THEORY OF COMPUTING, Volume 12 (7), 2016, pp. 1–27
www.theoryofcomputing.org

Efficient Indexing of Necklaces and Irreducible Polynomials over Finite Fields

Swastik Kopparty* Mrinal Kumar† Michael Saks‡

Received April 2, 2015; Revised March 6, 2016; Published August 13, 2016

Abstract: We study the problem of *indexing* irreducible polynomials over finite fields, and give the first efficient algorithm for this problem. Specifically, we show the existence of poly$(n, \log q)$-size circuits that compute a bijection between $\{1, \ldots, |S|\}$ and the set S of all irreducible, monic, univariate polynomials of degree n over a finite field $\mathbb{F}_q$. This has applications to pseudorandomness, and answers an open question of Alon, Goldreich, Håstad, and Peralta (1992).

Our approach uses a connection between irreducible polynomials and necklaces (equivalence classes of strings under cyclic rotation). Along the way, we give the first efficient algorithm for indexing necklaces of a given length over a given alphabet, which may be of independent interest.

ACM Classification: F.2.1, G.2.1, I.1.2

AMS Classification: 11T06, 68R15, 68W40

Key words and phrases: indexing algorithms, necklaces, irreducible polynomials, explicit constructions

1 Introduction

For a finite field $\mathbb{F}_q$ and an integer n, let S be the set of all monic irreducible polynomials in 1 variable over $\mathbb{F}_q$ of degree exactly n. There is a well known formula for $|S|$ (which is approximately q^n/n). We

A conference version of this paper appeared in the Proceedings of the 41st ICALP, 2014 [16].
*Supported in part by a Sloan Fellowship and NSF grant CCF-1253886.
†Supported in part by NSF grant CCF-1253886.
‡Supported in part by NSF grants CCF-0832787 and CCF-1218711.

DOI: 10.4086/toc.2016.v012a007

consider the problem of giving an efficiently computable *indexing* of irreducible polynomials, i. e., finding a bijection $f : \{1,\dots,|S|\} \to S$ such that $f(i)$ is computable in time $\mathsf{poly}(\log|S|) = \mathsf{poly}(n\log q)$. Our main result is that indexing of irreducible polynomials can be done efficiently given $O(n\log q)$ bits of advice. This answers a problem posed by Alon, Goldreich, Håstad, and Peralta [3], and is the polynomial analogue of the the well-known problem of "giving a formula for the n-bit primes." Note that today it is not even known (in general) how to produce a single irreducible polynomial of degree n in time $\mathsf{poly}(n\log q)$ without the aid of either advice or randomness.

The main technical result we show en route is an efficient indexing algorithm for *necklaces*. Necklaces are equivalence classes of strings modulo cyclic rotation. We give an $\mathsf{poly}(n\log|\Sigma|)$-time computable bijection $g : \{1,2,\dots,|\mathcal{N}|\} \to \mathcal{N}$, where $\mathcal{N}$ is the set of necklaces of length n over the alphabet Σ.

1.1 The indexing problem

We define an *indexing* of a finite set S to be a bijection from the set $\{1,\dots,|S|\}$ to S. Let us formalize indexing as a computational problem. Suppose that L is an arbitrary language over alphabet Σ and let L^n be the set of strings of L of length n. We want to "construct" an indexing function A^n for each of the sets L^n. Formally, this means giving an algorithm A which takes as input a size parameter n and an index j and outputs $A^n(j)$, so that the following properties hold for each n.

- A^n maps the set $\{1,\dots,|L^n|\}$ bijectively to L^n.
- If $j > |L^n|$, then $A^n(j)$ returns **too large**.

An indexing algorithm is considered to be efficient if its running time is $\mathsf{poly}(n)$.

A closely related problem is *reverse-indexing*. A reverse-indexing of L is a bijection from L^n to $\{1,\dots,|L^n|\}$, and we say it is efficient if it can be computed in time $\mathsf{poly}(n)$.

We can use the above formalism for languages to formulate the indexing and reverse-indexing problems for any combinatorial structure, such as permutations, graphs, partitions etc., by using standard efficient encodings of such structures by strings.

1.2 Indexing, enumeration, counting and ranking

Indexing is closely related to the well-studied *counting*, *enumeration* and *ranking* problems for L. The counting problem is to give an algorithm that, on input n, outputs the size of L^n. The enumeration problem is to give an algorithm that, on input n, outputs a list containing all elements of L^n. A counting or enumeration algorithm is said to be efficient if it runs in time $\mathsf{poly}(n)$ or $|L^n|\cdot\mathsf{poly}(n)$ respectively.

Other important algorithmic problems associated with combinatorial objects include the *ranking* and *unranking* problems. For the ranking problem, one is given an ordering of L^n (such as the lexicographic order) and the goal is to compute the rank (under this order) of a given element of L^n. For the unranking problem, one has to compute the inverse of this ranking map. It is easy to see that unranking algorithms for any ordering are automatically indexing algorithms, and ranking algorithms for any ordering are automatically reverse-indexing algorithms.[1]

[1]We use the terms indexing and reverse-indexing instead of the terms unranking and ranking to make an important distinction:

There is well developed complexity theory for counting problems, starting with the fundamental work of Valiant [33]. For combinatorial structures, counting problems are (of course) at the heart of combinatorics, and many basic identities in combinatorics (such as recurrence relations that express the number of structures of a particular size in terms of the number of such structures of smaller sizes) can also be viewed as giving efficient counting algorithms for these structures.

The enumeration and ranking problems for combinatorial structures has also received a large amount of attention. See the books [22, 17, 25, 5] for an overview of some of the work on this topic.

Counting and enumeration can be easily reduced to indexing: Given an indexing algorithm A we can compute $|L^n|$ by calling $A^n(j)$ on increasing powers of 2 until we get the answer "**too large**" and then do binary search to determine the largest j for which $A^n(j)$ is not too large. Enumeration can be done by just running the indexing algorithm on the integers $1,2,\ldots$ until we get the answer "**too large**."

Conversely, in many cases, such as for subsets, permutations, set partitions, integer partitions, trees, spanning trees (and many more), the known counting algorithms can be modified to give efficient indexing (and hence enumeration) algorithms. This happens, for example, when the counting problem is solved by a recurrence relation that is proved via a bijective proof.

However, it seems that not all combinatorial counting arguments lead to efficient indexing algorithms. A prime example of this situation is when we have a finite group acting on a finite set, and the set we want to index is the set of orbits of the action. The associated counting problem can be solved using the Burnside counting lemma, and there seems to be no general way to use this to get an efficient indexing algorithm.

This leads us to one of the indexing problems studied here: Fix an alphabet Σ and consider two strings x and y in Σ^n to be equivalent if one is a rotation of the other, i. e., we can find strings x^1, x^2 such that $x = x^1x^2$ and $y = x^2x^1$ (here uv denotes the concatenation of the strings u and v). The equivalence classes of strings are precisely the orbits under the natural action of the cyclic group $\mathbb{Z}_n$ on Σ^n. These equivalence classes are often called *necklaces* because if we view the symbols of a string as arranged in a circle, then equivalent strings give rise to the same arrangement. We are interested in the problem of efficiently indexing necklaces. We apply the indexing algorithm for necklaces to the problem of indexing irreducible polynomials over a finite field.

1.3 Main results

Our main result is an efficient algorithm for indexing irreducible polynomials.

Theorem 1.1. *Let q be a prime power, and let $n \geq 1$ be an integer. Let $I_{q,n}$ be the set of monic irreducible polynomials of degree n over $\mathbb{F}_q$.*

There is an indexing algorithm for $I_{q,n}$, which takes $O(n \log q)$ bits of advice and runs in $\mathsf{poly}(n, \log q)$ time.

in indexing and reverse-indexing the actual bijection between $\{1,\ldots,|S|\}$ and S is of no importance whatsoever, but in ranking and unranking the actual bijection is part of the problem. We feel this difference is worth highlighting, and hence we introduced the new terms indexing and reverse-indexing for this purpose. Note that some important prior work on ranking/unranking distinguishes between these notions [21].

We remark that it is not known today[2] how to deterministically produce (without the aid of advice or randomness) even a single irreducible polynomial of degree n in time $\mathsf{poly}(n \log q)$ for all choices of n and q. Our result shows that once we take a little bit of advice, we can produce not just one, but all irreducible polynomials. For constant q, where it is known how to deterministically construct a single irreducible polynomial in $\mathsf{poly}(n)$ time without advice [29], our indexing algorithm can be made to run with just $O(\log n)$ bits of advice.

Using a known correspondence [12] between necklaces and irreducible polynomials over finite fields, indexing irreducible polynomials reduces to the problem of indexing necklaces. Our main technical result (of independent interest) is an efficient algorithm for this latter problem.

Theorem 1.2. *There is an algorithm for indexing necklaces of length n over the alphabet $\{1,\dots,q\}$, which runs in time* $\mathsf{poly}(n \log q)$.

Our methods also give an efficient reverse-indexing algorithm for necklaces (but unfortunately this does not lead to an efficient reverse-indexing algorithm for irreducible polynomials; this has to do with the open problem of efficiently computing the discrete logarithm).

Theorem 1.3. *There is an algorithm for reverse-indexing necklaces of length n over the alphabet $\{1,\dots,q\}$, which runs in time* $\mathsf{poly}(n \log q)$.

The indexing algorithm for irreducible polynomials can be used to make a classical ε-biased set construction from [3] based on linear-feedback shift register sequences constructible with logarithmic advice (to put it at par with the other constructions in that paper). It can also be used to make the explicit subspace designs of [13] very explicit (with small advice).

Agrawal and Biswas [2] gave a construction of a family of nearly-coprime polynomials, and used this to give randomness-efficient black-box polynomial identity tests. The ability to efficiently index irreducible polynomials enables one to do this even more randomness-efficiently (using a small amount of advice).

Similarly, the string fingerprinting algorithm by Rabin [24], which is based on choosing a random irreducible polynomial can be made more randomness-efficient by choosing the random irreducible polynomial via first choosing a random index and then indexing the corresponding irreducible polynomial using our indexing algorithm. This application also requires a small amount of advice.

