Generating a Gray code for prefix normal words in amortized polylogarithmic time per word*

Péter Burcsi Gabriele Fici Zsuzsanna Lipták Rajeev Raman Joe Sawada

Abstract

A prefix normal word is a binary word with the property that no substring has more 1s than the prefix of the same length. By proving that the set of prefix normal words is a bubble language, we can exhaustively list all prefix normal words of length n as a combinatorial Gray code, where successive strings differ by at most two swaps or bit flips. This Gray code can be generated in $\mathcal{O}(\log^2 n)$ amortized time per word, while the best generation algorithm hitherto has $\mathcal{O}(n)$ running time per word. We also present a membership tester for prefix normal words, as well as a novel characterization of bubble languages.

keywords: prefix normal words, binary languages, combinatorial Gray code, combinatorial generation, jumbled pattern matching

1 Introduction

A binary word of length n is *prefix normal* if for all $1 \leq k \leq n$, no substring of length k has more 1s than the prefix of length k. For example, the following are the 14 prefix normal words of length $n = 5$:

$$11111, 11110, 11101, 11100, 11011, 11010, 11001, 11000, 10101, 10100, 10010, 10001, 10000, 00000.$$

The word 10011, for instance, is not prefix normal because the substring 11 has more 1s than the prefix 10; similarly 11100110110 is not prefix normal, because the substring 11011 has more 1s than the prefix of length 5. The number *pnw*(n) of prefix normal words for $n = 1, 2, \ldots, 12$ is

$$2,\ 3,\ 5,\ 8,\ 14,\ 23,\ 41,\ 70,\ 125,\ 218,\ 395,\ \text{and } 697,$$

respectively. This enumeration sequence is included as sequence A194850 in *The On-Line Encyclopedia of Integer Sequences* (OEIS) [30], listing *pnw*(n) up to $n = 50$. It is not difficult to show that *pnw*(n) grows exponentially. Some bounds and partial enumeration results were presented in [12], and it was conjectured there that *pnw*$(n) = 2^{n-\Theta(\log^2 n)}$. This conjecture was recently proved by Balister and Gerke in [5]. Finding a closed form formula or generating function for *pnw*(n), however, remains an open problem.

Prefix normal words were originally introduced in [18] by two of the current authors, in the context of *Binary Jumbled Pattern Matching (BJPM)*: Given a binary string w of length n, and a pair of non-negative integers (x, y), decide whether w has a substring with x 1s and y 0s. While the online version of this problem can be solved naively in $O(n)$ time, the indexed version has attracted much attention during the past decade [1,

*Some of the results contained in this paper were presented in a preliminary form at CPM 2014 [11] and FUN 2014 [10].

2, 3, 4, 8, 9, 13, 14, 17, 21, 22, 23, 24]. As was shown in [12, 18], every binary word w can be assigned two canonical prefix normal word, called its *prefix normal form*, which can then be used to answer BJPM queries in constant time.

1.1 Our contributions

In this paper, we deal with the question of *generating* all prefix normal words of a given length n. In combinatorial generation, the aim is to exhaustively list all instances of a combinatorial object. Typically, the number of these instances grows exponentially, and time is measured per object, and *excluding* the time for outputting the objects. For an introduction to combinatorial generation, see [26].

The current best generation algorithm for prefix normal words runs in $\mathcal{O}(n)$ time per word [15]. Our algorithm improves on this considerably, using amortized $\mathcal{O}(\log^2 n)$ time per word.[1] It is based on the theory of *bubble languages* [27, 28, 31], an interesting class of binary languages defined by the following property: $\mathcal{L}$ is a bubble language if, for every word in $\mathcal{L}$, replacing the first occurrence of 01 (if any) by 10 results in another word in $\mathcal{L}$ [27, 28]. Many important languages are bubble languages, including binary necklaces, Lyndon words, and k-ary Dyck words[2]. A generic generation algorithm for bubble languages was given in [28], yielding *cool-lex* Gray codes for each subset of a bubble language containing all strings of a fixed length and weight (number of 1s). In general, a *(combinatorial) Gray code* is an exhaustive listing of all instances of a combinatorial object such that successive objects in the listing are "close" in some well-defined sense. In the case of *cool-lex* order, the strings differ by at most two swaps. The generic algorithm's efficiency depends only on a language-dependent subroutine called an *oracle*, which in the best case leads to CAT (constant amortized time generation) algorithms.

In the following, we show that the set of all prefix normal words forms a bubble language; it is the first new and interesting language shown to be a bubble language since the original exposition [27, 28]. We develop an oracle for prefix normal words and apply the generic generation algorithm to obtain a cool-lex ordering of prefix normal words with length n and weight d. Concatenating together these lists in increasing order of weight, we obtain a Gray code for all prefix normal words of length n where successive words differ by at most two swaps or by a swap and a bit flip. We then present an optimized oracle for prefix normal words, and, based on recent results from [5], we prove that our new generation algorithm runs in amortized time $O(\log^2 n)$ per word. Even though the previous $\mathcal{O}(n)$ time per word algorithm of [15] also provided a Gray code for prefix normal words (albeit with respect to a different measure of closeness), we are achieving a very considerable improvement in running time.

As an example, the listing of prefix normal words of length $n = 7$ that results from our algorithm, partitioned by weight, is given in Table 1.

A second contribution of this paper is a *new characterization of bubble languages*. We show that bubble languages can be described in terms of a closure property in the computation tree of a simple recursive generation algorithm for *all* binary strings. We believe that this view could aid other researchers in applying the powerful tool of bubble languages and their accompanying Gray codes. In fact, it was the discovery that

[1] This algorithm was originally presented in [11], where we proved that it ran in amortized $\mathcal{O}(n)$ time per word, and conjectured amortized $\Theta(\log n)$ time per word. Based on the result of [5] on the asymptotic number of prefix normal words, we have been able to prove the amortized $\mathcal{O}(\log^2 n)$ running time per word.

[2] Those languages are actually 10-bubble, while prefix normal words are 01-bubble: the difference is simply exchanging the role of 0 and 1 in the definition, see [27, 28].

$d=0$	$d=1$	$d=2$	$d=3$	$d=4$	$d=5$	$d=6$	$d=7$
0000000	1000000	1010000	1101000	1101100	1110110	1110111	1111111
		1001000	1010100	1110100	1111010	1111011	
		1000100	1100100	1101010	1101101	1111101	
		1000010	1010010	1100110	1110101	1111110	
		1000001	1100010	1110010	1101011		
		1100000	1010001	1101001	1110011		
			1001001	1010101	1111001		
			1100001	1100101	1111100		
			1110000	1100011			
				1110001			
				1111000			

Table 1: All prefix normal words of length 7 as output by our algorithm.

prefix normal words formed a bubble language that led to an efficient generation algorithm and Gray code for our language.

The final part of the paper deals with *membership testing*, i.e. deciding whether a given binary word is prefix normal. Several quadratic-time membership testers for prefix normal words were given in [12, 18]. The best worst-case time tester can be obtained by using the connection to Indexed Binary Jumbled Pattern Matching (BJPM), for which the current best algorithm, by Chan and Lewenstein, runs in $O(n^{1.864})$ time [13]. We present a new membership tester for prefix normal words which applies a simple two-phase approach and is conjectured to run in average-case $O(n)$ time, where the average is taken over all words of length n.

1.2 Related work

In addition to the connection to jumbled indexing, prefix normal words are also increasingly being studied for their own sake. Enumeration and language-theoretic results were given by Burcsi et al. in [12], and Balister and Gerke strengthened some results in [5]: in particular, they proved a conjecture about the asymptotic growth behaviour of the number of prefix normal words and gave a new result about the size of the equivalence classes. Cicalese et al. gave a generation algorithm in [15], with linear running time per word, and studied infinite prefix normal words in [16]. Prefix normal words and prefix normal forms have been applied to a certain family of graphs by Blondin-Massé et al. [7], and were shown to pertain to a new class of languages connected to the Reflected Binary Gray Code by Sawada et al. [29]. Very recently, Fleischmann et al. presented some results on the size of the equivalence classes in [**?**].

