Space Efficient Data Structures and Algorithms in the Word-RAM Model

by

Hicham El-Zein

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Gerth Stølting Brodal
Professor,
Department of Computer Science, Aarhus University

Supervisor(s): J. Ian Munro
University Professor,
Chertion School of Computer Science, University of Waterloo

Internal Member: Anna Lubiw
Professor,
Chertion School of Computer Science, University of Waterloo
Eric Blais
Associate Professor,
Chertion School of Computer Science, University of Waterloo

Internal-External Member: Gordon B. Agnew
Associate Professor,
School of Engineering, University of Waterloo

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis we study space-efficient data structures for various combinatorial objects. We focus on succinct and compact data structures. Succinct data structures are data structures whose size is within the information theoretic lower bound plus a lower order term, whereas compact data structures are data structures whose size is a constant factor from the information theoretic lower bound.

We start by discussing the compact representation of unlabeled permutations, where the goal is to store a permutation π if we are permitted to reassign the labels of the elements, while supporting the following query: given k and i compute $\pi^k(i)$ quickly.

We study this problem in several scenarios. In the first scenario the queries are answered by just examining the labels of the queried elements. In the second scenario we assign labels to the n elements from the label set $\{1, \ldots, cn\}$ where $c \geq 1$ is a constant. In the third scenario we assign labels to the n elements from the label set $\{1, \ldots, cn^{1+\varepsilon}\}$ where c is a constant and $0 < \varepsilon < 1$. We give tight upper and lower bounds in all the three scenarios and we are able to answer queries in constant time independent of k. Finally, as an application we show how to improve the representation of general (labeled) permutations using our results.

We then deviate from the general scheme of designing space-efficient data structures to designing space-efficient algorithms. We cover the problem of powering permutations in place. Given a permutation of n elements, stored as an array, we address the problem of replacing the permutation by its k^{th} power while using $o(n)$ bits of extra storage. To this end, we first present an algorithm for inverting permutations that uses $O(\lg^2 n)$ additional bits and runs in $O(n \lg n)$ worst case time. This result is then generalized to the situation in which the permutation is to be replaced by its k^{th} power. We present an algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\varepsilon}\})$ additional bits.

Next, we cover data structures for range reporting. In range reporting problems a set $\mathcal{S}$ of n points in $\mathbb{R}^d$ is preprocessed such that the following query can be efficiently computed. Given an axis-aligned box $\mathcal{Q}$ return an aggregate function over $\mathcal{S} \cap \mathcal{Q}$. Range reporting problems have fundamental importance in computational geometry, and are interesting to study both for their optimality with respect to space and query time, and as tools employed to provide efficient solutions to various geometric problems.

We start by presenting a data structure that answers multi-dimensional range mode queries that improves a result by Chan et al. Then we present succinct data structures for one dimensional approximate color counting, one dimensional approximate median reporting, and one dimensional color reporting.

Our data structure for one dimensional approximate color counting answers queries in constant time, thus improving a result by Saladi, and our data structure for approximate median reporting in the special case when the points are in the rank space uses only $O(n)$ bits, thus improving a result by Bose et al. Moreover, we show, somewhat counter-intuitively, that it is not necessary to store colors of the points in order to answer approximate color counting queries, nor the value of the points in order to answer approximate median reporting queries.

Finally, we present a dynamic data structure with restricted updates for one dimensional color reporting in the case when the points are in the rank space, and we present a fully dynamic succinct data structure for one dimensional range reporting.

Acknowledgements

I offer my sincere gratitude to my supervisor J. Ian Munro who has supported me throughout my research with his patience and knowledge while allowing me to work on topics that I like. I also wish to thank my coauthors Yakov Nekrich, Venkatesh Raman, and Sharma V. Thankachan for the insightful discussions that we had. I would like to thank my readers Gerth Stølting Brodal, Anna Lubiw, Eric Blais, and Gordon B. Agnew. I would like to thank Wendy Rush and Helen Jardine for offering their help whenever I needed it. Finally, I would like to thank my friends and all the fellow students that helped in creating for me a supporting environment.

Dedication

To my parents

Table of Contents

List of Figures

Chapter 1

Introduction and Motivation

1.1 Motivation

In modern computation the volume of data-sets has increased dramatically. Since the majority of these data-sets are stored in internal memory, reducing their storage requirement is an important research topic. One way of reducing storage is using succinct and compact data structures which maintain the data in compressed form with extra data structures over it in a way that allows efficient access and query of the data.

Succinct and compact data structures are data structures that emphasize space efficiency. The goal is to occupy as little space as possible while maintaining an efficient query time. More precisely, succinct data structures are data structures whose size is within the information theoretic lower bound plus a lower order term, while compact data structures are data structures whose size is constant factor from the information theoretic lower bound.

In general, the goal is to dramatically reduce the storage cost for structural information. For example, a suffix tree is a data structure that permits us to find all occurrences of a query string in time linear to the query. This is crucial in many applications including queries about the human genome. In that particular case, the suffix tree, naively implemented, takes about 80 times as much space as the raw data [78]. Succinct methods reduces this to a factor of about 2. Moreover, this enables data structures to be stored in a faster(smaller) level of memory and so permits queries to be answered much more quickly.

In what follows, we give a high level description of this thesis contributions.

1.2 Thesis Outline and Contribution

The main theme of this thesis is designing succinct and compact data structures for various combinatorial objects, though we shift slightly from that theme to designing in-place or space-efficient algorithms in Chapter 4.

In Chapter 3 we discuss the compact representation of unlabeled permutations (i.e. permutations where the labels of the elements can be reassigned). Given an arbitrary unlabeled permutation π, we store it compactly such that $\pi^k(i)$ can be computed quickly for any i and any integer power k. We consider the problem in several scenarios.

In the first scenario we assign labels to elements so that queries are answered by just examining the labels of the queried elements. We show that a label space of $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \cdot i$ is necessary and sufficient. In other words, $2 \lg n$ bits of space are necessary and sufficient for representing each of the labels.[1] In the second scenario we assign labels to the n elements from the label set $\{1, \ldots, cn\}$ where $c \geq 1$ is a constant. We show that $\Theta(\sqrt{n})$ bits are necessary and sufficient to represent the permutation. Moreover, we support queries in such a structure in $O(1)$ time. Finally, in the third scenario we assign labels to the n elements from the label set $\{1, \ldots, cn^{1+\varepsilon}\}$ where c is a constant and $0 < \varepsilon < 1$. We show that $\Theta(n^{(1-\varepsilon)/2})$ bits are necessary and sufficient to represent the permutation. We can also support queries in such a structure in $O(1)$ time. We note that the results of Chapter 3 are published in [34].

On the topic of permutations, we cover the problem of powering permutations in place in Chapter 4. Given a permutation of n elements, stored as an array, we address the problem of replacing the permutation by its k^{th} power. We aim to perform this operation quickly using $o(n)$ bits of extra storage. To this end, we first present an algorithm for inverting permutations that uses $O(\lg^2 n)$ additional bits and runs in $O(n \lg n)$ worst case time. This result is then generalized to the situation in which the permutation is to be replaced by its k^{th} power. An algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\varepsilon}\})$ additional bits is presented. We note that the results of Chapter 4 are published in [33].

Then, we cover a bunch of data structures for range reporting problems. Range reporting problems are problems where a point set is preprocessed so that certain information

[1] We use $\lg n$ to denote $\log_2 n$

about a query region can be efficiently computed. These problems are of fundamental importance in computational geometry, both in the study of their optimality with respect to space and query time, and as tools employed to provide efficient solutions to various geometric problems.

More formally, in range reporting problems a set $\mathcal{S}$ of n points in $\mathbb{R}^d$ is preprocessed such that the following query can be efficiently computed. Given a range $\mathcal{Q} = [l_1, r_1] \times \ldots \times [l_d, r_d]$ return an aggregate function over $\mathcal{S} \cap \mathcal{Q}$.

In Chapter 5, we present a data structure for multi-dimensional range mode queries. In this problem we have to preprocess a set of colored points $\mathcal{S}$. Given a query range $\mathcal{Q}$, the aim is to report the most frequent color (i.e., a mode) of the multiset of colors corresponding to the points in $\mathcal{S} \cap \mathcal{Q}$. When $d = 1$, Chan et al. [20] gave a data structure that requires $O(n + (n/\Delta)^2/w)$ words and supports range mode queries in $O(\Delta)$ time for any $\Delta \geq 1$, where $w = \Omega(\log n)$ is the word size. Chan et al. also proposed a data structure for higher dimensions (i.e., $d \geq 2$) with $O(s_n + (n/\Delta)^{2d})$ words and $O(\Delta \cdot t_n)$ query time, where s_n and t_n denote the space and query time of a data structure that supports orthogonal range counting queries on the set $\mathcal{S}$. We show that the space can be improved to $O(s_n + (n/\Delta)^{2d}/w)$ words without any increase to the query time. When $d = 1$, the space and query time costs of our data structure match those achieved by the current best known one-dimensional data structure. We note that the results of Chapter 5 are published in [27, 28].

In Chapter 6 we present succinct data structures for one dimensional approximate color counting, one dimensional approximate median reporting, and one dimensional color reporting. We show, somewhat counter-intuitively, that it is not necessary to store colors of the points in order to answer approximate color counting queries, nor the value of the points in order to answer approximate median reporting queries.

In one dimensional color counting we are given a set of n points with integer coordinates in the range $[1, m]$ and every point is assigned a color from the set $\{1, \ldots, \sigma\}$. A color counting query asks for the number of distinct colors in $[a, b]$. We describe a succinct data structure that answers approximate color counting queries in $O(1)$ time and uses $\mathcal{B}(n, m) + O(n) + o(\mathcal{B}(n, m))$ bits, where $\mathcal{B}(n, m) \approx n \lg m/n$ is the minimum number of bits required to represent an arbitrary set of size n from a universe of m elements. In the special case when points are in the rank space (i.e., when $n = m$), our data structure needs only $O(n)$ bits. Also, we show that $\Omega(n)$ bits are necessary in that case.

We then extend the techniques presented to describe a data structure for the one dimensional approximate median reporting problem. We are given a set of n points with integer coordinates in the range $[1, m]$ and every point is assigned a value from the set

$\{1,\ldots,U\}$. An approximate-median reporting query asks for an element whose rank is between $(\lfloor k/2 \rfloor - \alpha k)$ and $(\lfloor k/2 \rfloor + \alpha k)$ in the query interval $[a,b]$, where k is the number of points in $[a,b]$, and α is the approximation factor. We describe a succinct data structure that answers approximate range median queries in $O(1)$ time and uses $\mathcal{B}(n,m) + O(n) + o(\mathcal{B}(n,m))$ bits. In the special case when points are in the rank space, our data structure needs only $O(n)$ bits, thus improving a result from [16]. Also, we show that $\Omega(n)$ bits are necessary in that case.

Then, we turn to one dimensional color reporting. We are given a set of n points with integer coordinates in the range $[1,m]$ and every point is assigned a color from the set $\{1,\ldots,\sigma\}$. A color reporting query asks for the list of distinct colors that occur in a query interval $[a,b]$. We describe a data structure that uses $\mathcal{B}(n,m) + n\mathcal{H}_d(S) + o(\mathcal{B}(n,m)) + o(n \lg \sigma)$ bits and answers queries in $O(k+1)$ time, where k is the number of colors in the answer, and $\mathcal{H}_d(S)$ $(d = \log_\sigma n)$ is the d-th order empirical entropy of the color sequence. We also consider succinct color reporting under restricted updates. Our dynamic data structure uses $n\mathcal{H}_d(S) + o(n \lg \sigma)$ bits and supports queries in $O(k+1)$ time.

We note that the results of Chapter 6 are published in [31].

Finally, in Chapter 7 we present a succinct dynamic data structure for the one-dimensional range reporting problem where the goal is to maintain (under insertion and deletion) a set of integers $\mathcal{S}$ from a universe of size m to answer range reporting queries: Given an interval $[a,b]$ for some $a,b \in [m]$, find a point in $\mathcal{S} \cap [a,b]$. We describe a succinct data structure that supports updates in $O(\lg^\varepsilon m)$ time and answers queries in optimal $O(k)$ time where k is the number of points in the answer. This is the first dynamic data structure for this problem that uses succinct space and achieves optimal query time. We note that the results of Chapter 7 are to be published in [32].

Chapter 2

Preliminaries

In this chapter we discuss terminology and some standard data structures and techniques that are useful throughout this thesis. We first define the word RAM model which is the model of computation to be used throughout this thesis (Section 2.1), then we give a more formal definition of space efficient data structures (Section 2.2), then we discuss bit vectors and sequences in details (Sections 2.3 and 2.4), finally, we present rank space reduction which is a useful technique in range reporting problems (Section 2.5).

2.1 Word RAM Model

The computational model used throughout this thesis is the word random access machine model, or the word RAM model [1, 40]. The word RAM model is a realistic and natural model for describing modern computers although it does not take into account multilevel storage. Data is stored in words consisting of $w \in \Omega(\lg n)$ bits, where n is the input size. Words can be read and written in constant time. Moreover, arithmetic (addition, subtraction, multiplication, and division) and bitwise boolean operations (AND, OR, NOT, XOR, SHIFT etc.) can be performed in $O(1)$ time on w-bit integers. We measure the running time of an algorithm in this model by counting the number of memory accesses and operations performed on words. The space cost can be measured by counting the number of words or the number of bits used by the algorithm.

2.2 Space Efficient Data Structures

Since we can measure the space cost of a data structure in the word RAM model in terms of the number of bits used, a natural question to ask is: What is the smallest number of bits needed to represent an arbitrary combinatorial object of type $\mathcal{X}$? The answer to this question is a lower bound for any data structure that represents $\mathcal{X}$.

The *information theoretic lower bound* for storing an element from a set $\mathcal{X}$ is $\lg N$ bits, where $N = |\mathcal{X}|$ is the cardinality of $\mathcal{X}$. This is best illustrated with the archetypal example of representing a binary tree of n nodes. In this example $\mathcal{X}$ is the set of all binary trees of size n, and in this case $N = \binom{2n+1}{n}/(2n+1)$ (the n^{th} Catalan number). Thus, to represent a binary tree of n nodes $\lg N = 2n - o(n)$ bits are required. We note that this is significantly better than the traditional pointer representation where each node stores a pointer to each of its left and right children (and potentially subtree size and other information), since such a representation requires $\Theta(n \lg n)$ bits.

We say that a data structure is *succinct* if the space it uses matches the information theoretic lower bound plus a lower order term (i.e. $\lg N + o(\lg N)$ bits). On the other hand, a data structure is *compact* if the space it uses is asymptotically the same as the information theoretic lower bound (i.e. $\Theta(\lg N)$ bits). Moreover, in both cases, operations should be done efficiently.

This area of research goes back to the late eighties, started by Jacobson [59, 58]. However, in the context of space-efficient data structures, the first use of the word *succinct* was by Turán [95] in 1984, where he showed how to store a planar graph of size n in $12n$ bits. By our definitions his data structure is compact not succinct. Moreover, interestingly enough, the first use of the word *compact* in this context was by Van Dam and Evans [96] in 1967. For readable summaries about the area we refer the reader to the following summaries [76, 73]. We also note that the book *Compact Data Structures:A Practical Approach* [78] by Gonzalo Navarro is an excellent source on the topic.

2.3 Bit Vectors

For the sake of completeness and since bit vectors are used extensively throughout this thesis, we cover them in detail in this section. A bit vector is a simple way to represent a set S whose elements come from the universe $[m]$, where $[m]$ denotes the set $\{0, 1, \ldots, m-1\}$. If $i \in S$, we set the i^{th} bit in the bit vector to 1, otherwise we set it to 0. It is not hard to see that membership queries (checking whether a given element in $[m]$ belongs to S) can be answered in constant time by probing a single bit. In addition to membership queries, we would also like to support the following operations:

- rank (i): returns the number of 1s up to and including position i.
- select (i): return the position of the i^{th} 1.

Given a bit vector of length m, Jacobson [59] gave a structure that takes $o(m)$ additional bits of space and can support rank and select by making $O(\lg m)$ bit inspections. However, the bits inspected were not necessarily contiguous and might depend on previous values read. Munro [69] (full details in [23]) enhanced this structure to support both operations in constant time, without increasing the space bound. In this section, we describe the details of this structure.

Supporting Rank. To answer rank queries in constant time, we store the following:

- We break the vector into blocks of size $\lceil \lg^2 m \rceil$, and we store in a table T_1 the number of 1s up to the last position of each block. We also store in T_1 references to all tables T_{2i} (described below) where $0 \le i \le \lceil n/\lceil \lg^2 m \rceil \rceil$. The space used by T_1 is $O(m/\lg m)$ bits.
- We break the blocks into sub-blocks of size $\frac{1}{2}\lceil \lg m \rceil$, and for each block i we store in a table T_{2i} the number of 1s from the start position of the block up to the last position of each sub-block. The space used by T_{2i} is $O(\lg m \lg \lg m)$ bits. The total space required by all such tables is $O(m \lg \lg m/\lg m)$ bits.
- For every possible sub-block, we store a table T_3 that gives the number of 1s up-to every possible position. Since there is $O(\sqrt{m})$ distinct sub-blocks, T_3 requires $O(\sqrt{m} \lg m \lg \lg m)$ bits.

To answer rank (x), let $i = \lfloor x/\lceil \lg^2 m \rceil \rfloor$ be the index of the block containing x, we compute j_1 the number of ones up to position $(i \cdot \lceil \lg^2 m \rceil)$ using table lookup on T_1. Let $k = \lfloor (x - i \cdot \lceil \lg^2 m \rceil)/(\frac{1}{2}\lceil \lg m \rceil) \rfloor$ be the index of the sub-block containing x, using table lookup on T_{2i} we compute j_2 the number of ones up to the last position in the $(k-1)$-th sub-block of the i^{th} block of S. Finally using table lookup on T_3, we get j_3 the number of ones up to position $(x - i \cdot \lceil \lg^2 m \rceil - k \cdot \frac{1}{2}\lceil \lg m \rceil)$ in the k^{th} sub-block of the i^{th} block of S, and we return $(j_1 + j_2 + j_3)$.

Supporting Select. Supporting select queries is more complex than supporting rank queries. We store the following:

- In a table T_1, we store the position of every $\lceil \lg m \lg\lg m \rceil$)-th 1 bit in the bit vector. Also, we store in T_1 references to all tables T_{2i} (described below) where $0 \leq i \leq \lceil n/\lceil \lg m \lg\lg m \rceil \rceil$. T_1 requires $O(m/\lg\lg m)$ bits.
- Let r be the sub-range between the i^{th} 1 and the $(i{+}1)^{\text{th}}$ 1 in T_1. If $r \geq \lceil \lg m \lg\lg m \rceil^2$, we store all the positions of all ones in this subrange in the table T_{2i}. In this case, T_{2i} requires $O(\lg^2 m \lg\lg m)$ bits. However, there can be at most $m/\lceil \lg m \lg\lg m \rceil^2$ such sub-ranges. Thus, the total space required by such tables is $O(m/\lg\lg m)$ bits. If $r < \lceil \lg m \lg\lg m \rceil^2$ we store the position of every $\lceil \lg r \lg\lg m \rceil$-th one bit in the sub-range. In this case, T_{2i} requires $O(r/\lg\lg m)$ bits, and the total space of such tables is $O(m/\lg\lg m)$ bits.
- After one more level of subdivision, the range size will be at most $(\lg\lg m)^4$. We use a precomputed table T_3 that requires $o(m)$ bits to store answers of all select queries on every possible bit vector of that size.

To answer select (x), we check if x is a multiple of $\lceil \lg m \lg\lg m \rceil$. If so we can answer *select*(x) using table lookup on T_1. Let $i = \lfloor x/\lceil \lg m \lg\lg m \rceil \rfloor$. Using table lookup on T_1, we get j_1 the index of the $(i \cdot \lceil \lg m \lg\lg m \rceil)$-th one in S and j_2 the index of the $((i+1) \cdot \lceil \lg m \lg\lg m \rceil)$-th one in S. If $r = j_2 - j_1 \geq \lceil \lg m \lg\lg m \rceil^2$ we get j_3 the index of the $(x - i \cdot \lceil \lg m \lg\lg m \rceil)$-th one in the subdivision between j_1 and j_2 using table lookup on T_{2i}, and we return $(j_1 + j_3)$. Let $k = \lfloor (x - i \cdot \lceil \lg m \lg\lg m \rceil)/\lceil \lg r \lg\lg m \rceil \rfloor$. Using table lookup on T_{2i}, we get j_4 the index of the $(k \cdot \lceil \lg r \lg\lg m \rceil)$-th one in the subdivision between j_1 and j_2, and j_5 the index of the $((k+1) \cdot \lceil \lg r \lg\lg m \rceil)$-th one in the subdivision between j_1 and j_2. Finally using table lookup on T_3, we get j_6 the index of the $(x - i \cdot \lceil \lg m \lg\lg m \rceil - k \cdot \lceil \lg r \lg\lg m \rceil)$-th one in the subdivision between j_4 and j_5, then we return $(j_1 + j_4 + j_6)$.

An immediate use of rank and select queries, is the ability to support the successor and predecessor queries.

Supporting Predecessor. The predecessor of an element x, is the largest element $y < x$ such that $y \in S$. To answer predecessor (x) we return select (rank $(x) - 1$) if $x \in S$, otherwise we return select (rank (x)).

Supporting Successor. The successor of an element x, is the smallest element $y > x$ such that $y \in S$. To answer successor (x) we return select (rank $(x) + 1$).

Theorem 1 ([69]). *A bit vector of length m can be represented in $m + o(m)$ bits, such that rank, select, membership, predecessor and successor queries can be answered in constant time.*

In special cases when the number of 1s and the number of 0s in the bit vector are not proportional to each other, the space in the previous theorem can be improved.

Theorem 2 ([87]). *A bit vector of length m can be represented in $\lg \binom{m}{n} + O(m \lg\lg m / \lg m)$ bits where n is the number of 1s in the vector, such that rank, select, membership, predecessor and successor queries can be answered in constant time.*

2.4 Sequences

In classical information theory, one assumes that there is an infinite source that emits elements according to some distribution. A fundamental result of information theory is that, if the elements are being emitted independently, the minimum possible average code length for unambiguous codes is the *Shannon entropy* defined as $\mathcal{H} = -\sum_i (p_i \lg p_i)$, where p_i is the probability of symbol i, and $0 \lg 0$ is assumed to be 0.

A *sequence* is a string of finite size over an alphabet $\Sigma = \{1, \ldots, \sigma\}$. Given a sequence S over an alphabet $\Sigma = [1 \ldots \sigma]$, we obtain the *empirical entropy* of S by taking its Shannon entropy and substituting p_i with (n_i/n), where n_i is the number of occurrences of symbol i in S and n is the length of S. The zero'th order empirical entropy is defined as: $\mathcal{H}_0 = -\sum_i ((n_i/n) \lg (n_i/n))$, and $n\mathcal{H}_0(S)$ would be the size of an ideal compressor that uses $-\lg (n_i/n)$ bits to represent symbol i.