As another application of the indexing algorithm for necklaces, we give a $\mathsf{poly}(n)$ time algorithm for computing any given entry of the $k \times 2^n$ generator matrix or the $(2^n - k) \times 2^n$ parity check matrix of BCH codes for all values of the designed distance (this is the standard notion of strong explicitness for

[2]The problem of efficiently constructing an irreducible polynomial over $\mathbb{F}_q$ of degree n has been extensively studied. It is known how to solve this problem under the following relaxations:

- with randomness (folklore),
- under the Generalized Riemann Hypothesis (Adleman-Lenstra [1]),
- when $\mathbb{F}_q$ has small characteristic (Shoup [28]),
- given a construction of $\mathbb{F}_{q^n}$ (Lenstra [18]),
- if we only require an irreducible polynomial of degree approximately n (Adleman-Lenstra [1]).

error-correcting codes). Earlier, it was only known how to compute this entry explicitly for very small values of the designed distance (which is usually the setting where BCH codes are used).

1.4 Related Work

There is an extensive literature on enumeration algorithms for combinatorial objects (see the books [25, 14, 17, 22, 5]). Some of these references discuss necklaces in depth, and some also discuss the ranking/unranking problems for various combinatorial objects.

The lexicographically smallest element of a rotation class is called a *Lyndon word*, and much is known about them. Algorithmically, the problem of enumerating/indexing necklaces is essentially equivalent to the problem of enumerating/indexing Lyndon words. Following a long line of work [10, 11, 26, 8, 6, 27, 7], we now know $O(1)$-amortized time enumeration algorithms for Lyndon words/necklaces.

In [20] and [25], it was noted that the problem of efficient ranking/unranking of the lexicographic order on Lyndon words is an open problem. Our indexing algorithms in fact give a solution to this problem too: we get an efficient ranking/unranking algorithm for the lexicographic order on Lyndon words.

Recent work of Andoni, Goldberger, McGregor and Porat [4] studied a problem that may be viewed as an approximate version of reverse indexing of necklaces. They gave a randomized algorithm for producing short fingerprints of strings, such that the fingerprints of rotations of a string are determined by the fingerprint of the string itself. This fingerprinting itself was useful for detecting proximity of strings under misalignment.

Recent independent work. A preliminary version of this paper appeared as [16]. At about the same time, similar results were published by Kociumaka, Radoszewski and Rytter [15]. The work in these two papers was done independently. The papers both have polynomial-time algorithms for indexing necklaces; the authors in [15] exercised more care in designing the algorithm to obtain a better polynomial running time. Their approach to alphabets of size greater than 2 is cleaner than ours. On the other hand, we put the results in a broader context and have some additional applications (indexing irreducible polynomials and explicit constructions).

1.5 Organization of the paper

The rest of the paper is organized as follows. We give the algorithm to index necklaces in Section 2. In Section 3, we use our indexing algorithm for necklaces to give an indexing algorithm for irreducible polynomials over finite fields. In Section 4, we give an application to the explicit construction of generator and parity check matrices of BCH codes. We conclude with some open problems in Section 5. In Appendix A, we give an alternate algorithm for indexing binary necklaces of prime length. In Appendix B, we make some preliminary observations about the complexity theory of indexing in general.

2 Indexing necklaces

2.1 Strategy for the algorithm

We first consider a very basic indexing algorithm which will inspire our algorithms. Given a directed acyclic graph D on vertex set V and distinguished subsets S and T of nodes, there is a straightforward indexing algorithm for the set of paths that start in S and end in T: Fix an arbitrary ordering on the nodes, and consider the induced lexicographic ordering on paths (i. e., path $P_1P_2\ldots$ is less than path $Q_1Q_2\ldots$ if $P_i < Q_i$ where i is the least integer such that $P_i \neq Q_i$). Our indexing function will map the index j to the jth path from S to T in lexicographic order. There is a simple dynamic program which computes for each node v, the number $N(v)$ of paths from v to a vertex in T. Let $v_1,\ldots,v_r$ be the nodes of S listed in order. Given the input index j, we find the first source v_i such that the number of paths to T starting at nodes $v_1,\ldots,v_i$ is at least j; if there is no such source then the index j is larger than the number of paths being indexed. Otherwise, v_i is the first node of the desired path, and we can proceed inductively by replacing the set S by the set of children of v_i and updating j appropriately.

This approach can be adapted to the following situation. Suppose the set S we want to index is a set of strings of fixed length n over alphabet Σ. A *read-once branching program* of length n over alphabet Σ is an acyclic directed graph with vertex layers numbered from 0 to n, where (1) layer 0 has a single start node, (2) there is a designated subset of accepting nodes at level n, and (3) every non-sink node has one outgoing arc corresponding to each alphabet symbol, and these arcs connect the node to nodes at the next level. For nodes v and w and alphabet symbol σ we write $v \to_\sigma w$ to mean that there is an arc from v to w labelled by σ. The size of a branching program B, denoted by $|B|$ is the number of vertices in it.

Such a branching program takes words from Σ^n and, starting from the start node, follows the path corresponding to the word to either the accept or reject node. Given a read-once branching program for S, there is a 1-1 correspondence between strings in S and paths from the start node to an accepting node. We can use the indexing algorithm for paths given above to index S.

This suggests the following approach to indexing necklaces. For each equivalence class of strings (necklace) identify a canonical representative string of the class (such as the lexicographically smallest representative). Then build a branching program B which, given string y, determines whether y is a canonical representative of its class. By the preceding paragraph, this would be enough to index all of the canonical representatives, which is equivalent to indexing equivalence classes.

In fact, we are able to implement this approach provided that $q = 2$ and n is prime (see Appendix A). However, we have not been able to make it work in general. For this we need another approach, which still uses branching programs, but in a more involved way.

First some notation. For a given string y, we write the string obtained from y after cyclically rotating it rightwards by i positions as $\mathsf{Rot}^i(y)$. We define $\mathsf{Orbit}(y)$ to be the set containing y and all its distinct rotations. $\mathsf{Orbit}(y)$ will also be referred to as the *equivalence class* of y. A string y is said to be periodic with period p if it can be written as ${y_1}^m$ for some $y_1 \in \Sigma^p$ and $m = n/p$. A string is said to have fundamental period p if it is periodic with period p and not periodic with any period smaller than p. We will denote the fundamental period of a string y by $\mathsf{FP}(y)$. Note that for any string y, $|\mathsf{Orbit}(y)| = \mathsf{FP}(y)$.

If E is an orbit and x is a string, we say that $E < x$ if E contains at least one string y that is lexicographically less than x. (Notice that under our definition, if x and y are strings then we might have both that the orbit of x is less than y and the orbit of y is less than x.)

Let t be the total number of orbits. Let $\mathcal{C}_x$ be the set of orbits that are less than x. Our main goal will be to design an efficient algorithm which, given string x, returns $|\mathcal{C}_x|$. We now show that if we can do this, then we can solve both the indexing and reverse indexing problems.

For the indexing problem, we want a 1-1 function ψ that maps $j \in \{1,\dots,t\}$ to a string so that all of the image strings are in different orbits. The map ψ will be easily computable given a subroutine for $|\mathcal{C}_x|$.

Define the *minimal representative* of an orbit to be the lexicographically least string in the orbit. Let $y^1 < \cdots < y^t$ denote the minimal representatives in lex order. Our map ψ will map j to y^j. This clearly maps each index to a representative of a different orbit.

It suffices to show how to compute $\psi(j)$. Note that $|\mathcal{C}_x|$ is equal to the number of y^i that precede x, and is thus a nondecreasing function of x. Therefore, $\psi(j) = y^j$ is equal to the lexicographically largest string with $|\mathcal{C}_x| < j$. Furthermore, since $|\mathcal{C}_x|$ is a nondecreasing function of x, we can find $\psi(j)$ by doing binary search on the set of strings according to the value of $|\mathcal{C}_x|$.

Similarly, we can solve the reverse indexing problem: given a string x we can find the index of the orbit to which it belongs by first finding the lexicographically minimal representative y^i of its orbit and then computing $|\mathcal{C}_{y^i}|+1$.

Lemma 2.1. To efficiently index and reverse index necklaces of length n over an alphabet Σ, it suffices to have an efficient algorithm that takes as input a string $x \in \Sigma^n$ and outputs $|\mathcal{C}_x|$.

The next section gives our algorithm to determine $|\mathcal{C}_x|$ for any input string x.

2.2 Computing $|\mathcal{C}_x|$

Let us define:

$$G_{x,p} = \bigcup_{E\in\mathcal{C}_x:|E|=p} E \qquad \text{and} \qquad G_{x,\le p} = \bigcup_{E\in\mathcal{C}_x:|E| \text{ divides } p} E\,.$$

In Section 2.2.2 we reduce the problem of computing of $|\mathcal{C}_x|$ to the problem of computing $|G_{x,\le p}|$ for various p. The main component of the indexing algorithm is a subroutine that computes $|G_{x,\le p}|$ given a string x and an integer p. This subroutine works by building a branching program with $n^{O(1)}$ nodes, which when given a string y accepts if and only if (1) the orbit of y has size dividing p and (2) $\mathsf{Orbit}(y) < x$. The quantity we want to compute, $|G_{x,\le p}|$, is therefore simply the number of inputs y accepted by this branching program (which, as noted above can be computed in polynomial time via a simple dynamic program).

2.2.1 Preliminaries

We state some basic facts about periodic strings without proof.

Fact 2.2. *Let y be a string of length n and let p be positive integer dividing n. Then, $|\mathsf{Orbit}(y)| = p$ if and only if y has fundamental period p. In particular, y can be written as ${y_1}^{n/p}$ for an aperiodic string $y_1 \in \Sigma^p$.*

Fact 2.3. *The fundamental period of a string is a divisor of any period of the string.*

In particular, the fundamental period of a string is unique.

Throughout this paper, for a natural number m, we denote by $[m]$, the set $\{0,1,2,\dots m-1\}$.

2.2.2 Reduction to computing $|G_{x,\leq p}|$

We begin with some simple transformations that reduce the computation of $|\mathcal{C}_x|$ to the computation of $|G_{x,\leq p}|$ (for various p).

