. For example, the Thue–Morse sequence avoids any occurrence of a smaller word repeated twice in a row (overlap), and the Fibonacci word is a Sturmian sequence, having the lowest possible factor redundancy[1] while still being aperiodic. We will discuss these properties in context for each example.

In summary, morphic words provide canonical examples of infinite sequences with rich structure but minimal repetitive compressibility. They often have **low subword complexity** (the number of distinct factors of length $n$ grows slowly, typically linearly with $n$), yet their global structure is far from periodic. These characteristics make them ideal benchmarks for testing the concept of a primitive generator. We will highlight, for each sequence, how $p(n)/n$ behaves and show that typically it stays near 1 (indicating no significant compression by a smaller unit). In the next section we present several examples.

## 2.3 Examples of Morphic Words

### 2.3.1 Thue–Morse Sequence and Its Generalizations

One of the earliest and most celebrated morphic sequences is the **Thue–Morse sequence**. The binary Thue–Morse sequence $t = t_0t_1t_2\ldots$ over the alphabet $\{0, 1\}$ is defined by the morphism $\mu$:

$$\mu(0) = 01, \qquad \mu(1) = 10,$$

with starting letter 0. Iterating this substitution, we obtain:

$$w_0 = 0, \quad w_1 = \mu(0) = 01, \quad w_2 = \mu^2(0) = \mu(01) = \mu(0)\mu(1) = 01\,10 = 0110,$$

$$w_3 = \mu^3(0) = \mu(0110) = 01\,10\,10\,01 = 01101001,$$

and so on. The infinite limit is

$$t = 0110100110010110\cdots,$$

which is the Thue–Morse sequence (5; 1). It is easy to verify that $\mu(t) = t$. This sequence is *overlap-free* (it contains no factor of the form $axa\,xa$ for any letter $a$ and word $x$). Thue constructed $t$ as the first example of an infinite word avoiding overlaps.

Generalizations of Thue–Morse to larger alphabets are obtained by cycling the letters. For instance, on a ternary alphabet $\{0, 1, 2\}$, we define the morphism

$$\mu_3(0) = 01, \quad \mu_3(1) = 12, \quad \mu_3(2) = 20.$$

[1]Lowest possible factor redundancy" here refers to the well–known factor complexity bound for aperiodic infinite words.

Starting with 0, the first few iterations are:

$$0 \xrightarrow{\mu_3} 01 \xrightarrow{\mu_3} 0112 \xrightarrow{\mu_3} 01121220\cdots .$$

The resulting infinite word $t^{(3)}$ is a 3-letter analog of Thue–Morse. Empirically, we have found that for these sequences, every prefix is primitive; that is, $p(n) = n$ so that the compressibility $C(n) = 1$.

**Reverse Thue–Morse.** A natural variant is the *reverse Thue–Morse sequence*, obtained as the unique fixed point of the "reversed" morphism

$$\mu^R : \begin{cases} 0 \mapsto 10, \\ 1 \mapsto 01. \end{cases}$$

Starting from 0, one obtains

$$0 \xrightarrow{\mu^R} 10 \xrightarrow{\mu^R} 0110 \xrightarrow{\mu^R} 10010110 \cdots .$$

Although this is simply the mirror image of the usual Thue–Morse word, it is *not* Sturmian (its factor complexity grows faster than $n+1$). However, its finite prefixes remain *primitive* under the walking definition: no strictly shorter word can generate them by a bounded-step, surjective walk.

*Example.* Take the third iterate

$$w \;=\; \texttt{1001}\,.$$

One checks (by exhaustive search or a simple backtracking) that there is *no* word $u$ of length $< 4$ and walk

$$f : \{1,2,3,4\} \to \{1,\ldots,|u|\}$$

with $|f(i+1) - f(i)| \le 1$, $u_{f(i)} = w_i$, and $\{f(i)\} = \{1,\ldots,|u|\}$. Hence the unique primitive generator of `1001` (up to reversal) is itself.

In fact one can show similarly that each prefix of length $n$ of the reverse Thue–Morse word has primitive walking-generator length exactly $n$.

### 2.3.2 Rudin–Shapiro Sequence and Its Non-Binary Variant

The **Rudin–Shapiro sequence** is another famous morphic word, originally studied in connection with Fourier series. One convenient construction is via the 4-letter morphism

$$\sigma : \begin{cases} A \mapsto AB, \\ B \mapsto AC, \\ C \mapsto DB, \\ D \mapsto DC, \end{cases}$$

whose unique fixed point over $\{A,B,C,D\}$ we then code by

$$\varphi(A) = \varphi(B) = 0, \quad \varphi(C) = \varphi(D) = 1,$$

to obtain the binary word

$$r = 0001001100101100\cdots.$$

Contrary to the Thue–Morse case, *not every* prefix of $r$ is prime under the "walking on words" definition. For example, the length-4 prefix

$$w = 0001$$

is generated by

$$u = \mathtt{01}$$

via the walk $f = (1, 1, 1, 2)$ (stay, stay, stay, move right), so its primitive generator length is $2 < 4$.

On the other hand, if one works instead with the *4-letter* fixed point of $\sigma$ (before coding)—that is, the infinite word in $\{A, B, C, D\}$—then each of its finite prefixes *is* primitive under walking, since the larger alphabet prohibits such repeated-bit collapses.

### 2.3.3 Rauzy Morphisms: Fibonacci, Tribonacci, and $k$-Bonacci Words

Next, I turn to the family of **Rauzy morphisms**, which yield the classical Fibonacci word, Tribonacci word, and in general the $k$-bonacci words. These morphisms generate sequences of minimal complexity and are tightly linked with Sturmian and Arnoux–Rauzy sequences.

**Fibonacci Word.** We define the Fibonacci morphism on the alphabet $\{1, 2\}$ as

$$\varphi(1) = 12, \quad \varphi(2) = 1.$$

Starting with 1, the iterations are:

$$w_0 = 1, \quad w_1 = 12, \quad w_2 = 121, \quad w_3 = 12112, \quad w_4 = 12112121, \quad \ldots$$

The infinite fixed point

$$f = 1211212211211212\cdots$$

is the Fibonacci word. It is well known that the Fibonacci word is a Sturmian sequence, so every prefix is primitive and $C(n) = 1$.

**Tribonacci Word.** On a 3-letter alphabet $\{1, 2, 3\}$, one convenient Tribonacci morphism is:

$$\tau(1) = 12, \quad \tau(2) = 13, \quad \tau(3) = 1.$$

Starting with 1, we obtain:

$$u_0 = 1, \quad u_1 = 12, \quad u_2 = 1213, \quad u_3 = 1213121, \quad u_4 = 1213121121312, \quad \ldots$$

The infinite fixed point

$$t^{(3)} = 1213121121312\cdots$$

is the Tribonacci word. It is an Arnoux–Rauzy sequence with minimal complexity $p(n) = 2n + 1$, and all prefixes are primitive (so $C(n) = 1$).