This compression ratio can be improved if the code of each symbol was a function of itself and the k symbols preceding it. For any k symbol string $W \in \Sigma^k$ let S_W denote the subsequence of S that contains the symbols following W in S. The k-the order empirical entropy is defined as: $\mathcal{H}_k = -\sum_i (|S_W|/n)\mathcal{H}_0(S_W)$, and $n\mathcal{H}_k(S)$ is a lower bound on storing S for any compression scheme that uses codes depending only on the k most recently seen symbols.

In the context of succinct data structures we need to store a sequence in a compressed form and support the following operations:

- access (i): returns the i-th character in S.
- rank (i, a): returns the count of the occurrences of symbol a in the first i positions of S.
- select (i, a): finds the position where the symbol a occurs for the i-th time.

The state of the art results on static sequences is due to the following theorem by Belazzougui1 and Navarro [11].

Theorem 3 ([11])**.** *A string S over an alphabet of size σ can be represented using $n\mathcal{H}_k(S)+o(n\lg\sigma)$ bits for any $k = o(n\lg_\sigma n)$ so that operation access can be solved in constant time, operation rank can be solved in time $O(\lg(\lg\sigma/\lg w))$, and operation select can be solved in time $O(f(n,\sigma))$ for any function $f(n,\sigma)$ satisfying $\omega(1) = f(n,\sigma) = o(\lg(\lg\sigma/\lg w))$.*

2.5 Reduction to Rank Space

Rank space reduction is a useful technique for range reporting problems first presented by Alstrup [3]. Using this technique, we can reduce a d-dimensional range reporting problem on a set P to the special case when all points have distinct coordinates that are integers bounded by n, where n is the number of points in the data structure. In what follows we briefly describe this reduction technique.

Given some integer x, for each i where $1 \leq i \leq d$ denote by $r_i(x)$ the rank of x among the i^{th} coordinate of all the points in P. Let P' be the set of n points formed by replacing the i^{th} coordinate of each point $p \in P$ denoted by p_i with $r_i(p_i)$. Any range reporting query $\mathcal{Q} = [a_1, b_1] \times [a_d, b_d]]$ on P is equivalent to a range reporting query $\mathcal{Q} = [r_1(a_1), r_1(b_1)] \times [r_d(a_d), r_d(b_d)]]$ on P'. Suppose that we can answer range reporting queries $\mathcal{Q}$ (e.g., a color reporting or a color counting query) on P' in time $t_q(n)$ using an $s(n)$-space data structure. Suppose that we can answer predecessor queries on some set in time $t'(n)$ using an $s'(n)$-space data structure. Then, it follows that we can answer range reporting queries $\mathcal{Q}$ on P in time $t(n) + d \cdot t'(n)$ using an $(s(n) + d \cdot s'(n))$-space data structure.

Chapter 3

Compact Unlabeled Permutations

3.1 Introduction and Motivation

A permutation π is a bijection from the set $\{1, \ldots, n\}$ to itself. Given a permutation π on an n element set, our problem is to preprocess the set, assigning a unique label to each element, to obtain a data structure with minimum space to support the following query: given a label i, determine $\pi^k(i)$ quickly where k is an arbitrary (not fixed) integer and π^k is the k^{th} power of π. We denote such queries by $\pi^k()$. Moreover, we assume that k is bounded by some polynomial function in n.

We are interested in *compact*, or highly-space efficient data structures. Our aim is to develop data structures whose size is within a constant factor of the information theoretic lower bound. Designing compact data structures is an area of interest in theory and practice motivated by the need of storing large amount of data using the smallest space possible.

Permutations are fundamental in computer science and are studied extensively. They are commonly used as a basic building block for space efficient encoding of strings [5, 45, 79, 92], binary relations [7, 9], integer functions [74] and many other combinatorial objects. Several papers have looked into problems related to permutation generation [93], permuting in place [38] etc. Others have dealt with the problem of space-efficient representation of restricted classes of permutations, like the permutations representing the lexicographic order of the suffixes of a string [50, 55], or the so-called approximately min-wise independent permutations [18], which are used for document similarity estimation. Since there are exactly $n!$ permutations, the number of bits required to represent a permutation of length n is $\lceil \lg(n!) \rceil \sim n \lg n - n \lg e + O(\lg n)$ bits. Munro et al. [74] studied the space efficient representation of general permutations where general powers can be computed quickly. They gave a representation taking the optimal $\lceil \lg(n!) \rceil + o(n)$ bits where $\pi()$ and $\pi^{-1}()$ can be computed in $O(\lg n / \lg \lg n)$ time, and a representation taking $((1+\varepsilon)n \lg n)$ bits where $\pi^k()$ can be computed in constant time for any k.

In this chapter we study the space-efficient representation of permutations where labels can be freely reassigned. This problem is similar to the problem of representing unlabeled equivalence relations [65, 27, 30]. In that problem one is given a partition of an n element set into equivalence classes. The goal is to preprocess the partition and assign a unique label to each element, obtaining a data structure to answer the following query: given two elements, are they in the same equivalence class. We note that a permutation of n elements can be decomposed into a set of disjoint cycles whose lengths form an integer partition of n. Two permutations are considered equivalent by relabelling if there cycles lengths form the same integer partition. In the case when the label space is n, both problems are similar since they become equivalent to storing an integer partition of n as we will show in Section 3.4. We note that the number of integer partitions of n is by the Hardy-Ramanujan formula [53] asymptotically equivalent to $\frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}$; thus the information theoretic lower bound for storing an integer partition is $\Theta(\sqrt{n})$ bits.

Our problem differs from representing equivalence relations when the label space exceeds n. For the case of equivalence relations, once the label space increases to $(1+\varepsilon)n$ for any constant $\varepsilon > 0$ the data structure size decreases from $\Theta(\sqrt{n})$ bits to $\Theta(\lg n)$ bits. This result is shown in [27, 30]. The main reason for this drastic decrease in auxiliary storage size is that as the label space increases, it is not necessary to keep track of the exact sizes of the equivalence classes; keeping track of an approximation of their sizes is sufficient. On the other hand, for the case of permutations it is always necessary to know the exact size of each cycle. Thus, as we increase the label space we will not witness such a decrease in auxiliary storage size.

We study this problem in several scenarios; thus, showing the tradeoffs between label space and auxiliary storage size for the stated problem. In Section 3.3, we cover the scenario where queries are to be answered by just examining the labels of the queried elements. We show that a label space of $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \cdot i$ is necessary and sufficient. Then, we show that with a label space of n^2 queries can be answered in constant time. In Section 3.4, we cover the scenario where labels can be assigned from the set $\{1, \ldots, n\}$. We show that $\Theta(\sqrt{n})$ bits are necessary and sufficient to represent the permutation. We use the same data structure as the main structure in [65]. However, we optimize it to achieve constant query time while using only $O(\sqrt{n})$ bits. Section 3.5 contains the main result of this chapter. We cover the scenario where labels can be assigned from the set $\{1, \ldots, cn^{1+\varepsilon}\}$ where c is a constant and $0 < \varepsilon < 1$. We show that $\Theta(n^{(1-\varepsilon)/2})$ bits are necessary and sufficient to represent the permutation, and we support queries in such a structure in $O(1)$ time in the standard word-RAM model.

Finally as an application to our new data structures, we give a representation of a labeled permutation that takes $s(n) + O(\sqrt{n})$ bits and can answer $\pi^k()$ in $O(t_f + t_i)$ time, where $s(n)$ denotes the number of bits required for a representation R to store a labeled permutation, and t_f and t_i are the time needed for R to support $\pi()$ and $\pi^{-1}()$. This result improves Theorem 3.3 in [74].

We note that the results of this chapter are published in [34].

3.2 Definitions

A *permutation* π is a bijection from the set $\{1, \ldots, n\}$ to itself, and we denote its inverse bijection as π^{-1}. We also extend the definition to arbitrary integer power of π as follows:

$$\pi^k(i) = \begin{cases} \pi^{k+1}(\pi^{-1}(i)) & k < 0 \\ i & k = 0 \\ \pi^{k-1}(\pi(i)) & k > 0 \end{cases}$$

A permutation can be viewed as a set of disjoint cycles. Since we are working with unlabeled permutations, we have the freedom to assign the labels in any way. In all our labeling schemes elements within the same cycle will get a block of consecutive labels. Furthermore, the blocks for cycles of the same length will be contiguous. For example the elements of the first cycle of length l will get labels from the interval $[s, s+l-1]$ for some integer s such that $\pi(i) = i+1$ for $i \in [s, s+l-2]$ and $\pi(s+l-1) = s$. The elements of the second cycle of length l will get labels in the range $[s+l, s+2l-1]$, and so on. Thus, given a label i and an integer k, to answer $\pi^k(i)$ it is sufficient to compute l the length of the cycle that i belongs to, and s the smallest index of an element that belongs to a cycle of length l. Now, it is not hard to verify that $\pi^k(i) = s + rl + ((p+k)\%l)$ where $r = \lfloor (i-s)/l \rfloor$, $p = i - (s + rl)$, and % denotes the modulo operation.

For example suppose that we have two cycles of length 5 which we assign labels from $[10, 14]$ and $[15, 19]$. To compute $\pi^2(16)$ notice that $l = 5$, $s = 10$, $r = \lfloor (16-10)/5 \rfloor = 1$, $p = 16 - (10+5) = 1$, so $\pi^2(16) = 10 + 5 + (1+2)\%5 = 18$.

Notice that the multiset formed by the cycle lengths of a given permutation π over an n-element set will form an integer partition of the integer n. An *integer partition* p of n is a multiset of positive integers that sum to n. We call these positive integers the *elements* of p, and we denote by $|p|$ this number of elements. We say that an integer partition p of n *dominates* an integer partition q of m where $n > m$ if q is a subset of p. For example, the integer partition $\{5, 5, 10\}$ of 20 dominates the integer partition $\{5, 5\}$ of 10, but not the integer partition $\{4, 6\}$ of 10. Given an integer partition p of n, we define a *part* q of size k to be a collection of elements in p that sum to k. We say that an integer s *fills* q if q contains $\lfloor k/s \rfloor$ integers s and one integer $k \bmod s$. Furthermore, we say that two parts *intersect* if they share at least one common element; otherwise, they are *non-intersecting*. For example the integer partition $\{1, 4, 5\}$ of 10 contains the following parts: part $\{1\}$ of size 1, part $\{4\}$ of size 4, part $\{5\}$ of size 5, part $\{1, 4\}$ of size 5, part $\{1, 5\}$ of size 6, part $\{4, 5\}$ of size 9 and part $\{1, 4, 5\}$ of size 10. We say that 5 fills the parts $\{5\}$ and $\{4, 5\}$ but not the part $\{1, 4, 5\}$. The parts $\{4, 5\}$ and $\{4\}$ are intersecting, while the parts $\{4, 5\}$ and $\{1\}$ are non-intersecting.

Finally, we give two observations that we will use repeatedly.

Observation 1. *M not necessarily distinct integers $m_0, \ldots, m_{M-1}$ ordered such that $m_i \leq m_{i+1}$ for $i \in [0, N-1]$ can be represented in $O(N + M)$ bits such that the i^{th} integer m_i can be accessed in $O(1)$ time.*

Proof. Store the values m_0 and $(m_i - m_{i-1})$ for $i = 1, \ldots, M-1$ represented in unary with a 0 separator between each two consecutive values in a bit vector ψ as described in Section 2.3. Also store a select structure on ψ to identify the 0s quickly, and a rank structure to count the 1s quickly. To get the integer m_i, count the number of 1s before the i^{th} 0 in ψ. □

Observation 2. *M positive integers $m_0, \ldots, m_{M-1}$ that sum to N can be represented in $O(N + M)$ bits such that the i^{th} integer m_i can be accessed in $O(1)$ time, the partial sum $\sum_{j=1}^{i} m_j$ can be computed in $O(1)$ time, and given an integer x we can compute the biggest index i such that $\sum_{j=1}^{i} m_j \leq x$ in $O(1)$ time.*

Proof. Store the values m_i for $i = 0, \ldots, M-1$ represented in unary with a 0 separator between each two consecutive values in a bit vector ψ as described in Section 2.3. Also store a select structure on ψ to identify the 1s and 0s quickly, and a rank structure to count the 1s and 0s quickly. To get the integer m_i, subtract the number of 1s before the i^{th} 0 from the number of 1s before the $(i-1)^{\text{th}}$ 0 in ψ. To compute the partial sum value $\sum_{j=1}^{i} m_j$, count the number of 1s before the i^{th} 0 in the bit vector ψ. Given x, to compute the biggest index i such that $\sum_{j=1}^{i} m_j \leq x$, get the index i of the x^{th} 1 ψ then return the number of 0s before i. □

Note that if we are allowed to reorder the numbers in Observation 2, we can reduce the size of the representation to $O(\sqrt{N})$ bits without compromising the constant runtime of the stated operations since the problem becomes equivalent to storing an integer partition of N.

3.3 Direct Labeling Scheme

In this section we cover the problem where queries are answered by computing directly from the labels without using any auxiliary storage except for the value of n. We show that a label space of $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \cdot i$ is necessary and sufficient to represent the permutation. Moreover, we show that with a label space of n^2, we can compute $\pi^k()$ in constant time.

Theorem 4. *Given a permutation π, a label space of $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \cdot i < n^2$ is necessary and sufficient to represent the permutation.*

Proof. To show that this many labels are necessary, consider a labeling scheme for this problem. It reserves a set of labels for each cycle to ensure that queries are answered correctly by looking only at the labels. Consider the labels assigned by such a scheme for the following collection C of n permutations. The i^{th} permutation C_i of C contains $\lfloor n/i \rfloor$ cycles each of length i and one cycle of length $n - \lfloor n/i \rfloor \cdot i$.

Note that for each C_i the labels assigned to the elements of the $\lfloor n/i \rfloor$ cycles of length i can not be reused for the elements of any cycle of length different than i. This happens because for any label x, we can obtain the length of the cycle that x belongs to by searching for the smallest positive integer k such that $\pi^k(x) = x$. Thus, a label space of $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \cdot i$ is necessary.

For the upper bound observe that there exist at most $\lfloor n/i \rfloor$ cycles of length i. We assign labels from the set of integers in the range $[0, n-1]$ for all the elements in cycles of length 1, and labels from the set of integers in the range $[\sum_{j=1}^{i-1}(\lfloor \frac{n}{j} \rfloor \cdot j)+(r-1)i, \sum_{j=1}^{i-1}(\lfloor \frac{n}{j} \rfloor \cdot j)+ri-1]$ for the elements in the r^{th} cycle of length i, where $1 \leq r \leq \lfloor n/i \rfloor$. Given a label x, to answer a query $\pi^k(x)$ we find the biggest integer l such that $s = \sum_{j=1}^{l-1} \lfloor \frac{n}{j} \rfloor \cdot j \leq x$. Next, we compute $r = \lfloor (x-s)/l \rfloor$ and $p = x - (s + rl)$ then we return $s + rl + ((p+k)\%l)$. □

To answer queries in constant time we extend the label space marginally to n^2. Then we assign labels from the set of integers in the range $[0, n-1]$ for all the elements in cycles of length 1, and labels from the set of integers in the range $[n(i-1)+(r-1)i, n(i-1)+ri-1]$ for the elements in the r^{th} cycle of length i, where $1 \leq r \leq \lfloor n/i \rfloor$. Given a label x, to answer a query $\pi^k(x)$ we find $l = \lfloor x/n \rfloor + 1$. Next, we compute $s = (l-1)n$, $r = \lfloor (x-s)/l \rfloor$ and $p = x - (s + rl)$ then we return $s + rl + ((p + k)\%l)$.

Theorem 5. *Given a permutation π, we can assign to each of the elements a label in the range of $\{1, \ldots, n^2\}$ such that $\pi^k()$ can be computed in constant time by looking only at the labels.*

3.4 Compact Data Structures with Label Space n

In this section we consider the scenario where the n elements are to be assigned labels in the range 1 to n. The queries can be answered by looking at an auxiliary data structure. Moreover, we have the freedom to assign the labels in any way.

Following [65], the information theoretic lower bound for the representation of a permutation is the number of partitions of n, which by the Hardy-Ramanujan formula [53] is asymptotically equivalent to $\frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}$. Thus, the information theoretic lower bound for representing a permutation is $\Theta(\sqrt{n})$ bits of space.

We will use the same data structure as the main structure in [65], however we will optimize it to achieve constant query time while using only $O(\sqrt{n})$ bits. Given π let m be the number of distinct cycle sizes in π and let $s_1, \ldots, s_m$ be the distinct sizes of the cycles. For $i = 1$ to m let n_i be the number of cycles of size s_i. We order the cycles in non-decreasing order by $\gamma_i = s_i n_i$ so that for $i = 1$ to $m - 1$, $s_i n_i \leq s_{i+1} n_{i+1}$. Notice that since

$$\textstyle\sum_{i=1}^{m} s_i n_i = n \text{ and } s_i n_i \geq i \text{ for } i = 1, \ldots, m,$$

m is at most $\sqrt{2n}$. The primary data structure is made up of two sequences:

- the sequence $\vec{\delta}$ that consists of $\delta_1 = s_1 n_1$ and $\delta_i = s_i n_i - s_{i-1} n_{i-1}$, for $i = 2, \ldots, m$ and
- the sequence $\vec{n}$ that consists of n_i, for $i = 1, \ldots, m$.

We represent the elements of the two sequences in binary. Since the lengths of the elements may vary, we store two other sequences that shadow the primary sequences. The shadow sequences have a 1 at the starting point of each element in the shadowed sequence and a 0 elsewhere. Also we store a select structure on the two shadow sequences in order to identify the 1s quickly. It is proved in [65] these sequences can be stored in $O(\sqrt{n})$ bits.

The sequences $\vec{\delta}$ and $\vec{n}$ give an implicit ordering of the elements. We assign the first $s_1 n_1$ labels to the elements of the cycles with length s_1, and then we assign the next $s_2 n_2$ labels to the elements of the cycles with length s_2, and so on.

Define the predecessor of an element x to be the maximum index j satisfying the condition that $\sum_{i=1}^{j} s_i n_i < x$. We store an array A where $A[i] = \max\{j \mid \sum_{t=1}^{j} s_t n_t \leq i(i+1)/2\}$, for $i = 1$ to $\sqrt{2n}$. Next, we prove a modified version of Lemma 2 in [65].

Lemma 6. *The predecessor $p(x)$ of an integer x in the sequence $\sum_{t=1}^{i} s_t n_t$, $i = 1$ to m is in the range $[A[\lfloor\sqrt{2x}\rfloor - 1], A[\lfloor\sqrt{2x}\rfloor - 1] + 5]$.*

Proof. Let $i = \lfloor\sqrt{2x}\rfloor - 1$. Without loss of generality assume that $i \geq 6$, since for $x < 25$ we can store $p(x)$ explicitly in $O(\lg n)$ bits. Notice that:

$$i(i+1)/2 \leq (\sqrt{2x} - 1)\sqrt{2x}/2 \leq x$$

and

$$x \leq \sqrt{2x}(\sqrt{2x} + 1)/2 \leq (i+2)(i+3)/2$$

For $j = A[i] + 1$, $\sum_{t=1}^{j-1} s_t n_t \leq i(i+1)/2$, so $j - 1 \leq i$ and $j \leq i + 1$. Since $\sum_{t=1}^{j} s_t n_t > i(i+1)/2$, $s_j n_j \geq i(i+1)/(2j) \geq i/2$. Hence, $\sum_{t=1}^{j+5} s_t n_t \geq (i+2)(i+3)/2 \geq x$. □

We can obtain the actual value of $p(x)$ by checking at most six numbers. Moreover, we can store A using $O(\sqrt{n})$ bits using the method described in Observation 1.

In the standard word-RAM model, computing $\sqrt{x}$ is not a constant time operation. The standard Newton's iterative method uses $O(\lg\lg n)$ operations. Following [65], we can use a look-up to precomputed tables and finds $\sqrt{x}$ in constant time. We use two tables, one when the number of bits up to the most significant bit of x is odd, denoted by O, and one when the number of bits is even, denoted by E. For $i = 1, \ldots, \lceil\sqrt{2n}\rceil$, we store in $E[i]$ the value of $\lfloor\sqrt{i2^{\lceil \lg i \rceil}}\rfloor$, and in $O[i]$ the value of $\lfloor\sqrt{i2^{\lceil \lg i \rceil - 1}}\rfloor$. E and O can be stored in $O(\sqrt{n})$ bits by storing them using the method described in Observation 1. We summarize with:

Lemma 7. *For $i \leq n$, $\lfloor \sqrt{i} \rfloor$ can be computed in constant time using a precomputed table of $O(\sqrt{n})$ bits.*

For each i where at least one of the bit locations of δ_i in $\vec{\delta}$ is a multiple of $(\varepsilon \lg n)$, we store the partial sum value $\sum_{j=1}^{i} (s_j n_j)$ and the value of $s_i n_i$. Moreover, for every possible sequence of δ values $\delta_1, \delta_2, \ldots, \delta_o$ of length $(\varepsilon \lg n)$ and its corresponding shadow sequence, we store in a table T the values $\sum_{j=1}^{h} (\sum_{f=1}^{j} \delta_f)$. To compute $\sum_{j=1}^{i} (s_j n_j)$ for an arbitrary index i, we find the biggest index $v \leq i$ that has its partial sum value stored. Notice that $\sum_{j=1}^{i} (s_j n_j) = \sum_{j=1}^{v} (s_j n_j) + (i - v) s_v n_v + \sum_{j=v+1}^{i} (\sum_{f=v+1}^{j} \delta_f)$. Since we can obtain these values using table lookup on T, we can compute the partial sum at an arbitrary index in constant time. Moreover, we can compute the value of $s_i n_i$ for an arbitrary index i by computing the partial sum at $i - 1$ and subtracting it from the partial sum at i. Finally, we can compute s_i by computing $s_i n_i$ and dividing it by n_i. By choosing $\varepsilon < 1/4$, the size of T becomes $o(\sqrt{n})$ bits.

Answering Queries: Given a label x, to compute $\pi^k(x)$ we first find the predecessor $p(x)$ of x by querying A and checking at most 6 different values. Next we compute the partial sum value $s = \sum_{i=1}^{p(x)-1} (n_i s_i)$. Then, we compute $r = \lfloor (x - s)/s_{p(x)} \rfloor$ and $p = x - (s + r s_{p(x)})$ then we return $s + r s_{p(x)} + ((p + k) \% l)$.

Theorem 8. *Given an unlabeled permutation of n elements, $\Theta(\sqrt{n})$ bits are necessary and sufficient for storing the permutation if each element is to be given a unique label in the range $\{1, 2, \ldots, n\}$. Moreover, there is a structure of $\Theta(\sqrt{n})$ bits such that $\pi^k()$ can be computed in $O(1)$ time.*

3.5 Compact Data Structures with Extended Label Space

In this section we consider the scenario where the n elements are to be assigned labels in the range 1 to $cn^{1+\varepsilon}$ where c is a constant and $0 < \varepsilon < 1$. As in Section 3.4 we assign an implicit ordering of the elements, and queries can be answered by looking at an auxiliary data structure.