1.3 Overview

The paper is organized as follows. In Section 2, we give the necessary terminology and some basic facts about prefix normal words, and we develop a result on the average critical prefix length of a prefix normal word. This result will later be used in the analysis of our generating algorithm. In Section 3, we give a simple generation algorithm which, based on the result of [5], is proved to run in amortized $\mathcal{O}(n)$ per word. In Section 4, we present our novel view of bubble languages. In Section 5, we introduce our new generation algorithm, which uses the bubble framework. In Section 6, we present the new membership tester. We close with some open problems in Section 7.

2 Preliminaries

A *binary word* (or *string*) $w = w_1 \cdots w_n$ over $\Sigma = \{0, 1\}$ is a finite sequence of elements from Σ. Its length n is denoted by $|w|$, and the i-th symbol of a word w by w_i, for $1 \leq i \leq |w|$. We denote by Σ^n the set of words over Σ of length n, by $\Sigma^* = \cup_{n \geq 0} \Sigma^n$ the set of all words over Σ, and by ε the empty word. Let $w \in \Sigma^*$. If $w = uv$ for some $u, v \in \Sigma^*$, we say that u is a *prefix* of w and v is a *suffix* of w. A *substring* of w is a prefix of a suffix of w. A *binary language* is any subset $\mathcal{L}$ of Σ^*. We denote by $|w|_c$ the number of occurrences in w of character $c \in \{0, 1\}$. The number of 1s in w, $|w|_1$, is also called the *weight* of w. For a binary language $\mathcal{L}$, let $\mathcal{L}(n)$ denote the subset of all strings in $\mathcal{L}$ with length n, and $\mathcal{L}(n, d)$ that of all strings in $\mathcal{L}$ with length n and weight d.

We denote by *swap*(w, i, j) the string obtained from w by exchanging the characters in positions i and j.

We define combinatorial Gray codes, following [26, ch. 5]: Given a set of combinatorial objects $\mathcal{S}$ and a relation C on $\mathcal{S}$ (the closeness relation), a *combinatorial Gray code* for $\mathcal{S}$ is a listing $s_1, s_2, ..., s_{|\mathcal{S}|}$ of the elements of $\mathcal{S}$, such that $(s_i, s_{i+1}) \in C$ for $i = 1, 2, ..., |\mathcal{S}| - 1$. If we also require that $(s_{|\mathcal{S}|}, s_1) \in C$, then the code is called *cyclic*.

2.1 Prefix normal words

Let $w = w_1 w_2 \cdots w_n$ be a binary word. For each $i = 0, 1, \ldots, n$, we define $P(w, i) = |w_1 \cdots w_i|_1$, the weight of the prefix of length i, and $F(w, i) = \max\{|u|_1 : u \text{ is a substring of } w \text{ and } |u| = i\}$, the maximum weight of i-length substrings of w. The function F is sometimes called *maximum-ones function*, while in the context of compact data structures, function P is often called *rank*$_1(w, i)$ [25].

Definition 2.1 *A word $w \in \Sigma^*$ is called* prefix normal *if for all $1 \leq i \leq |w|$, $F(w, i) = P(w, i)$. We denote by $\mathcal{L}_{PN}$ the language of prefix normal words, and by pnw$(n) = |\mathcal{L}_{PN}(n)|$, the number of prefix normal words of length n.*

In [12, 18] it was shown that for every word w there exists a unique word w', called its *prefix normal form*, such that for all $1 \leq i \leq |w|$, $F(w, i) = F(w', i)$, and w' is prefix normal. We give the formal definition:

Definition 2.2 *Given a word $w \in \{0, 1\}^n$, the* prefix normal form of w*, denoted* $\mathrm{PNF}(w)$*, is the prefix normal word w' given by $w'_i = F(w, i) - F(w, i-1)$, for $i = 1, \ldots, n$. Two words w, v are* prefix normal equivalent *if* $\mathrm{PNF}(w) = \mathrm{PNF}(v)$.

As an example, the word $w = 11100110110$ has the maximum-ones function $F(w, \cdot) = 0, 1, 2, 3, 3, 4, 4, 5, 5, 6, 7, 7$, as can be checked easily. It is furthermore not difficult to see that for all $i < n$: $F(w, i+1) = F(w, i)$ or $F(w, i + 1) = F(w, i) + 1$. Thus the sequence of first differences $F(w, i + 1) - F(w, i)$ yields a binary word, in this case the word 11101010110. In Table 2 we list all prefix normal words of length 5 followed by the set of binary words with this prefix normal form.

The next lemma lists some properties of prefix normal words which will be needed in the following. Proofs can be found in [12].

Lemma 2.3 ([12]) *Let w be a binary word.*

1. *w is prefix normal if and only if all of its prefixes are prefix normal.*
2. *If w is prefix normal, then so is $w0$.*

$\mathcal{L}_{\text{PN}} \cap \Sigma^5$	Words with this prefix normal form	$\mathcal{L}_{\text{PN}} \cap \Sigma^5$	Words with this prefix normal form
11111	$\{11111\}$	11000	$\{11000, 011000, 00110, 00011\}$
11110	$\{11110, 01111\}$	10101	$\{10101\}$
11101	$\{11101, 10111\}$	10100	$\{10100, 01010, 00101\}$
11100	$\{11100, 01110, 00111\}$	10010	$\{10010, 01001\}$
11011	$\{11011\}$	10001	$\{10001\}$
11010	$\{11010, 10110, 01101, 01011\}$	10000	$\{10000, 01000, 00100, 00010, 00001\}$
11001	$\{11001, 10011\}$	00000	$\{00000\}$

Table 2: All prefix normal words of length 5 and their equivalence classes.

3. *Let w be prefix normal. Then the word $w1$ is prefix normal if and only if, for every suffix u of w, $|u|_1 < P(w, |u| + 1)$.*

4. *w is prefix normal if and only if* $\mathrm{PNF}(w) = w$.

We refer the interested reader to [12] for more on prefix normal words.

2.2 Critical prefix length

It will often be useful to write binary words $w \neq 1^n$ as $w = 1^s0^t\gamma$, where $s \geq 0, t \geq 1$ and γ is either ε or a binary word beginning with 1. In other words, s is the length of the first, possibly empty, 1-run of w, t is the length of the first 0-run, and γ the remaining, possibly empty, suffix. Note that this representation is unique.

Definition 2.4 *Let $w \in \{0,1\}^n \setminus \{1^n\}$, $w = 1^s0^t\gamma$, where $0 \leq s, 1 \leq t$ and $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$. We refer to 1^s0^t as w's* critical prefix*, and denote by $cr(w) = s + t$ the* critical prefix length *of w.*

For example, the critical prefix length of 11101010110 is 4, that of 11111000000 is 11, and that of 00101110110 is 2.