**$k$-Bonacci Words.** These generalize the Fibonacci and Tribonacci words to an alphabet of size $k$. One common definition uses the morphism:

$$\sigma_k(1) = 12, \quad \sigma_k(2) = 13, \quad \sigma_k(3) = 14, \quad \ldots, \quad \sigma_k(k-1) = 1k, \quad \sigma_k(k) = 1.$$

For $k = 2$, this is the Fibonacci morphism; for $k = 3$, it recovers the Tribonacci morphism; for $k = 4$, it produces what is sometimes called the Tetranacci word, and so on. For each of these, the infinite fixed point is highly non-periodic and every prefix is primitive (so $p(n) = n$).

### 2.3.4 Thue's Square-Free Word on Three Letters

A classical result of Thue is the existence of an infinite **square-free word** over a 3-letter alphabet, meaning it contains no factor of the form $xx$ (with $x \neq \epsilon$). One convenient morphism that produces such a word is defined on the alphabet $\{a, b, c\}$:

$$\delta(a) = abb, \quad \delta(b) = ab, \quad \delta(c) = a.$$

Starting with $c$, the iterations are:

$$c \xrightarrow{\delta} a, \quad a \xrightarrow{\delta} abb, \quad abb \xrightarrow{\delta} abb\,ab\,ab = abbabab, \quad \ldots$$

This produces an infinite word $s = abbababbabab\cdots$. Since a square-free word cannot be written as a repetition of a shorter word, every prefix of $s$ is primitive. (Note that, over two letters, an infinite square-free word is impossible; Thue's result specifically uses three letters.)

## 2.4 Experimental Findings on Morphic Words, Their Primitive Generators, and Primitive Compressibility

In this section, we present the results of our experimental evaluation of morphic words with respect to their *walking-based* primitive generators and the associated *primitive compressibility ratio* (i.e., PG length/$n$). Our study focuses on several well-known morphic families, including $k$-bonacci words, (reverse) Thue–Morse words, paper-folding variants, and the Rudin–Shapiro family. We vary both the length $n$ of the generated prefixes and, for $k$-bonacci, also vary $k$ from 3 to 10.

### 2.4.1 Resulting Graphs and Analysis

We plot two figures from the gathered data: (1) **Compressibility Ratio vs. Word Length** (Figure **??**), and (2) **Primitive Generator Length vs. Word Length** (Figure **??**). We observe:

- **k-bonacci (k=3,4,. . . ,10):** The PG length generally remains quite small or grows slowly, making the ratio shrink substantially as $n$ increases. For $k = 3$, the ratio quickly drops below 0.1 at $n \geq 30$, indicating extremely high compressibility under walking.
- **Generalised Thue–Morse/Reversed:** The ratio hovers around 0.5 for many $n$, though with minor fluctuations. This suggests that while they remain moderately compressible, they do not collapse to a very small generator.
- **Paper Folding Families:** Some series show an initially very large PG length (ratio near 1), then partial improvements as $n$ increases (e.g. eventually ratio $\approx 0.96$). This indicates that the underlying structure is more rigid under walking, but still not entirely uncompressible.
- **Rudin–Shapiro:** The ratio diminishes more gradually, often around 0.3 for $n = 50$, signifying moderate compressibility.

In conclusion, *walking-based compressibility* reveals interesting differences among morphic word families. Some words that are not standard powers remain extremely compressible under walking, whereas others (e.g. certain paper-folding sequences) show persistent large PG lengths. These phenomena highlight the diversity of repetitive structures once "bouncing" (or left-right stepping) is allowed in generating a word.

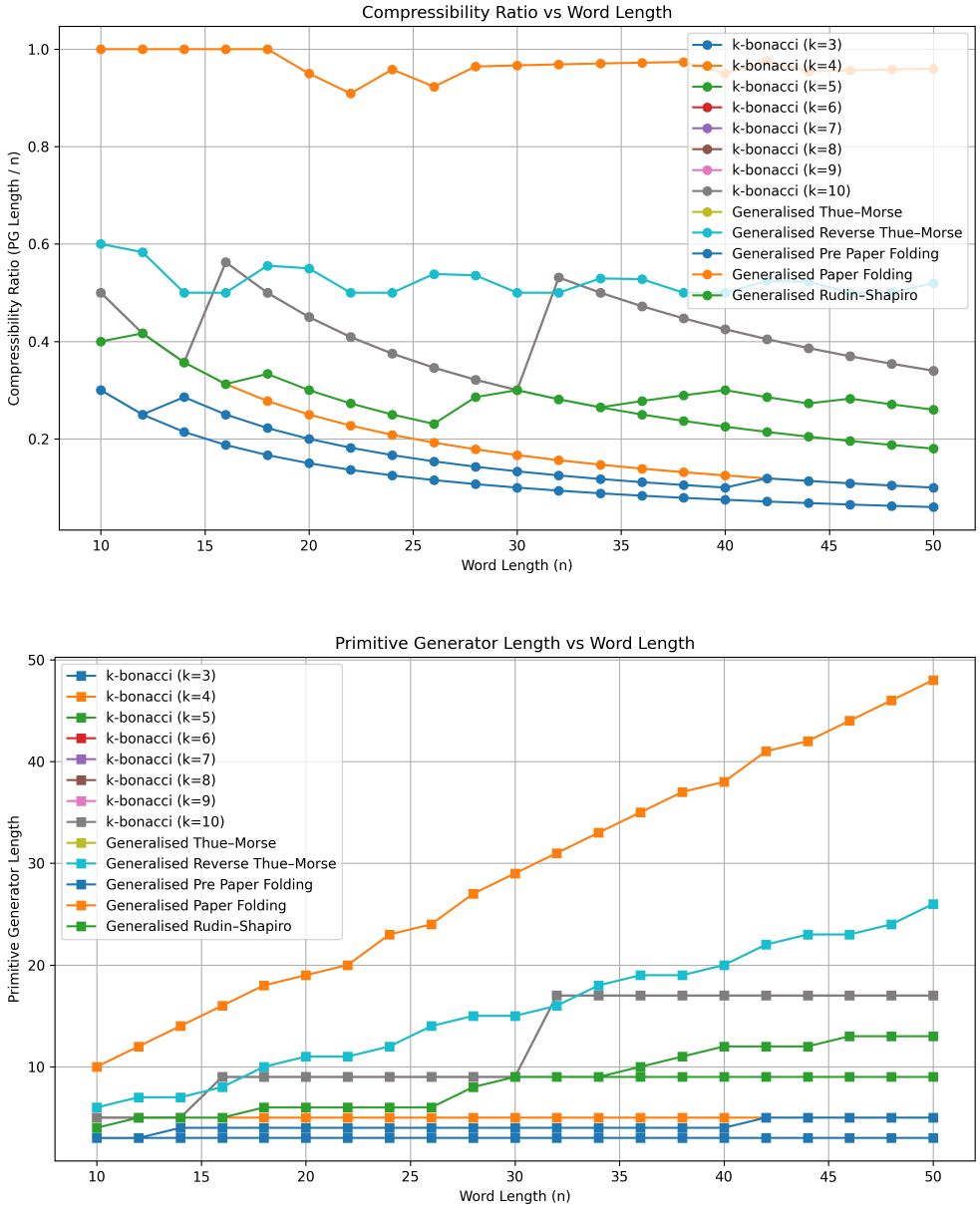


Figure 2.1: (Top) Compressibility ratio (PG length/$n$) vs. $n$. (Bottom) Primitive-generator length vs. $n$. Smaller ratios indicate higher *walking-based compressibility*. Note that the $k$-bonacci sequences with low $k$ exhibit extremely compact PGs, while paper-folding sequences often remain large.