Lemma 2.4. For all $x \in \Sigma^n$,

$$|\mathcal{C}_x| = \sum_{y \in G_{x,\leq n}} \frac{1}{|\mathsf{Orbit}(y)|} = \sum_{y \in G_{x,\leq n}} \frac{1}{\mathsf{FP}(y)}.$$

Proof. For $y \in G_{x,\leq n}$, $\mathsf{Rot}^i(y) \in G_{x,\leq n}$ for every positive integer i. Note that there are exactly $|\mathsf{Orbit}(y)|$ distinct strings of the form $\mathsf{Rot}^i(y)$. Thus for any orbit $E \in \mathcal{C}_x$, we have

$$\sum_{y \in E} \frac{1}{|\mathsf{Orbit}(y)|} = 1.$$

Therefore:

$$\sum_{y \in G_{x,\leq n}} \frac{1}{|\mathsf{Orbit}(y)|} = \sum_{E \in \mathcal{C}_x} \sum_{y \in E} \frac{1}{|\mathsf{Orbit}(y)|} = \sum_{E \in \mathcal{C}_x} 1 = |\mathcal{C}_x|.$$ □

The sum on the right hand side can be split on the basis of the period of y. From Lemma 2.4, Fact 2.2 and Fact 2.3, we have the following lemma.

Lemma 2.5. For all $x \in \Sigma^n$,

$$|\mathcal{C}_x| = \sum_{i|n} \frac{|G_{x,i}|}{i}.$$

So, to count $|\mathcal{C}_x|$ efficiently, it suffices to compute $|G_{x,i}|$ efficiently for each $i \mid n$. Now, from the definitions, we have the following lemma.

Lemma 2.6. For all $x \in \Sigma^n$,

$$|G_{x,\leq p}| = \sum_{i|p} |G_{x,i}|.$$

From the Möbius Inversion Formula (see Chapter 3 in [31] for more details), we have the following equality.

Lemma 2.7.

$$|G_{x,p}| = \sum_{i|p} \mu\left(\frac{p}{i}\right) |G_{x,\leq i}|.$$

Lemma 2.7 implies that it suffices to compute $|G_{x,\leq p}|$ efficiently for every divisor p of n. In the next few sections, we will focus on this sub-problem and design an efficient algorithm for this problem. We will first describe the algorithm when the alphabet is binary, and then generalize to larger alphabets.

2.2.3 Computing $|G_{x,\leq n}|$ efficiently for the binary alphabet

In this section, we will design an efficient algorithm that given a string $x \in \{0,1\}^n$ computes $|G_{x,\leq n}|$. On input x the algorithm will construct a branching program with the property that $|G_{x,\leq n}|$ is the number of accepting paths in the branching program. This number of accepting paths can be computed by a simple dynamic program as described at the beginning of Section 2.1.

Lemma 2.8. Given as input a branching program B of length n over alphabet Σ, we can compute the size of the set of accepted strings in time $\mathsf{poly}(|B|, \log n)$.

Proof. The number accepted strings is the number of paths from the start node to the accept node, and all such paths have length exactly n. If the start node is labelled i and the accept node is labelled j, then the number of accepted strings is the i,j entry in the n^{th} power of the adjacency matrix of the graph, and can thus be computed in time polynomial in the size of the graph and $\log n$ (by repeated squaring). □

We now describe how to construct, for each fixed string $x = x_1 x_2 \ldots x_n \in \{0,1\}^n$, a branching program B_x of size polynomial in n such that the strings accepted by B_x are exactly those in $G_{x,\leq n}$. Lemma 2.8 then implies that we can compute $|G_{x,\leq n}|$ in time polynomial in n.

For strings x, y, when is $y <_{\mathsf{lex}} x$? This happens and only if there exists an $i \in \{0,1,2,\ldots,n-1\}$ such that $y_j = x_j$ for every $j \leq i$ and $x_{i+1} > y_{i+1}$. In the case of binary strings of length n, we must have $x_{i+1} = 1$ and $y_{i+1} = 0$.

Definition 2.9. The set of witnesses for x, denoted L_x, is defined by:

$$L_x = \{s0 \mid s \in \{0,1\}^* \text{ such that } s1 \text{ is a prefix of } x\}.$$

We can summarize the discussion from the paragraph above as follows:

Observation 2.10. For $x, y \in \{0,1\}^n$, we have $y <_{\mathsf{lex}} x$ if and only if some prefix of y lies in L_x.

We will now generalize this observation to strings under rotation. For strings x, y, when is $\mathsf{Orbit}(y) < x$? Recall that $\mathsf{Orbit}(y) < x$ if for some $y' \in \mathsf{Orbit}(y)$, we have $y' <_{\mathsf{lex}} x$. From Observation 2.10, we know that this happens if and only if some $y' \in \mathsf{Orbit}(y)$ has some prefix w in L_x. Rotating back to y, two situations can arise. Either y contains w as a contiguous substring, or w appears as a “split substring” wrapped around the end of y. In the latter case, y has a prefix w_1 and a suffix w_2 such that $w_2 w_1 = w \in L_x$.

Recall that $G_{x,\leq n}$ is the set of y with $\mathsf{Orbit}(y) < x$. Thus, $y \in G_{x,\leq n}$ if and only if it has a contiguous substring as a witness, or it has a witness that is wrapped around its end. Let us separate these two cases out.

Definition 2.11. For a string $x \in \{0,1\}^n$,

$$G^c_{x,\leq n} = \{y \in \{0,1\}^n \mid y \text{ contains a string in } L_x \text{ as a contiguous substring}\},$$
$$G^w_{x,\leq n} = \{y \in \{0,1\}^n \mid y \text{ has a prefix } w_1 \text{ and suffix } w_2 \text{ such that } w_2 w_1 \in L_x\}.$$

From the discussion in the paragraph above, we have the following observation:

Observation 2.12.

$$G_{x,\le n} = G^c_{x,\le n} \cup G^w_{x,\le n}\,.$$

The branching program B_x will be obtained by combining two branching programs B^c_x and B^w_x, where the first accepts the strings in $G^c_{x,\le n}$ and the second accepts the strings in $G^w_{x,\le n}$. Each layer j of the branching program B_x is the product of layer j of B^c_x and layer j of B^w_x and we have arcs $(v,v') \to_\sigma (w,w')$ when $v \to_\sigma v'$ and $w \to_\sigma w'$. The accepting nodes at level $n+1$ are nodes (v,v') where v is an accepting node of B^c_x or v' is an accepting node of B^w_x. The resulting branching program clearly accepts the set of strings accepted by B^c_x or B^w_x.

Note that the branching programs B_x produced by the algorithm are never actually "run," but are given as input to the algorithm of Lemma 2.8 in order to determine $|G_{x,\le n}|$.

For a set of strings W, we will use $\mathsf{Prefix}(W)$ to denote the set of all prefixes of all strings in W (including the empty string Λ). Similarly, $\mathsf{Suffix}(W)$ denotes the set of all suffixes of of all strings in W (including the empty string Λ). Similarly, we will use $\mathsf{Substring}(W)$ for set of all contiguous substrings of strings in W.

For a string r, $Q(r)$ is the set of suffixes of r that belong to $\mathsf{Prefix}(L_x)$.

Constructing branching program B^c_x. We now present an algorithm which on input $x \in \{0,1\}^n$, runs in time polynomial in n and outputs a branching program B^c_x that recognizes L^c_x.

Definition 2.13. Branching program B^c_x.

1. Nodes at level j are triples (j,s,b) where $s \in \mathsf{Prefix}(L_x)$ and $b \in \{0,1\}$. (We want string s to be the longest suffix of $z_1z_2\ldots z_j$ that belongs to $\mathsf{Prefix}(L_x)$, and $b=1$ iff $z_1z_2\ldots z_j$ contains a substring that belongs to L_x.)

2. The start node is $(0,\Lambda,0)$ where Λ is the empty string.

3. The accepting nodes (n,s,b) are those with $b=1$.

4. For $j \le n$, the arc out of nodes $(j-1,s,b)$ labeled by alphabet symbol α is (j,s',b') where s' is the longest string in $Q(s\alpha)$ and $b'=1$ if s' contains a suffix in L_x and otherwise $b'=b$.

It is clear that the branching program can be constructed (as a directed graph) in time polynomial in n. It remains to show that it accepts those z that have a substring that belongs to L_x.

Fix a string $z \in \{0,1\}^n$. Let (j,s_j,b_j) be the jth vertex visited by the branching program on input z. Note that s_j is a suffix of $z_1\ldots z_j$. Let h_j be the index such that $s = z_{h_j}\ldots z_j$; if s is empty, we set $h_j = j+1$. For j between 1 and n let i_j be the least index such that $z_{i_j}\ldots z_j$ belongs to $\mathsf{Prefix}(L_x)$ (so $i_j = j+1$ if there is no such string). Note that $i_j \ge i_{j-1}$ since if $z_i\ldots z_j$ belongs to $\mathsf{Prefix}(L_x)$ so does $z_i\ldots z_{j-1}$.

The branching program is designed to make the following true:

Claim 2.14. *For j between 1 and n, $h_j = i_j$ and $b_j = 1$ if and only if a substring of $z_1\ldots z_j$ belongs to L_x.*

The claim for $b_j = 1$ implies that the branching program accepts the desired set of strings.

Proof. The claim follows easily by induction, where the basis $j = 0$ is trivial. Assume $j > 0$. First we show that $h_j = i_j$. By induction $h_{j-1} = i_{j-1}$ and by definition of h_j and i_j we have $i_j \leq h_j$. To show $h_j \leq i_j$, note that since $i_j \geq i_{j-1} = h_{j-1}$, the string $z_{i_j} \dots z_j$ is in $Q(s_{j-1}\alpha)$ and so is considered in the choice of s_j and thus $h_j = i_j$.