Lemma 2.5 *The expected critical prefix length of a binary string w of length n is $3 - \frac{n+3}{2^n}$.*

Proof. Let $w = 1^s0^t\gamma$, with $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$. Let X be a random variable with $X = cr(w)$. We have that $X = n$ if and only if $w = 1^s0^{n-s}$, with $s = 0, 1, \ldots, n$, so for $n + 1$ words. Otherwise $X = k$ for some $0 < k < n$, and

$$Pr(X = k) = \sum_{s=0}^{k-1} p^s(1-p)^{k-s}p = \sum_{s=0}^{k-1} (\frac{1}{2})^{k+1} = \frac{k}{2^{k+1}},$$

since the probability of having a 1 is $\frac{1}{2}$. Therefore,

$$Exp(X) = \sum_{k=0}^{n} k \cdot Pr(X = k) = \sum_{k=0}^{n-1} \frac{k^2}{2^{k+1}} + \frac{n(n+1)}{2^n} = 3 - \sum_{k=n}^{\infty} \frac{k^2}{2^{k+1}} + \frac{n(n+1)}{2^n} = 3 - \frac{n+3}{2^n},$$

where we have used in the last two equations that $\sum_{k\geq 1} \frac{k^2}{2^{k+1}} = 3$, and that the tail of the infinite sum has the closed form $2^{-n}(n^2+2n+3)$. □

Remark: It can be shown in a similar way that the expected critical prefix length of a randomly chosen infinite binary word is 3.

Lemma 2.6 *The sequence $C(n)$ of the sum of $cr(w)$ over all words w of length n obeys the recurrence $C(n) = 2C(n-1) + (n+1)$, with $C(0) = 0$.*

Proof. Consider all strings of length $n-1$ and what happens to their critical prefix length if one character is added. For those w with $cr(w) < n-1$, it just stays the same, and since we get two new strings $w1$ and $w0$, these $cr(w)$ are counted twice. The remaining strings are either of the form1^s0^{n-1-s}, with $s = 0, \ldots, n-2$, in which case adding a 0 will increase $cr(w)$ by 1, and adding a 1 will not; there are $n-1$ many of these. Else it is 1^{n-1}, in which case adding a 0 or a 1 will increase $cr(w)$ by 1. So altogether we get $C(n) = \sum_{|w|=n} cr(w) = \sum_{|w|=n-1} 2cr(w) + (n-1) + 2 = 2C(n-1) + (n+1)$. □

Incidentally, the sequence $C(n) = 3 \cdot 2^n - (n+3)$, is listed as sequence $A095151$ of the OEIS [30], along with the second-order recurrence $C(n) = 3C(n-1) - 2C(n-2) + 1$, for $n \geq 3$, and $C(0) = 0, C(1) = 2$. That it also obeys the first-order recurrence of Lemma 2.6 can be seen by computing the difference $C(n+1) - C(n)$ and substituting the recursive formula of order 2 for both.

Next we show that the expected critical prefix of the prefix normal form of a randomly chosen word is $\Theta(\log n)$. Note that this is not the same as the expected critical prefix length of a random prefix normal word, due to the fact that the equivalence class sizes of the prefix normal equivalence vary considerably (see [18], and Thm. 2 in [5]).

Lemma 2.7 *Given a random word w, let w' be the prefix normal form of w. Then the expected critical prefix length of w' is $\Theta(\log n)$.*

Proof. Let $w' = 1^{s'}0^{t'}\gamma'$, with $\gamma' \in 1\{0,1\}^* \cup \{\varepsilon\}$ and the r.v.'s $S' = s'$ and $T' = t'$. It is known that the expected maximum length of a run in a random word of length n is $\Theta(\log n)$ [19]. Clearly, S' equals the length of the longest run of 1's of w, thus $Exp(S') = \Theta(\log n)$. What about T'? Consider a 1-run of w of maximum length s'. If w has more than s' 1's, then there is a substring of w consisting of this 1-run and one more 1; the number of 0's in this substring is an upper bound on t'. Since these 0s form one single run, their number is again $O(\log n)$ in expectation. If w has exactly s' 1's, then $w' = 1^{s'}0^{n-s'}$, so $t' = n - s' \leq n$. The number of words with at most one 1-run is $\binom{n+1}{2} + 1$. So we have:

$$Exp(cr(w')) = Exp(S' + T') = \Theta(\log n) + (1 - \frac{\Theta(n^2)}{2^n})O(\log n) + \frac{\Theta(n^2)}{2^n}n) = \Theta(\log n).$$

□

It is easy to see that the number of prefix normal words grows exponentially (just note that $1^{|w|}w$ is prefix normal for every w). Balister and Gerke [5] recently proved a conjecture from [12] about the asymptotic number of prefix normal words:

Theorem 2.8 ([5], Thm. 1) *The number of prefix normal words of length n is $2^{n-\Theta(\log^2 n)}$.*

We will use this theorem to prove an upper bound on $Exp(cr(w))$ for prefix normal words w. We need the following lemma.

Lemma 2.9 *Let $t = o(n)$ and $t \geq 2 \log n$, and suppose that $pnw(n) \geq 2^{n-t}$. Let Z be a random variable taking values $cr(w)$ for prefix normal words $w \in \mathcal{L}_{PN}(n)$. Then $Exp(Z) = \mathcal{O}(t)$.*

Proof. Consider all prefix normal words $w \in \mathcal{L}_{\text{PN}}(n)$ with $cr(w) < 2t$. These contribute $\mathcal{O}(t)$ to $Exp(Z)$. Now consider all prefix normal words $w \in \mathcal{L}_{\text{PN}}(n)$ with $cr(w) \geq 2t$. There are at most $(2t+1)2^{n-2t}$ binary words with $cr(w) \geq 2t$, since these words must begin with one of the patterns $1^{2t}, 1^{2t-1}0, 1^{2t-2}00, \ldots, 0^{2t}$, and therefore, at most this number of prefix normal words with $cr(w) \geq 2t$. Each prefix normal word can only contribute at most n to the average. So the contribution to the average summed over all prefix normal words with $cr(w) \geq 2t$ is at most $n(2t+1)2^{n-2t}/pnw(n) \leq n(2t+1)2^{n-2t}/2^{n-t}$, which is $O(1)$, since $t \geq 2 \log n$, and hence negligible:

$$Exp(Z) = \frac{1}{pnw(n)} \left(\sum_{\substack{w \in \mathcal{L}_{\text{PN}}(n), \\ cr(w) < 2t}} cr(w) + \sum_{\substack{w \in \mathcal{L}_{\text{PN}}(n), \\ cr(w) \geq 2t}} cr(w) \right) \leq \frac{|\{w \in \mathcal{L}_{\text{PN}}(n) \mid cr(w) < 2t\}| \cdot 2t}{pnw(n)} +$$

$$+ \frac{(2t+1)2^{n-2t} \cdot n}{2^{n-t}} \leq \frac{|\{w \in \mathcal{L}_{\text{PN}}(n) \mid cr(w) < 2t\}| \cdot 2t}{pnw(n)} + \frac{(2t+1) \cdot n}{n^2} \leq 2t + \frac{(2t+1)}{n} = \mathcal{O}(t).$$

□

Theorem 2.10 *The expected length of the critical prefix of a prefix normal word of length n is $\mathcal{O}(\log^2 n)$.*

Proof. By Theorem 2.8, we know that there exists a constant $c > 0$ such that, for sufficiently large n, $pnw(n) \geq 2^{n - c \log^2 n}$. Applying Lemma 2.9, we get $Exp(cr(w)) = \mathcal{O}(\log^2 n)$, where w ranges over all prefix normal words of length n. □

3 A Simple Generation Algorithm for Prefix Normal Words

Our first generation algorithm uses Lemma 2.3: (1) A word is prefix normal if and only if all of its prefixes are prefix normal; (2) if w is prefix normal, so is $w0$, but not necessarily $w1$; and (3) $w1 \in \mathcal{L}_{\text{PN}}$ if and only if for every suffix u of w, the number of ones in u is strictly less than $P(w, |u|+1)$. Words $w \in \mathcal{L}_{\text{PN}}$ for which $w1$ is not prefix normal are called *extension critical*. Thus, whether a word is extension critical can be tested in linear time in $|w|$.

We can therefore generate all prefix normal words of length n by iteratively generating all prefix normal words of length k, for $k = 1, \ldots, n-1$, and extending each one by a 0 if it is extension critical, or by a 0 and a 1 if it is not. This yields a computation tree whose leaves are precisely the prefix normal words of length n. We refer to this algorithm as Simple Generation Algorithm.