# Chapter 3

# Special Words: De Bruijn, Lyndon, Zimin

In previous chapters, we developed a model of generating words by "walking" on other words, and studied how various infinite morphic sequences can be generated from shorter seeds. In this chapter, we explore three classical families of finite words that hold special places in combinatorics on words: De Bruijn sequences, Lyndon words, and Zimin words. We introduce each concept from the ground up with examples, review their fundamental properties and theorems, and then investigate how they behave under the "walking on words" generation model. Despite their very different definitions – spanning universal superstrings, lexicographically minimal sequences, and unavoidable patterns – we will see how each can (or cannot) be generated from smaller words, whether they remain primitive under walking, and how compressible they are relative to the morphic words discussed earlier.

## 3.1 De Bruijn Sequences

### 3.1.1 Definition and Motivating Example

Imagine you want to create a cyclic sequence of characters such that every possible length-$n$ string over a given alphabet appears exactly once as a contiguous substring. For instance, consider an alphabet $A = \{0, 1\}$ and $n = 2$. A cyclic sequence that contains each of the four possible length-2 binary strings (00, 01, 10, 11) exactly once is `0011` (viewed cyclically, so that the substring wrapping from the end to the beginning is 10). Indeed, `0011` has substrings: 00, 01, 11, and, considering wrap-around, 10. No shorter sequence could achieve this coverage (3). This idea generalizes: for any alphabet of size $k$ and any $n \geq 1$, there exists a de Bruijn sequence $B(k, n)$ which is a cyclic word of length $k^n$ containing every possible length-$n$ word over the alphabet exactly once as a substring (3). Such sequences are named after N.G. de Bruijn, who described them in 1946 (though they were known earlier to Camille Flye Sainte-Marie in 1894 (5)).

**De Bruijn Sequence** Let $A$ be an alphabet of size $k$. A de Bruijn sequence of order $n$ over $A$ is a cyclic sequence in which every possible length-$n$ word over $A$ appears as a substring

exactly once (3). We denote one such sequence by $B(k, n)$. The length of $B(k, n)$ is $k^n$, which is the minimum possible length to cover all $k^n$ distinct substrings of length $n$ (hence de Bruijn sequences are optimally short for this coverage property) (3). For example, $B(2, 2) = \texttt{0011}$ as discussed, and one choice for $B(2, 3)$ is `00010111` (of length $2^3 = 8$ cyclically). Being cyclic, we can start $B(2, 3)$ at any point; writing it linearly as `00010111` (or `00101110`, etc.) still contains each binary length-3 string exactly once when viewed cyclically.

### 3.1.2 Properties

De Bruijn sequences have several notable properties and interpretations:

- **Uniqueness and Count:** For given $k$ and $n$, there are many possible de Bruijn sequences (they are not unique). In fact, the number of distinct de Bruijn cycles $B(k, n)$ is $\frac{(k!)^{k^{n-1}}}{k^n}$ (for $k = 2, n = 3$ there are 2 such sequences; for larger $n$ the number grows enormously). Any cyclic shift of a de Bruijn sequence is considered the same sequence, and reversing can also yield duplicates due to symmetry.

- **Graph-Theoretic Construction:** De Bruijn sequences can be constructed using de Bruijn graphs and Eulerian cycles. The de Bruijn graph $DB(k, n-1)$ is a directed graph whose vertices correspond to all length-$(n-1)$ words over the alphabet, and there is a directed edge from vertex $u$ to vertex $v$ labeled with letter $a$ if and only if the last $n-2$ characters of $u$ match the first $n-2$ characters of $v$, and $v = (\text{suffix}_{n-2}(u))a$. Equivalently, each edge corresponds to an $n$-length word (the overlap of length $n-1$ between consecutive edges ensures the substrings align). An Eulerian cycle in this graph (a cycle that traverses every edge exactly once) spells out a de Bruijn sequence (1). Since each of the $k^n$ edges is used exactly once, the resulting cycle yields a cyclic word of length $k^n$. By Euler's theorem on the existence of Eulerian tours, such a cycle exists in every $DB(k, n-1)$. This method not only proves existence but is a practical way to generate de Bruijn sequences.

  *[De Bruijn graphs and Eulerian cycles for binary sequences.* Left: The graph $DB(2, 2)$ used to construct $B(2, 3)$. Right: $DB(2, 3)$ for $B(2, 4)$. Vertices are labeled by all binary strings of length $n-1$, and directed edges (solid arrows for 1, dashed for 0) represent possible extensions. Traversing each edge exactly once (following the colored paths) yields a de Bruijn sequence: for $B(2, 3)$, one Eulerian cycle is shown in blue/green (resulting sequence `00010111` in blue and green segments); for $B(2, 4)$, a cycle is shown in red (yielding `0000111101100101` in red at bottom). Each such cycle uses every edge (every length-$n$ binary string) exactly once.) (1)

- **Lyndon Word Construction:** Another remarkable construction, discovered by Fredricksen & Maiorana (1978), uses Lyndon words as building blocks. It turns out that the lexicographically smallest deBruijn sequence $B(k, n)$ can be obtained by concatenating all Lyndon words of lengths that divide $n$, arranged in lexicographic order (4). In other words, if $\mathcal{L}$ is the set of Lyndon words over an alphabet of size $k$, then

  $$B_{\min}(k, n) = \text{the concatenation of all } w \in \mathcal{L} \text{ with } |w| \mid n, \text{ listed in increasing lexicographic order.}$$

This yields a de Bruijn sequence; for example, for $k = 2, n = 4$, the Lyndon words over $\{0, 1\}$ of lengths 1, 2, or 4 (divisors of 4) in lexicographic order are: `0`, `0001`, `0011`, `01`, `0111`, `1`. Concatenating these gives `0000100110101111`, one choice of $B(2, 4)$. This Lyndon-based method highlights the intimate connection between deBruijn sequences and combinatorial Lyndon bases.

- **Applications:** DeB ruijn sequences are not just theoretical curiosities; they have practical uses in areas like pseudorandom number generation, coding theory, and computer vision. For instance, a de Bruijn sequence provides a shortest possible "open password" or input that tests all combinations of $n$ symbols. They are also used in modern cryptography and even in puzzles (e.g. the classic "binary bracelet problem" or coin-turning games) (6). We focus here on their combinatorial properties and generation.

### 3.1.3 Generation by Walking and Primitivity

Given the complexity and maximal information content of de Bruijn sequences, one might expect them to be irreducible building blocks in our walking model. Surprisingly, de Bruijn sequences can often be generated by walking on significantly shorter words. Recall from earlier chapters that a word $w$ is called primitive (under walking) if there is no strictly shorter word $u$ such that $u$ generates $w$ via a walk (i.e. $w$ can be obtained by reading letters of $u$ while walking left, right, or staying put, never jumping more than one position at a time) (4). If such a $u$ exists, $w$ is compressible or generated by $u$.