For the claim on b_j, if z has no substring in L_x, then b_j remains 0 by induction. If z has a substring in L_x, let $z_i \dots z_k$ be such a substring with k minimum. Then by the claim on t_k, $h_k \leq i$, and so $z_i \dots z_k$ is a suffix of s_k and so $b_k = 1$, and for all $j \geq k$, b_j continues to be 1. □

Constructing branching program B^w_x. In this section, the goal is to design a branching program which accepts the strings in $G^w_{x,\leq n}$, since such a string may not be in $G^c_{x,\leq n}$ and hence, may have been rejected by B^c_x. We now present an algorithm which on input $x \in \{0,1\}^n$, runs in time polynomial in n and outputs a branching program B^w_x that accepts the set of strings z that have a nonempty suffix u and nonemtpy prefix v such that uv belongs to L_x.

Definition 2.15. Branching program B^w_x.

1. Nodes at level j are triples (j, s, p) where $s \in \mathsf{Prefix}(L_x)$ and $p \in \mathsf{Substring}(L_x)$. (String s will be the longest suffix of $z_1 z_2 \dots z_j$ that belongs to $\mathsf{Prefix}(L_x)$ (as in B^c_x) and p is the longest prefix of $z_1 z_2 \dots z_j$ that belongs to $\mathsf{Substring}(L_x)$.)

2. The start node is $(0, \Lambda, \Lambda)$ where Λ is the empty string.

3. The accepting states are those states (n, s, p) such that p has a nonempty prefix p' and s has a nonempty suffix s' such that $s'p' \in L_x$.

4. For $j \leq n$, the arc out of state $(j-1, s, p)$ labeled by alphabet symbol α is (j, s', p') where s' is the longest string in $Q(s\alpha)$ and $p' = p\alpha$ if $|p| = j-1$ and $p\alpha \in \mathsf{Substring}(L_x)$, and $p' = p$ otherwise.

It is clear that the branching program can be constructed (as a directed graph) in time polynomial in n. It remains to show that it accepts L^w_x.

Fix a string $z \in \{0,1\}^n$. Let (j, s_j, p_j) be the jth node visited by the branching program on input z. Notice that s_j is calculated the same way in B^w_x as in B^c_x and so s_j is the longest suffix of $z_1 \dots z_j$ that belongs to $\mathsf{Prefix}(L_x)$.

An easy induction shows that p_j is the longest prefix of $z_1 \dots z_j$ belonging to $\mathsf{Substring}(L_x)$: Let k be the length of the longest prefix of z belonging to $\mathsf{Substring}(L_x)$. For $j \leq k$ we have $p_j = z_1 \dots z_j$ and for $j > k$, $p_j = z_1 \dots z_k$.

Finally, we need to show that the branching program accepts z if and only if z has a nonempty suffix s' and z has a nonempty prefix p' such that $s'p' \in L_x$. If the program accepts, then the acceptance condition and the fact that s_n is a suffix of z and p_n is a prefix of z implies that z has the required suffix and prefix. Conversely, if z has such a prefix p' and suffix s', then they each belong to $\mathsf{Prefix}(L_x)$. Since p_n is the longest prefix of z belonging to $\mathsf{Substring}(L_x)$, p' is a prefix of p_n and since s_n is the longest suffix of z belonging to $\mathsf{Prefix}(L_x)$, s' is a suffix of s_n. So the branching program will accept.

Putting things together. From the constructions, it is clear that the size of the branching programs B_x^w and B_x^c are polynomial in the size of L_x and hence polynomial in $n = |x|$. Moreover, by a product construction, we can efficiently construct the deterministic finite branching program B_x which accepts the strings accepted by B_x^w or B_x^c, which is $G_{x,\leq n}$. This observation, along with Lemma 2.8 implies the following lemma.

Lemma 2.16. There is an algorithm which takes as input a string x in $\{0,1\}^n$ and outputs the size of $G_{x,\leq n}$ in time polynomial in n.

2.2.4 Computing $|G_{x,\leq p}|$ efficiently for the binary alphabet

In this section, we will show that for every $p \mid n$, we can compute the quantity $|G_{x,\leq p}|$ efficiently. The algorithm will be a small variation of our algorithm for computing $|G_{x,\leq n}|$ from the previous section. Let p be a divisor of n with $p < n$. Every string $y \in G_{x,\leq p}$ is of the form $a^{n/p}$ for some $a \in \{0,1\}^p$, and every string in $\mathsf{Orbit}(y)$ is of the form $(\mathsf{Rot}^i(a))^{n/p}$, for some $i \leq p$. Let us write the string x as $x_1x_2\ldots x_{n/p}$ where for each i, x_i is of length exactly p. We will now try to characterize the strings in $G_{x,\leq p}$. From the definitions, $y = a^{n/p} \in G_{x,\leq p}$ if and only if there is a rotation $0 \leq i < p$ such that $(\mathsf{Rot}^i(a))^{n/p}$ has a prefix in L_x. This, in turn, can happen if and only if there is an $i < p$ such that one of the following is true.

- $\mathsf{Rot}^i(a) < x_1$ in lexicographic order, or
- there is j, $0 < j < n/p$, such that $\mathsf{Rot}^i(a) = x_1 = x_2 = x_3 = \cdots = x_i$ and $\mathsf{Rot}^i(a) < x_{i+1}$ in lexicographic order.

The strings $y = a^{n/p}$ for which a has a rotation which is less than x_1 in lexicographic order are exactly the strings of the form $c^{n/p}$ with $c \in G_{x_1,\leq p}$. Via the algorithm of the previous subsection, there is a polynomial in n time algorithm which outputs a branching program recognizing $G_{x_1,\leq p}$. The only strings which satisfy the second condition are of the form $c^{n/p}$, where c is a rotation of x_1 and $x_1 < x_{i+1}$ in lexicographic order. There are at most $|\mathsf{Orbit}(x_1)|$ such strings, and we can count them directly given x.

This gives us our algorithm for computing $|G_{x,\leq p}|$:

Computing $|G_{x,\leq p}|$:

Input:

- Integers n, p such that $p \mid n$.
- A string $x \in \{0,1\}^n$.

Algorithm:

1. Write x as $x = x_1x_2\ldots x_{n/p}$ where $|x_i| = p \quad \forall i \in [n/p]$.
2. Construct a branching program A_{x_1} such that $L(A_{x_i}) \cap \{0,1\}^p = G_{x_1,\leq p}$.
3. Let M be the number of strings of length p accepted by A_{x_1}.
4. If there is an $0 < i < n/p$ such that $x_1 = x_2 = x_3 = \cdots = x_i$, $x_1 < x_{i+1}$ in lexicographic order, and $x_1 \notin L(A_{x_1})$, then output $M + |\mathsf{Orbit}(x_1)|$, else output M.

From the construction in Section 2.2.3 and Lemma 2.16, it follows that we can construct A_{x_1} and count M in time polynomial in n. We thus have the following lemma.

Lemma 2.17. For any divisor p of n and string $x \in \{0,1\}^n$, we can compute the size of the set $G_{x,\leq p}$ in time $\mathsf{poly}(n)$.

We now have all the ingredients for the proof of the following theorem, which is a special case of Theorem 1.2 when the alphabet under consideration is $\{0,1\}$.

Theorem 2.18. *There is an algorithm for indexing necklaces of length n over the alphabet* $\{0,1\}$*, which runs in time* $\mathsf{poly}(n)$.

Proof. The proof simply follows by plugging together the conclusions of Lemma 2.5, Lemma 2.6, Lemma 2.7, Lemma 2.8 and Lemma 2.17. □

It is not difficult to see that the indexing algorithm can be used to obtain a reverse indexing algorithm as well and hence, we also obtain a special case of Theorem 1.3 for the binary alphabet.

2.2.5 Indexing necklaces over large alphabets

In this subsection we show how to handle the case of general alphabets Σ (with $|\Sigma| = q$). A direct generalization of the algorithm for the case of the binary alphabet, where the set L_x is appropriately defined, will run in time polynomial in n and q. Our goal here is to improve the running time to polynomial in n and $\log q$.

The basic idea is to represent the elements in Σ by binary strings of length $t \stackrel{\text{def}}{=} \lceil \log q \rceil$. Let $\mathsf{Bin} : \Sigma \to \{0,1\}^t$ be an injective map whose image is the set Γ of q lexicographically smallest strings in $\{0,1\}^t$. Extend this to a map $\mathsf{Bin} : \Sigma^n \to \{0,1\}^{tn}$ in the natural way.

We now use the map Bin to convert our indexing/counting problems over the large alphabet Σ to a related problem over the small alphabet $\{0,1\}$. For $x \in \Sigma^n$, we have $\mathsf{Bin}(\mathsf{Rot}^i(x)) = \mathsf{Rot}^{ti}(\mathsf{Bin}(x))$. For an orbit $E \subseteq \Sigma^n$ and $x \in \{0,1\}^{tn}$, we say $E < x$ if some element $z \in E$ satisfies $\mathsf{Bin}(z) <_{\mathsf{lex}} x$.

Let $\mathcal{C}_x$ be the set of orbits $E \subseteq \Sigma^n$ which are less than x. For each $x \in \{0,1\}^{tn}$ and $p \mid n$, define:

$$G_{x,p} = \bigcup_{E<x,\, |E|=p} E \qquad \text{and} \qquad G_{x,\leq p} = \bigcup_{E<x,\, |E| \text{ divides } p} E\,.$$

The following identity allows us to count $G_{x,\leq n}$:

$$|G_{x,\leq n}| = |\{y \in \{0,1\}^{tn} \mid y \in \Gamma^n, \exists i < n \text{ s.t. } \mathsf{Rot}^{it}(y) <_{\mathsf{lex}} x\}|\,.$$

It is easy to efficiently produce a branching program A_0 such that $L(A_0) \cap \{0,1\}^{tn} = \Gamma^n$. As we will describe below, the methods of the previous section can be easily adapted to efficiently produce a branching program A_x such that

$$L(A_x) \cap \{0,1\}^{tn} = \{y \in \{0,1\}^{tn} \mid \exists i < n \text{ s.t. } \mathsf{Rot}^{it}(y) <_{\mathsf{lex}} x\}\,.$$

The following lemma will be crucial in the design of this branching program.