Theorem 3.1 *The Simple Generation Algorithm generates all prefix normal words of length n in $O(n)$ amortized time per word.*

Proof. Notice that an extension critical test is performed in each inner node, taking $O(k)$ time if the node is at depth k. An inner node at depth k corresponds to a prefix normal word of length k, so the number of tests equals the total number of prefix normal words of length smaller than n. From Theorem 2.8 (Balister and Gerke, 2019 [5]) it follows that most prefix normal words are not extension critical, in particular more than half of all prefix normal words of a given length can be extended. Therefore, $\mathit{pnw}(n) \geq \frac{3}{2}\mathit{pnw}(n-1)$, and by induction $\sum_{k=1}^{n-1} \mathit{pnw}(k) \leq 2\mathit{pnw}(n)$, implying that the total time taken by the algorithm to generate all prefix normal words of length n is

$$\sum_{k=1}^{n-1} k \cdot \mathit{pnw}(k) \leq \sum_{k=1}^{n-1} n \cdot \mathit{pnw}(k) = O(n) \cdot \mathit{pnw}(n).$$

□

4 Bubble Languages and Combinatorial Generation

In this section we give a brief introduction to bubble languages, mostly summarising results from [27, 28]. However, our presentation is different in that it presents the generation of a bubble language as a restriction of an algorithm for generating *all* binary words. This view also yields a new characterization of bubble languages in terms of the computation tree of this generation algorithm (Prop. 4.2).

Definition 4.1 ([27, 28]) *A language $\mathcal{L} \subseteq \{0,1\}^*$ is called* a first-01 bubble language *if, for every word $w \in \mathcal{L}$, exchanging the first occurrence of* 01 *(if any) by* 10 *results in another word in $\mathcal{L}$. It is called a* a first-10 bubble language *if, for every word $w \in \mathcal{L}$, exchanging the first occurrence of* 10 *(if any) by* 01 *results in another word in $\mathcal{L}$. If not further specified, by* bubble language *we mean first-01 bubble.*

For example, the languages of binary Lyndon words and necklaces are 10-bubble languages. As was shown in [27], a language $\mathcal{L} \subseteq \{0,1\}^n$ is a bubble language if and only if each of its fixed-weight subsets $\mathcal{L}(n,d)$ is a bubble language. This implies that for generating a bubble language, it suffices to generate its fixed-weight subsets.

Next we consider combinatorial generation of binary strings. Let w be a binary string of length n, let d be its weight, and let $i_1 < i_2 < \ldots < i_d$ denote the positions of the 1s in w. Clearly, we can obtain w from the word 1^d0^{n-d} with the following algorithm: first swap the last 1 with the 0 in position i_d, then swap the $(d-1)$st 1 with the 0 in position i_{d-1} etc. Note that every 1 is moved at most once, and in particular, once the k'th 1 is moved into the position i_k, the suffix $w_{i_k} \cdots w_n$ remains fixed for the rest of the algorithm.

These observations lead to the RECURSIVE SWAP GENERATION ALGORITHM (Algorithm 1). Starting from the string $1^s0^t\gamma$, it generates recursively all n-length binary strings with weight d and fixed suffix γ, where $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$. The call RECURSIVESWAP$(d, n-d, \varepsilon)$ generates all binary strings of length n with weight d. The algorithm swaps the last 1 of the first 1-run with each of the 0s of the first 0-run, thereby generating a new string each, for which it makes a recursive call. During the execution of the algorithm, the current string resides in a global array w. The function SWAP(i,j) swaps the values stored in w_i and w_j. In the subroutine VISIT() we can print the contents of this array, or increment a counter, or check some property of the current string. Crucially, VISIT() is called on every string exactly once.

Let T_d^n denote the computation tree of RECURSIVESWAP$(d, n-d, \varepsilon)$. As an example, Fig. 1 illustrates T_4^7 (ignore for now the highlighted words). In slight abuse of notation, in the following we identify a node v with

Algorithm 1 Recursive Swap Generation Algorithm to generate all binary strings of length n.

```
1: procedure RecursiveSwap(s, t, γ)
2:     if s > 0 and t > 0 then
3:         for i ← 1 to t do
4:             Swap(s,s+i)
5:             RecursiveSwap(s−1, i, 10^{t−i}γ)
6:             Swap(s,s+i)
7:     Visit()

8: for d ← 0 to n do
9:     RecursiveSwap(d, n − d, ε)
```

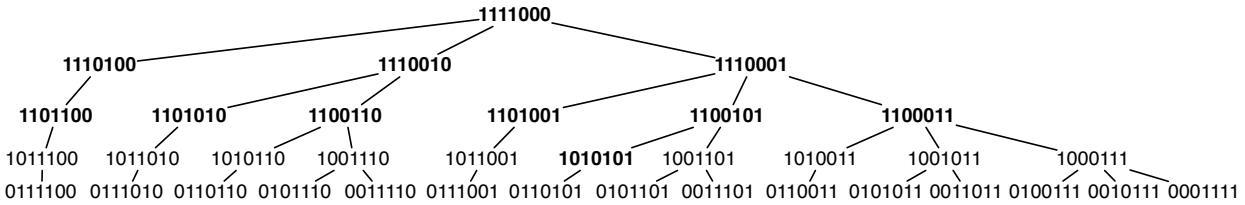

Figure 1: The computation tree T_d^n for $n = 7, d = 4$. Prefix normal words in bold.

the string it represents. The depth of T_d^n equals d, the number of 1s, while the maximum degree (number of children) is $n-d$, the number of 0s. Consider the subtree rooted at $v = 1^s0^t\gamma$: its depth is s and the maximum degree of nodes is t; the number of children of v itself is exactly t, and v's ith child is $1^{s-1}0^i10^{t-i}\gamma$. Note that suffix γ remains unchanged in the entire subtree; that the computation tree is isomorphic to the computation tree of 1^s0^t; and that the critical prefix length strictly decreases along any downward path in the tree. The algorithm performs a post-order traversal of the tree, yielding a listing of the strings of length n with weight d, in what is referred to as *cool-lex order* [27, 28, 31].

We can express the property of bubble language in terms of the computation tree T_d^n as follows:

Proposition 4.2 *A language $\mathcal{L} \subseteq \{0,1\}^n$ is a bubble language if and only if, for every $d = 0, \ldots, n$, its fixed-density subset $\mathcal{L}(n,d)$ is closed w.r.t. parents and left siblings in the computation tree T_d^n of the Recursive Swap Generation Algorithm. In particular, if $\mathcal{L}(n,d) \neq \emptyset$, then it forms a subtree rooted in 1^d0^{n-d}.*

Proof. Follows immediately from the definition of bubble languages. □

Using Prop. 4.2, the RECURSIVE SWAP GENERATION ALGORITHM can be applied to generate *any* fixed-weight bubble language $\mathcal{L}$, as long as we have a way of deciding, for a node $w = 1^s0^t\gamma$, already known to be in $\mathcal{L}$, which is its *rightmost child* (if any) that is still in $\mathcal{L}$. If such a child exists, and it is the jth child $u = 1^{s-1}0^j10^{t-j}\gamma$, then the bubble property ensures that all children to its left are also in $\mathcal{L}$. Thus, line 2. in Algorithm 1 can simply be replaced by "for $i = 1, \ldots, j$".