Given π, we divide the cycles in π into four different groups and handle each group appropriately. Let k_3 be the number of cycles of size $\leq n^{(1+\varepsilon)/2}$, and let $\{s_1, \ldots, s_{k_3}\}$ be the sizes of those cycles. For $i = 1$ to k_3 let n_i be the number of cycles of size s_i. Without loss of generality we define k_1 and k_2 such that:

- $\gamma_i = s_i n_i \le (\sqrt{c} n^{(1+\varepsilon)/2})/2 = \eta$, for $1 \le i \le k_1$.
- $s_i \le n^{(1-\varepsilon)/2}$ and $\gamma_i > \eta$, for $k_1 < i \le k_2$.
- $n^{(1-\varepsilon)/2} < s_i \le n^{(1+\varepsilon)/2}$ and $\gamma_i > \eta$, for $k_2 < i \le k_3$.

Let $l_{k_3+1}, \ldots, l_{k_4}$ be the size of the cycles that are bigger than $n^{(1+\varepsilon)/2}$. Note that the l_i ($i = k_3 + 1$ to k_4) values are not necessarily unique.

Case 1 ($1 \le i \le k_1$): We reserve the first $(cn^{1+\varepsilon})/4$ labels to handle all possible cycle sizes when $\gamma_i \le \eta$. We assign labels to the elements in the cycles that satisfy this criteria in a similar method to the labeling scheme described in Theorem 5. To be more specific, we assign labels from the set of integers in the range $[0, \eta - 1]$ for all the elements in cycles of length 1, and assign labels from the set of integers in the range $[\eta(j-1), \eta j - 1]$ for all the elements in cycles of length j, where $2 \le j \le \eta$. This covers all the elements of the cycles of sizes $s_1, \ldots, s_{k_1}$, and increases the label space by at most $\eta^2 = (cn^{1+\varepsilon})/4$. Let $B_1 = (cn^{1+\varepsilon})/4$.

Case 2 ($k_1 + 1 \le i \le k_2$): We order the s_i values in increasing order and make all cycles of size s_i fill a part whose length is $c_i\eta$, a multiple of η. Notice that $(k_2 - k_1) < n/\eta$ since $\gamma_i > \eta$, so the label space will increase by at most n. Since $\sum_{i=k_1+1}^{k_2}(c_i) \le (2n)/\eta = O(n^{(1-\varepsilon)/2})$, we can store the c_i values in $O(n^{(1-\varepsilon)/2})$ bits using the method described in Observation 2. Moreover, we store a bit vector ψ of size $n^{(1-\varepsilon)/2}$ to identify the s_i values, and we store a select structure on ψ to identify the 1s quickly. We assign labels in the range $[B_1, B_1 + c_{(k_1+1)}\eta - 1]$ to the elements in cycles of size $s_{(k_1+1)}$, then we assign the next $c_{(k_1+2)}\eta$ labels to elements in cycles of size $s_{(k_1+2)}$, and so on. Let $B_2 = B_1 + \sum_{j=k_1+1}^{k_2} c_j\eta$.

Case 3 ($k_2 + 1 \le i \le k_3$): We make all cycles of size s_i fill a part whose length is $c_i\eta$, a multiple of η. As in case 2, we store the c_i values in $O(n^{(1-\varepsilon)/2})$ bits using the method described in Observation 2. To identify the s_i values: we order them in increasing order of $r_i = s_i \% (16n^{(1-\varepsilon)/2}/c)$ and store the r_i values in $O(n^{(1-\varepsilon)/2})$ bits using the method described in Observation 1, then we store the value of $q_i = s_i/(16n^{(1-\varepsilon)/2}/c) \le (cn^{\varepsilon}/16)$ in the label of each element that is in a cycle of size s_i. Now $s_i = q_i(16n^{(1-\varepsilon)/2}/c) + r_i$. Let β_1 be equal to $\sum_{i=k_2+1}^{k_3} c_i\eta$. We assign labels in the range

$$\left[B_2 + q_i 2^{\lceil \lg(\beta_1) \rceil} + \sum_{j=k_2+1}^{i-1} c_j\eta, \quad B_2 + q_i 2^{\lceil \lg(\beta_1) \rceil} + \sum_{j=k_2+1}^{i} c_j\eta - 1\right]$$

to the elements in the cycles of size s_i. The label space will increase by at most $(cn^{\varepsilon}/16)2^{\lceil \lg(\beta_1) \rceil} + \beta_1 \le (cn^{1+\varepsilon})/4 + O(n)$. Let $B_3 = B_2 + (cn^{\varepsilon}/16)2^{\lceil \lg(\beta_1) \rceil} + \beta_1$.

Case 4 ($k_3+1 \le i \le k_4$): For the cycles of length l_i we make each cycle fill a part whose length is $c_i\eta$, a multiple of η. As in the previous cases, store the c_i values in $O(n^{(1-\varepsilon)/2})$ bits using the method described in Observation 2. To identify the l_i values: we order them by $r_i = (l_i\%\eta)\%(8n^{(1-\varepsilon)/2}/\sqrt{c})$ and store the r_i values in $O(n^{(1-\varepsilon)/2})$ bits using the method described in Observation 1, then store the value of $q_i = (l_i\%\eta)/(8n^{(1-\varepsilon)/2}/\sqrt{c}) \le (cn^{\varepsilon}/16)$ in the label of each element that is in a cycle of size l_i. Now $l_i = q_i(8n^{(1-\varepsilon)/2}/\sqrt{c})+r_i+(c_i-1)\eta$. Let β_2 be equal to $\sum_{i=k_3+1}^{k_4} c_i\eta$. Assign labels in the range

$$\left[B_3 + q_i 2^{\lceil \lg(\beta_2)\rceil} + \sum_{j=k_3+1}^{i-1} c_j\eta, \quad B_3 + q_i 2^{\lceil \lg(\beta_2)\rceil} + \sum_{j=k_3+1}^{i} c_j\eta - 1\right]$$

to the elements in the cycle of size l_i.

The total size of the structures used is $O(n^{(1-\varepsilon)/2})$ bits, and the total address space increased to at most $(3cn^{1+\varepsilon})/4 + O(n) \le cn^{1+\varepsilon}$ as required.

Answering Queries: Given a label x, to compute $\pi^k(x)$ we distinguish between four different cases:

Case 1 $x < B_1$**:** We Compute the value of $l = \lfloor x/\eta \rfloor + 1$, $s = (l-1)\eta$, $r = \lfloor (x-s)/l \rfloor$, and $p = x - (s + rl)$. Then, we return $s + rl + ((p + k)\%l)$.

Case 2 $B_1 \le x < B_2$**:** We compute the value $m = (x - B_1)/\eta$. Then we get the biggest index i such that $\sum_{j=k_1+1}^{i} c_j \le m$. This operation can be done in $O(1)$ time using the structure from Observation 2. Next, we find l the index of the i^{th} one in ψ; l is the size of the cycle that x belongs to. We compute $s = B_1 + \sum_{j=k_1+1}^{i-1} c_j\eta$, $r = \lfloor (x-s)/l \rfloor$, and $p = x - (s + rl)$. Then, we return $s + rl + ((p + k)\%l)$.

Case 3 $B_2 \le x < B_3$**:** We compute the value $m = ((x - B_2)\%\beta_1)/\eta$. Then we get the biggest index i such that $\sum_{j=k_2+1}^{i} c_j \le m$. Next we calculate $q_i = \lfloor (x - B_2)/2^{\lceil \lg(\beta_1)\rceil} \rfloor$ and $l = q_i(16n^{(1-\varepsilon)/2}/c) + r_i$; l is the size of the cycle that x belongs to. We compute $s = B_2 + q_i 2^{\lceil \lg(\beta_1)\rceil} + \sum_{j=k_2+1}^{i-1} c_j\eta$, $r = \lfloor (x-s)/l \rfloor$, and $p = x - (s + rl)$. Then, we return $s + rl + ((p + k)\%l)$.

Case 4 $B_3 \le x$**:** We compute the value $m = ((x - B_3)\%\beta_2)/\eta$. Then we get the biggest index i such that $\sum_{j=k_3+1}^{i} c_j \le m$. Next we calculate $q_i = \lfloor (x - B_3)/2^{\lceil \lg(\beta_2)\rceil} \rfloor$ and $l = q_i(8n^{(1-\varepsilon)/2}/\sqrt{c})+r_i+(c_i-1)\eta$; l is the size of the cycle that x belongs to. We compute $s = B_3 + q_i 2^{\lceil \lg(\beta_2)\rceil} + \sum_{j=k_3+1}^{i-1} c_j\eta$, $r = \lfloor (x-s)/l \rfloor$, and $p = x - (s + rl)$. Then, we return $s + rl + ((p + k)\%l)$.

All operations used take constant time, so $\pi^k(x)$ can be computed in $O(1)$ time.

Theorem 9. *Given an unlabeled permutation of n elements, $\Theta(n^{(1-\varepsilon)/2})$ bits are sufficient for storing the permutation if each element is to be given a unique label in the range $\{1, \dots, cn^{1+\varepsilon}\}$ for any constant $c > 1$ and $\varepsilon < 1$. Moreover, there is a structure of $\Theta(n^{(1-\varepsilon)/2})$ bits such that $\pi^k()$ can be computed in $O(1)$ time.*

Note that ε doesn't need to be a constant. By setting $\varepsilon = \alpha + \beta \lg\lg n/\lg n$ where α and β are constants, and $0 < \alpha < 1$ we get the following theorem:

Theorem 10. *Given an unlabeled permutation of n elements, $\Theta(n^{(1-\alpha)/2}/\lg^{\beta/2} n)$ bits are sufficient for storing the permutation if each element is to be given a unique label in the range $\{1, \dots, cn^{1+\alpha}\lg^{\beta} n\}$ for any constant c, α, β where $0 < \alpha < 1$. Moreover, there is a structure of $\Theta(n^{(1-\alpha)/2}/\lg^{\beta/2} n)$ bits such that $\pi^k()$ can be computed in $O(1)$ time.*

3.6 Lower Bounds

In this section we provide lower bounds on the auxiliary data size as the label space increases.

3.6.1 Lower Bound for Auxiliary Data with Label Space cn

In [30] El-Zein showed that for the problem of representing unlabeled equivalence relations, increasing the label space by a constant factor causes the size of the auxiliary data structure to decrease from $O(\sqrt{n})$ to $O(\lg n)$ bits.

In contrast to the problem of representing unlabeled equivalence relations, in this section we show that for the problem of representing unlabeled permutations increasing the label space by a constant factor will not affect the size of the auxiliary data structure asymptotically.

For any integer $c > 1$, let S_{cn} be the set of all partitions of $\lfloor cn \rfloor$ and S_n the set of all partitions of n. Without loss of generality assume that $\sqrt{n}$ is an integer that is divisible by c. While one partition of cn can dominate many partitions of n, we argue that at least $\binom{c\sqrt{n}}{\sqrt{n}/c}/\binom{\sqrt{n}}{\sqrt{n}/c}$ partitions of cn are necessary to dominate all partitions of n. Let $\mathcal{S}$ be the smallest set of partitions of cn that dominates all the partitions of n. We claim that:

Lemma 11. $|\mathcal{S}| \geq \binom{c\sqrt{n}}{\sqrt{n}/c}/\binom{\sqrt{n}}{\sqrt{n}/c}$.

Proof. Divide n into $\sqrt{n}/c$ parts each of size $c\sqrt{n}$. Let Q be the set formed by filling each part with a distinct size in the range $[1, c\sqrt{n}]$, clearly $|Q| = \binom{c\sqrt{n}}{\sqrt{n}/c}$.

A partition of cn can dominate at most $\sqrt{n}$ distinct parts, hence at most $\binom{\sqrt{n}}{\sqrt{n}/c}$ partitions in Q. Therefore, to dominate Q we need a minimum of $\binom{c\sqrt{n}}{\sqrt{n}/c}/\binom{\sqrt{n}}{\sqrt{n}/c}$ partitions of cn. Since Q is a subset of S_n our claim holds. □

The information theoretic lower bound for the space needed to represent a permutation of size n once labels are assigned from the set $\{1, \ldots, cn\}$ is

$$\begin{aligned} \lg(|\mathcal{S}|) &\geq \lg(\binom{c\sqrt{n}}{\sqrt{n}/c}/\binom{\sqrt{n}}{\sqrt{n}/c}) \\ &\in \Omega(\sqrt{n}). \end{aligned}$$

Theorem 12. *Given an unlabeled permutation of n elements, $\Theta(\sqrt{n})$ bits are necessary and sufficient for storing the permutation if each element is to be given a unique label in the range $\{1, \ldots, cn\}$ for any constant $c > 1$. Moreover, there is a structure of $\Theta(\sqrt{n})$ bits such that $\pi^k()$ can be computed in $O(1)$ time.*

3.6.2 Lower Bound for Auxiliary Data with Label Space $cn^{1+\varepsilon}$

Using techniques that are similar to the techniques presented in the previous subsection, we show that for the problem of representing unlabeled permutations an auxiliary data structure of size $O(n^{(1-\varepsilon)/2})$ bits is necessary when the label space is $cn^{1+\varepsilon}$, where c is any constant and $0 < \varepsilon < 1$.

Denote by $S_{cn^{1+\varepsilon}}$ the set of all partitions of $cn^{1+\varepsilon}$ and by S_n the set of all partitions of n. We argue that at least $\binom{(c+1)n^{(1+\varepsilon)/2}}{n^{(1-\varepsilon)/2}/(c+1)}/\binom{cn^{(1+\varepsilon)/2}/(c+1)}{n^{(1-\varepsilon)/2}/(c+1)}$ are necessary to dominate all partitions of n. Let $\mathcal{S}$ be the smallest set of partitions of $cn^{1+\varepsilon}$ that dominates all partitions of n. We claim that:

Lemma 13. $|\mathcal{S}| \geq \binom{(c+1)n^{(1+\varepsilon)/2}}{n^{(1-\varepsilon)/2}/(c+1)}/\binom{cn^{(1+\varepsilon)/2}/(c+1)}{n^{(1-\varepsilon)/2}/(c+1)}$.

Proof. Divide n into $n^{(1-\varepsilon)/2}/(c+1)$ parts each of size $(c+1)n^{(1+\varepsilon)/2}$. Let Q be the set formed by filling each part with a distinct size in the range $[1, (c+1)n^{(1+\varepsilon)/2}]$, clearly $|Q| = \binom{(c+1)n^{(1+\varepsilon)/2}}{n^{(1-\varepsilon)/2}/(c+1)}$.

A partition of $cn^{(1+\varepsilon)}$ can dominate at most $cn^{(1+\varepsilon)/2}/(c+1)$ distinct parts, hence at most $\binom{cn^{(1+\varepsilon)/2}/(c+1)}{n^{(1-\varepsilon)/2}/(c+1)}$ partitions in Q. Therefore, to dominate Q we need a minimum of $\binom{(c+1)n^{(1+\varepsilon)/2}}{n^{(1-\varepsilon)/2}/(c+1)}/\binom{cn^{(1+\varepsilon)/2}/(c+1)}{n^{(1-\varepsilon)/2}/(c+1)}$ partitions of $cn^{(1+\varepsilon)/2}$. Since Q is a subset of S_n our claim holds. □

The information theoretic lower bound for space to represent a permutation of size n once labels are assigned from the set $\{1, \ldots, cn^{1+\varepsilon}\}$ is

$$\begin{aligned} \lg(|\mathcal{S}|) &\geq \lg(\binom{(c+1)n^{(1+\varepsilon)/2}}{n^{(1-\varepsilon)/2}/(c+1)}/\binom{cn^{(1+\varepsilon)/2}/(c+1)}{n^{(1-\varepsilon)/2}/(c+1)}) \\ &\in \Omega(n^{(1-\varepsilon)/2}). \end{aligned}$$

Theorem 14. *Given an unlabeled permutation of n elements, $\Theta(n^{(1-\varepsilon)/2})$ bits are necessary and sufficient for storing the permutation if each element is to be given a unique label in the range $\{1, \ldots, cn^{1+\varepsilon}\}$ for any constant $c > 1$ and $\varepsilon < 1$. Moreover, there is a structure of $\Theta(n^{(1-\varepsilon)/2})$ bits such that $\pi^k()$ can be computed in $O(1)$ time.*

3.7 Application

As an application to our data structures, we give a representation of a labeled permutation that takes $s(n) + O(\sqrt{n})$ bits and can answer $\pi^k()$ in $O(t_f + t_i)$ time, where $s(n)$ denotes the number of bits required for a representation R to store a labeled permutation, and t_f and t_i are the time needed for R to support $\pi()$ and $\pi^{-1}()$.

This result improves Theorem 3.3 in [74] which says that suppose there is a representation R taking $s(n)$ bits to store an arbitrary permutation π on $\{1, \ldots, n\}$, that supports $\pi()$ in time t_f, and $\pi^{-1}()$ in time t_i. Then, there is a representation for an arbitrary permutation on $\{1, \ldots, n\}$ taking $s(n) + O(n \lg n / \lg\lg n)$ bits in which $\pi^k()$ can be supported in $t_f + t_i + O(1)$ time, and one taking $s(n) + O(\sqrt{n} \lg n)$ bits in which $\pi^k()$ can be supported in $t_f + t_i + O(\lg\lg n)$ time.

Theorem 15. *Suppose there is a representation R taking $s(n)$ bits to store an arbitrary permutation π on $\{1, \ldots, n\}$, that supports $\pi()$ and $\pi^{-1}()$ in time t_f and t_i. Then there is a representation for an arbitrary permutation on $\{1, \ldots, n\}$ taking $s(n) + O(\sqrt{n})$ bits in which $\pi^k()$ can be supported in $t_f + t_i + O(1)$ time.*

Proof. Given π, treat it as an unlabeled permutation and build the data structure from Theorem 8 on it. Call this structure P. Notice that the bijection between the labels generated by P and the real labels of π forms a permutation. Store this permutation using the given scheme in a structure P'. Now $\pi^k(i) = \pi_{P'}^{-1}(\pi_P^k(\pi_{P'}^1(i)))$ can be computed in $t_f + t_i + O(1)$ time, and the total space used is $s(n) + O(\sqrt{n})$ bits. $\square$

Chapter 4

Powering Permutations

4.1 Introduction and Motivation

In this chapter, we study the problem of transforming a permutation π to its k^{th} power π^k in place for arbitrary k. By "in place," we mean that the algorithm runs while using "very little" extra space. Ideally, we want the algorithm to use only a polylogarithmic number of bits in addition to the input. The algorithm we present uses several new techniques that are of interest in their own right and could find broader applications. We note that this work is an extension to the work done by Robertson [89].

One interesting application of inverting a permutation in place was encountered in the context of data ware-housing by Aruna, Inc. [24]. The permutation which corresponds to the rows of a relation sorted by some given key is stored explicitly. The inverse of a segment of the permutation is required to perform certain joins. The amount of space occupied by this permutation is substantial, and doubling that space to store the permutation inverse for the purpose of improving the time to compute certain joins is not practical, and indeed was not in the work leading to [24].

As mentioned in the previous chapter, since there are $n!$ permutations of length n, the number of bits required to represent a permutation is $\lceil \lg(n!) \rceil \sim n \lg n - n \lg e + O(\lg n)$ bits. Munro et al. [74] studied the space efficient representation of general permutations where general powers of individual elements can be computed quickly. They gave a representation taking the nearly optimal $\lceil \lg(n!) \rceil + o(n)$ bits, that can compute the image of a single element of $\pi^k()$ in $O(\lg n/\lg\lg n)$ time; and a representation taking $(1+\varepsilon)n\lg n$ bits where $\pi^k()$ can be computed in constant time. The preprocessing for these representations as presented in [74] requires an extra $O(n)$ words of space, so a solution that involves building them as an intermediate step will not be considered in place and therefore does not apply to our current problem.

Throughout this chapter, we assume that the permutation is stored in its *standard representation*. That is, it is stored in an array $A[1,\ldots,n]$ of n words that contains the value $\pi(i)$ at index i for $i \in \{1,\ldots,n\}$. At the termination of the algorithm this array will contain the value $\pi^k(i)$ at index i for $i \in \{1,\ldots,n\}$. Storing A requires $n\lceil\lg n\rceil = n\lg n + n(\lceil\lg n\rceil - \lg n)$ bits. When $(\lceil\lg n\rceil - \lg n)$ is "big," we can reduce the space required by this representation by encoding a constant number c of consecutive elements into a single object. This object is essentially the c-digit base n number $\pi[i]\pi[i+1]\ldots\pi[i+c-1]$. Encoding these n/c objects of size $\lceil c\lg n\rceil$ bits each, totals to $n\lg n + n/c$ bits (which is still more than the optimal representation by $(n/c + n\lg e - O(\lg n))$ bits). To decode a value, we need a constant number of arithmetic operations. This saving of memory at the cost of c accesses to interpret one element of A carries through all of our work.

This chapter is organized as follows. In Section 4.2, we review previous work on permuting data in place [38], on which we base our work. In Section 4.3, we present an algorithm for inverting permutations with a worst case time complexity of $O(n\lg n)$ using only $O(\lg^2 n)$ additional bits. Then we face the problem that while $\pi^{-1}()$ leaves the cycle structure as it was, higher powers may create more (smaller) cycles. This causes further difficulty which is addressed in Section 4.4 where we generalize the algorithm from Section 4.3 to the situation in which the permutation is to be replaced by its k^{th} power. An algorithm whose worst case running time is $O(n\lg n)$ and uses $O(\lg^2 n + \min\{k\lg n, n^{3/4+\varepsilon}\})$ additional bits is presented. Our solution relies on Rubinstein's [90] work on finding factorizations into small terms modulo a parameter. The final result can be improved if better factorization is applied. However, we show that obtaining a better factorization is probably difficult since it would imply Vinogradov's conjecture [98]. We conclude this chapter in Section 4.5.

We note that the results of this chapter are published in [33].

4.2 Background and Related Work

Fich et al. studied the problem of permuting external data in place according to a given permutation [38]. That is, given an array B of length n and a permutation π given by an oracle or read only memory, rearrange the elements of B in place according to π.

It is not sufficient to simply assign $B[\pi(i)] \leftarrow B[i]$ for all $i \in \{1, \cdots, n\}$, because an element in B may have been modified before it has been accessed. A permutation can be thought of as a collection of disjoint cycles. The procedure ROTATE rotates the values in B according to π by calling ROTATECYCLE on the leader of each cycle as illustrated in Figure 4.1. A *cycle leader* is a uniquely identifiable element in each cycle. The smallest element in a cycle, or *cycle minimum*, is a simple example of a cycle leader; though it does have shortcomings in term of detection.

```
procedure ROTATE(B)
    for i ← 0 to n − 1 do
        if ISLEADER(i) then
            ROTATECYCLE(B, i)
```

```
procedure ROTATECYCLE(B, leader)
    i ← π(leader)
    while i ≠ leader do
        SWAP(B[i], B[leader])
        i ← π(i)
```

Figure 4.1: Procedures to rotate the values in B according to a permutation π.