Lemma 2.19. Let $y \in \{0,1\}^{tn}$. There exists $i < n$ such that $\mathsf{Rot}^{it}(y) <_{\mathsf{lex}} x$ if and only if at least one of the following events occurs:

1. there exists $w \in L_x$ such that w appears as a contiguous substring of y starting at a coordinate j with $j \equiv 0 \mod t$ (where the coordinates of x are $0, 1, \ldots, tn-1$),

2. there exist strings w_1, w_2 such that $w_1 w_2 \in L_x$, w_2 is a prefix of y, w_1 is a suffix of y, and $|w_1| \equiv 0 \mod t$.

Given this lemma, the construction of A_x follows easily via the techniques of the previous subsections. The main addition is that one needs to remember the value of the current coordinate mod t, which can be done by blowing up the number of states of the branching program by a factor t.

Intersecting the accepted sets of A_x and A_0 gives us our desired branching program which allows us to count $|G_{x,\leq n}|$. This easily adapts to also count $|G_{x,\leq p}|$ for each $p \mid n$.

We conclude using the ideas of Section 2.2.2. We can now compute $|G_{x,p}|$ for each x and each $p \mid n$. From Lemma 2.5, Lemma 2.6 and Lemma 2.7, it follows that for every x, we can compute $|\mathcal{C}_x|$ efficiently. We thus get our main indexing theorem for necklaces from Lemma 2.1.

Theorem 2.20. *There are* $\mathsf{poly}(n, \log|\Sigma|)$*-time indexing and reverse-indexing algorithms for necklaces of length n over Σ.*

Furthermore, there are $\mathsf{poly}(n, \log|\Sigma|)$*-time indexing and reverse-indexing algorithms for necklaces of length n over Σ with fundamental period exactly n.*

3 Indexing irreducible polynomials

In the previous section, we saw an algorithm for indexing necklaces of length n over an alphabet Σ of size q, which runs in time polynomial in n and $\log q$. In this section, we will see how to use this algorithm to efficiently index irreducible polynomials over a finite field. More precisely, we will use an indexing algorithm for necklaces with fundamental period exactly equal to n (which is also given by the methods of the previous sections).

Let q be a prime power, and let $\mathbb{F}_q$ denote the finite field of q elements. For an integer $n > 0$, let $I_{q,n}$ denote the set of monic, irreducible polynomials of degree n in $\mathbb{F}_q[T]$.

Theorem 3.1. *For every q, n as above, there is an algorithm that runs in* $\mathsf{poly}(n, \log q)$ *time, takes $O(n \log q)$ bits of advice, and indexes $I_{q,n}$.*

Proof. To prove this theorem, we start by first describing the connection between the tasks of indexing necklaces and indexing irreducible polynomials. Let $P(T) \in I_{q,n}$. Note that $P(T)$ has all its roots in the field $\mathbb{F}_{q^n}$. Let $\alpha \in \mathbb{F}_{q^n}$ be one of the roots of $P(T)$. Then we have that $\alpha, \alpha^q, \ldots, \alpha^{q^{n-1}}$ are all distinct, and:

$$P(T) = \prod_{i=0}^{n-1} \left(T - \alpha^{q^i}\right).$$

Conversely, if we take $\alpha \in \mathbb{F}_{q^n}$ such that $\alpha, \alpha^q, \dots, \alpha^{q^{n-1}}$ are all distinct, then the polynomial

$$P(T) = \prod_{i=0}^{n-1} \left(T - \alpha^{q^i}\right)$$

is in $I_{q,n}$.

Define an action of $\mathbb{Z}_n$ on $\mathbb{F}_{q^n}^*$ as follows: for $k \in \mathbb{Z}_n$ and $\alpha \in (\mathbb{F}_{q^n})^*$, define:

$$k[\alpha] = \alpha^{q^k}.$$

This action partitions $\mathbb{F}_{q^n}^*$ into orbits. By the above discussion, $I_{q,n}$ is in one-to-one correspondence with the orbits of this action with size exactly n. Thus it suffices to index these orbits.

Let g be a generator of the the multiplicative group $(\mathbb{F}_{q^n})^*$. Define a map $E : \mathbb{Z}_{q^n-1} \to \mathbb{F}_{q^n}^*$ by:

$$E(a) = g^a.$$

We have that E is a bijection. Via this bijection, we have an action of $\mathbb{Z}_n$ on $\mathbb{Z}_{q^n-1}$, where for $k \in \mathbb{Z}_n$ and $a \in \mathbb{Z}_{q^n-1}$,

$$k[a] = q^k \cdot a.$$

Now represent elements of $\mathbb{Z}_{q^n-1}$ by integers in $\{0, 1, \dots, q^n - 2\}$. Define $\Sigma = \{0, 1, \dots, q-1\}$. For $a \in \mathbb{Z}_{q^n-1}$, consider its base-q expansion $a_\sigma \in \Sigma^n$. This gives us a bijection between $\mathbb{Z}_{q^n-1}$ and $\Sigma^n \setminus \{(q-1, \dots, q-1)\}$. Via this bijection, we get an action of $\mathbb{Z}_n$ on $\Sigma^n \setminus \{(q-1, \dots, q-1)\}$. This action is precisely the standard rotation action!

This motivates the following algorithm.

The Indexing Algorithm:

Input: q (a prime power), $n \geq 0$, $i \in [|I_{q,n}|]$.

Advice: 1. A description of $\mathbb{F}_q$.

2. An irreducible polynomial $F(T) \in \mathbb{F}_q[T]$ of degree n, whose root is a generator g of $(\mathbb{F}_{q^n})^*$ (a. k. a. primitive polynomial).

1. Let $\Sigma = \{0, 1, \dots, q-1\}$.
2. Use i to index a necklace $\sigma \in \Sigma^n \setminus \{(q-1, q-1, \dots, q-1)\}$ with fundamental period exactly n (via Theorem 2.20).
3. View σ as the base q expansion of an integer $a \in \{0, 1, \dots, q^n - 2\}$.
4. Use $F(T)$ to construct the finite field $\mathbb{F}_{q^n}$ and the element $g \in \mathbb{F}_{q^n}^*$. (This can be done by setting $\mathbb{F}_{q^n} = \mathbb{F}_q[T]/F(T)$, and taking the class of the element T in that quotient to be the element g.)
5. Set $\alpha = g^a$.
6. Set $P(T) = \prod_{i=0}^{n-1}(T - \alpha^{q^i})$.
7. Output $P(T)$.

For constant q, this algorithm can be made to work with $O(\log n)$ bits of advice. Indeed, one can construct the finite field $\mathbb{F}_{q^n}$ in deterministic $\mathsf{poly}(q,n)$ time, and a wonderful result of Shparlinski [30] and Shoup [29] constructs a set of $n^{O(1)}$ elements in $\mathbb{F}_{q^n}$, one of which is guaranteed to be a generator (see also [23] for improved quantitative dependence on q). The advice is then the index of an element of this set which is a generator. □

4 Explicit generator matrices and parity check matrices for BCH codes

In this section, we will apply the indexing algorithm for necklaces to give a strongly explicit construction for generator and the parity check matrices for BCH codes. More precisely, we use the fact that our indexing algorithm is in fact an unranking algorithm for the lexicographic ordering on (lexicographically least representatives of) necklaces.

BCH codes [19] are classical algebraic error-correcting codes based on polynomials over finite extension fields. They have played a central role since the early days of coding theory due to their remarkable properties (they are one of the few known families of codes that has better rate/distance tradeoff than random codes in some regimes). Furthermore, their study motivated many advances in algebraic algorithms.

Using our indexing algorithm for necklaces, we can answer a basic question about BCH codes: we construct strongly explicit generator matrices and parity check matrices for BCH codes. For the traditionally used setting of parameters (constant designed distance), it is trivial to construct generator matrices and parity check matrices for BCH codes. But for large values of the designed distance, as far as we are aware, this problem was unsolved.

4.1 Generator matrices

Let q be a prime power, and let $n \geq 1$ and $0 \leq D < q^n - 1$. The BCH code associated with these parameters will be of length q^n over the field $\mathbb{F}_q$, where the q^n coordinates are identified with the big field $\mathbb{F}_{q^n}$. Let:

$$V = \{\langle P(\alpha)\rangle_{\alpha\in\mathbb{F}_{q^n}} \mid P(X) \in \mathbb{F}_{q^n}[X],\ \deg(P) \leq D,\ \text{s. t.}\ \forall\alpha \in \mathbb{F}_{q^n}, P(\alpha) \in \mathbb{F}_q\}\,.$$

In words: this is the $\mathbb{F}_q$-linear space of all $\mathbb{F}_{q^n}$-evaluations of $\mathbb{F}_{q^n}$-polynomials of low degree, which have the property that all their evaluations lie in $\mathbb{F}_q$. In coding theory terminology, this is a subfield subcode of Reed-Solomon codes.

The condition that $P(\alpha) \in \mathbb{F}_q$ for each $\alpha \in \mathbb{F}_{q^n}$ can be expressed as follows:

$$P(X)^q \equiv P(X) \mod X^{q^n} - X\,.$$

Thus, if $P(X) = \sum_{i=0}^{D} a_i X^i$, then the above condition is equivalent to:

$$\sum_{i=0}^{D} a_i^q X^{iq} \equiv \sum_{i=0}^{D} a_i X^i \mod X^{q^n} - X,$$

which simplifies to:

$$\forall i, a_{iq \bmod (q^n-1)} = a_i^q\,.$$

Thus:

1. for every i, if ℓ is the smallest integer such that $iq^\ell \bmod (q^n-1) = i$, then

$$a_i \in V_\ell = \{\alpha \in \mathbb{F}_{q^n} \mid \alpha^{q^\ell} = \alpha\} = \mathbb{F}_{q^\ell}\,,$$

2. specifying $a_i \in V_\ell$ automatically determines $a_{iq \bmod (q^n-1)}, a_{iq^2 \bmod (q^n-1)}, \ldots$,

3. a_i can take any value in V_ℓ.