The framework provided in [27, 28] to list the strings in $\mathcal{L}(n,d)$ for a given bubble language $\mathcal{L}$, can thus be viewed as a restriction of the RECURSIVE SWAP GENERATION ALGORITHM: Given a string $w = 1^s0^t\gamma \in \mathcal{L}$, compute the largest integer j such that $1^{s-1}0^j10^{t-j}\gamma \in \mathcal{L}$, in other words, the rightmost child of node $1^s0^t\gamma \in \mathcal{L}$ which is still in $\mathcal{L}$, called the *bubble upper bound*[3]. This simple framework is outlined

[3]In [27, 28], actually a "bubble lower bound" is computed. Because we feel it simplifies the discussion, here we introduce a

in Algorithm 2 for a given bubble language $\mathcal{L}$, where the current word $w = w_1 w_2 \cdots w_n$ is stored globally. The function ORACLE(s, t) returns the bubble upper bound for w with respect to $\mathcal{L}$. The membership tester MEMBER$(\mathcal{L}, w)$ returns true if and only if $w \in \mathcal{L}$. The initial call is GENBUBBLE$(d, n-d)$ with w initialized to $1^d 0^{n-d}$.

Algorithm 2 Generic algorithm to list $\mathcal{L}(n, d)$ for a given bubble language $\mathcal{L}$ in cool-lex order.

```
1:  function ORACLE(s, t)
2:      j ← 1
3:      while j ≤ t and MEMBER(L, 1^{s-1}0^j 10^{t-j}γ) do j ← j + 1
4:      return j − 1

5:  procedure GENBUBBLE(s, t)
6:      if s > 0 and t > 0 then
7:          for i ← 1 to ORACLE(s, t) do
8:              SWAP(s, s+i)
9:              GENBUBBLE(s−1, i)
10:             SWAP(s, s+i)
11:     VISIT()
```

It was further shown in [27] that cool-lex order, the order in which the generic algorithm visits the strings of $\mathcal{L}(n, d)$, gives a Gray code. This can be seen on the tree T_d^n as follows:

Lemma 4.3 *Let u be a node in the computation tree T_d^n. Then each of the following can be obtained from u by a single swap operation: (a) any sibling of u, (b) $parent(u)$, and (c) any node on the leftmost path in the subtree rooted in u.*

Proof. Let u and u' be siblings, and let their parent be $v = 1^s 0^t \gamma$. Then there exist i, j such that $u = 1^{s-1} 0^i 1 0^{t-i} \gamma$ and $u' = 1^{s-1} 0^j 1 0^{t-j} \gamma$. Then $u' = swap(u, s+i, s+j)$, while $v = parent(u) = swap(u, s+i, s)$. For (c), let $u = 1^s 0^t \gamma$ and $u' = 1^k 0 1^{s-k} 0^{t-1} \gamma$ for some k; then $u' = swap(u, k+1, s+1)$. □

We now report the main result on bubble languages from [27, 28], for which we give a proof using Prop. 4.2.

Proposition 4.4 ([27, 28]) *Any fixed-length bubble language $\mathcal{L}(n)$, where $\mathcal{L}(n, d) \neq \emptyset$ for all $d = 0, \ldots, n$, can be generated such that subsequent strings differ by at most two swaps, or by a swap and a bit flip. Given a membership tester* MEMBER$(\mathcal{L}, w)$ *which runs in $\mathcal{O}(m)$ time, this generation algorithm takes amortized $\mathcal{O}(m)$ time per word.*

Proof. For a fixed-weight subset $\mathcal{L}(n, d)$, let $T_{\mathcal{L}}$ denote the subtree of T_d^n corresponding to $\mathcal{L}(n, d)$. Note that in a post-order traversal of $T_{\mathcal{L}}$, we have:

$$next(u) = \begin{cases} parent(u) & \text{if } u \text{ is rightmost child} \\ \text{leftmost descendant of } u\text{'s right sibling} & \text{otherwise.} \end{cases}$$

related value called the "bubble upper bound". The bubble lower bound is equal to t minus the bubble upper bound.

By Prop. 4.2, we have that the leftmost descendant of any node in $T_{\mathcal{L}}$ lies on the leftmost path in T_d^n. Thus, by Lemma 4.3, $next(u)$ can be reached in one or two swaps.

By concatenating the lists for weights $0, 1, \ldots, n$, the procedure GENERATEALL(n) shown in Algorithm 3 will exhaustively list $\mathcal{L}(n)$ for a given bubble language $\mathcal{L}$. To see the Gray code property, notice that for any weight d, the last string visited is 1^d0^{n-d}, while the first string visited for the next weight $d+1$ is the leftmost descendant of $1^{d+1}0^{n-d-1}$, i.e. a string of the form $1^i01^{d+1-i}0^{n-d-2}$, which is one swap and one bit flip away from 1^d0^{n-d}.

For the running time, notice that for $w = 1^s0^t\gamma$, we do at most $j+1$ membership tests, where j is the bubble upper bound for w. The j successful tests can be charged to the j children of w, while the possible last unsuccessful test can be charged to w itself. □

Algorithm 3 A Gray code to exhaustively list $\mathcal{L}(n)$ for a given bubble language $\mathcal{L}$.

```
1: procedure GENERATEALL(n)
2:     for d ← 0 to n do
3:         w_1 w_2 ··· w_n ← 1^d 0^{n−d}
4:         GENBUBBLE(d, n − d)
```

Remark: It is even possible to give a cyclic Gray code for $\mathcal{L}(n)$, by giving the fixed-weight subsets listed first by the odd weights (increasing), followed by the even weights (decreasing).

The oracle of Algorithm 2 applies a simple membership tester to compute the bubble upper bound for given $w \in \mathcal{L}$. However, we do not actually need a *general* membership tester, since all we want to know is which of the children of a node *already known to be in* $\mathcal{L}$ are in $\mathcal{L}$; moreover, the membership tester is allowed to use other information, which it can build up iteratively while examining earlier nodes. In the next section, we will apply this method to the language of prefix normal words.

5 A Gray Code for Prefix Normal Words

In this section, we prove that the set of prefix normal words $\mathcal{L}_{\text{PN}}$ is a bubble language. Then using the bubble framework and applying a basic quadratic-time membership tester, we show how to generate all words in $\mathcal{L}_{\text{PN}}(n, d)$ in Gray code order. By concatenating the lists together for all weights in increasing order, we obtain an algorithm to list $\mathcal{L}_{\text{PN}}(n)$ as a Gray code in $O(n^2)$ amortized time per word. By then providing an enhanced membership tester for prefix normal words specific to the bubble framework, we further show how this Gray code can be generated in $O(\log^2 n)$ amortized time.

Theorem 5.1 *$\mathcal{L}_{PN}$ is a bubble language.*

Proof. Let w be a prefix normal word containing an occurrence of 01. Let w' be the word obtained from w by replacing the first occurrence of 01 with 10. Then $w = u01v$, $w' = u10v$ for some $u, v \in \Sigma^*$. Let z be a substring of w'. We have to show that $|z|_1 \leq P(w', |z|)$.

Note that for any k, $P(w,k) \leq P(w',k)$. In fact, $P(w', |u|+1) = P(w, |u|) + 1$, and for every $k \neq |u|+1$, $P(w,k) = P(w',k)$. Now if z is contained in u or in v, then z is a substring of w, and thus $|z|_1 \leq P(w,|z|) \leq P(w',|z|)$. If $z = u'10v'$, with u' suffix of u and v' prefix of v, then $|z|_1 = |u'01v'|_1 \leq P(w,|z|) \leq P(w',|z|)$. If $z = 0v'$, with v' prefix of v, then $|z|_1 < |1v'|_1$, and $1v'$ is a substring of w,

Figure 2: Illustration of Lemma 5.2. On the right we have the two cases of a substring u (in gray) of w' which may violate the prefix normal property.

thus $|z|_1 \leq P(w, |z|) \leq P(w', |z|)$. Else $z = u'1$, with u' suffix of u. We can assume that u' is a proper suffix of u. Let z' be the substring of w' of the same length as z and starting one position before z (in other words, z' is obtained by shifting z to the left by one position). Since u does not contain 01 as a substring, we have $u = 1^n0^m$ for some $n \geq 1, m \geq 0$. If z' is a power of 0's, then $|z|_1 = 1$ and the claim holds. Else, $|z|_1 = |z'|_1$, and z' is a substring of w. Thus $|z|_1 \leq P(w, |z|) \leq P(w', |z|)$. □

Since there is a membership tester for prefix-normal words that runs in $O(n^2)$ time, e.g. as described in Algorithm 4, the aforementioned Gray codes for both $\mathcal{L}_{\text{PN}}(n, d)$ and $\mathcal{L}_{\text{PN}}(n)$ can be generated in $O(n^2)$ amortized time (Prop. 4.4). We show the computation tree T_4^7 in Fig. 1, with prefix normal words in bold. The complete listing for $d = 0, 1, \ldots, 7$ is given in Table 1.