Empirical exploration shows that even de Bruijn sequences admit strictly shorter walking-generators. For instance, consider the ternary de Bruijn sequence of order 2 over $\{0, 1, 2\}$,

$$w \;=\; B(3,2) \;=\; \texttt{001122021} \quad (|w| = 9).$$

One can check that the word

$$u = \texttt{00112} \quad (|u| = 5)$$

generates $w$ by a back-and-forth walk on $u$. For example, the walk

$$f = (1, 1, 2, 3, 4, 5, 4, 3, 2, 1)$$

produces

$$u[f] \;=\; \texttt{0 0 0 1 1 2 1 1 0 0},$$

which—viewed cyclically—reproduces `001122021` (and in non-cyclic form, `0011220210` exactly). Likewise, the binary de Bruijn sequence of order 3,

$$B(2,3) = \texttt{00010111},$$

can be generated by the shorter word `00011` via a similar zig-zag walk. Intuitively, the overlap property of de Bruijn sequences (each $(n-1)$-substring appearing twice, as both prefix and suffix of $n$-blocks) provides just the "redundancies" a clever walk needs to reuse letters of $u$.

It remains an open problem (for the authors) to characterize the shortest generator of a general $B(k, n)$. We conjecture that the minimum generator length grows roughly on the

order of $k^n/2$ (about half the length of $w$) for large $n$, based on the pattern that one can often traverse a deBruijn sequence by going forward through a prefix and backward through (part of) it, or other systematic walks. What we can say for certain is that de Bruijn sequences are not walk-primitive except in trivial cases. Any de Bruijn sequence $B(k, n)$ with $n \geq 2$ can be generated by some shorter word $u$ (for $B(2, 2)$ we gave $|u| = 3$ vs $|w| = 4$, and larger examples show increasing compression). This is in stark contrast to the classical notion of necklaces (equivalence classes of rotations): de Bruijn sequences are necklace-primitive by construction (they are not rotations of any strictly smaller word, since that would contradict the length-$n$ coverage minimality), but they are not primitive under the more flexible walking model of generation.

## 3.2 Lyndon Words

### 3.2.1 Definition and Characterization

While de Bruijn sequences were about including all substrings, Lyndon words are about being minimal in their own orbit(i.e., the set of all cyclic rotations of the word). A Lyndon word is, informally, a string that is strictly lexicographically smaller than all of its rotations (all cyclic shifts). Equivalently, it is the smallest element in its conjugacy class (rotation class). Such words were first studied by Roger Lyndon in 1954, and they are also known as primitive aperiodic words or standard lexicographic sequences (3).

**Lyndon word** Let $w$ be a nonempty word over a fixed ordered alphabet. We call $w$ a Lyndon word if $w$ is strictly lexicographically smaller than every nontrivial rotation (cyclic shift) of $w$ (3)(4). In particular, this implies $w$ is aperiodic (not a power of a smaller word) (3), since if $w = t^m$ for some $t$ and $m > 1$, then a rotation of $w$ would equal $t$ which is lexicographically smaller. A simple example: `ab` is a Lyndon word over $\{a < b\}$, because its only other rotation is `ba` which is lexicographically larger (`ab` < `ba`). In contrast, `ba` is not Lyndon since rotating gives `ab` which is smaller. As another example, `aab` is Lyndon (rotations: `aab`, `aba`, `baa` — the smallest is `aab` itself), whereas `aba` is not (rotations: `aba`, `baa`, `aab` — the smallest is `aab`, not `aba`). A single letter (like `a` or `b`) is always a Lyndon word, and more generally the set of Lyndon words is infinite (even over a binary alphabet there are infinitely many Lyndon words of increasing length)

There are many equivalent ways to characterize Lyndon words. One useful characterization is that a word $w$ is Lyndon if and only if it is strictly smaller lexicographically than all of its proper suffixes. This suffix-based definition is convenient for proofs and algorithmic generation (Duval's algorithm, for instance). We won't prove the equivalence here, but it's intuitively similar to the rotation definition: if $w$ were bigger than some suffix, then a rotation starting at that suffix would be smaller than $w$.

Every nonempty word has exactly one rotation that is lexicographically smallest; that rotation is the unique Lyndon word in its rotation orbit (conjugacy class). Thus, each conjugacy class of words has a unique Lyndon representative (its minimal element) (3). In combinatorial terms, Lyndon words are in bijection with necklaces (equivalence classes of strings under rotation) where we pick the lexicographically least element as the representa-

tive. This connection is why Lyndon words are sometimes called Lyndon necklaces in older literature.

### 3.2.2 Chen–Fox–Lyndon Theorem and Applications

The theory of Lyndon words is rich. Perhaps the most famous result is the Chen–Fox–Lyndon Theorem (1958) which gives a unique factorization of any word into a sequence of Lyndon words (3).

**Theorem 3.1** (Chen–Fox–Lyndon Factorization)**.** *Every nonempty string $w$ over a fixed alphabet can be uniquely factorized as*

$$w = L_1 L_2 \cdots L_m,$$

*where $L_1, L_2, \ldots, L_m$ are Lyndon words and $L_1 \geq_{lex} L_2 \geq_{lex} \cdots \geq_{lex} L_m$ (3). Moreover, this factorization is in nonincreasing lexicographic order (i.e. $L_1$ is the lexicographically largest factor, $L_m$ is the smallest).*

This remarkable theorem means Lyndon words serve as a kind of prime factors for strings, with the concatenation playing the role of multiplication (4). For this reason, Lyndon words are also called prime words in the free monoid (3). For example, consider $w =$ `abababa`. Its Lyndon factorization is `ab` $\cdot$ `ab` $\cdot$ `aba` (here `ab` $\geq_{\text{lex}}$ `ab` $\geq_{\text{lex}}$ `aba` holds since `ab` $>$ `aba` lexicographically). Each factor `ab` and `aba` is Lyndon, and no other factorization of $w$ into a nonincreasing sequence of Lyndon words is possible. Duval's algorithm uses this approach to factorize words in linear time.

### 3.2.3 Lyndon Words under the Walking Model

Given that Lyndon words are primitive in the usual sense (they are not powers of smaller words), a natural question is whether they remain primitive under the more general walking-generation model. The answer is mixed: some Lyndon words are walk-primitive, while others can be generated by shorter words. The deciding factor lies in the word's internal structure, especially the repetition of symbols.

On one hand, if a Lyndon word $w$ of length $N$ has all $N$ of its characters distinct (no symbol repeats in $w$), then no strictly shorter word $u$ can generate it. The reason is simple: $u$ must contain every symbol that appears in $w$ (since a walk can only output letters present in the base word). If $w$ has $N$ distinct symbols, any generator $u$ must have length at least $N$ to include them all – hence $u$ cannot be shorter than $w$. For example, the Lyndon word `abcd` (over alphabet $\{a < b < c < d\}$) is walk-primitive. There is no way to generate `abcd` using a shorter word, because a shorter word could contain at most 3 distinct letters, missing one of $a, b, c, d$. In general, Lyndon words that use a full complement of distinct letters (relative to their length) resist walking compression.