This motivates the following choice of basis for BCH codes. Define an equivalence relation $\sim$ on $\{0,\ldots,q^n-1\}$ as follows: $i_1 \sim i_2$ iff $i_2 = i_1 \cdot q^k \mod (q^n-1)$ for some k. Let $\mathcal{G}$ denote the set of all equivalence classes. Let $\mathcal{F} = \{S \cap \{0,1,\ldots,D\} \mid S \in \mathcal{G}\}$. Thus $\mathcal{F}$ is a partition of $\{0,1,\ldots,D\}$. Let $\alpha_{S,1},\ldots,\alpha_{S,|S|}$ be a basis for $V_{|S|}$ over $\mathbb{F}_q$ (note that when $\ell \mid n$, we have that

$$V_\ell = \{\alpha \in \mathbb{F}_{q^n} \mid \alpha^{q^\ell} = \alpha\} = \mathbb{F}_{q^\ell}$$

is an $\mathbb{F}_q$-linear subspace of $\mathbb{F}_{q^n}$ of dimension ℓ). For $S \in \mathcal{F}$, define $m_S = \min_{i\in S} i$. For $S \in \mathcal{F}$ and $j \in [|S|]$, define:

$$P_{S,j}(X) = \sum_{k=0}^{|S|-1} \alpha_j^{q^k} X^{m_S q^k \bmod (q^n-1)}\,.$$

It is easy to see from the above description that $(P_{S,j})_{S\in\mathcal{F},j\in[n]}$ forms an $\mathbb{F}_q$ basis for the BCH code V. Thus it remains to show that one can index the sets of $\mathcal{F}$.

If we write all the elements of $S \in \mathcal{F}$ in base q, we soon realize that the S are precisely in one-to-one correspondence with those rotation orbits of Σ^n (with $\Sigma = \{0,1,\ldots,q-1\}$) where all elements of the orbit are lexicographically $\le$ some fixed string in Σ^n (in this case the fixed string turns out to be the base q representation of the integer D). By the results of Section 2 (with a small change replacing "lexicographically $\ge$" by "lexicographically $\le$"), $\mathcal{F}$ can be indexed efficiently. Thus we can compute any given entry of a generator matrix for BCH codes.

4.2 Parity-check matrices

For constructing parity-check matrices of BCH codes, we use an alternate characterization of BCH codes (which is classical [19]). We begin by giving this characterization.

Let d be a given integer, the "designed distance." For $\alpha \in \mathbb{F}_{q^n}$, let v_α denote the vector

$$(1,\alpha,\alpha^2,\ldots,\alpha^{d-1}) \in \mathbb{F}_{q^n}^d\,.$$

Note that for any d distinct α, the corresponding vectors v_α are linearly independent over $\mathbb{F}_{q^n}$ (and hence over $\mathbb{F}_q$); this follows from the observation that these vectors form a Vandermonde matrix.

The BCH code with designed distance d is the set of $\mathbb{F}_q$-valued functions:

$$\left\{ f : \mathbb{F}_{q^n} \to \mathbb{F}_q \text{ s.t.} \sum_{\alpha\in\mathbb{F}_{q^n}} f(\alpha) v_\alpha = 0 \right\}.$$

(This is viewed as a code in the usual way: the coordinates of the code are indexed by $\mathbb{F}_{q^n}$). We remark that the set of $\mathbb{F}_{q^n}$-valued functions with this property is simply the Reed-Solomon code.

Our goal is to give a strongly-explicit construction for a parity-check matrix for the BCH code: i. e., a full-row-rank matrix with $\mathbb{F}_q$ entries whose null-space equals the BCH code. We first give a classical description of such a matrix; we will then proceed to give an algorithm that computes its entries efficiently.

Define an equivalence relation $\sim$ on $\{0,\ldots,q^n-1\}$ as follows: $i_1 \sim i_2$ iff $i_2 = i_1 \cdot q^k \mod (q^n-1)$ for some k. Let $\mathcal{G}$ denote the set of all equivalence classes. We have the following simple observation: for $\alpha \in \mathbb{F}_q$, $E \in \mathcal{G}$, and $e \in E$:

$$(\alpha^e)^{q^{|E|}} = \alpha^e\,,$$

and so:

$$\alpha^e \in \mathbb{F}_{q^{|E|}}\,.$$

For each equivalence class $E \subseteq \{1,\ldots,q^n-1\}$, let m_E denote the smallest element of E. Let

$$\mathcal{H} = \{E \mid m_E < d\}$$

be the set of all equivalence classes whose smallest element is at most $d-1$. Now for $\alpha \in \mathbb{F}_{q^n}$, define the vector

$$u_\alpha \in \prod_{E\in\mathcal{H}} \mathbb{F}_{q^{|E|}}$$

by

$$u_\alpha = (\alpha^{m_E})_{E\in\mathcal{H}}\,.$$

The remarkable dimension-distance tradeoff of BCH codes is based on the following simple, but extremely important, fact: if $f : \mathbb{F}_{q^n} \to \mathbb{F}_q$, then

$$\sum_{\alpha\in\mathbb{F}_{q^n}} f(\alpha)v_\alpha = 0$$

if and only if

$$\sum_{\alpha\in\mathbb{F}_{q^n}} f(\alpha)u_\alpha = 0\,.$$

The forward direction of this implication is trivial, since the vector u_α is obtained from v_α by deleting certain coordinates. The reverse direction is based on the fact that

$$\left(\sum_\alpha f(\alpha)\alpha^e\right)^q = \sum_\alpha f(\alpha)\alpha^{eq}\,.$$

In particular, we have that every d distinct u_α are linearly independent over $\mathbb{F}_q$.

For each $j \geq 1$, let $\psi_j : \mathbb{F}_{q^j} \to \mathbb{F}_q^j$ be an $\mathbb{F}_q$-linear isomorphism. We will use the maps ψ_j to convert the u_α into vectors with $\mathbb{F}_q$ entries. Define $r = \sum_{E\in\mathcal{H}} |E|$. Define $\tilde{u}_\alpha \in \mathbb{F}_q^r$ to be the vector:

$$\tilde{u}_\alpha = \left(\psi_{|E|}(\alpha^{m_E})\right)_{E\in\mathcal{H}}\,.$$

By construction, the vectors $\tilde{u}_\alpha$ satisfy the same $\mathbb{F}_q$-dependencies as the vectors u_α.

Let M be the $r \times q^n$ matrix whose columns are $\tilde{u}_\alpha$ for $\alpha \in \mathbb{F}_{q^n}$. Then M is a parity check matrix of the BCH code. The fact that the nullspace of M equals the BCH code follows from the discussion above. The fact that the row rank of M is full follows from the fact that for any $(c_E)_{E\in\mathcal{G}}$ with $c_E \in \mathbb{F}_{q^{|E|}}$, the function $g : \mathbb{F}_{q^n} \to \mathbb{F}_q$ given by:

$$g(x) = \sum_{E\in\mathcal{G}} \mathsf{Tr}_{\mathbb{F}_{q^{|E|}}/\mathbb{F}_q}(c_E x^{m_E}),$$

is not identically 0, provided the c_E are not all 0. This is proved by expanding the right hand side as a polynomial $\bmod (x^{q^n} - x)$ and observing that there is a monomial with a nonzero coefficient. We omit the (well-known) details.

We now use the components of our indexing algorithm for necklaces to give a strongly explicit construction for M (after choosing a convenient ordering of the rows). The key observation is that when the elements of $\{0,1,\ldots,q^n-1\}$ are written in base q, the equivalence classes under the relation $\sim$ are precisely the necklaces of length n over the alphabet $\{0,1,\ldots,q-1\}$. We will now also refer to the E as necklaces.

For an integer j dividing n, let $\mathcal{H}_j = \{E \in \mathcal{H} \mid |E| = j\}$. The necklaces in $\mathcal{H}_j$ are precisely those necklaces with fundamental period j which contain some string which is lexicographically at most the base-q representation of $d-1$. By the results of Section 2, $|\mathcal{H}_j|$ can be efficiently computed, and the necklaces in $\mathcal{H}_j$ can be efficiently indexed by a map $h_j : [|\mathcal{H}_j|] \to \mathcal{H}_j$.

To efficiently compute the entries of M, we first re-index the rows. The rows are indexed by tuples (j,i,s), where $j \mid n$, $i \in \{1,2,\ldots,|\mathcal{H}_j|\}$ and $s \in \{1,2,\ldots,j\}$: we may assume this form of indexing of the rows for our strongly explicit construction since one can efficiently compute a bijection between $s \in \{1,2,\ldots,r\}$, and the collection of all such tuples (this uses the fact that $|\mathcal{H}_j|$ can be computed). The columns are indexed by $\alpha \in \mathbb{F}_{q^n}$. The entry of M corresponding to row (j,i,s) and column α is then:

$$(\psi_j(\alpha^{m_{h_j(i)}}))_s .$$

Note that this is efficiently computable. In words, one first uses i to index a necklace E in $\mathcal{H}_j$; next one computes m_E and uses this to compute α^{m_E}; then one applies ψ_j to this to get a vector in $\mathbb{F}_q^j$; and finally one takes the sth coordinate of this vector.

This completes the description of the strongly-explicit construction of a parity check matrix for BCH codes.

5 Open problems

We conclude with some open problems.

1. Can the orbits of group actions be indexed in general?

   One formulation of this problem is as follows: Let G be a finite group acting on a set X, both of size $\mathsf{poly}(n)$. Suppose G and its action on X are given as input explicitly. For a finite alphabet Σ, consider the action of G on Σ^X (by permuting coordinates according to the action on X). Can the orbits of this action be indexed? Can they be reverse-indexed?

2. Let G be the symmetric group S_n. Consider its action on $\{0,1\}^{\binom{[n]}{2}}$, where G acts by permuting coordinates. The orbits of this action correspond to the isomorphism classes of n-vertex graphs. Can these orbits be indexed?

More ambitiously, can these orbits be reverse-indexed? This would imply that graph isomorphism is in P.

3. It would be interesting to explore the complexity theory of indexing and reverse-indexing. Which languages can be indexed efficiently? Can this be characterized in terms of known complexity classes? We include some simple observations about these problems in Appendix B.

A Alternative indexing algorithm for binary necklaces of prime length

In this section we give another algorithm for indexing necklaces in $\{0,1\}^n$ in the special case where n is prime.