Algorithm 4 Test if $w_1w_2 \cdots w_n \in \mathcal{L}_{\text{PN}}$ in $O(n^2)$ time.

```
1: function MEMBER( L_PN, w_1 w_2 ··· w_n )
2:     p_0 ← 0
3:     for i ← 1 to n do  p_i ← p_{i−1} + w_i
4:     for i ← 2 to n do
5:         f ← 0
6:         for j ← i to n do
7:             f ← f + w_j
8:             if f > p_{j−i+1} then return FALSE
9:     RETURN TRUE
```

5.1 A More Efficient Approach

Now we develop a more efficient membership tester for $\mathcal{L}_{\text{PN}}$ that is specific to one required by an oracle for bubble languages. In particular, membership tests are only made on strings of the form $w = 1^{s-1}0^j10^{t-j}\gamma$, given that $1^s0^t\gamma \in \mathcal{L}_{\text{PN}}$.

Lemma 5.2 *Let $w = 1^s0^t\gamma$ be a word in $\mathcal{L}_{PN}$ where $s \geq 0$, $t \geq 1$ and $\gamma \in 1\{0,1\}^* \cup \{\varepsilon\}$. Let $w' = b_1b_2 \cdots b_n = swap(w, s, s + j)$ for some $1 \leq j \leq t$. Then w' is* **not** *in $\mathcal{L}_{PN}$ if and only if either*

1. *$F(\gamma 0^{s+t}, s + j - 1) \geq s$, or*
2. *$|b_{s+j}b_{s+j+1} \cdots b_{2(s+j-1)}|_1 \geq s$.*

Proof. The proof is illustrated in Fig. 2. Note that $w' = 1^{s-1}0^j10^{t-j}\gamma$.

($\Leftarrow$) The prefix of $b_1b_2 \cdots b_{s+j-1}$ of w' has $s - 1$ ones. If $F(\gamma 0^{s+t}, s + j - 1) \geq s$, then it must be that γ contains a substring of length $s+j-1$, or less, with at least s ones. Similarly, if $|b_{s+j}b_{s+j+1} \cdots b_{2(s+j-1)}|_1 \geq$

s then w' also contains a substring of length $s+j-1$ with at least s ones. Thus, if either $F(\gamma 0^{s+t}, s+j-1) \geq s$ or $|b_{s+j}b_{s+j+1}\cdots b_{2(s+j-1)}|_1 \geq s$ then w' is not in $\mathcal{L}_{\text{PN}}$.

($\Rightarrow$) Assume w' is not in $\mathcal{L}_{\text{PN}}$. Then there is a shortest substring $u = b_i b_{i+1}\cdots b_{i+m-1}$ with length m in w' such that $|u|_1 > P(w', m)$. Clearly $i > 1$ and since m is minimal $b_i = b_{i+m-1} = 1$. Suppose $i+m-1 \leq s+j$. Then u can have at most $s-1$ ones since $i > 1$ and thus $|u|_1 \leq P(w', m)$, a contradiction. Thus $i + m - 1 > s + j$. We now consider three cases for i. If $i < s + j$ then since $b_i = 1$, $2 \leq i < s$. But since $b_1 b_2 \cdots b_{i-1} = 1^{i-1}$, this means $P(w', m) \geq |u|_1$. Thus $i \geq s + j$. Suppose $i = s + j$. If $m > s + j - 1$ then the prefix of w' of length m overlaps with u, i.e. we can write $b_1 b_2 \cdots b_m = vv'$ and $u = v'u'$ for some non-empty v' containing the swapped 1. Since $|u|_1 > P(w', m)$, this implies that also u' has more 1s than the prefix of the same length, a contradiction to our choice of u. Thus $m \leq s + j - 1$. Since w' starts with 1^{s-1} and $|u|_1 > P(w', m)$ it must be that $|u|_1 \geq s$. By extending u to have length $s + j - 1$ we have $|b_{s+j}b_{s+j+1}\cdots b_{2(s+j-1)}|_1 \geq s$. Finally, suppose $i > s + j$. Then because $b_i = 1$, u is a substring of γ and hence a substring of w. Since $w \in \mathcal{L}_{\text{PN}}$ we have $|u|_1 \leq P(w, m)$. Since $P(w, m) = P(w', m)$ for all $m < s$ and $m \geq s + j$, and $|u|_1 > P(w', m)$, it must be that $s \leq m \leq s + j - 1$. For each of these possible values for m, $P(w', m) = s - 1$. Thus $|u|_1 \geq s$ which means $F(\gamma, m) \geq s$. Finally, since the length of $\gamma 0^{s+t}$ is at least $s + j - 1$, we also have $F(\gamma 0^{s+t}, s + j - 1) \geq s$. Considering all cases, we must either have $F(\gamma 0^{s+t}, s + j - 1) \geq s$, or $|b_{s+j}b_{s+j+1}\cdots b_{2(s+j-1)}|_1 \geq s$. □

Let f_i denote the value $F(\gamma 0^{s+t}, i)$. By maintaining $f_1, f_2, \ldots, f_{s+t}$ as GENBUBBLE iterates through the prefix normal words, we can apply the previous lemma to optimize a membership tester. Pseudocode is given in Algorithm 5. This function requires the passing of the variables s and j from the function ORACLE, recalling that the current string $w_1 w_2 \cdots w_n$ is stored globally.

Algorithm 5 Membership testing for $\mathcal{L}_{\text{PN}}$ specific to cool-lex framework in $O(s + t)$ time.

```
1: function MEMBERPN(s, j)
2:     ones ← 1                          ▷ first 1 accounted for by the (proposed) swap of a 1 to w_{s+j}
3:     for i ← s + j + 1 to 2(s + j − 1) do
4:         if w_i = 1 then ones ← ones + 1
5:     if ones ≥ s or f_{s+j−1} ≥ s then return FALSE
6:     RETURN TRUE
```

The resulting membership tester clearly runs in $O(s+t)$ time since $j \leq t$. In order to apply the oracle, we must maintain the data structure $f_1, f_2, \ldots, f_{s+t}$ as we proceed through the recursive generation algorithm. Since the length of the critical prefix $1^s 0^t$ decreases as we go deeper in the computation tree, it is sufficient to update $f_1, f_2, \ldots, f_{s+i}$ as the bits in positions s and $s + i$ get swapped. Observe that this swap changes γ by replacing the prefix 0^i with 1. Thus, to update the f values we can simply scan the first $s + i$ bits (of the updated γ) as illustrated in UPDATEF(x) of Algorithm 6, where x corresponds to $s + i$. This function should be inserted just before the recursive call is made in GENBUBBLE(s, t) on line 9 of Algorithm 2. Since these values need to be restored *after* the recursive call, we need to first save the initial values $f_1, f_2, \ldots, f_{s+i}$ in a temporary array so they can be restored. A complete C implementation is given in Appendix A.