Each element will be tested to see whether it is a cycle leader, by traversing its cycle only in the forward direction until we determine the element is the cycle leader or that it is not. Clearly, the cycle minimum as leader would take $\Theta(n^2)$ value inspections in total in the worst case. A leader that we call the *local min leader* can be used to permute data in $O(n \lg n)$ worst case time complexity using only $O(\lg^2 n)$ additional bits [38]. We use the name local min leader since from that leader in a cycle we are able to identify local minima, as explained next. As stated in [38], the local min leaders of a permutation π are characterized as follows. Let $E_1 = \{1, \ldots, n\}$ and $\pi_1 = \pi$. For positive integers $r > 1$, define E_r as the set of local minima in E_{r-1} encountered following the cycle representation of the permutation π_{r-1} and define π_r as the permutation that maps each element of E_r to the next element of E_r that is encountered following π_{r-1}. More formally, $E_r = \{i \in E_{r-1} | \pi_{r-1}^{-1}(i) > i < \pi_{r-1}(i)\}$ and $\pi_r : E_r \to E_r$ is defined such that $\pi_r(i) = \pi_{r-1}^m(i)$ where $m = \min\{m > 0 | \pi_{r-1}^m(i) \in E_r\}$. Since at most half the elements in each cycle are local minima, $|E_r| < |E_{r-1}|/2$ and $r \leq \lg n$. The local min leader of a cycle is the unique element i, such that $\pi_{r-1}(\ldots(\pi_1(i))) \in E_r$. For example, if $\pi = (1\ 7\ 2\ 9\ 4\ 5\ 3\ 10\ 6\ 8)$ as illustrated in Figure 4.2 (similar to Figure 6 in [38]), then

$$E_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \qquad \pi_1 = (1\ 7\ 2\ 9\ 4\ 5\ 3\ 10\ 6\ 8)$$
$$E_2 = \{1, 2, 4, 3, 6\}, \qquad \pi_2 = (1\ 2\ 4\ 3\ 6)$$
$$E_3 = \{1, 3\}, \qquad \pi_3 = (1\ 3)$$
$$E_4 = \{1\}, \qquad \pi_4 = (1)$$

The local min leader of the only cycle in π is the element 9 since $\pi_3\pi_2\pi_1(9) = 1$.

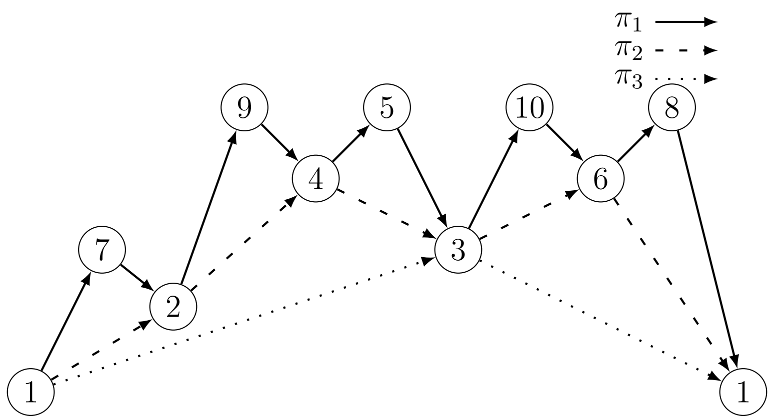


Figure 4.2: The cycles generated from π.

The procedure IsLocalMinLeader (see Figure 4.3) checks if element i in the permutation is the local min leader of its cycle. The procedure executes at most $4n$ steps on the permutation for a single element, and a total of $O(n \lg n)$ steps on the permutation for all elements ([38], Theorem 2.3). We treat the local min leader technique as a black box. There are a few occasions where we need details so we provide the procedure to make this chapter more self contained.

```
procedure IsLocalMinLeader(i)
    elbow[0] ← elbow[1] ← i
    for r ← 1, 2, ... do
        //loop invariant:
        {elbow[r] = π_{r−1} ... π_1(i)}
        Next(r)
        if elbow[r] > elbow[r − 1] then
            elbow[r] ← elbow[r − 1]
            Next(r)
            if elbow[r] > elbow[r − 1] then
                return false
            elbow[r + 1] ← elbow[r]
      else if elbow[r] = elbow[r − 1] then
            return true

procedure Next(r)
    if r = 1 then
        elbow[0] ← π(elbow[1])
    else
        while elbow[r − 1] < elbow[r − 2] do
            elbow[r − 1] ← elbow[r − 2]
            Next(r − 1)
        while elbow[r − 1] > elbow[r − 2] do
            elbow[r − 1] ← elbow[r − 2]
            Next(r − 1)
```

Figure 4.3: Procedures to check if element i is a local min leader.

4.3 Inverting Permutations

In this section we present an algorithm for inverting a permutation that uses $O(\lg^2 n)$ additional bits and runs in $O(n \lg n)$ time. We note that this algorithm is a modified version of the algorithm presented in [89]. The modifications are needed to make the analysis of the algorithm correct.

As a warm-up we first review two algorithms presented in [89]. The first explained in this section uses $O(b + \lg n)$ additional bits for any $b \leq n$ and runs in $O(n^2/b)$ worst case time. The second explained in section 4.3.1 runs in $O(n \lg n)$ time, but using $O(\sqrt{n} \lg n)$ additional bits.

To invert a permutation we can use the structure of the algorithm described in Figure 4.1, but invert the cycles instead of rotating the data. Figure 4.4 shows how to invert a cycle. The algorithm checks each element from 0 to $n - 1$ to see if it is a cycle leader, and inverts each cycle only on its leader. For this approach to work, a cycle leader must be used that will remain unchanged once the cycle is inverted. An example of such a cycle leader is the cycle minimum.

```
procedure InvertCycle(A, leader)
    current ← A[leader]
    previous ← leader
    while current ≠ leader do
        next ← A[current]
        A[current] ← previous
        previous ← current
        current ← next
    A[leader] ← previous
```

Figure 4.4: Procedure to invert a cycle.

Inverting a permutation using cycle minimum as a leader will use $O(\lg n)$ additional bits and take $\Theta(n)$ time if the permutation consists of one large cycle in increasing order; or $\Theta(n^2)$ time if the permutation consists of one large cycle in decreasing order. We note that for a random cycle of length n this total cost would be about $\Theta(n \lg n)$. The analysis is similar to the bidirectional distributed algorithm for finding the smallest of a set of n uniquely numbered processors arranged in a circle [56]. However, our interest is in finding algorithms with good worst case performance.

We can invert a permutation in linear time using a n-bit vector. We iterate over the permutation checking each element from 0 to $n-1$, if its corresponding bit is not marked its cycle is inverted. As the cycle is inverted, we mark the bits corresponding to the elements in the cycle. Since each cycle will be traversed once, the total runtime is $O(n)$.

Using a technique presented in [38], we can shrink the bit vector to b bits by conceptually dividing the permutation into $\lceil n/b \rceil$ sections each of size b (except possibly the last section will be smaller). We reset the b-bit vector at the start of each section and use it to keep track of which elements are encountered in the section being processed. We iterate over the permutation checking each element from 0 to $n-1$. If the element under consideration for being a cycle leader has a corresponding bit with value 0, we traverse its cycle searching for a smaller element. As the cycle is traversed, we mark the bits corresponding to the elements in the cycle of the current section. If no smaller element is found, then the element is a cycle leader and the cycle is inverted. On the other hand, if the element under consideration has a corresponding bit with value 1, then the element was previously encountered as part of a cycle containing a smaller element in the section, and hence it is not a cycle leader. Each cycle will be traversed at most n/b times, thus the total runtime is n^2/b and the space used is $b + O(\lg n)$.

Theorem 16. [89] *In the worst case a permutation of length n stored in its standard representation can be replaced with its inverse in $O(n^2/b)$ time using $b + O(\lg n)$ extra bits of space for any integer b.*

By setting $b = \sqrt{n}$ we get the following corollary.

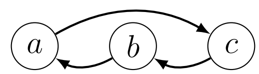
Corollary 17. [89] *In the worst case a permutation of length n stored in its standard representation can be replaced with its inverse in $O(n\sqrt{n})$ time using $O(\sqrt{n})$ extra bits of space.*

4.3.1 Inversion in $O(n \lg n)$ Time Using $O(\sqrt{n} \lg n)$ Bits

In this section we continue our revision of [89] and present an algorithm for inverting permutations that runs in $O(n \lg n)$ time using $O(\sqrt{n} \lg n)$ additional bits.

The local min leader of a cycle will, in general, change after the cycle has been inverted. Figure 4.5 shows a simple example of this: b is the leader of the cycle, but if it were inverted, c would become the leader. Since $c > b$, the algorithm in Figure 4.4 will invert the cycle once on b and then again on c because c will look like a leader when it is reached in the outer loop. Inverting the cycle the second time will undo the work of inverting it the first time. We will call a cycle with this problem a *bad cycle*.

Definition 1. [89] *A* bad cycle *is a cycle with the property that if inverted, it has a new cycle local min leader not yet processed, i.e., larger than the original leader.*



$a < b < c$

Figure 4.5: [89] An example of a bad cycle.

It is not hard to build a permutation that will have $\Theta(n)$ bad cycles. Such a permutation could just repeat our bad cycle pattern and create exactly $\lfloor n/3 \rfloor$ bad cycles. So, there is not enough space to use even 1 bit to mark these cycles.

Definition 2. [89] *A* tail *of a cycle is the element that points to its local min leader, i.e., if t is the tail of a cycle c with local min leader l, then $\pi(t) = l$.*

Theorem 18. [89] *In the worst case a permutation of length n stored in its standard representation can be replaced with its inverse in $O(n \lg n)$ time using $O(\sqrt{n} \lg n)$ extra bits of space.*

Proof. Although the permutation π can contain up to n cycles, the number of distinct cycle lengths in π which we denote by k is less than $\lfloor \sqrt{2n} \rfloor$ (since $\sum_{i=1}^{\lceil \sqrt{2n} \rceil} i > n$). First, we store these cycle lengths in an array L of size $O(\sqrt{n} \lg n)$ bits. This can be done in $O(n \lg n)$ time by iterating over the permutation and computing the length of every cycle as it is detected on its local min leader using the procedure IsLocalMinLeader (see Figure 4.3). After a length is detected, we query a balanced binary search tree H to check if the length computed was already encountered; if it was not encountered, we insert the new length to L and H. The cycles's lengths are ranked according to their position in L.

We iterate over the permutation checking each element from 0 to $n-1$, and we invert each cycle only on its local min leader. To check if a cycle is bad we can test each element in the inverted cycle for leadership to find the inverted cycle local min leader. If a bad cycle c was detected, we modify the tail of the inverted cycle c^{-1} to point to the rank of the length of the cycle instead of back to the leader of the inverted cycle. That is, if we find that element i is the local min leader of cycle c (of length l), we invert c. If j is the leader of c^{-1} and $j > i$, we set $A[m] = \text{rank}\,(l)$ where m is the element in c^{-1} that points to j and $\text{rank}\,(l)$ is the index in L such that $L[\text{rank}\,(l)] = l$.

When pointing to the ranks of the cycles' lengths we have to use values in the range of 1 to n, otherwise the size of each entry in A may increase to $\lceil \lg n \rceil + 1$ bits and we may end up using n additional bits. The problem now is that A does not distinguish between pointing to a cycle length rank, or pointing to a different element in the cycle. This can be solved with a table T of size $O(\sqrt{n}\lg n)$ bits that stores the elements of the permutation that point to its first k elements. T will initially store $\pi^{-1}(1), \dots, \pi^{-1}(k)$. We set T initially by traversing the permutation, then we update it as cycles are inverted.

While testing for the leadership of an element i, if an element t is found such that $A[t] \leq k$, then t can be checked against T in $O(1)$ time to determine if $A[t]$ points to a cycle length rank or an element in the cycle. If it is the first case we abort the procedure IsLocalMinLeader and we do not invert the cycle. If the length traversed so far matches the cycle length stored in L at rank $A[t]$, then the element i is the local min leader of an already inverted cycle. We restore the cycle by setting $A[t] = i$.

The total time spent is $O(n \lg n)$, and the space used is $O(\sqrt{n}\lg n + \lg^2 n)$. □

4.3.2 Reducing Extra Space to $O(\lg^2 n)$ Bits

Next, we extend the approach presented in the previous subsection to achieve an algorithm for inverting permutations with $O(n \lg n)$ worst case time complexity while using only $O(\lg^2 n)$ bits. First we start with some definitions.

Given a permutation π, the *depth* of an element $e \in \pi$ is the maximum index d such that $\pi_{d-1}(\dots(\pi_2(\pi(e)))) \in E_d$.[1] For example, the depth of 10 in Figure 4.2 is 3 since $\pi_2(\pi(10)) = 1 \in E_3$ and $\pi_3(\pi_2(\pi(10))) = 3 \notin E_4$. Let c be a cycle in π of size l with local min leader s_1. We define S_1 as the following sequence: $s_1, s_2, \dots, s_l$ where $s_i = \pi(s_{i-1})$ for $i > 1$; s_l is the tail of the cycle c. For $i > 1$, S_i is a subsequence of S_{i-1} formed by the local minima in S_{i-1} excluding S_{i-1}'s first and last elements. The *limited depth* of an element $e \in \pi$ is the maximum index d such that $\pi_{d-1}(\dots(\pi_2(\pi(e)))) \in S_d$. The values $s_1, \dots, s_{i-1}$ are not needed to evaluate the limited depth of s_i, but only the values $s_i, \dots, s_l$ are required. The limited depth of an element is upper bounded by its depth. Notice that the first element in S_i is always $\pi_{i-1}(\dots(\pi(s_1))$, since s_1 is the local min leader of c. Moreover, the limited depth d of a cycle's local min leader is either unique or shared by at most one other element $\pi_1^{-1}(\dots(\pi_{d-1}^{-1}(\pi_d(\dots(\pi_2(\pi(s_1)))))))$ in the cycle. That's because if there are more than two elements in S_d, the limited depth of s_1 will be at least $d+1$. The depth and limited depth of an element can be computed in a manner similar to the procedure IsLocalMinLeader with the same space and time complexity.

We say that a cycle is *broken* if its tail points to an element other than its local min leader. We call this element the broken cycle's *intersection*. We define the *spine*

[1]For the definition of π_i where $i \in \{1, \dots, d\}$ check Section 4.2.

to be the path from the leader to the intersection, and the *loop* to be the cycle containing the intersection and the tail. Figure 4.6 demonstrates these terms.

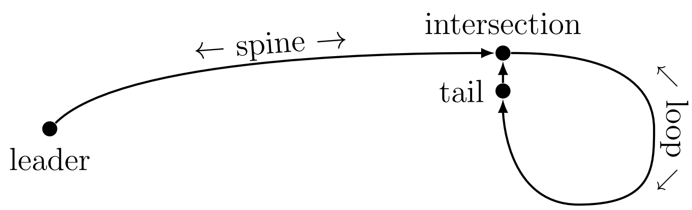


Figure 4.6: [89] An example of a broken cycle.

Following the algorithm described previously, when a cycle c is detected it is replaced by its inverse; if c is detected to be a bad cycle, the tail of c^{-1} is modified to store the limited depth of c^{-1}'s local min leader k. In that case, the tail of c^{-1} will be modified to point to the unique element whose limited depth is the same as k if that element was encountered before k, thus making c^{-1} a broken cycle. Finally, c^{-1} will be restored once k is encountered. As in the previous subsection, for A to distinguish between pointing to a limited depth, or pointing to a different element in the cycle we use a table T of size $O(\lg^2 n)$ bits that stores the elements of the permutation that point to its first $\lg n$ elements.

The algorithm iterates over the permutation checking each element from 0 to $n-1$. At each element i, it interleaves four scans $\mathscr{F}$, $\mathscr{L}$, $\mathscr{T}$ and $\mathscr{H}$. For every operation run on $\mathscr{F}$, a constant number of operations are run on $\mathscr{L}$; and for every operation run on $\mathscr{L}$ a constant number of operations are run on $\mathscr{T}$ and $\mathscr{H}$. $\mathscr{F}$ is used to determine whether i is the local min leader of its cycle (c or c^{-1}), $\mathscr{L}$ is used to determine the limited depth of i, and $\mathscr{T}$ and $\mathscr{H}$ are used to determine if i's cycle was broken, and to restore it. The $\mathscr{T}$ and $\mathscr{H}$ scans have two phases:

- The first phase is the classic tortoise and hare algorithm for cycle detection. It is used to check if i's cycle is broken. $\mathscr{T}$ (for tortoise) and $\mathscr{H}$ (for hare) both start at element i, $\mathscr{T}$ proceeds at one step per iteration and $\mathscr{H}$ proceeds at two steps until they meet at element j. Phase one will consist of no more than l iterations, where l is the length of i's cycle. This is because at each iteration, the forward distance (i.e. the distance from $\mathscr{H}$ to $\mathscr{T}$ traversing forward in the cycle) between the two pointers will decrease by one; or if the cycle was broken, the distance decreases once both pointers enter the broken cycle's loop. If one of the scans encounters a limited depth or if i is reachable from j, $\mathscr{T}$ and $\mathscr{H}$ are aborted while $\mathscr{F}$ and $\mathscr{L}$ continue. Otherwise, we know that the cycle is broken and we proceed to the second phase.

- The aim of the second phase is to find the tail of the broken cycle c^{-1}. Let λ be the length of c^{-1}'s loop, μ be the distance from i to c^{-1}'s intersection, and δ be the distance from the intersection to j. Denote by d_t and d_h the distance traveled by the pointers in $\mathscr{T}$ and $\mathscr{H}$ respectively. $d_t = \mu + \delta$ and $d_h = \mu + k\lambda + \delta$ where $k \in \mathbb{Z}^+$. We know

$$\begin{aligned} 2d_t &= d_h \\ 2(\mu + \delta) &= \mu + k\lambda + \delta \\ \mu &= k\lambda - \delta \ . \end{aligned}$$

  Thus, if we reset $\mathscr{T}$'s pointer to element i, while $\mathscr{H}$ remains at j, and as in the first phase, $\mathscr{T}$ proceeds at one step per iteration and $\mathscr{H}$ proceeds at two steps: $\mathscr{T}$ and $\mathscr{H}$ will meet at c^{-1}'s intersection. Then, c^{-1}'s tail can be found by iterating through c^{-1}'s loop till an element that points to the intersection is reached. After finding the tail, the limited depth of the intersection (which will always be the same as the limited depth of c^{-1}'s leader) is computed.

The $\mathscr{L}$ scan aims to compute the limited depth of element i. To do so, $\mathscr{L}$ should identify the tail of c or c^{-1}. $\mathscr{L}$ identifies the tail correctly if it encounters an element storing a limited depth (then that element is the tail), or if the cycle is broken and the tail is computed by the $\mathscr{T}$ and $\mathscr{H}$ scans (as is the case when the cycle is broken and i is on its spine). In the other cases, the $\mathscr{L}$ scan assumes that the tail is the element pointing to i. It returns a correct value if i is a local min leader, and it may not return a correct value otherwise. However, returning an incorrect value in the other cases does not affect the correctness of the algorithm.

The $\mathscr{F}$ scan tests whether i is the local min leader of c or c^{-1}. If $\mathscr{F}$ encounters a limited depth or if the scans $\mathscr{T}$ and $\mathscr{H}$ detect that c^{-1} is broken, $\mathscr{F}$ will behave as if the tail of c^{-1} points to i. The $\mathscr{F}$ scan terminates on one of the following cases:

- The first case is $\mathscr{F}$ determines that i is not a local min leader. If so, the entire process of all four scans is aborted.

- The second case is $\mathscr{F}$ determines the element is a local min leader. Then, two cases can occur:

  - If c^{-1} was broken or a limited depth was encountered, then we know that the cycle is already inverted. Compare the limited depth of i that is computed by $\mathscr{L}$ to the limited depth stored or computed by $\mathscr{T}$ and $\mathscr{H}$. If the two values are equal make the tail point to i. Alternatively, abort all four scans.

- Otherwise, the cycle c is not inverted. Invert c and if it was bad store in its tail the limited depth of c^{-1}'s local min leader.

Analysis: All four scans use $O(\lg^2 n)$ extra bits. The time complexity is bounded by the time complexity of $\mathscr{F}$, since the runtime of $\mathscr{L}$, $\mathscr{T}$ and $\mathscr{H}$ is at most a constant factor times the runtime of $\mathscr{F}$. For each cycle c, the time spent by F testing for leadership before inverting the cycle is $O(l \lg l)$ where l is the length of c. Inverting c and properly setting its tail if it was bad will take $O(l)$ time. After inverting c, if c^{-1} is bad at most one intermediate broken cycle can be formed, since the limited depth of the local min leader is unique or shared by at most one other element. This fact is crucial to our analysis, and it is the reason why the $\mathscr{L}$ scan is introduced. The time spent testing for leadership for indices in c^{-1} is divided into the following cases:

- c^{-1} is broken and the element i being tested is in c^{-1}'s loop.
- Otherwise either c^{-1} is broken and i is in the spine, or c^{-1} is not broken and the tail stores the limited depth of the leader.
  - If $\mathscr{T}$ does not inspect the tail, then the runtime will be the same as testing whether i is the local min leader of c^{-1}.
  - Otherwise, the procedure will test if i is the local min leader of the cycle formed by pointing the tail of c^{-1} to i. It will iterate at most 4 times from i to the tail [38]. So, the time complexity will be at most 4 times the time complexity of testing weather i is the local min leader of c^{-1}.

In all cases the runtime is bounded by $O(l \lg l)$. Thus, the total runtime per cycle is $O(l \lg l)$ and the total runtime for the whole algorithm is $O(n \lg n)$.

Theorem 19. *In the worst case a permutation of length n stored in its standard representation can be replaced with its inverse in $O(n \lg n)$ time using $O(\lg^2 n)$ extra bits of space.*

4.4 Arbitrary Powers

As in the previous Chapter, the k^{th} power of a permutation π is π^k defined as follows:

$$\pi^k(i) = \begin{cases} \pi^{k+1}(\pi^{-1}(i)) & k < 0 \\ i & k = 0 \\ \pi^{k-1}(\pi(i)) & k > 0 \end{cases}$$

where k is an arbitrary integer. In this section we extend the techniques presented in the previous section to cover the situation in which the permutation is to be replaced by its k^{th} power for an arbitrary integer k. We present an algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\varepsilon}\})$ additional bits.

Without loss of generality, we assume that k is positive. If k is negative, we invert the permutation then raise it to the power of $-k$. Raising a cycle to an arbitrary power can result in several disjoint cycles as illustrated in Figure 4.7.