For convenience, we will denote the n coordinates of $\{0,1\}^n$ by $0,1,\ldots,n-1$, and identify them with elements of $\mathbb{Z}_n$. For a binary string x, $\mathsf{wt}(x)$ denotes the Hamming weight of the string x.

Definition A.1. Let $x \in \{0,1\}^n$. We say x is top-heavy if for every j, $0 \le j < n$:

$$\sum_{k=0}^{j}\left(x_k - \frac{\mathsf{wt}(x)}{n}\right) \ge 0\,.$$

In words: every prefix of x has normalized Hamming weight at least as large as the normalized Hamming weight of x.

The next lemma by Dvoretzky and Motzkin [9] shows that every aperiodic string of prime length has a unique top-heavy rotation.

Lemma A.2 ([9])**.** Let n be prime. For each $x \in \{0,1\}^n \setminus \{0^n, 1^n\}$, there exists a unique i, $0 \le i < n$ such that $\mathsf{Rot}^i(x)$ is top-heavy.

Proof. Define $f : \{0,1\}^n \times \mathbb{N} \to \mathbb{R}$ by:

$$f(x,j) = \sum_{k=0}^{j}\left(x_{k \bmod n} - \frac{\mathsf{wt}(x)}{n}\right).$$

Then the top-heaviness of x is equivalent to $f(x,j) \ge 0$ for all $j \in \mathbb{N}$.

We make two observations.

1. If $j = j' \mod n$, then $f(x,j) = f(x,j')$. This follows from the fact that:

$$\sum_{k=0}^{n-1}\left(x_k - \frac{\mathsf{wt}(x)}{n}\right) = 0\,.$$

2. For nonnegative integers j,ℓ with $j < n$, we have:

$$f(\mathsf{Rot}^j(x),\ell) = f(x,j+\ell) - f(x,j)\,.$$

Putting these two facts together, we get that:

$$f(\mathsf{Rot}^j(x),\ell) = f(x,(j+\ell) \bmod n) - f(x,j)\,. \tag{A.1}$$

Now fix $x \in \{0,1\}^n \setminus \{0^n,1^n\}$. Define $i \in \{0,1,\dots,n-1\}$ to be such that $f(x,i)$ is minimized. By Equation (A.1), we get that $f(\mathsf{Rot}^i(x),\ell) \geq 0$ for all nonnegative integers ℓ. This proves the existence of i.

For uniqueness of i, we make two more observations:

1. If $f(x,j) > f(x,i)$, then

$$f(\mathsf{Rot}^j(x),n+i-j) = f(x,n+i) - f(x,j) = f(x,i) - f(x,j) < 0\,,$$

   and thus $\mathsf{Rot}^j(x)$ is not top-heavy.

2. If $f(x,j) = f(x,j')$, then $j = j' \mod n$. To see this, first note that we may assume $j < j'$. Then:

$$\begin{aligned} 0 &= f(x,j') - f(x,j) \\ &= \sum_{k=j+1}^{j'} \left(x_{k \bmod n} - \frac{\mathsf{wt}(x)}{n} \right) \\ &= \left(\sum_{k=j+1}^{j'} x_{k \bmod n} \right) - (j'-j) \cdot \frac{\mathsf{wt}(x)}{n}\,. \end{aligned}$$

   Thus, since the first term is an integer, we must have that $(j'-j)\cdot\mathsf{wt}(x)$ must be divisible by n, and by our hypothesis on x, we have that $j' = j \mod n$.

Thus $i \in \{0,1,\dots,n-1\}$, for which $\mathsf{Rot}^i(x)$ is top-heavy, is unique. □

The above lemma implies that each orbit E contains a unique top-heavy string. We define the canonical element of E to be that element.

We now show that there is a branching program A such that $L(A) \cap \{0,1\}^n$ precisely equals the set of top-heavy strings. By the discussion in Section 2.1, this immediately gives an indexing algorithm for orbits of E.

How does a branching program verify top-heaviness? In parallel, for each $\ell \in \{1,\dots,n-1\}$, the branching program checks if condition C_ℓ holds, where C_ℓ is

$$\text{“}(\forall 0 \leq j < n) \left(\sum_{k=0}^{j} x_k \geq \frac{k\cdot\ell}{n} \right)\text{.”}$$

At the same time, it also computes the weight of x. At the final state, it checks if $C_{\mathsf{wt}(x)}$ is true. The string x is top-heavy if and only if $C_{\mathsf{wt}(x)}$ is true.

This completes the description of the indexing algorithm.

We also know an extension of this approach that can handle n which have $O(1)$ prime factors. The key additional ingredient of this extension is a new encoding of strings that enables verification of properties like top-heaviness by automata.

B Complexity of indexing

In this section, we explore some basic questions about the complexity theory of indexing and reverse indexing. We would like to understand what sets can be indexed/reverse-indexed efficiently.

The outline of this section is as follows. We first deal with indexing and reverse-indexing in a nonuniform setting. Based on some simple observations about what cannot be indexed/reverse-indexed, we make some naive, optimistic conjectures characterizing what is efficiently indexable/reverse-indexable, and then proceed to disprove these conjectures. We then make some natural definitions for indexing and reverse-indexing in a uniform setting, and conclude with some analogous naive, optimistic conjectures.

B.1 Indexing and reverse-indexing in the nonuniform setting

By simple counting, most sets $S \subseteq \{0,1\}^n$ cannot be indexed or reverse-indexed by circuits of size $\mathsf{poly}(n)$. We now make two naive and optimistic conjectures.

- For every $c > 0$, there exists $d > 0$, such that for all sufficiently large n, if $S \subseteq \{0,1\}^n$ has a n^c-size circuit recognizing it, then there is a n^d-size circuit for indexing S.
- For every $c > 0$, there exists $d > 0$, such that for all sufficiently large n, if $S \subseteq \{0,1\}^n$ has a n^c-size circuit recognizing it, then there is a n^d-size circuit for reverse-indexing S.

Note that the simple observations about indexing made in the Introduction are consistent with these conjectures.

We now show that these conjectures are false (unless the polynomial hierarchy collapses). Assuming the conjectures, we will give Σ_4 algorithms to count the number of satisfying assignments of a given Boolean formula ϕ of size n^c on n variables. By Toda's theorem [32], this would imply that the polynomial hierarchy collapses.

Suppose we are given a Boolean formula ϕ of size at most n^c on n variables (with n sufficiently large). Let $S \subseteq \{0,1\}^n$ be the set of satisfying assignments of ϕ. We know that S can be recognized by a circuit of size n^c (namely ϕ). By the conjectures, there are circuits C_i and C_r of size n^d for indexing S and reverse-indexing S. We will now see that a Σ_4 algorithm can get its hands on these circuits, and then use these circuits to count the number of elements in S.

Indexing. Consider the Σ_4 algorithm that does the following on input ϕ. Guess a circuit $C : \{0,1\}^n \to \{0,1\}^n \cup \{$"**too large**"$\}$ of size n^d, and an integer $K < 2^n$ and then verify the following properties:

- for all $i \in [K]$, $C(i) \neq$ **too large** and $\phi(C(i)) = 1$,
- for all $i \notin [K]$, $C(i) =$ **too large**,
- for all $x \in \{0,1\}^n$, if $\phi(x) = 1$, then there exists a unique $i \in [K]$ for which $C(i) = x$.

If $C = C_i$, and $K = |S|$, then these properties hold. It is also easy to see that if all these properties hold, then C is an indexing circuit for S, and $K = |S|$. Thus the above gives a Σ_4 algorithm to compute $|S|$.

Reverse-indexing. Consider the Σ_4 algorithm that does the following on input ϕ. Guess a circuit $C:\{0,1\}^n \to \{0,1\}^n \cup \{\text{“}\mathbf{false}\text{”}\}$ of size n^d, and an integer $K < 2^n$ and then verify the following properties:

- for all $x \in \{0,1\}^n$, either ($\phi(x) = 1$ and $C(x) \in [K]$) or ($\phi(x) = 0$ and $C(x) = \mathbf{false}$),
- for all $i \in [K]$, there exists a unique $x \in \{0,1\}^n$ such that $C(x) = i$.

If $C = C_r$, and $K = |S|$, then these properties hold. It is also easy to see that if all these properties hold, then C is a reverse-indexing circuit for S, and $K = |S|$. Thus the above gives a Σ_4 algorithm to compute $|S|$.

B.2 Indexing and reverse-indexing in the uniform setting

We now introduce a natural framework for talking about indexing in the uniform setting.

Let $L \subseteq \Sigma^* \times \Sigma^*$ be a pair-language. For $x \in \Sigma^*$, define $L_x = \{y \mid (x,y) \in L\}$. An algorithm $M(x,i)$ is said to be an indexing algorithm for L if for every $x \in \Sigma^*$, the function $M(x,\cdot)$ is an indexing of the set L_x. An algorithm $M(x,y)$ is said to be a reverse indexing algorithm for L if for every $x \in \Sigma^*$, the function $M(x,\cdot)$ is a reverse indexing of the set L_x. Indexing/reverse-indexing algorithms are said to be efficient if they run in time $\mathsf{poly}(|x|)$.

We now make some preliminary observations about the limitations of efficient indexing/reverse-indexing.

1. If L can be efficiently indexed, then the counting problem for L can be solved efficiently. (Recall that the counting problem for L is the problem of determining $|L_x|$ when given x as input. The counting problem can be solved via binary search using an indexing algorithm.)
2. If L can be efficiently reverse indexed, then L must be in P. Indeed, the reverse indexing algorithm $M(x,y)$ immediately tells us whether $(x,y) \in L$.

In the absence of any other easy observations, we gleefully made the following optimistic conjectures.

1. Every pair-language $L \in P$ for which the counting problem can be solved efficiently can be efficiently indexed.
2. Every pair-language $L \in P$ can be efficiently reverse indexed.

Using ideas similar to those used in the nonuniform case, one can show that the latter of these conjectures is not true (unless the polynomial hierarchy collapses). However we have been unable to say anything interesting about the first conjecture, and we leave it as an open problem.