5.2 Analysis

For each prefix normal word w generated by GENERATEALL(n), using the optimized membership tester for prefix normal words, the algorithm requires $O(cr(w))$ time to update $f_1, f_2, \ldots, f_{cr(w)}$. Thus, the overall

Algorithm 6 Update required values for f where $x = s + i$.

```
1: procedure UPDATEF(x)
2:     ones ← 0
3:     for k ← x to 2x do
4:         if w_k = 1 then ones ← ones + 1
5:         f_{k−x+1} ← MAX(f_{k−x+1}, ones)
```

work done by the algorithm is proportional to $C(n) = \sum_{w \in \mathcal{L}_{\text{PN}}(n)} cr(w) = pnw(n) \cdot O(\log^2 n)$, by Theorem 2.10.

Theorem 5.3 *The set of words $\mathcal{L}_{PN}(n)$, where $n > 1$, can be generated in amortized $O(\log^2 n)$ time per word.*

Prefix normal words are the first interesting example of a bubble language for which no $O(1)$ amortized time generation algorithm is known.

6 Membership Testing

The best membership testing algorithm uses the fact that a word is prefix normal if and only if it equals its prefix normal form (see Lemma 2.3). As mentioned before, the most efficient algorithm for computing the F-function of a word w, and thus its the prefix normal form, is from [13] and runs in time $\mathcal{O}(n^{1.864})$. Here we present a simple two-phase membership tester which, even though $\mathcal{O}(n^2)$ in the worst-case, could outperform other algorithms in practice.

Now consider the following two-phase approach. Suppose there is an $O(n)$ test that rejects X_n binary strings of length n (Phase I). Then, for the remaining $Y_n = 2^n - X_n$ strings, apply the worst case $O(n^2)$ algorithm (Phase II). On average this will lead to a membership algorithm that runs in time:

$$\frac{c_1 n \cdot X_n + c_2 n^2 \cdot Y_n}{2^n},$$

for some constants c_1 and c_2. This expression will be less than $(c_1 + c_2)n$ if $Y_n \leq 2^n/n$, which implies an $O(n)$ average case tester. Thus, when designing an $O(n)$ time rejection tester in Phase I, we are aiming to reject a number proportional to $2^n - 2^n/n$. Without knowing exactly how many strings get rejected by a particular tester, we can focus on the following ratio:

$$ratio = \frac{nY_n}{2^n}.$$

As n grows, if this ratio is decreasing and bounded by a constant c, then $Y_n \leq c2^n/n$, which implies an $O(n)$ average case time tester.

Applying this approach, we try the following trivial $O(n)$ test for Phase I: a string will *not* be prefix normal if the longest run of 1s is not a prefix. Applying this test as the first phase, the resulting ratios for some increasing values of n are given in Table 3(a). Since the ratios are increasing as n increases, we require a more advanced rejection test.

The next attempt uses a more compact *run-length* representation for w. Let w be represented by a series of c blocks, which are maximal substrings of the form 1^*0^*. Each block B_i is composed of two integers

Algorithm 7 Membership tester: returns whether or not $w = w_1 w_2 \cdots w_n \in \mathcal{L}_{\text{PN}}$.

```
1: function MEMBERPNF( w )
2:     (s_1, t_1)(s_2, t_2) ··· (s_c, t_c) ← the run-length block encoding of w
3:     ▷ Phase I: linear time rejection tests
4:     for i ← 2 to c do
5:         if s_i > s_1 then return FALSE
6:         if s_{i-1} + t_{i-1} + s_i ≤ s_1 + t_1 and s_{i-1} + s_i > s_1 then return FALSE
7:     ▷ Phase II: call O(n^2) membership tester
8:     return MEMBER( L_PN, w )
```

(s_i, t_i) representing the number of 1s and 0s respectively. For example, the string 11100101011100110 can be represented by $B_1B_2B_3B_4B_5 = (3,2)(1,1)(1,1)(3,2)(2,1)$ where $c = 5$. Such a representation can easily be found in $O(n)$ time. A word w will *not* be prefix normal word if it contains a substring of the form $1^i0^j1^k$ such that $i + j + k \leq s_1 + t_1$ and $i + k > s_1$ (the substring is not longer, yet has more 1s than the critical prefix). Thus, a word is not prefix normal if for some $2 \leq i \leq c$:

$$s_{i-1} + t_{i-1} + s_i \leq s_1 + t_1 \quad \text{and} \quad s_{i-1} + s_i > s_1.$$

By applying this additional test in our first phase, we obtain Algorithm 7. The ratios that result from this algorithm are given in Table 3(b). Similar decreasing ratios also occur for odd n.

n	$ratio$
10	2.500
12	2.561
14	2.602
16	2.631
18	2.656
20	2.675
22	2.693
24	2.708

(a)

n	$ratio$
10	2.168
12	2.142
14	2.121
16	2.106
18	2.093
20	2.083
22	2.075
24	2.067

(b)

Table 3: (a) Ratios from the trivial rejection test. (b) Ratios by adding secondary rejection test.

Since the ratios achieved by the combination of the two tests are decreasing as n increases (see Table 3(b)), we make the following conjecture:

Conjecture 6.1 *The membership tester* MEMBERPNF*(w) for $\mathcal{L}_{PN}$ runs in $O(n)$-time on average, where the average is taken over all words of length n.*

We note that one can conceive of several other (perhaps more advanced) rejection tests that run in $O(n)$ time, however, these two were sufficient to obtain our desired experimental results.

7 Conclusion and Open Problems

The main result of this paper is a generating algorithm for prefix normal words, which is shown to run in amortized $\mathcal{O}(\log^2 n)$ time per word, as opposed to the hitherto best $\mathcal{O}(n)$ time per word algorithm. Our algorithm is based on the fact that prefix normal words form a bubble language, thus the general framework for bubble languages can be applied, and the algorithm outputs the language as a Gray code.

We further presented a novel view of bubble languages, in terms of subtrees of the computation tree of a generating algorithm for all binary strings. We hope that this view will aid readers to apply the bubble framework to other binary languages. Finally, we gave a membership tester for prefix normal words, which we conjecture to run in average linear time over all binary words.

We conclude with the following open problems on prefix normal words:

1. Given a prefix normal word w, efficiently list all words with prefix normal form w (i.e., its equivalence class). The maximum size of an equivalence class is listed in the OEIS as sequence A238110 [30]. Note that in the recent article [5], the authors prove that the maximum equivalence class size is asymptotically $2^{n-O(\sqrt{n \log n})}$.

2. Derive a closed form enumeration formula for the number $pnw(n)$ of prefix normal words of length n, or a generating function for $pnw(n)$.

3. Develop an algorithm to exhaustively list all prefix normal words in constant amortized time per word.

4. Develop a general membership tester for prefix normal words which runs in $o(n^{1.864})$ time in the worst case.

Acknowledgements

We thank Frank Ruskey for useful discussions. We further thank the organizers of the Dagstuhl Seminar no. 18281 [6], which took place in July 2018, and which gave some of the authors an opportunity to collaborate on prefix normal words.

References

[1] P. Afshani, I. van Duijn, R. Killmann, and J. S. Nielsen. A lower bound for jumbled indexing. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, (SODA 2020)*, pages 592–606, 2020.

[2] A. Amir, A. Apostolico, T. Hirst, G. M. Landau, N. Lewenstein, and L. Rozenberg. Algorithms for jumbled indexing, jumbled border and jumbled square on run-length encoded strings. *Theoret. Comput. Sci.*, 656:146–159, 2016.

[3] A. Amir, A. Butman, and E. Porat. On the relationship between histogram indexing and block-mass indexing. *Philosophical Transactions of The Royal Society A: Mathematical Physical and Engineering Sciences*, 372(2016), 2014.

[4] A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein. On hardness of jumbled indexing. In *41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*, volume 8572 of *LNCS*, pages 114–125, 2014.