Lemma 20. *Raising a cycle of length l to its k^{th} power, will produce $\gcd(k, l)$ cycles each of length $l/\gcd(k, l)$.*

Proof. Suppose μ cycles are produced. Since they are all identical, they will have the same length λ. λ is the smallest positive integer such that $(\pi^k)^\lambda(i) = \pi^{k\lambda}(i) = i$, so $k\lambda = cl$ for an integer c that is relatively prime with λ. Now

$$l = \lambda\mu$$
$$k = cl/\lambda = c\mu,$$

but c is relatively prime with λ, so $\mu = \gcd(k, l)$ and $\lambda = l/\gcd(k, l)$. □

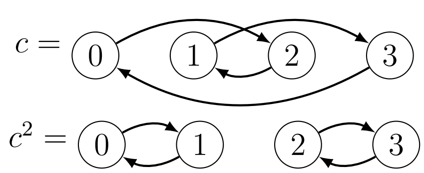


Figure 4.7: The cycles created by raising c to its second power.

Given a cycle, it is not hard to raise the cycle to its k^{th} power while using $O(k)$ words or $O(k \lg n)$ bits. Figure 4.8 shows how to achieve this task. Starting from element i, we store $i, \pi(i), \pi^2(i), \ldots, \pi^{k-1}(i)$ in an array B using $O(k \lg n)$ bits. We replace $A[i]$ with $A[\pi^{k-1}(i)]$, then we replace $A[\pi(i)]$ with $A[\pi^k(i)]$, and so on until we reach $A[\pi(i)^{l-k}]$ where l is the length of the cycle. Then, we replace $A[\pi(i)^{l-k}]$ till $A[\pi(i)^{l-1}]$ with the values stored in B. When the procedure terminates, $A[i], A[i+1], \ldots, A[i + \gcd(k, l) - 1]$ will contain an element from each resulting cycle.

To raise a permutation to its k^{th} power, we use the same algorithm as the one presented in subsection 4.3.2, however, we modify the $\mathcal{T}$ scan so that it raises cycles to their k^{th} power instead of inverting them once they are detected at their leaders. Furthermore, once the

```
procedure PowerCycle(A, k, leader)
    power ← leader
    count ← 0
    while count < k do
     temp[count] ← power
     power ← A[power]
     count ← count + 1
    while count < cycleLength do
     next ← A[leader]
     A[leader] ← power
     leader ← next
     power ← A[power]
     count ← count + 1
    count ← 0
    while count < k do
     next ← A[leader]
     A[leader] ← temp[count]
     leader ← next
     count ← count + 1
```

Figure 4.8: Procedure to raise a cycle to its k^{th} power.

$\mathcal{T}$ scan raises a cycle (of size l) to its k^{th} power, it iterates through every cycle of the resulting $\gcd(k, l)$ cycles computing each cycle's leader and checking which cycles are bad; for each bad cycle, the $\mathcal{T}$ scan computes the cycle's limited depth and stores that value in the cycle's tail.

Theorem 21. *In the worst case a permutation of length n stored in its standard representation can be replaced with its k^{th} power when k is bounded by some polynomial function of n in $O(n \lg n)$ time using $O(\lg^2 n + k \lg n)$ extra bits of space.*

Theorem 21 is useful if the value of k is small. In the next subsection, we show how to power permutations using $o(n)$ extra bits of space.

4.4.1 Powering Permutations in $O(n \lg n)$ Time using $o(n)$ Extra Bits

To improve the space complexity we only have to modify the way we are raising cycles to their k^{th} power. To raise a cycle c of length l to its k^{th} power, we split the algorithm into two cases.

- **First Case: k and l are relatively prime**
  In this case, we use the following theorem given by Rubinstein [90]:

  Theorem 22 (Rubinstein [90], Theorem 4.3)**.** *Let* $\gcd(N, a) = 1$ *and* R *be a rectangle. Then,* $c_R(N, a)$*, the number of solutions* (x, y) *to* $xy = N \bmod a$ *with* (x, y) *lying in the rectangle* R *is equal to*

  $$\frac{\text{area}(R)}{a^2}\phi(a) + O(a^{1/2+\varepsilon})$$

  for any $\varepsilon > 0$*, where* ϕ *is Euler's totient function.*

  In particular, there exists a point (x, y) *where* $xy = N \bmod a$ *in any square* R *with side length at least* $a^{3/4+\varepsilon}$ *(*R *must be larger than* $a^{3/2+\varepsilon}$*).*

  In this case $\gcd(k, l) = 1$ so there always exist two integers $x, y < l^{3/4+\varepsilon}$ such that $xy = k \bmod l$. To find x and y we do a linear search which takes $O(l^{3/4+\varepsilon})$ time. Then we raise c to the x^{th} power followed by the y^{th} power using the method described in the previous subsection. The total runtime is $O(l)$ and the space used is $O(l^{3/4+\varepsilon})$.

- **Second Case: k and l have a common factor other than 1**
  In this case $\gcd(k, l) = f > 1$. We first raise c to its f^{th} power producing f different cycles using a reduction to the first case. Then, we raise each of the f resulting cycles to its $(d = (k/f))^{\text{th}}$ power.

  We modify the permutation π to form the permutation π' that results from adding an additional element e to the cycle c in π to form the cycle c' in π'. More formally, π' is defined as follows:

  - Let a be an element in the cycle c; for all elements $i \in \pi$ except $\pi^{-1}(a)$, $\pi'(i) = \pi(i)$.
  - $\pi'(\pi^{-1}(a)) = e$ (where e is a new element).
  - $\pi'(e) = a$.

This modification can be done by storing a and two extra words where the first word stores the inverse of a, and the second stores the image of e ($\pi'(e)$). Each time the array A is accessed at an index i, if $A[i]$ is equal to a, i is checked against the first word stored. If they match, then $A[i]$ points to a otherwise $A[i]$ points to e. Doing this eliminates the need for increasing the word size.

We rename the elements in c to reflect how they get split to different cycles once c is raised to its f^{th} power. Let $\{c_{ij} | 0 \leq i < l/f, 0 \leq j < f\}$ be the elements of c, such that

- $\pi(c_{ij}) = c_{i(j+1)}$ if $j < f - 1$
- $\pi(c_{ij}) = c_{(i+1 \bmod l/f)0}$ if $j = f - 1$

Raising c to its f^{th} power will result in f cycles such that the j^{th} cycle c_j will contain the elements $\{c_{ij} | 0 \leq i < l/k\}$, where $\pi^f(c_{ij}) = c_{(i+1 \bmod l/f)j}$. This naming can be observed in Figure 4.9.

Without loss of generality assume that $a = c_{00}$. Since the length of c' is $l + 1$ and $\gcd(l + 1, f) = 1$ (since f divides l), raising c' to its f^{th} power will result in only one cycle. Observe that if we traverse forward in c'^f starting from e, the first l/f elements are $c_{0(f-1)}, c_{1(f-1)}, \ldots, c_{((l/f)-1)(f-1)}$. That is, the elements in c_{f-1} ordered correctly. Moreover, the next l/f elements are the elements of c_{f-2}, and so on...

After modifying π to π' we raise c' to its f^{th} power using the same technique presented in the first case. Then, we iterate l/f elements starting from e, we set $A[c_{((l/f)-1)(f-1)}]$ to $c_{0(f-1)}$ and we raise c_{f-1} to its d^{th} power also using the same technique presented in the first case. We find the local min leader of c_{f-1} and store the limited depth of the leader in the tail of c_{f-1} if c_{f-1} is a bad cycle. We then repeat the same process for the rest of the cycles $c_{f-2}, \ldots, c_0$. This process is illustrated in Figure 4.9.

Theorem 23. *In the worst case a permutation of length n stored in its standard representation can be replaced with its k^{th} power when k is bounded by some polynomial function of n in $O(n \lg n)$ time using $O(\lg^2 n + \min\{k \lg n, n^{3/4+\varepsilon}\})$ extra bits of space.*

The space complexity in Theorem 23 can be improved if better factoring is applied. More precisely, if for any N and a where $\gcd(N, a) = 1$, we can find $g(a)$ factors $x_1, \ldots, x_{g(a)} \leq f(a)$ such that $x_1 x_2 \ldots x_{g(a)} = N \bmod a$ in $h(a)$ time, then we can achieve an algorithm with running time $O((n + h(n)) \lg n + g(n)n)$ that uses $O(\lg^2 n + \min\{k \lg n, f(n) \lg n\})$ extra bits of space.

Note that given any factoring algorithm as described above, any quadratic non-residue $(\bmod\ p)$ can be factored to factors smaller than $f(p)$. Since at least one of the factors

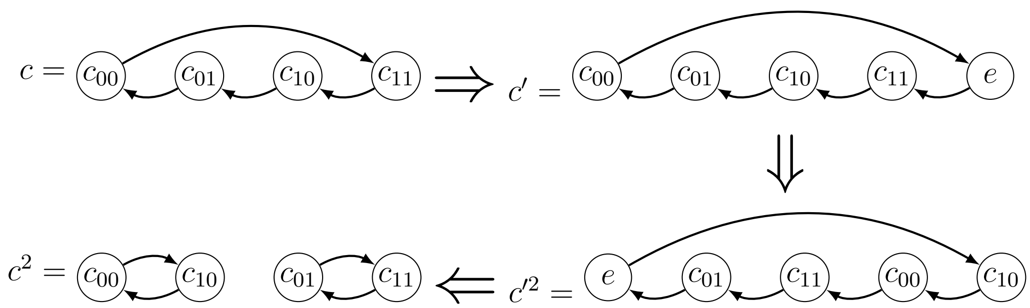


Figure 4.9: The process to raise a cycle to its k^{th} power when the cycle length and k are not coprime.

must also be a quadratic non-residue, this implies that the least quadratic non-residue (mod p) is smaller than $f(p)$. Thus, reducing $f(n)$ to $O(n^{\varepsilon})$ is probably difficult since this improvement would imply Vinogradov's conjecture [98] (that the least quadratic non-residue (mod p) lies below p^{ε}).

4.5 Conclusion

In this chapter we presented an algorithm for inverting a permutation that runs in $O(n \lg n)$ worst case time and uses $O(\lg^2 n)$ additional bits. This algorithm is then extended to an algorithm for raising a permutation to its k^{th} power that runs in $O(n \lg n)$ time and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\varepsilon}\})$ extra bits of space. Both algorithms presented rely on the cycle's local min leader presented in [38]. Moreover, they can easily be adapted to utilize any different cycle leader. A different leader may yield a better algorithm without adding to the worst case time or space complexity for both problems as well as the problem of permuting in place [38].

Chapter 5

Range Mode

5.1 Introduction

In this chapter we investigate data structures for the range mode query problem in a multi-dimensional setting:

Range Mode: Given a set of n points in d dimensions $\mathcal{S}$ such that each point is a assigned a color, a *range mode query* $\mathcal{Q} = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$ asks for the most frequent color in $\mathcal{S} \cap \mathcal{Q}$.

Although the one-dimensional range query problem has received significant attention [20, 64, 84, 85, 47], only limited attention has been paid to the multi-dimensional problem. The first solution for the multi-dimensional case was proposed recently by Chan et al. [20]. They gave a data structure that requires $O(s_n + (n/\Delta)^{2d})$ words and supports d-dimensional range mode queries in $O(\Delta \cdot t_n)$ time for any $\Delta \geq 1$, where s_n is the space of an orthogonal range counting data structure in d dimensions with query time t_n. The model of computation is the standard Word RAM model with word size $w = \Omega(\lg n)$, also d is assumed to be a constant. In this chapter we show that the space of the range mode query data structure can be improved to $O(s_n + (n/\Delta)^{2d}/w)$ words while maintaining the same query time. That is, our data structure achieves the same asymptotic space and query time costs as those of the current best known range mode query data structure for one-dimensional data [20].

We note that the results of this chapter are published in [27, 28].

5.1.1 Related Work

The first range mode data structure (on arrays) was proposed by Krizanc et al. [64], requiring $O(n)$ words for $O(\sqrt{n} \lg\lg n)$ query time. Krizanc et al. also considered data structures that use more than linear space. They described data structures that provide constant query time using $O(n^2 \lg\lg n / \lg n)$ words, and $O(n^\varepsilon \lg n)$ query time using $O(n^{2-2\varepsilon})$ words. Later, Petersen and Grabowski [85] improved the first bound to constant time using $O(n^2 \lg\lg n / \lg^2 n)$ words. Peterson [84] then improved the second bound to $O(n^\varepsilon)$ query time using $O(n^{2-2\varepsilon})$ words for any $\varepsilon \in (0, 1/2]$. Chan et al. [20] further improved the previous bound to $O(n^\varepsilon)$ query time using $O(n^{2-2\varepsilon}/\lg n)$ words. Moreover, using reductions from boolean matrix multiplication, they show that query time significantly lower than $\sqrt{n}$ is unlikely for this problem with linear space [20]. Finally, Greve et al. [47] proved a lower bound of $\Omega(\lg n / \lg(s \cdot w/n))$ time for any data structure that supports range mode queries on arrays using s memory cells of w bits in the cell probe model.

Given a fixed $\alpha \in (0, 1]$ and a range $\mathcal{Q}$, the objective of an approximate range mode query is to return an element whose frequency in $\mathcal{S} \cap \mathcal{Q}$ is at least $\alpha \cdot m$, where m denotes the frequency of the mode of $\mathcal{S} \cap \mathcal{Q}$. Bose et al. [16] gave a data structure that requires $O(n/(1-\alpha))$ words and answers approximate range mode queries in $O(\log\log_{1/\alpha}(n))$ time, as well as a data structure that answers queries in constant time when $\alpha \in \{1/2, 1/3, 1/4\}$, using $O(n \lg n)$, $O(n \lg\lg n)$, and $O(n)$ words respectively. Greve et al. [47] improved previous results by giving a data structure that supports range mode queries in $O(1)$ time using $O(n)$ words when $\alpha = 1/3$, and $O(\lg(\alpha/(1-\alpha)))$ time using $O(n\alpha/(1-\alpha))$ words when $\alpha \in [1/2, 1)$.

Another related question is the problem of finding a least frequent element (with frequency at least one) in a one dimensional range. Chan et al. [21] gave the first solution with linear space and $O(\sqrt{n})$ query time. Later, Durocher et al. [29] improved the query time to $O(\sqrt{n/w})$. Our improved data structure for the range mode query problem is based on the encoding ideas from [29]. See the recent survey by Skala [94] for further reading.

5.2 Framework

A point $p \in \mathcal{S}$ is represented by a $(d+1)$-tuple $(p_1, p_2, \ldots, p_d, p_c)$, where for each i, p_i is p's coordinate in dimension i, and p_c is the color associated with p. When d is a constant, we can map the input set $\mathcal{S}$ to the rank space using standard techniques as described in subsection 2.5, requiring $O(n)$ words of additional space and an $O(\lg n)$ additive increase to query time. Throughout this chapter we assume that points are in the rank space. That is for any point $p \in \mathcal{S}$ and any $i \in \{1, \ldots, d\}$, $p_i \in \{0, \ldots, n-1\}$. Moreover, if $p \neq q$, then $p_i \neq q_i$ for all $i \in \{1, \ldots, d\}$. This ensures the following lemma:

Lemma 24. *The number of points of $\mathcal{S}$ in a rectangle $\mathcal{Q} = [\alpha_1, \beta_1] \times \ldots \times [\alpha_d, \beta_d]$ is at most the minimum element in $\{\beta_i - \alpha_i + 1 \mid 1 \leq i \leq d\}$.*

Definition 3. *Let $\Delta \geq 1$ be an integer. A Δ-box is a region $R = [\alpha_1, \beta_1] \times \ldots \times [\alpha_d, \beta_d]$, where for all i, α_i and β_i are multiples of Δ.*

There are $\Theta((n/\Delta)^{2d})$ distinct Δ-boxes in our grid, which includes empty boxes, i.e., boxes with $\alpha_i = \beta_i$ for some $i \in [1, d]$. Each Δ-box $R = [\alpha_1, \beta_1] \times \ldots \times [\alpha_d, \beta_d]$ can be identified using a unique index, given by:

$$rank(R, \Delta) = \sum_{i=1}^{d} (\alpha_i/\Delta) \cdot \phi^{2i-2} + (\beta_i/\Delta) \cdot \phi^{2i-1}$$

where $\phi = \lfloor n/\Delta \rfloor + 1$. Notice that $rank(R, \Delta)$ can be computed in $O(d)$ time (i.e. constant time when d is a constant) given any R and Δ.

5.3 Data Structure of Chan et al.

In this section we review the data structure presented by Chan et al. [20]. The data structure relies on the following observation [64]. A mode of $\mathcal{Q}_1 \cup Q_2$ (i.e. the most frequent color occurring in $\mathcal{Q}_1 \cup Q_2$) is either a mode of $\mathcal{Q}_1$ or the color of an element in $\mathcal{Q}_2$.

Data Structure The data structure consists of two components:

1. An array A of length $(1 + n/\Delta)^{2d}$, such that $A[i]$ stores a mode of the Δ-box R with $rank(R, \Delta) = i$.
2. For each color c, the data structure maintain an orthogonal range counting data structure over the set of points in $\mathcal{S}$ with color c. The total space and query time can be bounded by s_n and t_n, where s_n is the space of an orthogonal range counting data structure over n points in d dimensions and t_n is its query time.

Thus, the total space used is $O(s_n + (n/\Delta)^{2d})$ words.

Query Algorithm To answer a query $\mathcal{Q} = [a_1, b_1] \times \ldots \times [a_d, b_d]$, we first find the largest rectangle $\mathcal{Q}' = [a'_1, b'_1] \times \ldots \times [a'_d, b'_d]$ inside $\mathcal{Q}$, where $a'_i = \Delta\lceil a_i/\Delta\rceil$ and $b'_i = \Delta\lfloor b_i/\Delta\rfloor$. If $a'_i \geq b'_i$ for some i, then $\mathcal{Q}'$ is empty. Otherwise, a mode of $\mathcal{Q}'$ is given by $A[rank(\mathcal{Q}', \Delta)]$. Recall that $rank(\mathcal{Q}', \Delta)$ can be computed in constant time when d is a constant. Notice that the number of points in the region $\mathcal{Q} \setminus \mathcal{Q}'$ (the region within $\mathcal{Q}$, but outside $\mathcal{Q}'$) is at most $2d\Delta$ (refer to Lemma 24). Moreover, the mode of $\mathcal{Q}$ is either the mode of $\mathcal{Q}'$ or the color of one of the points among the $O(\Delta)$ points in $\mathcal{Q} \setminus \mathcal{Q}'$. We call these $O(\Delta)$ colors the *candidate* colors. Using the range counting structure, for each candidate color c we count the number of points with color c in $\mathcal{Q}$ and report the one with the maximum count. The query time is $O(2d\Delta \cdot t_n) = O(\Delta \cdot t_n)$.

Theorem 25 (Chan et al. [20])**.** *There exists a data structure that supports orthogonal range mode queries on a set of n points in d dimensions in $O(\Delta \cdot t_n)$ time while using $O(s_n + (n/\Delta)^{2d})$ words.*

The current best orthogonal range counting data structure requires:

$$s_n = O(n(\lg n/\lg\lg n)^{d-2})$$

words and supports queries in:

$$t_n = O((\lg n/\lg\lg n)^{d-1})$$

time [60]. The following result can be obtained by choosing Δ such that $s_n = (n/\Delta)^{2d}$. That is $\Delta = n^{(1-\frac{1}{2d})}(\lg n/\lg\lg n)^{(\frac{1}{d}-\frac{1}{2})}$.

Corollary 26 (Chan et al. [20])**.** *There exists data structure that supports orthogonal range mode queries on a set of n points in d dimensions in $O(n^{(1-\frac{1}{2d})}(\lg n/\lg\lg n)^{(d+\frac{1}{d}-\frac{3}{2})})$ time while using $O(n(\lg n/\lg\lg n)^{d-2})$ words.*

5.4 Improved Data Structure

Again we assume that the input point set $\mathcal{S}$ has been transformed to the rank space, and we denote by s_n and t_n the space and query time of an orthogonal range counting data structure on $\mathcal{S}$. The main idea is to maintain the array A in $\Theta((n/\Delta)^{2d})$ bits as opposed to $\Theta((n/\Delta)^{2d})$ words. Doing so increases the cost of accessing an entry of A from constant to $O(\Delta \cdot t_n)$ time. However, the total query cost does not increase.

We now describe how to encode A in less space. We use the following common notation: let $\lg^{(h)} n = \lg(\lg^{(h-1)} n)$ for $h > 1$, let $\lg^{(1)} n = \lg n$, and let $\lg^* n$ be the smallest integer k such that $\lg^{(k)} n \le 2$. Let $\Delta_h = \Delta \lg^{(h)} n$ (rounded to the next highest power of 2) and let A_h be an array of length $(1 + n/\Delta_h)^{2d}$ such that $A_h[i]$ stores the most frequent color in the Δ_h box with $rank(\cdot, \Delta) = i$. Notice that Δ_i is a multiple of Δ_{i+1}, and $\Delta_{\lg^* n} = \Theta(\Delta)$.

Lemma 27. *There exists a scheme where A_h can be encoded in $S(h)$ bits and any entry in A_h can be decoded in $T(h)$ time, where*

$$S(h) = \begin{cases} O((n/\Delta_1)^{2d} \lg n) & \text{if } h = 1 \\ S(h-1) + O((n/\Delta_h)^{2d} \lg^{(h)} n) & \text{if } h > 1, \end{cases}$$

$$T(h) = \begin{cases} O(1) & \text{if } h = 1 \\ T(h-1) + t_n \cdot O(\Delta / \lg^{(h)} n) & \text{if } h > 1. \end{cases}$$

Proof. Let A'_h be the desired encoding. The base case can be achieved by storing A_1 explicitly (i.e., $A_1 = A'_1$). For $h > 1$, given an encoding A'_{h-1} we obtain A'_h by storing an additional array B_h of size $(1 + n/\Delta_h)^{2d}$ where each entry has size $O(\lg^h(n))$ bits. Let R be a Δ_h box and R' be the largest (possibly empty) Δ_{h-1} box within R. We distinguish between two cases:

1. If the mode of R and R' are the same, then we simply store a special symbol \$ in $B_h[rank(R, \Delta_h)]$.
2. Else, there must exists a point p in the region $R \setminus R'$, where p_c is the mode of R. Moreover the distance (say τ) from p to the boundary of R is at most Δ_{h-1}. We store $B_h[rank(R, \Delta_h)] = \lceil \tau / \delta_h \rceil$, an approximate value of τ, where $\delta_h = \Delta / \lg^{(h)} n$. This approximate distance can be encoded in $O(\lg(\Delta_{h-1}/\delta_h)) = O(\lg^{(h)} n)$ bits.

Since the space occupied by B_h is $O((n/\Delta_h)^{2d} \lg^{(h)} n)$ bits, the equation $S(h) = S(h-1) + O((n/\Delta_h)^{2d} \lg^{(h)} n)$ follows.