Acknowledgements

We would like to thank Joe Sawada for making us aware of the work of Kociumaka et al. [15], and Igor Shparlinski for helpful suggestions and references. Thanks to the anonymous referees for many valuable comments.

References

[1] LEONARD M. ADLEMAN AND HENDRIK W. LENSTRA, JR.: Finding irreducible polynomials over finite fields. In *Proc. 18th STOC*, pp. 350–355. ACM Press, 1986. [doi:10.1145/12130.12166] 4

[2] MANINDRA AGRAWAL AND SOMENATH BISWAS: Primality and identity testing via Chinese remaindering. *J. ACM*, 50(4):429–443, 2003. Preliminary version in FOCS'99. [doi:10.1145/792538.792540] 4

[3] NOGA ALON, ODED GOLDREICH, JOHAN HÅSTAD, AND RENÉ PERALTA: Simple construction of almost k-wise independent random variables. *Random Structures and Algorithms*, 3(3):289–304, 1992. Preliminary version in FOCS'90. [doi:10.1002/rsa.3240030308] 2, 4

[4] ALEXANDR ANDONI, ASSAF GOLDBERGER, ANDREW MCGREGOR, AND ELY PORAT: Homomorphic fingerprints under misalignments: sketching edit and shift distances. In *Proc. 45th STOC*, pp. 931–940. ACM Press, 2013. [doi:10.1145/2488608.2488726] 5

[5] JÖRG ARNDT: *Matters Computational*. Springer, 2011. [doi:10.1007/978-3-642-14764-7] 3, 5

[6] JEAN BERSTEL AND MICHEL POCCHIOLA: Average cost of Duval's algorithm for generating Lyndon words. *Theoret. Comput. Sci.*, 132(1-2):415–425, 1994. [doi:10.1016/0304-3975(94)00013-1] 5

[7] KEVIN CATTELL, FRANK RUSKEY, JOE SAWADA, MICAELA SERRA, AND C. ROBERT MIERS: Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over GF(2). *J. Algorithms*, 37(2):267–282, 2000. [doi:10.1006/jagm.2000.1108] 5

[8] JEAN-PIERRE DUVAL: Génération d'une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornée. *Theoret. Comput. Sci.*, 60(3):255–283, 1988. [doi:10.1016/0304-3975(88)90113-2] 5

[9] ARYEH DVORETZKY AND THEODORE S. MOTZKIN: A problem of arrangements. *Duke Math. J.*, 14(2):305–313, 1947. [doi:10.1215/S0012-7094-47-01423-3] 20

[10] HAROLD FREDRICKSEN AND IRVING J. KESSLER: An algorithm for generating necklaces of beads in two colors. *Discrete Math.*, 61(2-3):181–188, 1986. [doi:10.1016/0012-365X(86)90089-0] 5

[11] HAROLD FREDRICKSEN AND JAMES A. MAIORANA: Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Math.*, 23(3):207–210, 1978. [doi:10.1016/0012-365X(78)90002-X] 5

[12] SOLOMON W. GOLOMB: Irreducible polynomials, synchronization codes, primitive necklaces and the cyclotomic algebra. In *Combinatorial Math. and Its Appl.*, pp. 358–370. Univ. N. Carolina Press, Chapel Hill, NC., 1969. 4

[13] VENKATESAN GURUSWAMI AND SWASTIK KOPPARTY: Explicit subspace designs. In *Proc. 54th FOCS*, pp. 608–617. IEEE Comp. Soc. Press, 2013. Preliminary version in ECCC. 4

[14] DONALD E. KNUTH: Generating all trees, history of combinatorial generation. In *The Art of Computer Programming*, volume 4. Addison-Wesley Professional, 2006. ACM DL. 5

[15] TOMASZ KOCIUMAKA, JAKUB RADOSZEWSKI, AND WOJCIECH RYTTER: Computing k-th Lyndon word and decoding lexicographically minimal de Bruijn sequence. In *Combinat. Pattern Matching*, volume 8486 of *LNCS*, pp. 202–211. Springer, 2014. [doi:10.1007/978-3-319-07566-2_21, arXiv:1510.02637] 5, 23

[16] SWASTIK KOPPARTY, MRINAL KUMAR, AND MICHAEL E. SAKS: Efficient indexing of necklaces and irreducible polynomials over finite fields. In *Proc. 41st Internat. Colloq. on Automata, Languages and Programming (ICALP'14)*, volume 8572, pp. 726–737. Springer, 2014. [doi:10.1007/978-3-662-43948-7_60, arXiv:1504.00572] 1, 5

[17] DONALD L. KREHER AND DOUGLAS ROBERT STINSON: *Combinatorial Algorithms: Generation, Enumeration, and Search*. Volume 7 of *Discrete Math. and its Appl.* CRC Press, 1999. 3, 5

[18] HENDRIK W. LENSTRA, JR.: Finding isomorphisms between finite fields. *Math. Comput.*, 56(193):329–347, 1991. [doi:10.2307/2008545] 4

[19] F. JESSICA MACWILLIAMS AND NEIL J. A. SLOANE: *The Theory of Error-Correcting Codes*. North-Holland Publ. Co., 2nd edition, 1978. 16, 17

[20] CONRADO MARTÍNEZ AND XAVIER MOLINERO: An efficient generic algorithm for the generation of unlabelled cycles. In *Math. and Comput. Sci. III*, Trends in Math., pp. 187–197. Springer, 2004. [doi:10.1007/978-3-0348-7915-6_19] 5

[21] WENDY J. MYRVOLD AND FRANK RUSKEY: Ranking and unranking permutations in linear time. *Inf. Process. Lett.*, 79(6):281–284, 2001. [doi:10.1016/S0020-0190(01)00141-7] 3

[22] ALBERT NIJENHUIS AND HERBERT S. WILF: *Combinatorial Algorithms for Computers and Calculators*. Volume 1. Academic Press, 1978. 3, 5

[23] GRIGORIĬ IOSIFOVICH PEREL'MUTER AND IGOR EVGEN'EVICH SHPARLINSKI: On the distribution of primitive roots in finite fields. *Russ. Math. Surv. (Uspekhi Mat. Nauk)*, 45(1):223–224 (original pp. 185–186), 1990. Russian original freely downloadable from Math-Net.Ru. [doi:10.1070/RM1990v045n01ABEH002330] 16

[24] MICHAEL O. RABIN: *Fingerprinting by Random Polynomials*. Center for Research in Comput. Techn., Aiken Comput. Lab., Harvard Univ., 1981. 24 pp. See also at Harvard CS Tech Report version TR 15-81 (12 pp). 4

[25] FRANK RUSKEY: Combinatorial Generation, 2003. Draft available at `http://www.1stworks.com/ref/RuskeyCombGen.pdf`. 3, 5

[26] FRANK RUSKEY, CARLA D. SAVAGE, AND TERRY MIN YIH WANG: Generating necklaces. *J. Algorithms*, 13(3):414–430, 1992. [doi:10.1016/0196-6774(92)90047-G] 5

[27] FRANK RUSKEY AND JOE SAWADA: An efficient algorithm for generating necklaces with fixed density. *SIAM J. Comput.*, 29(2):671–684, 1999. Preliminary version in SODA'99. [doi:10.1137/S0097539798344112] 5

[28] VICTOR SHOUP: New algorithms for finding irreducible polynomials over finite fields. *Math. Comput.*, 54(189):435–447, 1990. Preliminary version in FOCS'88. [doi:10.1090/S0025-5718-1990-0993933-0] 4

[29] VICTOR SHOUP: Searching for primitive roots in finite fields. *Math. Comput.*, 58(197):369–380, 1992. Preliminary version in STOC'90. [doi:10.1090/S0025-5718-1992-1106981-9] 4, 16

[30] IGOR EVGEN'EVICH SHPARLINSKI: On primitive elements in finite fields and on elliptic curves. *Math. USSR Sb.*, 71(1):41–50, 1992. [doi:10.1070/SM1992v071n01ABEH001389] 16

[31] RICHARD P. STANLEY: *Enumerative Combinatorics: Volume 1*. Cambridge Univ. Press, 2011. 8

[32] SEINOSUKE TODA: PP is as hard as the polynomial-time hierarchy. *SIAM J. Comput.*, 20(5):865–877, 1991. Preliminary version in FOCS'89. [doi:10.1137/0220053] 22

[33] LESLIE G. VALIANT: The complexity of computing the permanent. *Theoret. Comput. Sci.*, 8(2):189–201, 1979. [doi:10.1016/0304-3975(79)90044-6] 3

AUTHORS

Swastik Kopparty
Rutgers University, New Brunswick, NJ
swastik.kopparty@rutgers.edu
http://www.math.rutgers.edu/~sk1233/

Mrinal Kumar
Rutgers University, New Brunswick, NJ
mrinal.kumar@rutgers.edu
https://dragon.rutgers.edu/~mk1029/

Michael Saks
Rutgers University, New Brunswick, NJ
saks@math.rutgers.edu
http://www.math.rutgers.edu/~saks/

ABOUT THE AUTHORS

SWASTIK KOPPARTY received his Ph. D. in Computer Science from M.I.T. in 2010, and was advised by Madhu Sudan. During 2010-2011, he was a postdoc at the Institute for Advanced Study, and he has been at Rutgers University since then. He is interested in complexity theory, error-correcting codes, finite fields, randomness and pseudorandomness.

MRINAL KUMAR is a Ph. D. student in the Department of Computer Science at Rutgers University, where he is advised by Swastik Kopparty and Shubhangi Saraf. His research interests are in computational complexity, algebraic complexity and error-correcting codes.

MICHAEL SAKS received his Ph. D. in Mathematics from M.I.T. in 1980, where he was advised by Daniel Kleitman. He was a postdoctoral fellow at UCLA, and held positions at Bell Communications Research and the Computer Science and Engineering Department at UCSD.

He has worked in a variety of areas in the theory of computing and discrete mathematics: lower bounds for data structures, circuits, communication complexity, branching programs and decision trees, streaming algorithms, sublinear algorithms, satisfiability algorithms, online algorithms, distributed computing, and derandomization, and extremal problems for graphs, hypergraphs and partially ordered sets.