[5] P. Balister and S. Gerke. The asymptotic number of prefix normal words. *Theoret. Comput. Sci.*, 784:75–80, 2019.

[6] J. Barbay, J. Fischer, S. Kratsch, and S. R. Satti. Synergies between Adaptive Analysis of Algorithms, Parameterized Complexity, Compressed Data Structures and Compressed Indices (Dagstuhl Seminar 18281). *Dagstuhl Reports*, 8(7):44–61, 2019.

[7] A. Blondin Massé, J. de Carufel, A. Goupil, M. Lapointe, É. Nadeau, and É. Vandomme. Leaf realization problem, caterpillar graphs and prefix normal words. *Theoret. Comput. Sci.*, 732:1–13, 2018.

[8] P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták. Algorithms for Jumbled Pattern Matching in Strings. *International Journal of Foundations of Computer Science*, 23:357–374, 2012.

[9] P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták. On approximate jumbled pattern matching in strings. *Theory Comput. Syst.*, 50(1):35–51, 2012.

[10] P. Burcsi, G. Fici, Zs. Lipták, F. Ruskey, and J. Sawada. Normal, abby normal, prefix normal. In *Proc. of the 7th International Conference on Fun with Algorithms (FUN 2014)*, volume 8496 of *LNCS*, pages 74–88, 2014.

[11] P. Burcsi, G. Fici, Zs. Lipták, F. Ruskey, and J. Sawada. On combinatorial generation of prefix normal words. In *Proc. of the 25th Ann. Symp. on Comb. Pattern Matching (CPM 2014)*, volume 8486 of *LNCS*, pages 60–69, 2014.

[12] P. Burcsi, G. Fici, Zs. Lipták, F. Ruskey, and J. Sawada. On prefix normal words and prefix normal forms. *Theoret. Comput. Sci.*, 659:1–13, 2017.

[13] T. M. Chan and M. Lewenstein. Clustered integer 3SUM via additive combinatorics. In *Proc. of the 47th Ann. ACM on Symp. on Theory of Computing (STOC 2015)*, pages 31–40, 2015.

[14] F. Cicalese, E. S. Laber, O. Weimann, and R. Yuster. Approximating the maximum consecutive subsums of a sequence. *Theoret. Comput. Sci.*, 525:130–137, 2014.

[15] F. Cicalese, Zs. Lipták, and M. Rossi. Bubble-flip - A new generation algorithm for prefix normal words. *Theoret. Comput. Sci.*, 743:38–52, 2018.

[16] F. Cicalese, Zs. Lipták, and M. Rossi. On infinite prefix normal words. In *Proc. of the 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2019)*, pages 122–135, 2019.

[17] L. F. I. Cunha, S. Dantas, T. Gagie, R. Wittler, L. A. B. Kowada, and J. Stoye. Faster jumbled indexing for binary RLE strings. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, pages 19:1–19:9, 2017.

[18] G. Fici and Zs. Lipták. On prefix normal words. In *Proc. of the 15th Intern. Conf. on Developments in Language Theory (DLT 2011)*, volume 6795 of *LNCS*, pages 228–238. Springer, 2011.

[19] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, USA, 1 edition, 2009.

[20] P. Fleischmann, D. Nowotka, M. Kulczynski, and D. B. Poulsen. On collapsing prefix normal words. In *Proc. of the 14th International Conference Language and Automata Theory and Applications (LATA 2020), Milano, Italy, March 4-6, 2020*, 2020.

[21] T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015.

[22] E. Giaquinta and S. Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14–16):538–542, 2013.

[23] T. Kociumaka, J. Radoszewski, and W. Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. *Algorithmica*, 77(4):1194–1215, 2017.

[24] T. M. Moosa and M. S. Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discr. Alg.*, 10:5–9, 2012.

[25] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2, 2007.

[26] F. Ruskey. *Combinatorial Generation*. 2003.

[27] F. Ruskey, J. Sawada, and A. Williams. Binary bubble languages and cool-lex order. *J. Comb. Theory, Ser. A*, 119(1):155–169, 2012.

[28] J. Sawada and A. Williams. Efficient oracles for generating binary bubble languages. *Electr. J. Comb.*, 19(1):P42, 2012.

[29] J. Sawada, A. Williams, and D. Wong. Inside the Binary Reflected Gray Code: Flip-Swap languages in 2-Gray code order. Unpublished manuscript, 2017.

[30] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. Available electronically at `http://oeis.org`.

[31] A. Williams. *Shift Gray Codes*. PhD thesis, University of Victoria, Canada, 2009.

Appendix A: C code

```
//-----------------------------------------------
// COOL-LEX GRAY CODE for Prefix Normal Words
// OEIS:  http://oeis.org/A194850
//-----------------------------------------------
#include <stdio.h>
#define MAX(a,b) ((a > b) ? a : b)

int a[100], F[100], N, NO_OUTPUT=0, COLEX=0;
long long int total = 0;

//---------------------------------
void Visit() {
int i;

        if (!NO_OUTPUT) {
                for (i=1; i<=N; i++) {
                        if (a[i] == 0) printf("1");
                        else printf("0");
                }
                printf("\n");
        }
        total++;
}
//---------------------------------
void Swap(int i, int j) {
int tmp;

        tmp = a[i];     a[i] = a[j];      a[j] = tmp;
}
//---------------------------------
int Member_PNF(int s, int j) {
int i, ones=1;

        for (i=s+j+1; i<=2*(s+j-1); i++) if (a[i] == 1) ones++;
        if (ones >= s || F[s+j-1] >= s) return 0;
        return 1;
}
//---------------------------------
int Oracle_PNF(int s, int t) {
int j=1;

        while (j <= t && Member_PNF(s,j)) j++;
        return j-1;
}
//---------------------------------
int UpdateF(int x) {
int j, ones=0;

        for (j=x; j<=2*x; j++) {
                if (a[j] == 1) ones++;
                F[j-x+1] = MAX(F[j-x+1],ones);
        }
}
//----------------------------
// COOL LEX GRAY CODE or COLEX
//----------------------------
void Gen(int s, int t) {
int i, j, k, G[100];

        if (COLEX) Visit();
        if (s > 0 && t > 0) {
```

```
                j = Oracle_PNF(s,t);
                for (i=1; i<=j; i++) {
                        Swap(s,s+i);
                        for (k=s+i; k<=2*(s+i); k++) G[k-(s+i)+1] = F[k-(s+i)+1];

                        UpdateF(s+i);
                        Gen(s-1,i);

                        for (k=s+i; k<=2*(s+i); k++) F[k-(s+i)+1] = G[k-(s+i)+1];
                        Swap(s,s+i);
                }
        }
        if (!COLEX) Visit();
}
//---------------------------------
int main( ) {
int i, j, output, D;

        //-------
        // INPUT
        //-------
        printf("\nSELECT output [1]Cool-lex Gray code [2]Co-lex [3]Just counts: ");
        scanf("%d", &output);

        if (output == 2) COLEX = 1;
        if (output == 3) NO_OUTPUT = 1;

        printf("ENTER length n: ");    scanf("%d", &N);
        printf("ENTER # of a's (or -1 for all PN words): ");    scanf("%d", &D);
        printf("\n");

        for (i=1; i<=N; i++) F[i] = 0;

        //-----------
        //GENERATION
        //-----------
        if (D == -1) {
                for (j=0; j<=N; j++)  {
                        for (i=1; i<=j; i++)   a[i] = 1;
                        for (i=j+1; i<=2*N; i++) a[i] = 0;      // PAD WITH N 1s
                        Gen(j,N-j);
                }
        }
        else {
                for (i=1; i<=D; i++)     a[i] = 1;
                for (i=D+1; i<=2*N; i++) a[i] = 0;              // PAD WITH N 1s
                Gen(D,N-D);
        }

        printf("Total = %lld\n", total);
}
```