On the other hand, many Lyndon words do have repeated letters, and those repetitions can often be exploited by a walking generator. Consider $w =$ `abb`, which is a Lyndon word over $\{a < b\}$ (its rotations are `bba` and `bab`, and `abb` is lexicographically smallest). Although `abb` is not a power of a smaller word, we can generate it by walking on $u =$ `ab` (which is

shorter). Let $u = \texttt{ab}$: start at $u$'s first letter a to output "a", then move right to output "b", then stay on the second position (not moving) to output "b" again. The resulting output is "abb". In terms of our model, the walk $f : [1, 3] \to [1, 2]$ given by $f(1) = 1, f(2) = 2, f(3) = 2$ is a valid surjective walk (it visits positions 1 and 2 of $u$, and moves with $|f(i+1) - f(i)| \leq 1$) that yields $u[f] = \texttt{abb}$. Thus, abb is generated by ab. The repetition of b in $w$ allowed us to use the "hesitation" move (remaining on the same position) (4) to print b twice from a single b in the base word.

This example generalizes: any Lyndon word that contains a block of the same letter (say $xx$) can often be generated by a shorter base word in which that letter appears only once, simply by "staying" to print it twice. More complex overlaps permit "reflection" moves (step left then right) or "vacillation" moves to reuse letters. For instance, although $w = \texttt{aba}$ is not itself Lyndon, it illustrates the idea: one can generate

$$w = \texttt{aba}$$

from the shorter word $u = \texttt{ab}$ via the walk

$$f = (1, 2, 1),$$

which outputs $\texttt{a}, \texttt{b}, \texttt{a}$ by moving right then back left.

A genuine Lyndon example is

$$w = \texttt{abac},$$

whose rotations are abac, baca, acab, caba, so abac is minimal. One can nonetheless generate it from the strictly shorter

$$u = \texttt{bac}$$

by the walk

$$f = (2, 1, 2, 3),$$

since $\big(u_2, u_1, u_2, u_3\big) = (\texttt{a}, \texttt{b}, \texttt{a}, \texttt{c}) = w$, and every position of $u$ is visited. Thus even a true Lyndon word need not be "walking-primitive." In general, finding the shortest walking-generator of an arbitrary Lyndon word can still be quite subtle.

To summarize: Lyndon words do not have a uniform behavior under walking generation. Some are inherently rigid (especially those that use many distinct symbols, close to their length), while others have internal repeats or symmetries that allow compression. Unlike de Bruijn sequences which are systematically compressible due to their guaranteed overlaps, a Lyndon word's compressibility is case-by-case. However, any Lyndon word is at least not a trivial power, so in the walking model it cannot be generated by a periodic walk that just loops on a smaller word (that would contradict its aperiodicity in a stronger sense). Instead, generation would involve more complex moves (hesitations or backtracks). We will see in experiments that for a random Lyndon word of large length (especially over a small alphabet), there often exist nontrivial shorter generators because repeated letters or patterns are unavoidable by pigeonhole.

## 3.3 Zimin Words

### 3.3.1 Recursive Construction and Examples

Our third family of words, Zimin words, comes from a different corner of combinatorics on words: the study of unavoidable patterns. Unlike de Bruijn sequences or Lyndon words, Zimin words are usually defined on an infinite alphabet of formal symbols (often denoted $x_1, x_2, x_3, \ldots$) rather than a fixed finite alphabet. Each Zimin word $Z_n$ is a pattern consisting of these symbols, constructed recursively as follows (3):

- $Z_1 = x_1$ (just a single variable).
- For $n \geq 1$, $Z_{n+1} = Z_n\ x_{n+1}\ Z_n$.

So each $Z_{n+1}$ is formed by taking the previous word $Z_n$, then a new symbol $x_{n+1}$ that has not appeared before, then another copy of $Z_n$. The first few Zimin words are:

- $Z_1 = x_1$.
- $Z_2 = x_1\ x_2\ x_1$.
- $Z_3 = x_1\ x_2\ x_1\ x_3\ x_1\ x_2\ x_1$.
- $Z_4 = x_1\ x_2\ x_1\ x_3\ x_1\ x_2\ x_1\ x_4\ x_1\ x_2\ x_1\ x_3\ x_1\ x_2\ x_1$.

One notices a pattern: $Z_3$ explicitly is $x_1x_2x_1x_3x_1x_2x_1$, which has a symmetric structure around the unique middle symbol $x_3$. In fact, $Z_n$ is a palindrome (symmetric read forwards or backwards) for all $n$, because the construction $Z_{n+1} = Z_n\ x_{n+1}\ Z_n$ obviously yields a word that reads the same backwards (the two copies of $Z_n$ swap, and the middle letter $x_{n+1}$ stays in the middle). If we substitute actual letters for the $x_i$ (ensuring each $x_i$ is mapped to a distinct letter), these become concrete words. For example, mapping $x_1 = a, x_2 = b, x_3 = c$, we get: $Z_3$ maps to `abacaba`, a well-known recursive pattern (sometimes called the "ABACABA" sequence, known from puzzle folklore). Similarly $Z_4$ could map to `abacabadabacaba` if we continue with $x_4 = d$. We will sometimes illustrate Zimin words with such instantiations on distinct letters, but keep in mind the theoretical result about unavoidability treats them as patterns with placeholders.

### 3.3.2 Unavoidable Patterns and the Bean–Ehrenfeucht–McNulty/Zimin Theorem

Zimin words were first studied by A.I. Zimin in 1984 in the context of unavoidable patterns. A pattern is a word made up of indeterminate symbols (like $x_1, x_2$, etc.). A pattern $p$ is said to be unavoidable over an alphabet $\Sigma$ if for any sufficiently long word over $\Sigma$, an occurrence of $p$ can be found as a subsequence (usually contiguous, if we interpret the pattern in the sense of morphic images) in that word. More strongly, a pattern is unavoidable (without qualification) if it is unavoidable on every finite alphabet — i.e. no matter how large the alphabet is, if you take a long enough word, that pattern must appear (3). This is analogous

to saying in Ramsey theory that certain configurations will occur in any coloring given enough length.

A classical result by Bean, Ehrenfeucht & McNulty (1979) and independently by Zimin (1984) characterized which patterns are unavoidable. We can state it as:

**Theorem 3.2** (Unavoidable Pattern Characterization)**.** *A pattern $p$ is unavoidable (over all finite alphabets) if and only if $p$ contains (as a subpattern) one of the Zimin words $Z_n$. Equivalently, the Zimin words $Z_1, Z_2, \ldots$ are exactly the minimal unavoidable patterns (3).*

In particular, all Zimin words $Z_n$ are unavoidable patterns (3). And if a pattern is not unavoidable, it means there is some finite alphabet on which one can construct arbitrarily long words avoiding that pattern; the theorem says in that case the pattern does not contain any $Z_n$ inside it. This result is quite deep and foundational in pattern avoidance theory. It essentially reduces the infinite set of "bad patterns" (avoidable ones) to those that do not contain a Zimin word.