We now describe how to decode the original value of an entry in A'_h. The array A'_1 is stored explicitly, therefore $T(1) = O(1)$. For $h > 1$, assume that we can decode entries of A'_{h-1} in the desired time. An entry in A'_h corresponding to a Δ_h-box R can be decoded as follows:

1. If $B_h[rank(R, \Delta_h)] = \$$, then the mode of R is same as the mode of R', the largest Δ_{h-1} box within R. The mode of R' is equal to $A_h[rank(R', \Delta_{h-1})]$ so the time for decoding it is $T(h) = T(h-1) + O(1)$.

2. Otherwise, $\delta_h \cdot B_h[rank(R, \Delta_h)]$ represents the approximate distance (within an additive error at most $\delta_h = \Delta / \lg^{(h)} n$) from a point p on the boundary of R, such that p_c is the mode of R. Since the points are in rank space, the number of points satisfying this approximate distance criteria is at most $2d \cdot \delta_h$ and the color of a point among them is the mode of R. So, the mode of R (i.e., $A_h[rank(R, \Delta_h)]$) can be identified using $O(\delta_h)$ range counting queries. Thus, giving the equation: $T(h) = T(h-1) + t_n \cdot O(\Delta / \lg^{(h)} n)$.

By combining both cases, the equation $T(h) = T(h-1) + t_n \cdot O(\Delta / \lg^{(h)} n)$ follows. □

Note that

$$\begin{aligned}
S(\lg^* n) &= O\left(\sum_{h=1}^{\lg^* n} (n/\Delta_h)^{2d} \lg^{(h)} n\right) \\
&= O\left((n/\Delta)^{2d} \sum_{h=1}^{\lg^* n} \left(\frac{1}{\lg^{(h)} n}\right)^{2d-1}\right) \\
&= O\left((n/\Delta)^{2d}\right), \text{and} \\
T(\lg^* n) &= t_n \cdot O\left(\sum_{h=1}^{\lg^* n} \delta_h\right) \\
&= t_n \cdot O\left(\Delta \sum_{h=1}^{\lg^* n} \frac{1}{\lg^{(h)} n}\right) \\
&= t_n \cdot O(\Delta).
\end{aligned}$$

Therefore, by maintaining an $O((n/\Delta)^{2d})$-bit or $O((n/\Delta)^{2d}/w)$-word data structure structure (along with the range counting structures), we can compute the mode of the largest $\Delta_{\lg^* n}$ box $\mathcal{Q}'$ in any query $\mathcal{Q}$ in $t_n \cdot O(\Delta)$ time. Since the number of points in $\mathcal{Q} \setminus \mathcal{Q}'$ is at most $2d \cdot \Delta_{\lg^* n} = O(\Delta)$, the mode of $\mathcal{Q}$ can be computed within an additional $O(t_n \cdot \Delta)$ time. We summarize our results in the following theorem.

Theorem 28. *There exists a data structure that supports orthogonal range mode queries on a set of n points in d dimensions in $O(\Delta \cdot t_n)$ time while using $O(s_n + (n/\Delta)^{2d}/w)$ words.*

We get the following corollary by using the range counting data structure of Jájá et al. [60].

Corollary 29. *There exists a data structure that supports orthogonal range mode queries on a set of n points in* $d \geq 2$ *dimensions in* $O((n^{(1-\frac{1}{2d})}/w^{\frac{1}{2d}})(\lg n/\lg\lg n)^{(d+\frac{1}{d}-\frac{3}{2})})$ *time while using* $O(n(\lg n/\lg\lg n)^{d-2})$ *words.*

Chapter 6

One Dimensional Range Searching

6.1 Introduction

In this chapter we present data structures for the following problems.

- One dimensional color range reporting: Given a set of colored points $\mathcal{P}$, preprocess $\mathcal{P}$ into an efficient data structure so that for any range $\mathcal{Q} = [a, b]$ the distinct colors of points contained in $\mathcal{P} \cap \mathcal{Q}$ can be reported.

- One dimensional approximate color range counting: Given a set of colored points $\mathcal{P}$, preprocess $\mathcal{P}$ into an efficient data structure so that for any range $\mathcal{Q} = [a, b]$ a $(1 + \varepsilon)$-approximation of the number of distinct colors of points contained in $\mathcal{P} \cap \mathcal{Q}$ can be reported, where $\varepsilon < 1$ is a constant. That is, if the number of distinct colors in $\mathcal{Q} \cap \mathcal{P}$ is x, the data structure should return a number y satisfying $x \leq y \leq (1 + \varepsilon)x$.

- One dimensional approximate median reporting: Given a set of points $\mathcal{P}$ where every point is assigned a value from the set $\{1, \ldots, U\}$, preprocess $\mathcal{P}$ into an efficient data structure so that for any range $\mathcal{Q} = [a, b]$ an element whose rank is between $(\lfloor k/2 \rfloor - \alpha k)$ and $(\lfloor k/2 \rfloor + \alpha k)$ in the query interval $[a, b]$ is reported, where k is the number of points in $[a, b]$ and $\alpha < 1$ is a constant.

We study all three problems in the context of succinctness, where the goal is to achieve the optimal space requirement plus a lower order term, while maintaining fast query time.

We note that the results of this Chapter are published in [31].

Previous Work. If the input points are in the rank space, one-dimensional color reporting queries can be answered in $O(k+1)$ time using $nH_d(S) + o(n \lg \sigma) + O(n \lg\lg \sigma)$ bits [6, 13, 17], where σ is the number of distinct colors, $d = o(\log_\sigma n)$, and $H_d(S)$ is the *d-th* order empirical entropy of the given sequence of colors S. In the general case, one-dimensional color reporting queries can be answered in $O(\lg n + k)$ time in the static and dynamic scenarios as shown by Janardan and Lopez [61] and Gupta et al. [52]. Muthukrishnan [77] later described a static $O(n)$ space data structure that answers queries in $O(k+1)$ time when all point coordinates are bounded by n. His result implies an $O(n)$-words data structure that answer queries in $O(\min(\lg\lg m, \sqrt{\lg n/\lg\lg n}) + k)$ time using the reduction-to-rank-space technique, where $O(\min(\lg\lg m, \sqrt{\lg n/\lg\lg n}))$ is the time needed to answer a predecessor query [14, 40]. A dynamic data structure of Mortensen [67] supports queries and updates in $O(\lg\lg n + k)$ and $O(\lg\lg n)$ time respectively if the values of all elements are bounded by n. Finally, Nekrich and Vitter [82] presented an $O(n)$-words static data structure that answers queries in $O(k+1)$ time; their result is valid even in the case when point are not in the rank space. They also presented a dynamic version of their structure that uses the same space and achieves the same query time while handling updates in $O(\lg^\epsilon n)$ time.

One-dimensional color counting in the rank space was studied by Gagie et al. [41]. They gave a data structure that answers queries in $O(\lg^{1+\epsilon} n)$ time for any constant $\epsilon > 0$ and uses $nH_0(S) + O(n) + o(nH_0(S))$ bits. Nekrich [81] described a data structure that uses $O(n \lg n)$ bits and answers color counting queries in $O(\lg k/\lg\lg n)$ time, where k is the number of colors. A lower bound that follows from the predecessor problem [97, 10] holds for exact one-dimensional color counting, and does not permit constant query time for a data structure with space bounded by a polynomial function of n. We circumvent this lower bound by focusing on approximate color counting. If we combine a reduction of one-dimensional color counting to point counting in 2D with the result of Chan and Wilkinson [22], we obtain a data structure that uses $O(n \lg n)$ bits and answer approximate color counting queries in $O(\lg^\epsilon n)$ time. The data structure of Nekrich [22] also uses $O(n \lg n)$ bits but answers approximate color counting queries in $O(1)$ time. In both [81] and [22] it is assumed that points are in the rank space. In the general case, Saladi [86] presented a data structure that uses $O(n)$ words and answers queries in $O(\lg\lg U)$ time.

Bose et al. [16] studied the problem of one-dimensional approximate median reporting when the input points are in the rank space. They provided a data structure that uses $O(n)$ words and answers queries in constant time. Their data structure returns the value of an approximate-median. In that model $O(n)$ words are required because one can query each index separately and get its value. We relax that constraint and focus on data structures that report the index of an approximate-median. We show that in this relaxed model $O(n)$ bits are sufficient and necessary.

Our Results. We focus on studying the three problems presented in the succinct scenario. In Sections 6.2 and 6.3 we solve an open problem from [86] by presenting a data structure that answers approximate color counting queries in optimal $O(1)$ time. Our data structure uses $\mathcal{B}(n,m) + O(n) + o(\mathcal{B}(n,m))$ bits, where $\mathcal{B}(n,m) \approx n \lg (m/n)$ is the minimum number of bits required to store a set of size n from a universe of m elements. Thus, we demonstrate that is not necessary to store the colors of points in order to answer approximate color counting queries. If points are in the rank space, our data structure needs only $O(n)$ bits and does not require access to the original data set. That is, similar to data structures for answering range minimum queries [39] that can answer queries without storing the original data set, we can construct a data structure for a colored set of points S and discard the set S. Using our data structure, we are still able to obtain a constant factor approximation on the number of colors in $S \cap [a,b]$ for an arbitrary query interval $[a,b]$.

In Section 6.4 we use similar techniques to obtain a data structure that answers approximate range median queries in constant time using only $\mathcal{B}(n,m) + O(n) + o(\mathcal{B}(n,m))$bits. When the points are in the rank space, our data structure uses $O(n)$ bits, thus improving a result from [16].

Then we turn to the problem of reporting colors using succinct space. We describe a data structure that answers color reporting queries in $O(k+1)$ time while using $\mathcal{B}(n,m) + nH_d(S) + o(\mathcal{B}(n,m) + n \lg \sigma)$ bits in Section 6.5. This result is a succinct counterpart of the data structure from [82] that also achieves optimal query time but uses $O(n \lg n)$ bits.

Finally we consider dynamic succinct color reporting in the rank space. We present a succinct data structure that answers color reporting queries in optimal $O(k+1)$ time and updates in $O(\lg n)$ time while using $nH_d(S) + o(n \lg \sigma)$ bits. Our data structure supports an update operation that changes the color of a point in $O(\lg n)$ time.

Applications. Color reporting and counting queries are related to problems that arise in string processing and databases. Color searching queries are helpful when we are interested in (the number of) distinct object categories in a query range or look for distinct documents that contain a query substring. One prominent example is the document counting queries on a collection of documents. We keep documents (strings) $d_1, \ldots, d_D$ in a data structure so that for any query string P the number of documents that contain P can be calculated. This problem can be solved by answering color counting queries on the so called document array. Consider the generalized suffix tree for all the documents, this document array is the array of documents the leaves correspond to in order; see [77, 42] for a detailed description.

The document array, however, needs $O(n \lg D)$ bits of space in the worst case where n is the size of all the documents. If the number of documents is large and the alphabet size is small, the space usage of the document array can be significantly larger than the space needed to store the document collection. Using the result of Theorem 39, we can answer approximate document counting queries using $O(n)$ additional bits.

6.2 Approximate Color Range Counting

In this section we present a data structure that uses $\mathcal{B}(n,m) + O(n) + o(\mathcal{B}(n,m))$ bits of space and answers approximate color counting queries in constant time. A color range counting query for an interval returns the number of distinct colors of points contained within the interval. For any constant $\varepsilon > 0$, our color range counting data structure returns in constant time an approximate answer which is within a factor of at most $(1+\varepsilon)$ of the correct answer.

6.2.1 Approximate Color Range Counting in Rank Space

We begin by describing a data structure for the problem in the special case when the input points are in the rank space. The input consists of a sequence $S = s_1, \ldots, s_n$ of n colors. A query is a range $[a, b]$ where $a, b \in [n]$, and the answer is a $(1+\varepsilon)$-approximation of the number of distinct colors found in $s_a, \ldots, s_b$.

6.2.1.1 Space Inefficient Solution

First we describe a space inefficient solution that requires $O(n \lg^3 n)$ bits of space and answers one-dimensional approximate color counting queries in constant time.

Consider the complete binary tree $\mathcal{T}$, in which each leaf of $\mathcal{T}$ corresponds to an element of S, and every internal node has two children. Given a node $u \in \mathcal{T}$, $u_l(u_r)$ denotes the left(right) child of u, $S(u)$ denotes the set of all elements stored in the leaf descendants of u, and $a_u(b_u)$ denotes the rightmost(leftmost) element in $S(u_l)(S(u_r))$. These definitions are illustrated in Figure 6.1.

Let $\delta = 1 + \varepsilon$. For each node $u \in \mathcal{T}$ we store the values $l_1, \ldots, l_{\log_\delta n}$ in a fusion tree [1] [40], where l_i $(1 \le i \le \log_\delta n)$ is the maximum value satisfying the condition that

[1] Fusion trees have a branching factor of $w^{1/5} = \Omega(\lg^{1/5} n)$. If a fusion tree contains a polylogarithmic

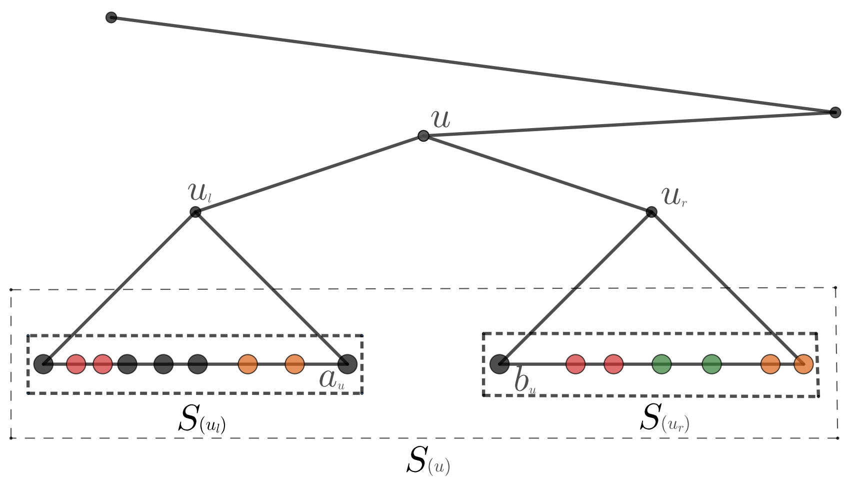


Figure 6.1: A sample node $u \in \mathcal{T}$ and the sets associated with u.

$s_{l_i}, \ldots, s_{a_u}$ contains δ^i distinct colors. Also, for each node $u \in \mathcal{T}$ and each i $(1 \leq i \leq \log_\delta n)$ we store the values $r_{i1}, \ldots, r_{i \log_\delta n}$ in a fusion tree [40], where r_{ij} $(1 \leq j \leq \log_\delta n)$ is the minimum value satisfying the condition that $s_{b_u}, \ldots, s_{r_j}$ contains δ^j distinct colors that are not present in $s_{l_i}, \ldots, s_{a_u}$.

Query: Given a query $[a, b]$ we find the lowest common ancestor u of a and b in $\mathcal{T}$. We query the fusion tree stored on $l_1, \ldots, l_{\log_\delta n}$ to find the predecessor l_i of a, then we query the fusion tree stored on $r_{i1}, \ldots, r_{i \log_\delta n}$ and find the successor r_{ij} of b. Finally we return $\delta^i + \delta^j$ as an estimate for the number of distinct colors in $[a, b]$.

Lemma 30. *The algorithm described above returns a $(1+\varepsilon)$-approximation of the number of distinct colors in $s_a, \ldots, s_b$.*

Proof. Denote by x the number of distinct colors in $s_a, \ldots, s_{a_u}$ and y the number of distinct colors in $s_{b_u}, \ldots, s_b$ that are not found in $s_a, \ldots, s_{a_u}$. Let y' denote the number of colors in s_{b_u}, …, s_b that do not occur in $l_i, \ldots, s_{a_u}$. By the definition of l_i and r_{ij}, $x \leq \delta^i \leq \delta \cdot x$ and $y' \leq \delta^j \leq \delta \cdot y'$. Since $y' \leq y$, $\delta^j \leq \delta \cdot y$. Hence $\delta^i + \delta^j \leq \delta(x+y)$. There are at most $\delta^i - x$

number of elements, as in our case, the height of the tree will be a constant and queries will be answered in constant time.

colors that occur in $l_i, \ldots, s_{a_u}$, but do not occur in $s_a, \ldots, s_{a_u}$. Hence $y - (\delta^i - x) \le y'$ and $y - (\delta^i - x) \le \delta^j$. If we add δ^i to both parts of the latter inequality, we obtain $y + x \le \delta^j + \delta^i$. Summing up

$$x + y \le \delta^i + \delta^j \le \delta(x + y)$$

which completes the proof.

□

Theorem 31. *There exists an $O(n \lg^3 n)$-bit data structure that supports one-dimensional $(1+\varepsilon)$-approximate color range counting queries in constant time when the input points are in the rank space.*

6.2.1.2 Lower Bound

Next, we show using a simple proof that $\Omega(n)$ bits are required for any data structure that answers one-dimensional $(1+\varepsilon)$-approximate color range counting queries in the rank space.

We assume without loss of generality that the number of colors $\sigma > \lfloor 1+\varepsilon \rfloor$, otherwise no data structure is needed since returning σ for any query would be a correct $(1+\varepsilon)$-approximation of the exact answer. Moreover, denote by $c_1, c_2, \ldots, c_k$ the first $k = \lfloor 1+\varepsilon \rfloor + 1$ colors. Divide a sequence S of size n to n/k blocks each of size k. We say that S satisfies property $(*)$ if for each block b in S one of the following two conditions hold:

- either b consists of the color c_1 repeated k times,
- or $b = c_1, c_2, \ldots, c_k$.

Clearly, the number of sequences that satisfy $(*)$ is $2^{(n/k)}$ since there exist n/k blocks in a sequence of size n and each block can satisfy one of two different conditions. Moreover for any two distinct sequences S_1 and S_2 satisfying $(*)$ differing at block b, there exist at least one $(1+\varepsilon)$-approximate range counting query, namely the query that asks for the number of different colors in b, that will return different values. Thus, the information theoretic lower bound for storing a one-dimensional $(1+\varepsilon)$-approximate range counting data structure is $\Omega(\lg 2^{(n/k)}) = \Omega(n/k) = \Omega(n/(1+\epsilon))$ bits.

Theorem 32. *Any one-dimensional $(1+\varepsilon)$-approximate range counting data structure requires $\Omega(n/(1+\epsilon))$ bits.*

6.2.1.3 Compact Data Structure

In this subsection we show how to make the data structure of Theorem 31 compact by bootstrapping.

Let $\delta = 1 + \varepsilon$. We define the functions: $f(n) = f^{(1)}(n) = \lg^4 n$ and $f^{(h)}(n) = f^{(h-1)}(f(n))$. The function $f^*(n)$ is defined as

$$f^*(n) = \begin{cases} 1 \text{ if } n \leq 2^{16} \\ 1 + f^*(f(n)) \text{ if } n > 1 \end{cases}$$

We note that the functions $f^{(i)}$ and f^* are a twist on the iterated logarithm function $\lg^*$, and $\lg^{(i)}(n) < f^{(i)}(n) < \lg^{(i/2)}(n)$.

We start by modifying the tree $\mathcal{T}$ from section 6.2.1.1 so that each leaf of $\mathcal{T}$ corresponds to a block of $f(n)$ consecutive elements of S (instead of a single element of S). Then, we define the family of trees $\mathcal{T}_{ij}$ where $1 \leq i \leq f^*(n)$ and $1 \leq j \leq n/f^{(i)}(n)$ as follows. Tree $\mathcal{T}_{ij}$ spans the i^{th} block of S of size $f^{(i)}(n)$ (i.e. $s_{((i-1)f^{(i)}(n)+1)}, \ldots, s_{(if^{(i)}(n))}$) and each leaf of $\mathcal{T}_{ij}$ correspond to a block of $f^{(i+1)}(n)$ consecutive elements. For each node $u \in \mathcal{T}_{ij}$ we store in separate fusion trees the sets of values: $\{l_p | 1 \leq p \leq \log_\delta f^{(i)}(n)\}$, and for each $1 \leq p \leq \log_\delta f^i(n)$ the set $\{r_{pq}|, 1 \leq q \leq \log_\delta f^{(i)}(n)\}$ as defined in Section 6.2.1.1. Finally, for every two indices a and b satisfying $1 \leq a \leq b \leq f(n)$ we store in a table B the index i such that a and b are in the same block of size $f^i(n)$ but in different blocks of size $f^{i+1}(n)$. In other words, i must satisfy the following conditions $\lfloor a/f^{(i)}(n) \rfloor = \lfloor b/f^{(i)}(n) \rfloor$ and $\lfloor a/f^{(i+1)}(n) \rfloor \neq \lfloor b/f^{(i+1)}(n) \rfloor$

Space Analysis: The number of nodes in $\mathcal{T}$ is reduced to $n/f(n)$ and the space used by $\mathcal{T}$ and fusion trees stored in its nodes is $O(n/\lg n)$ bits. The number of nodes in $\mathcal{T}_{ij}$ is $f^{(i)}(n)/f^{(i+1)}(n)$ and the space used by $\mathcal{T}_{ij}$ and fusion trees stored in its nodes is $O(f^{(i)}(n)/\lg(f^{(i)}(n)))$ bits. Thus, the total space used by all such trees is:

$$\begin{aligned}
\sum_{i=1}^{f^*(n)} \left(\sum_{j=1}^{n/f^{(i)}(n)} O\Big(f^{(i)}(n)/\lg(f^{(i)}(n))\Big) \right) &= \sum_{i=1}^{f^*(n)} \Big(n/f^{(i)}(n) \cdot O\Big(f^{(i)}(n)/\lg(f^{(i)}(n))\Big)\Big) \\
&= \sum_{i=1}^{f^*(n)} O\Big(n/\lg(f^{(i)}(n))\Big) \\
&= n \sum_{i=1}^{f^*(n)} O\Big(1/\lg(f^{(i)}(n))\Big) \\
&= O(n)
\end{aligned}$$

Finally, the table B uses $o(n)$ bits. Thus, the total space used is $O(n)$ bits.

Query: Given a query $[a, b]$, if a and b are in two different blocks of size $f(n)$, we can answer queries using $\mathcal{T}$ in the same way as described in Subsection 6.2.1.1. Otherwise, we query B on values $(a \mod f(n))$ and $(b \mod f(n))$ to find the index i satisfying the condition that a and b are in the same block of size $f^{(i)}(n)$ but in different blocks of size $f^{(i+1)}(n)$. Finally, we query $\mathcal{T}_{i\lfloor a/f^{(i)}n\rfloor}$ as we query $\mathcal{T}$.

Theorem 33. *There exists a compact $O(n)$-bit data structure that supports one-dimensional $(1+\varepsilon)$-approximate color range counting queries in constant time when the input points are in the rank space.*

6.3 General Approximate Range Counting

In this section, we consider the general case of approximate range counting where each point is assigned a coordinate from 1 to m as well as a color. Given a query $[a, b]$ where $a, b \in [m]$, the goal is to return a $(1+\varepsilon)$-approximation of the number of colors that occur within $[a, b]$. We present a data structure that uses $\mathcal{B}(n, m) + O(n) + o(\mathcal{B}(n, m))$ bits of space and answers $(1+\varepsilon)$-approximate color counting queries in constant time.