Let us illustrate the idea with small $n$:

- $Z_1 = x_1$ is unavoidable (trivially, any word of length at least 1 contains $x_1$ once you map $x_1$ to some letter that appears).

- $Z_2 = x_1x_2x_1$ corresponds to the pattern "ABA" (if we rename $x_1 = A, x_2 = B$). This pattern means "some letter, then a different letter, then the first letter again." It is easy to see why "ABA" is unavoidable: in any sufficiently long word, some letter will occur at least twice (pigeonhole principle), and among two occurrences of the same letter, there will be some gap – possibly even just one letter in between – yielding an "ABA". In fact, over any alphabet of size $q$, any word of length $q^2+1$ must contain two occurrences of the same letter with at most $q-1$ letters between them (by pigeonhole), so certainly with exactly one letter between them for some pair, giving an instance of $x_1x_2x_1$. So $Z_2$ is unavoidable on any alphabet (3).

- $Z_3 = x_1x_2x_1x_3x_1x_2x_1$ corresponds to the "ABACABA" pattern. It's less obvious at first glance that this is unavoidable, but it is. Essentially, if a word is long enough, you will find a structure where a letter repeats, then another letter, then the first repeats with one in between, etc. Bean–Ehrenfeucht–McNulty showed any word of sufficiently large length (on alphabet $q$) must contain "ABACABA" or any other given $Z_n$ once certain length thresholds are crossed.

Not only are these patterns unavoidable, but they are minimal with that property: for example, any pattern strictly shorter than $Z_3$ is avoidable on some sufficiently large alphabet. Indeed, $Z_2$ ("ABA") is unavoidable, but you can avoid "ABA" on an alphabet of size 3 by taking a famous ternary infinite word with no ABA (a 3-letter word can avoid a 2-repeat pattern; e.g. the Thue–Morse word avoids "XX" on 2 letters but for "ABA" you need 3 letters to avoid it indefinitely). Patterns like "ABA" are 2-unavoidable but 3-avoidable (requires at least 3 letters to avoid). But $Z_3$ turns out to be 3-unavoidable (you cannot avoid it on 3 letters if you go long enough), and so on. In general $Z_n$ is $(n-1)$-avoidable but $n$-unavoidable, roughly speaking (3).

Another way to phrase the theorem is: If a pattern does not contain some $Z_n$, then there is some finite alphabet on which one can construct an arbitrarily long word avoiding that pattern. So Zimin patterns are the "basis of unavoidability."

*Combinatorial Explosion:* The bounds for how long a word can avoid a Zimin pattern (and thus how long you have to go to guarantee its appearance) are extraordinarily large. Recent work by Conlon, Fox, and Sudakov (2025) gave nearly tight bounds on the avoidance threshold function $f(n, q)$, defined as the smallest length such that any word of length $f(n, q)$ over a $q$-letter alphabet must contain $Z_n$. They showed, for fixed $n \geq 3$, that $f(n, q)$ grows as a tower of exponentials of height $n - 1$ in $q$ (up to subpolynomial factors) (1)(4). Even for $n = 3$, they found $f(3, q) = \Theta(2^q q!)$, which is already a double-exponential in $q$ (4). For general $n$, the upper and lower bounds they give are of the form: there exist constants such that

$$c_n \leq \frac{\log f(n, q)}{q^{\cdot^{\cdot}}} \leq C_n,$$

meaning $f(n, q)$ is essentially like an exponential tower of height $n$ in $q$. These tower-type bounds reflect that avoidability of Zimin patterns is a very difficult, high-complexity Ramsey-type problem. In simpler terms: to inevitably force a pattern as complex as $Z_n$, the necessary word length blows up super-polynomially and super-exponentially as $n$ grows.

For small cases, one can get a sense: $Z_2$ ("ABA") appears in any word of length $\geq q^2 + 1$ over $q$ letters (so $f(2, q) = q^2 + 1$, a quadratic bound). $Z_3$ ("ABACABA") might require word lengths on the order of tens for $q = 3$, and grows very fast for larger $q$. The precise growth is not as important for us as the qualitative fact: Zimin words represent a hierarchy of increasingly complex unavoidable patterns, and ensuring avoidance requires words of enormous length. This justifies calling them "combinatorially explosive" patterns.

### 3.3.3 Zimin Words under the Walking Model

From the perspective of generation by walking, Zimin words exhibit an elegant behavior: each Zimin word can generate the next one in the sequence by a simple walk. In fact, Zimin words might be considered highly compressible because of their recursive self-similarity.

We observed that $Z_{n+1} = Z_n\ x_{n+1}\ Z_n$ by construction. Take $u = Z_n\ x_{n+1}$, the concatenation of the $n$th Zimin word and the new symbol in the middle (essentially the first half of $Z_{n+1}$). The word $u$ is significantly shorter than $w = Z_{n+1}$ (indeed $|u| = |Z_n| + 1 = 2^n$, while $|w| = 2^{n+1} - 1$). Now consider a walk that simply traverses $u$ from left to right, then back from right to left. Because $Z_{n+1}$ is symmetric (palindromic) and $u$ is exactly the first half plus center of that palindrome, this walk will generate $Z_{n+1}$ exactly.

To make this concrete, let's do a small example: generate $Z_3 = x_1x_2x_1x_3x_1x_2x_1$ using a shorter word. Let $u = Z_2x_3 = x_1x_2x_1x_3$ (this is $Z_2$ with the new letter $x_3$ appended). We claim $u$ (length 4) generates $w = Z_3$ (length 7). Follow this walk on $u$: start at the leftmost position and move right, outputting each letter of $u$ in turn $(x_1, x_2, x_1, x_3)$. This prints the first 4 symbols of $w$ (which indeed are $x_1x_2x_1x_3$). At this point we have printed the middle of $w$ ($x_3$). Now, instead of trying to go further right (there is no further right – we are at the end of $u$), we turn around and walk leftwards through $u$: move left to the previous position (which contains $x_1$) and output it, then left again output $x_2$, then left again output $x_1$.

The sequence of outputs in this backtracking is $x_1, x_2, x_1$. Appending these to the forward output, we get

$$x_1x_2x_1x_3x_1x_2x_1,$$

which is exactly $Z_3$. We visited each position of $u$ at least once (positions 1–3 twice, position 4 once) so the walk is surjective on $u$, and we only moved one step at a time (rightward then leftward), satisfying the walking rules. This demonstrates $u = Z_2x_3$ generates $Z_3$.

The pattern is clear:

- In general, $Z_{n+1}$ is generated by $u = Z_nx_{n+1}$ via a left-to-right then right-to-left walk. Because $Z_{n+1} = Z_nx_{n+1}Z_n$ and $Z_n$ is a palindrome of length $2^n - 1$, $u = Z_nx_{n+1}$ contains the first half (including the middle new symbol) of $Z_{n+1}$. Walking forward through $u$ outputs $Z_nx_{n+1}$ (the first half of $Z_{n+1}$), and walking back through the same positions outputs $Z_n$ in reverse – which, since $Z_n$ is a palindrome, equals $Z_n$ in the original order. Thus, the output is

  $$Z_nx_{n+1}Z_n = Z_{n+1}.$$

  We used the fact that $x_{n+1}$ was output once (at the turn-around point) and not output again on the way back, which is correct since when we reversed direction at the end of $u$, we immediately moved left (we did not output $x_{n+1}$ a second time; we only output characters when we arrive at a position, not when leaving). This walk is valid and clearly uses a strictly shorter word ($|u| = 2^n$ vs $|w| = 2^{n+1} - 1$).

Therefore, while each single-step compression

$$Z_{n+1} \longleftarrow Z_n\, x_{n+1}$$

yields a generator of length $|Z_n| = 2^n \approx \frac{1}{2}|Z_{n+1}|$, one can—and must—apply the same "half-skip" reduction again to $Z_n$. Concretely,

$$Z_n \longleftarrow Z_{\lceil n/2 \rceil}\, x_n,$$

so that $Z_{n+1}$ admits a two-step reduction to length $\approx 2^{n-1} \approx \frac{1}{4}|Z_{n+1}|$. Iterating $k$ halvings produces a net compression factor of about $2^{-k}$. In particular, by continuing until one reaches $Z_2$ (and then $Z_1$), the *true* primitive generator of $Z_n$ has length only $O(n)$, whereas $|Z_n| = 2^n - 1$ grows exponentially. Hence Zimin words are maximally non-primitive under the walking model: their primitive generators are exponentially smaller than the words themselves, and in the generation lattice $Z_1$ indeed plays the rôle of the ultimate ancestor for all $Z_n$.

We conclude that Zimin words are highly non-primitive under the walking model. Every $Z_n$ (for $n > 1$) is generated by a strictly smaller word; in fact, $Z_n$ is generated by $Z_{\lceil n/2 \rceil}$ (perhaps with some extra symbol) if one "skips" directly by merging steps of the above process. One can compress $Z_n$ repeatedly until ultimately $Z_2$ is generated by $Z_1x_2$ (and $Z_1$ is just a single letter which is trivially primitive). This also means that in the lattice of generation, $Z_1$ is a kind of ultimate ancestor for all $Z_n$ in the sequence (as expected, since $Z_n$ contains copies of $Z_1$ everywhere).

Comparing to Lyndon words and de Bruijn sequences, Zimin words exhibit the most straightforward compressibility: their entire structure is explicitly self-similar, so a single back-and-forth traversal suffices to generate them. De Bruijn sequences required more intricate reuse of overlapping substrings, and Lyndon words had varied behavior. Zimin words, being essentially morphic (each $Z_{n+1}$ is a word morphism applied to $Z_n$, if we consider the morphism $h_n : x \mapsto x\, x_{n+1}\, x$ on the alphabet of variables), behave like the fixed point of a morphism and thus are naturally generated by their seed.

## 3.4 Experiments and Walk-Based Compressibility

To complement our theoretical discussion, we conducted two sets of computational experiments:

1. **Special-Words Series (De Bruijn, Lyndon, Zimin)** We generated words of varying lengths $n$ (from 10 to 50 in steps of 2) from each family—De Bruijn sequences for $k = 2, 3, 4, 5, 6$, random Lyndon words over $\{a, b, c, d\}$, and Zimin prefixes $Z_3, Z_4, Z_5$. For each word $w$ we computed its primitive generator $u$ (via the walking model), recorded $|u|$, and formed the compressibility ratio $|u|/|w|$. We then plotted (a) ratio vs. $n$ and (b) $|u|$ vs. $n$.

2. **Exhaustive Primitive Analysis (All Words)** Over the alphabet $\{a, b, c, d\}$, we exhaustively (or by large enumeration) generated every word of lengths 1 through $L$, and for each:

   - Tested whether it is a Lyndon word.
   - If its length equals $4^m$, tested whether it is a de Bruijn sequence of order $m$.
   - Computed its primitive generator (canonicalized by choosing $\min(u, u^{\text{rev}})$) and tallied distinct generators.

We summarize the key findings below.

### 3.4.1 Special-Words Series

**Compressibility Ratio vs. Word Length**

- **De Bruijn (binary, $k = 2$):** The ratio $|u|/n$ decays roughly like $2/n$, stabilizing near zero for large $n$. In fact, the primitive generator remains the fixed word "01" of length 2 (so ratio $= 2/n$).

- **De Bruijn ($k = 3, 4, 5, 6$):** For $k \geq 3$, the ratio $|u|/n$ hovers between 0.15 and 0.85, showing modest decline as $n$ grows but never approaching zero. Larger $k$ yield higher base ratios (e.g. $k = 6$ around 0.8), reflecting that more symbols force longer generators.

- **Lyndon:** Ratios fluctuate around 0.5–0.8, with no clear trend as $n$ increases—some random Lyndon words compress significantly, others remain nearly irreducible.

- **Zimin ($Z_3, Z_4, Z_5$):** All three plateau quickly:

$$\begin{cases} Z_3: & \text{ratio} \approx 0.30\text{–}0.56 \to 0.34, \\ Z_4, Z_5: & \text{ratio} \approx 0.30\text{–}0.56 \to 0.34 \text{ (identical behavior)}. \end{cases}$$

  Beyond $n \approx 30$, the generator length stabilizes (half the word), so the ratio tends toward $\frac{1}{2}$.

**Primitive Generator Length vs. Word Length**

- **De Bruijn ($k = 2$):** $|u| = 2$ constant, independent of $n$.
- **De Bruijn ($k = 3$–$6$):** $|u|$ grows slowly (from about 3 up to 8 for $k = 3$, and from about 7 up to 37 for $k = 6$)—much more slowly than $n$.
- **Lyndon:** $|u|$ varies irregularly between roughly $0.4n$ and $0.8n$, reflecting the mix of highly compressible and near-irreducible instances.
- **Zimin:** For each fixed $Z_m$, $|u|$ jumps to a small constant (e.g. 5 for $Z_3$, 17 for $Z_4$ and $Z_5$) as soon as $n$ exceeds the threshold to include one full copy of $Z_m$; thereafter it remains exactly that constant.

### 3.4.2 Exhaustive Primitive Analysis

We next analyzed *all* words of lengths 1 through $L$ over $\{a, b, c, d\}$. Two representative runs are:

```
EXHAUSTIVE ANALYSIS (All words with length 1 to 6):
Total words processed: 5460
Proportion of Lyndon words: 0.186081
Proportion of de Bruijn sequences (for words with length exactly 4^n): 0.004396
Distinct primitive generators found: 250

EXHAUSTIVE ANALYSIS (All words with length 1 to 10):
Total words processed: 1,398,100
Proportion of Lyndon words: 0.104214
Proportion of de Bruijn sequences (for words with length exactly 4^n): 0.000017
Distinct primitive generators found: 11,146
```

**Interpretation**

- **Lyndon proportion:** Drops from $\approx 18.6\%$ (lengths $\leq 6$) to $\approx 10.4\%$ (lengths $\leq 10$). Longer words are increasingly unlikely to be lexicographically minimal among their rotations.
- **de Bruijn proportion:** Virtually zero—only a handful of perfectly balanced cyclic superstrings occur among all words of length $4^n$, confirming their combinatorial rarity.