Let $\delta = 1 + \varepsilon$ and let $x_1, \ldots, x_n$ be the coordinates of the n given colored points $\mathcal{P}$ in sorted order. Denote by $\mathcal{P}_{\lceil \lg^3 n \rceil}$ the set of points whose x-coordinate rank is a multiple of $\lceil \lg^3 n \rceil$. For each point $p \in \mathcal{P}$ denote by $\mathrm{L}(p)$ the set of points to the left of p, and by $\mathrm{R}(p)$ the set of points to the right of p.

For each point $p \in \mathcal{P}_{\lceil \lg^3 n \rceil}$ we store in a fusion tree [40] the unique values $l_1, \ldots, l_{\log_\delta n}$ where l_i ($i \in [\log_\delta n]$) is the maximum value satisfying the condition that $s_{l_i}, \ldots, s_p$ contains δ^i unique colors. Also, for each point $p \in \mathcal{P}_{\lceil \lg^3 n \rceil}$ and each $i \in [\log_\delta n]$ we store in a fusion tree [40] the unique values $r_{i1}, \ldots, r_{i \log_\delta n}$ where r_{ij} ($j \in [\log_\delta n]$) is the minimum value satisfying the condition that $s_{p+1}, \ldots, s_{r_j}$ contains δ^j unique colors not present in $s_{l_i}, \ldots, s_p$. We also store a succinct point reporting structure [46] on $\mathcal{P}_{\lceil \lg^3 n \rceil}$.

Next, we divide $x_1, \ldots, x_n$ into $n/\lceil \lg^3 n \rceil$ blocks each of size $\lceil \lg^3 n \rceil$, except for the last one. Using $O(n \lg^{4/5} m)$ bits [83] we store predecessor and successor data structures for each block independently. Since the size of each block is at most $\lceil \lg^3 n \rceil$, answering predecessor and successor queries within a block takes constant time. Finally, we store in $O(n)$ bits the compact data structure from Theorem 39 for answering queries in the rank space.

Query: Given a query $[a, b]$ we check if a point $p \in \mathcal{P}_{\lceil \lg^3 n \rceil}$ is in $[a, b]$. If so, we query the fusion tree stored on $l_1, \ldots, l_{\log_{1+\varepsilon} n}$ to find l_i the predecessor of a, then we query the fusion tree stored on $r_{i1}, \ldots, r_{i \log_{1+\varepsilon} n}$ to find r_{ij} the successor of b, afterwards we return $(1+\varepsilon)^i + (1+\varepsilon)^j$.

If such a point p does not exist, then both a and b are in one of the blocks whose size is $\lceil \lg^3 n \rceil$. Using the reporting data structure stored on $\mathcal{P}$ we get the rank of an arbitrary point in $[a, b]$ then determine which block does a and b belong to. Afterwards, using the predecessor and successor structures, we determine the rank of a and b. Since the query is now reduced to the rank space, we can answer it in constant time.

Theorem 34. *There exists an* $(\mathcal{B}(n, m) + O(n) + O(n \lg^{4/5} m))$*-bit data structure that supports one-dimensional* $(1 + \varepsilon)$*-approximate color range counting queries in constant time.*

Next, we describe how to reduce the space of the predecessor and successor data structures. We split the universe $[m]$ into n subranges $r_1, \ldots, r_n$ each of size m/n. We also use succinct rank and select data structures that store a bit vector of size n using $n + o(n)$ bits and answers rank and select queries in constant time [69]. For each non-empty subrange r_i we store a predecessor and successor structure for every block of $\lg^2 n$ consecutive elements and a point reporting structure P_i on all the points within r_i. These structures are stored consecutively in an array A. To locate the data structures for any range r_i within A, we count the number of points in the ranges r_j for $j < i$ then scale that number. For that purpose, we construct a bit vector B of size $2n$ bits, with rank and select queries, that stores a zero for each range r_i followed by n_i ones, where n_i is the number of points in the range r_i. To count the number of points preceding r_i, we use a select query to get the position k of the i^{th} zero in B, then with a rank query we count the number of ones before position k.

Given a non-empty query range $[a, b]$ such that there exist at most $\lg^3 n$ points between a and b, a belongs to r_i where $i = \lfloor a/(m/n) \rfloor$ and b belongs to r_j where $j = \lfloor b/(m/n) \rfloor$, we find the rank of a in the following manner. First, we map a to $a' = a - im/n$ and b to $b' = b - jm/n$. If the range $[a', m/n]$ is empty in P_i, we use rank and select queries to get s the number of ones before the $(i+1)^{\text{th}}$ zero in B, the rank of a will be $s + 1$. Otherwise, we find a point p in P_i within the range $[a', m/n]$ if i and j are different or within the range $[a', b']$ if i and j are the same. If p's rank within r_i is k, we query the $\lfloor k/\lg^3 n \rfloor$ successor data structure to find the rank of a' in r_i. Then, we add the number of points occurring in each range r_l where $l < i$ to this rank to get the rank of a. We obtain the rank of b in a similar manner.

The extra space used is $o(\mathcal{B}(n,m))$ bits for the point reporting structures stored on the ranges $r_1,\ldots,r_n$, $O(n\lg^{4/5}(m/n)) = o(\mathcal{B}(n,m))$ bits for the predecessor and successor data structures, and $O(n)$ bits for the bit vector B.

Theorem 35. *There exists an $(\mathcal{B}(n,m)+O(n)+o(\mathcal{B}(n,m))$-bit data structure that supports one-dimensional $(1+\varepsilon)$-approximate color range counting queries in constant time.*

6.4 Approximate Median Range Reporting

In this section we present a data structure that uses $\mathcal{B}(n,m)+O(n)+o(\mathcal{B}(n,m))$ bits of space and answers approximate median reporting queries in constant time. A median range query returns the median of a query interval. Given a query interval with k points, an approximate-median reporting query returns an element whose rank is between $(\lfloor k/2\rfloor - \alpha k)$ and $(\lfloor k/2\rfloor + \alpha k)$ in the interval, where α is the approximation factor.

6.4.1 Approximate Median Range Reporting in Rank Space

We begin by describing a data structure for the problem in the special case when the input points are in the rank space. The input consists of a sequence $S = s_1,\ldots,s_n$ of n values. A query is a range $[a,b]$ where $a,b\in[n]$, and the answer is the index of an approximate median of $s_a,\ldots,s_b$.

6.4.1.1 Space Inefficient Solution

Like the preceding section, to illustrate the main idea of our final solution, we first describe a space inefficient solution that requires $O(n\lg^3 n)$ bits of space and answers approximate range median queries in constant time.

Let $\delta = 1+(\alpha/2)$. We store the unique values $\{m_{kij} : 1\le k\le n \text{ and } 1\le i,j\le \log_\delta n\}$, where m_{kij} is the median of the interval $s_a,\ldots,s_b$ such that $a = k-\delta^i$ and $b = k-\delta^i+\delta^j$.

Query: Given a query $[a,b]$ we pick an arbitrary point k in $[a,b]$ then compute the biggest indices i and j such that $k-\delta^i < a$ and $k-\delta^i+\delta^j < b$. Then, we return m_{kij}.

Lemma 36. *The algorithm described above returns an approximate median of $s_a,\ldots,s_b$ with an approximation ratio α.*

Proof. Denote by $x = (k - \delta^i) - a$ and $y = b - (k - \delta^i + \delta^j)$. By the way we compute i and j:

$$x \leq (\alpha/2)(k - a + 1) \leq (\alpha/2)(b - a + 1)$$
$$y \leq (\alpha/2)(b - k + \delta^i + 1) \leq (\alpha/2)(b - a + 1)$$

so the number of elements in $s_x, \ldots, s_y$ is at least $(b - a + 1) - \alpha(b - a + 1)$. Thus, since we are returning the index of an exact median in $s_x, \ldots, s_y$, that index would correspond to an approximate median in $s_a, \ldots, s_b$. □

Theorem 37. *There exists an $O(n \lg^3 n)$-bit data structure that supports one-dimensional approximate median queries in constant time when the input points are in the rank space.*

6.4.1.2 Lower Bound

Next, we show using a simple proof that $\Omega(n)$ bits are required for any data structure that answers one-dimensional approximate median range queries in the rank space.

Given an approximation factor α, divide the sequence S of size n to n/k blocks each of size $k = 1/\alpha$. We say that S satisfies property $(\star)$ if for each block b in S one of the following two conditions hold:

- either b consists of three 1s followed by $(k-3)/2$ 0s followed by $(k-3)/2$ 2s,
- or b consists $(k-3)/2$ 0s followed by $(k-3)/2$ 2s followed by three 1s.

Clearly, the number of sequences that satisfy $(\star)$ is $2^{(n/k)}$ since there exist n/k blocks in a sequence of size n and each block can have one of two different values. Moreover for any two distinct sequences S_1 and S_2 satisfying $(\star)$ differing at block b, there exist at least one approximate range median query, namely the query that asks for an approximate median of b, that will return different values. Thus, the information theoretic lower bound for storing an approximate range median data structure is $\Omega(\lg 2^{(n/k)}) = \Omega(n/k) = \Omega(\alpha n)$ bits.

Theorem 38. *Any one-dimensional approximate range median data structure requires $\Omega(\alpha n)$ bits where α is the approximation factor.*

6.4.1.3 Compact Data Structure

Using a similar approach to the one used in Subsection 6.4.1.1, in this subsection we use bootstrapping to make the data structure of Theorem 37 compact. Let $\delta = 1 + (\alpha/2)$ and the functions $f(n)$, $f^{(h)(n)}$, and $f^*(n)$ be defined as in Subsection 6.4.1.1.

The main idea is to store the values m_{kij} for all k values that are a multiple of $\lg^4 n$ using $O(n/\lg n)$ bits, then, recursively bootstrap on the individual blocks of size $\lg^4 n$. More precisely we store the tables $T_0, \ldots, T_{f^*(n)}$. The table T_0 contains the values m_{kij} as described in Section 6.4.1.1 for all k values that are a multiple of $\lg^4 n$. The table T_s where $1 \leq s \leq f^*(n)$ contains the values m_{bkij} where:

$$
\begin{aligned}
&1 \leq b \leq n/f^s(n) \\
&1 \leq k \leq f^{(s)}(n)/f^{(s+1)}(n) \\
&1 \leq i, j \leq \lg_\delta f^{(s)}(n)
\end{aligned}
$$

and m_{sbkij} is the median of the interval $s_x, \ldots, s_y$ such that:

$$
\begin{aligned}
x &= b \cdot f^{(s)}(n) + k \cdot f^{(s+1)}(n) - \delta^i \\
y &= b \cdot f^{(s)}(n) + k \cdot f^{(s+1)}(n) - \delta^i + \delta^j
\end{aligned}
$$

Finally, for every two indices x and y satisfying $1 \leq x \leq y \leq f(n)$ we store in a table B the index i such that x and y are in the same block of size $f^s(n)$ but in different blocks of size $f^{s+1}(n)$. In other words, s must satisfy the following conditions $\lfloor x/f^{(s)}(n) \rfloor = \lfloor y/f^{(s)}(n) \rfloor$ and $\lfloor x/f^{(s+1)}(n) \rfloor \neq \lfloor y/f^{(s+1)}(n) \rfloor$.

Space Analysis: The table T_0 uses $O(n/\lg n)$ bits since it contains $n/\lg^2 n$ entries each of size $\lg n$. Each entry in table T_s where $1 \leq s \leq f^*(n)$ can be stored in $\lg(f^{(s)}(n))$ bits. Moreover, the number of entries in T_s is:

$$
\begin{aligned}
&(n/f^s(n)) \cdot (f^{(s)}(n)/f^{(s+1)}(n)) \cdot \lg_\delta^2(f^{(s)}(n)) \\
&= (n/f^{(s+1)}(n)) \cdot \lg_\delta^2(f^{(s)}(n)) \\
&= (n/\lg_\delta^4(f^{(s)}(n))) \cdot \lg_\delta^2(f^{(s)}(n)) \\
&= (n/\lg_\delta^2(f^{(s)}(n)))
\end{aligned}
$$

so T_s will use $O(n/\lg(f^{(s)}(n)))$ bits. Thus, the total space used by all tables T_s ($1 \leq s \leq f^*(n)$) is: $\sum_{s=1}^{f^*(n)} O(n/\lg(f^{(s)}(n))) = O(n)$ bits. Finally, the table B uses $o(n)$ bits. Thus, the total space used is $O(n)$ bits.

Query: Given a query $[a, b]$, if a and b are in two different blocks of size $f(n)$, we can answer queries using the same way as described in Subsection 6.4.1.1. Otherwise, we query B on values $(a \mod f(n))$ and $(b \mod f(n))$ to find the index s satisfying the condition that a and b are in the same block of size $f^{(s)}(n)$ but in different blocks of size $f^{(s+1)}(n)$. Then, we proceed in a query similar to the one described in Subsection 6.4.1.1 using the entries in table T_s.

Theorem 39. *There exists a compact $O(n)$-bit data structure that supports one-dimensional approximate range median queries in constant time when the input points are in the rank space.*

6.4.2 General Approximate Range Median

In this section, we consider the general case of approximate median reporting where each point is assigned a coordinate from 1 to m as well as a value. Given a query $[a, b]$ where $a, b \in [m]$, the goal is to return an approximate median of the values occurring within $[a, b]$. We present a succinct data structure that uses $\mathcal{B}(n, m) + O(n) + o(\mathcal{B}(n, m))$ bits of space and answers approximate range median queries in constant time.

Let $\delta = 1 + (\alpha/2)$ and let $x_1, \ldots, x_n$ be the coordinates of the n given points $\mathcal{P}$ in sorted order. Denote by $\mathcal{P}_{\lceil \lg^3 n \rceil}$ the set of points whose coordinate rank is a multiple of $\lceil \lg^3 n \rceil$. For each point $p \in \mathcal{P}$ denote by $\mathrm{L}(p)$ the set of points to the left of p, and by $\mathrm{R}(p)$ the set of points to the right of p.

For each point $p \in \mathcal{P}_{\lceil \lg^3 n \rceil}$ we store in a fusion tree [40] the unique values $l_1, \ldots, l_{\log_\delta n}$ where l_i ($i \in [\log_\delta n]$) is the coordinate of the δ^i point before p. Also, for each point $p \in \mathcal{P}_{\lceil \lg^3 n \rceil}$ and each $i \in [\log_\delta n]$ we store in a fusion tree [40] the unique values $r_{i1}, \ldots, r_{i \log_\delta n}$ where r_{ij} ($j \in [\log_\delta n]$) is the coordinate of the δ^j point after l_i. We also store the median of the interval $[l_i, r_{ij}]$ as satellite data associated with r_{ij}. In addition, we store a succinct point reporting structure [46] on $\mathcal{P}_{\lceil \lg^3 n \rceil}$.

Next, we divide $x_1, \ldots, x_n$ into $n / \lceil \lg^3 n \rceil$ blocks each of size $\lceil \lg^3 n \rceil$, except for the last one. Using $o(\mathcal{B}(n, m))$ bits as described in Section 6.3 we store predecessor and successor data structures that can answer queries in each block independently. Since the size of each block is at most $\lceil \lg^3 n \rceil$, answering predecessor and successor queries within a block takes constant time. Finally, we store in $O(n)$ bits the compact data structure from Theorem 39 for answering queries in the rank space.

Query: Given a query $[a, b]$ we check if a point $p \in \mathcal{P}_{\lceil \lg^3 n \rceil}$ is in $[a, b]$. If so, we query the fusion tree stored on $l_1, \ldots, l_{\log_{1+\varepsilon} n}$ to find l_i the predecessor of a, then we query the fusion tree stored on $r_{i1}, \ldots, r_{i \log_{1+\varepsilon} n}$ to find r_{ij} the successor of b, afterwards we return the median of $[l_i, r_{ij}]$.

If such a point p does not exist, then both a and b are in one of the blocks whose size is $\lceil \lg^3 n \rceil$. Using the reporting data structure stored on $\mathcal{P}$ we get the rank of an arbitrary point in $[a, b]$ then determine which block does a and b belong to. Afterwards, using the predecessor and successor structures, we determine the rank of a and b. Since the query is now reduced to the rank space, we can answer it in constant time.

Theorem 40. *There exists an $(\mathcal{B}(n,m)+O(n)+o(\mathcal{B}(n,m))$-bit data structure that supports one-dimensional approximate range median queries in constant time.*

6.5 1D Color Range Reporting

Using similar techniques to those used in the previous sections, we present in this section a succinct data structure that uses $\mathcal{B}(n,m) + nH_d(S) + o(\mathcal{B}(n,m) + n \lg \sigma)$ bits of space and answers color reporting queries in optimal $O(k+1)$ time.

If the input points are in the rank space (i.e. the x-coordinates of the input points are 1,...,n and the input consists of a sequence $S = s_1, \ldots, s_n$ of n colors, a query is a range $[a, b]$ where $a, b \in [n]$, and the answer is the distinct colors found in $s_a, \ldots, s_b$), one-dimensional color range reporting can be solved in $O(k+1)$ time using $nH_d(S)+o(n)\lg\sigma + O(n \lg\lg\sigma)$ bits [6, 13, 17].

This solution can be extended to general one-dimensional range reporting by storing the x-coordinates of the points in sorted order in an indexable dictionary that supports select queries in constant time using $\mathcal{B}(n,m) + o(\mathcal{B}(n,m))$ bits [87] in addition to the data structure described in [6, 13, 17]. We can find the predecessor or successor of any x-coordinate in $O(\lg n)$ time by answering $O(\lg n)$ select queries. Hence, we can reduce any query $[a, b]$ to the rank space in $O(\lg n)$ additional time.

Theorem 41. *There exists an $(\mathcal{B}(n,m)+nH_d(S)+o(\mathcal{B}(n,m)+n\lg\sigma))$-space data structure that supports one-dimensional color range reporting queries in $O(\lg n + k)$time.*

6.5.1 Improved Data Structure

Next, we show how to improve the query time obtained from Theorem 41 to $O(k+1)$, while using the same amount of space.

Let $x_1, \ldots, x_n$ be the coordinates in sorted order of the n given colored points $\mathcal{P}$. We denote by $\mathcal{P}_{\lceil \lg^2 n \rceil}$ the set of points whose x-coordinate rank is a multiple of $\lceil \lg^2 n \rceil$. For each point $p \in \mathcal{P}$ we denote by $\mathrm{L}\,(p)$ the set of points to the left of p, and by $\mathrm{R}\,(p)$ the set of points to the right of p. For every color z the set $\mathrm{Min}\,(p)$ contains the minimal element $e \in \mathrm{L}\,(p)$ of color z, and the set $\mathrm{Max}\,(p)$ contains the maximal element $e \in \mathrm{R}\,(p)$ of color z.

Data Structure: For each point $p \in \mathcal{P}_{\lceil \lg^2 n \rceil}$, we store the smallest $\lceil \lg n \rceil$ elements of $\mathrm{Min}\,(p)$ and the largest $\lceil \lg n \rceil$ elements of $\mathrm{Max}\,(p)$. We also store two succinct one-dimensional point reporting data structures [46], one on every point in $\mathcal{P}$, and the other on every point in $\mathcal{P}_{\lceil \lg^2 n \rceil}$. Next, we store a data structure similar to the one used in subsection 6.3 that can find in constant time the ranks of a query $[a, b]$ if $[a, b]$ is not empty, and a and b belong to the same block of size $\lg^2 n$, Finally, we store the data structure from Theorem 41.

Answering Queries: We report all colors in a query range $[a, b]$ as follows. Using the reporting data structure stored on $\mathcal{P}_{\lceil \lg^2 n \rceil}$, we search for some $p \in \mathcal{P}_{\lceil \lg^2 n \rceil} \cap [a, b]$.

If such a point p exist, we traverse the list $\mathrm{L}\,(p)$ until an element $p' > b$ is found or the end of $\mathrm{L}\,(p)$ is reached. We also traverse the list $\mathrm{R}\,(p)$ until an element $p' < a$ is found or the end of $\mathrm{R}\,(p)$ is reached. If we reach neither the end of $\mathrm{L}\,(p)$ nor the end of $\mathrm{R}\,(p)$, then all distinct colors in $[a, b]$ are reported. Otherwise, the range $[a, b]$ contains more than $\lg n$ distinct colors. In that case we use the data structure from Theorem 41.

If a and b belong to a continuous block of $\lg^2 n$ points, we find their ranks in a similar manner to subsection 6.3, then solve the problem in the rank space as described in the previous subsection.

Theorem 42. *There exists a $(\mathcal{B}(n, m) + nH_d(S) + O(n) + o(\mathcal{B}(n, m) + n \lg \sigma))$-bit data structure that supports one-dimensional color range reporting queries in $O(k + 1)$ time.*

Note that $n = o(n \lg \sigma)$ as long as σ is not a constant. If σ is a constant, we solve the problem using a different approach. We store a separate succinct range emptiness data structure [46] for every subset of points with a given color. To answer a query $[a, b]$, for each color c we query the range emptiness data structure associated with c to check if a point with color c occurs in the range $[a, b]$, if so we report c. The query runtime is a constant since the number of colors is constant and range emptiness queries take constant time. Hence, we obtain the following theorem.

Theorem 43. *There exists an $(\mathcal{B}(n, m) + nH_d(S) + o(\mathcal{B}(n, m) + n \lg \sigma))$-space data structure that supports one-dimensional color range reporting queries in $O(k + 1)$ time.*

6.6 Dynamic Color Reporting in Rank Space

Finally, we describe a succinct data structure that uses $nH_d(S) + o(n \lg \sigma)$ bits of space and answers color reporting queries in optimal $O(k+1)$ time when the input points are in the rank space, while supporting the following update operation in $O(\lg n)$ time: given an index i and a color c, set the color of the i^{th} element to c.

Theorem 44. *There exists an $(nH_d(S)+o(n \lg \sigma)+O(n))$-bit data structure that supports one-dimensional color range reporting queries in $O(k+1)$ time and updates of the form: given an index i and a color c set the color of the i^{th} element to c, in $O(\lg n)$ time when points are in the rank space.*

Proof. Let the input sequence be $S = s_1, \ldots, s_n$, and $\mathcal{T}$ be the complete balanced binary tree where every leaf of $\mathcal{T}$ corresponds to an element of S and every internal node has two children. For any node $u \in \mathcal{T}$, $S(u)$ denotes the set of all elements stored in the leaf descendants of u. For $i \in \{1, \ldots, n\}$ denote by $l_i(r_i)$ the height of the highest ancestor u of the node corresponding to i such that i is the leftmost(rightmost) element in $S(u)$ with color s_i.