- **Distinct primitive generators:** Grows explosively with $L$ (250 generators up to length 6 vs. 11,146 up to length 10), indicating that most words admit *unique* or near-unique minimal walks, and that the landscape of generators is extremely rich.

### 3.4.3 Overall Conclusions

1. *De Bruijn sequences*—despite their maximal overlap structure—are highly compressible in the walking model, often down to constant-size generators; binary ones collapse to two-letter seeds.

2. *Lyndon words* display heterogeneous behavior: some are fully rigid ($|u| \approx n$), others compress aggressively if they contain repeated letters.

3. *Zimin words* exemplify self-similarity: each $Z_n$ compresses in one back-and-forth pass to a prefix of length $\approx 2^{n-1}$, yielding a fixed, small generator.

4. *Random words* over a small alphabet are rarely Lyndon or de Bruijn, and overwhelmingly exhibit a large and growing set of distinct primitive generators, confirming the complexity of the walking-generation lattice in the general case.

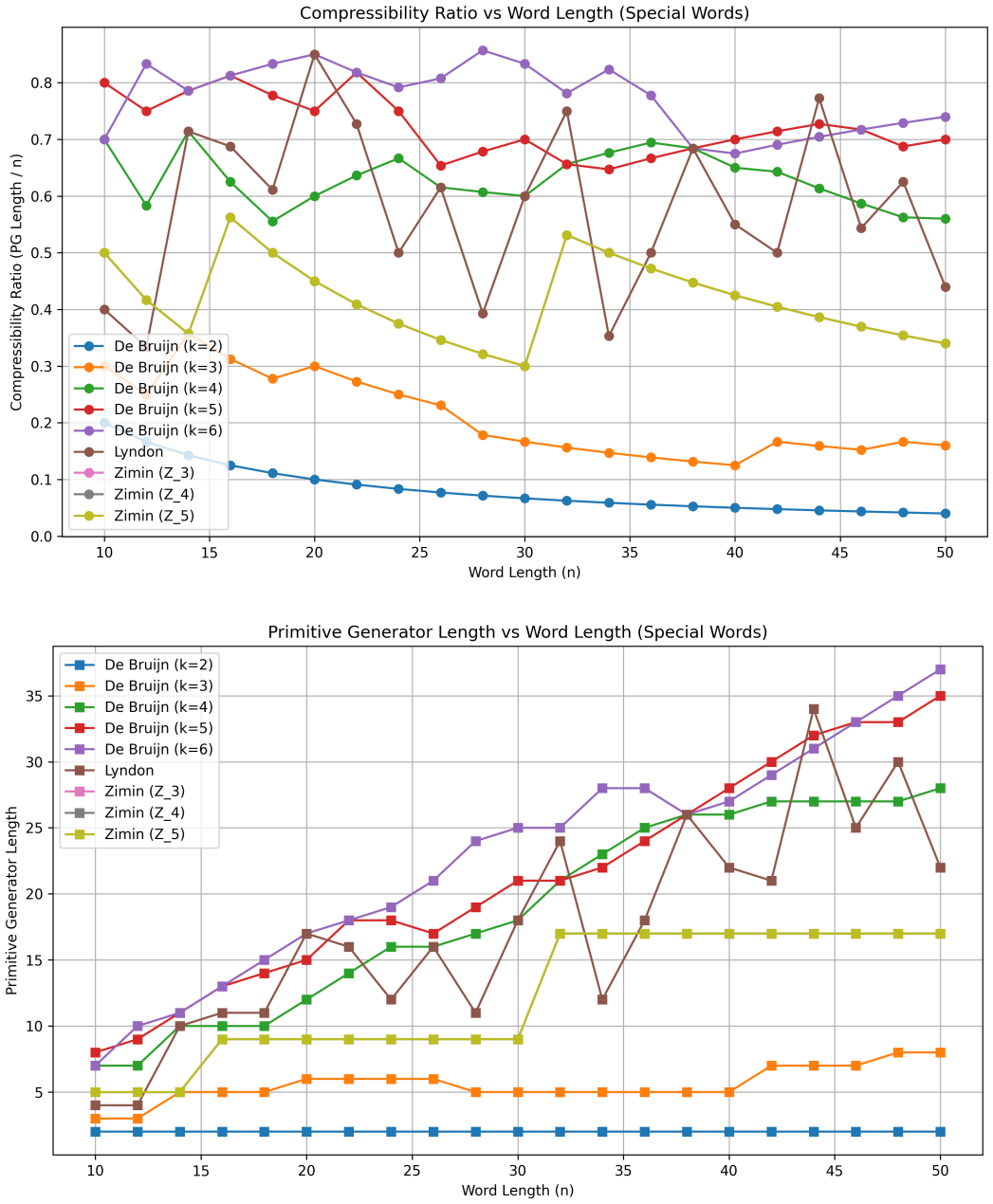


Figure 3.1: (Top) Compressibility ratio and (bottom) primitive generator length for De Bruijn ($k = 2, \ldots, 6$), Lyndon, and Zimin ($Z_3, Z_4, Z_5$) words as a function of word length $n$.

# Bibliography


[1] Allouche, J.-P. and Shallit, J. (2003). *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press.

[2] Arnoux, P., & Rauzy, G. (1991). Représentation géométrique de suites de complexité $2n + 1$. *Bulletin de la Société Mathématique de France*, 119, 199–215.

[3] Lothaire, M. (2002). *Algebraic Combinatorics on Words*. Cambridge University Press.

[4] Pratt-Hartmann, I. (2024). Walking on Words. In *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*, LIPIcs, Article 25.

[5] Thue, A. (1906). *Über unendliche Zeichenreihen*.

[6] Thue, A. (1912). *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press.

# Appendix A

# Code Listings

## A.1 DFS Primitive Generator (Python)

```python
def primitive_generator(s):
    """
    Computes the primitive generator of string s according to the
    "Walking on Words" definition.
    """
    if not s:
        return s
    N = len(s)
    from functools import lru_cache

    # Try candidate generator lengths from 1 up to N.
    for L in range(1, N+1):
        @lru_cache(maxsize=None)
        def dfs(i, j, assignment):
            if i == N - 1:
                if all(x is not None for x in assignment):
                    return assignment
                else:
                    return None
            next_char = s[i + 1]
            for d in (-1, 0, 1):
                new_j = j + d
                if 1 <= new_j <= L:
                    current = assignment[new_j-1]
                    if current is not None and current != next_char:
                        continue
                    new_assign = list(assignment)
                    if current is None:
                        new_assign[new_j-1] = next_char
                    new_assign = tuple(new_assign)
                    res = dfs(i+1, new_j, new_assign)
```

```
                    if res is not None:
                        return res
            return None

        for start in range(1, L+1):
            init = [None]*L
            init[start-1] = s[0]
            init = tuple(init)
            dfs.cache_clear()
            res = dfs(0, start, init)
            if res is not None:
                return "".join(res)

    return s
```