We store S in a dynamic data structure using $nH_d(S) + o(n \lg \sigma)$ bits that supports access in $O(1)$ time and Update, Rank, and Select in $O(\lg n / \lg \lg n)$ time [49]. We divide S into blocks of $\lg n$ elements each, then we subdivide each block to subblocks of size $\lg \lg n$ elements. For each subblock b_{ij} ($0 \le i < n/\lg n$ and $0 \le j < \lg n / \lg \lg n$) in block b_i we store:

- The maximum value m^l_{ij} of the sequence $l_{i \lg n + j \lg \lg n}, \ldots, l_{i \lg n+(j+1) \lg \lg n}$ and a succinct range maximum data structure [39] T^l_{ij} to answer range maximum queries on it.
- The maximum value m^r_{ij} of the sequence $r_{i \lg n + j \lg \lg n}, \ldots, r_{i \lg n+(j+1) \lg \lg n}$ and a succinct range maximum data structure [39] T^r_{ij} to answer range maximum queries on it.

The space used is $O(\lg \lg n)$ bits per subblock, which sums to $O(n)$ bits. For each block b_i we store:

- The sequence $m^l_{i0}, \ldots, m^l_{i \lg \lg n}$, its maximum value m^l_i, and a succinct range maximum data structure [39] T^l_i to answer range maximum queries on it.

- The sequence $m^r_{i0}, \ldots, m^r_{i \lg \lg n}$, its maximum value m^r_i, and a succinct range maximum data structure [39] T^r_i to answer range maximum queries on it.

The space used is $O(\lg n / \lg \lg n)$ bits per block, which sums to $O(n/ \lg \lg n)$ bits. Finally, using Lemma 1 from a result by Nekrich et al. [82] we store using $O(n)$ bits two-dimensional point reporting structures T^l and T^r containing the set of points (i, m^l_i) and (i, m^r_i) where $1 \leq i \leq n/ \lg n$. These structures support queries in $O(k+1)$ time and updates in $O(\lg^{\varepsilon} n)$ time.

Answering Queries: Given a query $[a, b]$, we find the lowest common ancestor u of a and b. Let $u_l(u_r)$ be the left(right) child of u, c be the rightmost child of u_l, and let h denote the height of u_l and u_r.

To get all distinct colors in $[a, c] = [a, b] \cap S(u_l)$, it is sufficient to report all colors s_i in that range with $r_i \geq h$. We maintain the invariant that each color is reported on its right most occurrence.

If $[a, c]$ was contained in a single subblock b_{ij}, we query T^r_{ij} for all the distinct colors as follows. We get the largest element r_d in $r_a, \ldots, r_c$, if s_d was previously reported we return, otherwise we report s_d and recurse on the interval $[d, c]$ followed by $[a, d]$. Note that it is important to recurse on $[d, c]$ before $[a, d]$ to maintain the invariant mentioned above, which guarantees that $r_d = \min(r_a, \ldots, r_c)$ will be smaller than h if the color s_d was previously reported.

Otherwise, if $[a, c]$ spans several subblocks but is contained in a single block b_i we proceed as follows. We first query the rightmost subblock partially spanned by $[a, c]$. Then, we query T^r_i to get all the subblocks b_{ij} spanned by $[a, c]$ satisfying the condition that $m^r_{ij} \geq h$ in order from right to left. We query each one of them in that order, then we query the leftmost subblock that is partially spanned by $[a, c]$.

Finally, if $[a, c]$ spans several blocks we first query the rightmost block partially spanned by $[a, c]$. Then, we query T^r to get all the blocks i spanned by $[a, c]$ satisfying the condition that $m^r_i \geq h$ in order from right to left. We query each one of them in that order, then we query the leftmost block that is partially spanned by $[a, c]$.

Similarly, to report all the distinct colors in $[c + 1, b] = [a, b] \cap S(u_r)$ it is sufficient to report all colors s_i in that range with $l_i \geq h$. We do this in a similar way to the method used to query $[a, c]$, while maintaining the invariant that each color is reported on its left most occurrence.

Updating the Sequence: If the color of position i was updated from c to c' the following values could get modified: r_i, r_a where a is the first index before i with color c,

r_b where b is the first index after i with color c', l_i, l_d where d is the first index after i with color c, and l_e where e is the first index before i with color c'.

We can find the value r_i of any index i in $O(\lg n/\lg\lg n)$ time by using Rank and Select queries to get the first index j before i with the same color as index i, then computing the lowest common ancestor of i and j. Similarly, to get the value l_i, we use Rank and Select queries to get the first index j after i with the same color as index i, then we compute the lowest common ancestor of i and j.

Since we don't store the values $r_1, \ldots, r_n$ and $l_1, \ldots, l_n$ explicitly, once one of them changes (say r_a where a is in subblock b_{ij}) we recompute all values r_j where $j \in b_{ij}$ and reconstruct T^r_{ij}. Recomputing all values r_j where $j \in b_{ij}$ takes $O(\lg\lg n \cdot \lg n/\lg\lg n) = O(\lg n)$ time and reconstructing T^r_{ij} takes $O(\lg\lg n)$ time. If m^r_{ij} changed, we rebuild T^r_i in $O(\lg n)$ time. Finally, if m_i changed we update its value in T^r in $O(\lg^{\varepsilon} n)$ time. Since only a constant number of values get updated, the runtime is $O(\lg n)$. □

If σ is a constant then the $O(n)$ additional bits stored by the data structure are no longer a lower order term, so we handle this case separately. We divide S into blocks of size $\lg n/2\lg\sigma$. We store a lookup table using $O(\sqrt{n}\lg^2 n)$ bits to answer color range queries over every possible block of this size. Also, we store the data structure from Theorem 44 on the sequence $S' = s'_1, \ldots, s'_{2\lg\sigma n/\lg n}$ with alphabet $\sigma' = 2^{\sigma}$, where s'_i denotes the subset of colors found on the i^{th} block of S. The total space used is $nH_d(S) + o(n\lg\sigma) + O(n/\lg n)$ bits. To answer a query $\mathcal{Q}$, we use the lookup table to get the colors in the (two) blocks which are not completely spanned by $\mathcal{Q}$, then we use the data structure from Theorem 44 to get the colors in the blocks that are fully spanned by $\mathcal{Q}$. Each color will be reported at most a constant number of times. The query time is $O(k+1) = O(1)$ and update time is $O(\lg n)$.

Theorem 45. *There exists an $(nH_d(S) + o(n\lg\sigma))$-bit data structure that supports one-dimensional color range reporting queries in $O(k+1)$ time and updates of the form: given an index i and a color c set the color of the i^{th} element to c, in $O(\lg n)$ time when points are in the rank space.*

Chapter 7

Succinct Dynamic One Dimensional Point Reporting

7.1 Introduction

This chapter studies the dynamic one-dimensional range reporting problem where the goal is to maintain (under insertion and deletion) a set of integers S from a universe of size m to answer range reporting queries efficiently: Given an interval $[a, b]$ for some $a, b \in [m]$, report all points in $S \cap [a, b]$. We note that this operation is equivalent to the operation $\text{FindAny}(a, b)$ which reports an arbitrary point c in $S \cap [a, b]$. This follows since if $[a, b]$ is not empty, we can recurse on $[a, c-1]$ and $[c+1, b]$ after obtaining c to get all points in $[a, b]$.

We study this problem in the context of succinctness. The goal is to occupy as little, or close to as little, space as possible while maintaining an efficient query time. We describe a dynamic data structure that answers reporting queries in optimal $O(k+1)$ time, where k is the number of points in the answer, and supports updates (insertions and deletions) in $O(\lg^{\varepsilon} m)$ expected time. Our data structure uses $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits where $\mathcal{B}(n, m) \approx n \lg(m/n)$ is the minimum number of bits required to represent a set of size n from a universe of m elements

Related Work One-dimensional range reporting is a well studied problem. Miltersen et al. [66] presented a data structure for the static version of this problem that uses $O(n \lg m)$

words and answers queries in constant time per reported element. Alstrup et al. [4] later presented an improved data structure with the same query time that uses $O(n)$ words, i.e., $O(n \lg m)$ bits. Goswami et al. [46] presented a succinct data structure that further improved the space usage to $\mathcal{B}(n,m) + o(\mathcal{B}(n,m))$ bits while preserving the query time.

For the dynamic version of this problem Mortensen et al. [68] presented a data structure that uses a linear number of words and answers queries in $O(t_q)$ time and updates in expected $O(t_u)$ time where:

$$t_q \geq \lg\lg\lg m,\ \lg\lg m/\lg\lg\lg m \leq t_u \leq \lg\lg m : \ t_u = O(\lg_{t_q} \lg m) + t_{pred},$$

$$\text{or } t_q \leq \lg\lg\lg m,\ t_u \geq \lg\lg m : \ 2^{t_q} = O(\lg_{t_u} \lg m).$$

The most appealing point of this trade-off in the context of succinct data structures is when the query time is constant and the update time is $O(\lg^{\varepsilon} m)$ time for a fixed $\varepsilon > 0$.

Our Results. We focus on studying one-dimensional range reporting in the succinct scenario. Our results depend on the ability to construct a static succinct one dimensional point reporting structure in $O(n \lg^{\varepsilon})$ time using $o(n)$ workspace. We defer the details of this construction to the end in Section 7.5 due to its technical nature. We present some preliminaries in Section 7.2. In Section 7.3 we present a semi-dynamic[1] succinct range reporting data structure that supports deletions in expected $O(\lg^{\varepsilon} m)$ time and queries in constant time. In Section 7.4 we present a fully-dynamic succinct range reporting data structure that supports updates in expected $O(\lg^{\varepsilon} m)$ time and queries in constant time.

We note that the results of this chapter are to be published in [32].

7.2 Preliminaries

In this section we review some previous results that will be used in the rest of this paper.

7.2.1 One-Dimensional Point Reporting

First we review the data structure of Alstrup et al. [4] for static one-dimensional range reporting. We start by defining some notations. Let $x \oplus y$ denote the binary exclusive-or of x and y. Given a w-bit integer x let $x \downarrow i = x/2^i$ denote the rightmost w bits of the result

[1]A semi-dynamic data structure supports queries and deletions but not insertions.

of shifting x i bits to the right. Similarly let $x \uparrow i = x \cdot 2^i \mod 2^w$ denote the rightmost w bits of the result of shifting x i bits to the left. Finally, denote by $\mathrm{msb}(x)$ the position of the most significant bit (or leftmost one bit) of x.

Given a set of integers S the goal is to store S while supporting the query $\mathrm{FindAny}(a, b)$ which returns an element in $S \cap [a, b]$. Denote by T the classic binary tree with 2^w leaves where all leaves have depth w. The leaves are numbered $0, \ldots, 2^w - 1$ from left to right while the internal nodes are labeled in a manner similar to an implicit binary heap. The root is the first node, and the children of a node v are $2v$ and $2v + 1$. As noted in [4] the d^{th} ancestor of v is $v \downarrow d$ and the lowest common ancestor of two leaves a and b is the $(1 + \mathrm{msb}(a \oplus b))^{\text{th}}$ ancestor of a or b. Thus the lowest common ancestor of two leaves can be computed in constant time.

Given a node $v \in T$ let $\mathrm{left}(v)$ and $\mathrm{right}(v)$ denote the left and right children of v, and let S_v denote the subset of S that is in the subtree rooted at v. A node v is branching if both $S_{\mathrm{left}(v)}$ and $S_{\mathrm{right}(v)}$ are not empty. To answer a query $\mathrm{FindAny}(a, b)$ it is sufficient to compute the lowest common ancestor v of a and b; when v is computed, either $\max S_{\mathrm{left}(v)}$ or $\min S_{\mathrm{right}(v)}$ is in $[a, b]$, or $[a, b]$ is empty. Thus by storing the values $\max S_{\mathrm{left}(v)}$ and $\min S_{\mathrm{right}(v)}$ for all nodes v with non-empty S_v in $O(nw)$ words, range reporting queries can be answered in constant time.

To improve the space Alstrup et al. [4] observe the following. Let v be the nearest branching ancestor of the lowest common ancestor of a and b, and let $v_l(v_r)$ be the nearest branching node in v's left(right) subtree if one exists, otherwise $v_l = v (v_r = v)$ if there is no branching node in v's left(right) subtree. Then either $\max S_{\mathrm{left}(v_l)}$, $\min S_{\mathrm{right}(v_l)}$, $\max S_{\mathrm{left}(v_r)}$, or $\min S_{\mathrm{right}(v_r)}$ is in $[a, b]$, or $[a, b]$ is empty. Thus they store a $O(n)$ word data structure that consists of:

B, D: vectors of size $O(n\sqrt{w} \lg w)$ bits that return the nearest branching ancestor of the nodes in T with non empty-subtrees.

V: a vector storing for each branching node v the values $\max S_v$ and $\min S_v$, in addition to two pointers to the nearest branching nodes in the left and right subtrees of v.

For the full details we refer the reader to [4].

7.2.2 Tree Representation

In their paper Geary and Raman [44] present a succinct ordinal tree representation that answers level ancestor queries. In their tree representation the tree is partitioned into mini-trees of size $O(\lg^4 n)$, and then the mini-trees are partitioned into micro-trees of size $O(\lg n)$. Internally a node x is referred to by $\tau(x) = (\tau_1(x), \tau_2(x), \tau_3(x))$ where $\tau_1(x)$ is the id of x's mini-tree, $\tau_2(x)$ is the id of x's micro tree, and $\tau_3(x)$ is the id of x in its micro tree. If two nodes x and y are in the same micro tree μ then $\tau_1(x) = \tau_1(y) = p(\mu)$ where $p(\mu)$ is the id of the micro tree μ. Note that micro trees can intersect only at their roots, and if a node is in different micro trees (i.e. it is the root of several micro trees) it can have different τ names. That is, if a node x is a root of two different micro-trees μ_1 and μ_2, it will have two different τ names where in the first one $\tau_2(x) = p(\mu_1)$ and in the second $\tau_2(x) = p(\mu_2)$. Both names are valid and we can select any one of them.

Geary and Raman show how to compute the preorder number of x given $\tau(x)$ in constant time using an index of size $o(n)$ bits. This index can be constructed in $O(n)$ time using a workspace of $O(n)$ words. Given a tree T partitioned using the above scheme and a node $x \in T$ we denote by $\text{root}\,(x)$ the root of the mini-tree that x belongs to.

7.2.3 Sparse Arrays

We will use the following Theorem from [57]:

Theorem 46 ([57])**.** *There is an $(m, n, O(n))$-family of perfect hash functions $\mathcal{H}$ such that any hash function $h \in \mathcal{H}$ can be represented in $\Theta(n \lg\lg n)$ bits and evaluated in constant time for $m \leq 2^w$. The perfect hash function can be constructed in expected $O(n)$ time.*

As noted in [4] a corollary of the previous theorem is the following.

Corollary 47. *A sparse array of size $m \geq n$ with n initialized entries that contain $b = \Omega(\lg\lg n)$ bits each can be stored using $O(nb)$ bits, so that any initialized entry can be accessed in $O(1)$ time. The expected preprocessing time of this data structure is $O(n)$.*

7.3 Semi-Dynamic Succinct One-Dimensional Point Reporting

Although Goswami et al. [46] presented a succinct data structure for one-dimensional range reporting, it is not clear what is the construction time of their data structure. In Section 7.5 we utilize succinct data structure techniques to improve the data structure in [4] so that it uses $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits and can be constructed in $O(n \lg^{\varepsilon} m)$ time using $o(n)$ extra bits of space. The details are deferred to Section 7.5 due to their technical nature.

Theorem 48. *There exists a succinct $\mathcal{B}(n,m)+o(\mathcal{B}(n,m))$-bit data structure that supports one-dimensional range reporting queries in $O(k+1)$ time where k is the number of points within the query. Additionally given the point set in sorted order, this data structure can be constructed in expected $O(n\lg^{\varepsilon} m)$ time using $o(n)$-bits workspace.*

The data structure for one-dimensional range reporting can be dynamized so that queries are supported in deterministic $O(k)$ time and updates in expected $O(\lg^{\varepsilon} m)$ time while the space usage is $O(n)$ words [68]. Our aim is to reduce the space to the information theoretic lower bound plus a lower order term. In this section we present a semi-dynamic succinct one-dimensional range reporting data structure that supports queries and deletions but does not support insertions.

Data Structure We store the data structure from Theorem 48 and call it P. We divide the points into blocks of size $\lg^2 m$ and we store predecessor and successor data structures that can answer queries in each block independently using $o(\mathcal{B}(n,m))$ bits as described subsection 6.3. We also store a dynamic data structure [68] D on the endpoints of each block. Furthermore, each block is divided into subblocks of size $\lg n/2$ and stores a dynamic data structure [68] D_i $(1 \leq i \leq n/\lg^2 m)$ on the ranks (within the block) of the endpoints of each subblock. We also store a compressed bit vector([**?**], Theorem 2) B of size n that indicates which points were deleted. Finally, we store a lookup table T that can report for any range the 0 bits in a bit vector of size $\lg n/2$.

Query To report the points within an interval $[a,b]$ we query D on the interval. Then for each point reported with rank k we query the $(\lfloor k/2 \rfloor)^{\text{th}}$ and $(\lfloor k/2 \rfloor + 1)^{\text{st}}$ blocks.

To query the k^{th} block we first reduce the problem to the rank space by finding the rank of the successor of a and the predecessor of b within the block. Next, we query D_k for the non-empty subblocks within the block and use T to report the points in the subblock.

If the query to D does not return any point then either $[a,b]$ is empty or $[a,b]$ is contained fully within a block. To determine which block contains $[a,b]$ we query P to get the rank of a random point in $[a,b]$ from that we determine which block contains $[a,b]$. Afterwards we proceed within the block as described above.

Deletions To delete a point p we first query to check that the interval $[p,p]$ is not empty. We obtain the rank k of p by querying P, and then we set the k^{th} bit in T to 1. Now we know that the point p is in the $s = (2(k \mod \lg^2 m)/\lg n)^{\text{th}}$ subblock of the $b = (k/\lg^2 m)^{\text{th}}$ block. We check if the s^{th} subblock is empty. If that is so we remove its endpoints from $D_{(k/\lg^2 m)}$. Then we check if the b^{th} block is empty. In that case we remove its endpoints from D. The expected running time is $O(\lg^{\varepsilon} m)$.

Space Analysis P uses $\mathcal{B}(n,m)+o(\mathcal{B}(n,m))$ bits and D contains $O(n/\lg^2 m)$ points thus uses $O(n/\lg m)$ bits. Each D_i ($1 \leq i \leq n/\lg^2 m$) contains $O(\lg^2 m/\lg n)$ points from a universe if size $\lg^2 m$ thus uses $O(\lg^2 m \lg\lg m/\lg n)$ bits. The D_i structures use $O(n\lg\lg m/\lg n)$ bits in total. If $\lg\lg m \notin o(\lg n)$ then $n < \lg^c m$ for some constant c. In that case we use a slightly different approach. We reduce the problem to the rank space from the beginning to make the universe size n, so D uses $O(n/\lg n)$ bits and the D_i structures use $O(n\lg\lg n/\lg n)$ bits in total. The table T uses $O(\sqrt{n}\lg^3 n\lg\lg n)$ bits and finally the compressed bit vector uses $o(n)$ as long as the number of deletions is $o(n)$. In total the space remains $\mathcal{B}(n,m)+o(\mathcal{B}(n,m))$ bits.

Construction Time and Workspace P can be constructed in expected $O(n\lg^{\varepsilon} m)$ time using $o(n)$ extra bits of space. D can be constructed in expected $O(n/\lg^{2-\varepsilon} m)$ time using $O(1)$ extra words of space. Each D_i can be constructed in expected $O((\lg^2 m/\lg n)\lg^{\varepsilon}\lg m)$ time using $O(1)$ extra words of space, so all the D_i's can be constructed in expected $O((n/\lg n)\lg^{\varepsilon}\lg m)$ time using $O(1)$ extra words of space. T can be constructed in $o(n)$ time using $o(n)$ extra bits of space. In total the construction time and workspace are dominated by the cost of constructing P and remain the same as in Theorem 48.

Theorem 49. *There exists a semi-dynamic succinct $\mathcal{B}(n,m)+o(\mathcal{B}(n,m))$-bit data structure that supports one-dimensional range reporting queries in $O(k+1)$ time where k is the number of points within the query, and point deletions in expected $O(\lg^{\varepsilon} m)$ time as long as the number of deletions is $o(n)$. Additionally given the point set in sorted order, this data structure can be constructed in expected $O(n\lg^{\varepsilon} m)$ time using $o(n)$-bits workspace.*

7.4 Fully-Dynamic Succinct One-Dimensional Point Reporting

7.4.1 Fully-Dynamic Structure with Amortized Updates

We first present a fully dynamic solution that uses $\mathcal{B}(n,m)+o(\mathcal{B}(n,m))$ bits of space and supports queries in $O(k)$ time and updates in amortized expected $O(\lg^{\varepsilon} m)$ time.

We divide the universe of size m into $n/\lg^2 m$ chunks of equal size and maintain a fully dynamic [68] data structure B to keep track of the nonempty chunks. B is maintained throughout the data structure updates. Whenever a point is inserted we insert both endpoints of its chunk into B. Moreover whenever a chunk becomes empty we remove its endpoints from B. For each chunk b_i ($1 \le i \le n/\lg^2 m$) we maintain two data structures: S_i and D_i. S_i is the compressed semi-dynamic range reporting structure described in Theorem 49 and D_i is the fully dynamic data structure described in [68]. We maintain the invariant that $\text{size}(D_i) < \text{size}(S_i)/\lg^{\varepsilon} n$ for all i where $n = \sum_i \text{size}(S_i)$. Once $\text{size}(D_i) = \text{size}(S_i)/\lg^{\varepsilon} n$ we rebuild S_i and merge D_i with it. The time needed to rebuild S_i will be $O(\text{size}(S_i)\lg^{\varepsilon} m$ which we can charge to the elements inserted into D_i at a cost of $O(\lg^{2\varepsilon} m)$ per element. Moreover if the total number of elements increase by a constant factor or if $n/\lg^{\varepsilon} n$ elements were deleted from the collections S_i we rebuild the whole data structure. The time needed to rebuild the whole structure is $O(n \lg^{\varepsilon} m)$ and will be charged to the new elements inserted if the size doubles at a cost of $O(\lg^{\varepsilon} m)$ per element, or to the elements deleted at a cost of $O(\lg^{2\varepsilon} m)$ per element.

To report all the points within an interval $[a, b]$ we query B to get the non-empty chunks. Whenever a non-empty chunk i is reported we query both S_i and D_i. If $[a, b]$ is completely within one chunk we get its index $i = \lfloor b \lg^2 m/n \rfloor$, and then we query S_i and D_i.

The space used by B is at most $O(n/\lg m)$ bits. and the space used by all the D_i structures is:

$$\begin{aligned} O(n \lg(m \lg^2 m/n)/\lg^{\varepsilon} n) &= O((n \lg(m/n)/\lg^{\varepsilon} n) + (n \lg\lg n/\lg^{\varepsilon} n)) \\ &= o(\mathcal{B}(n, m)). \end{aligned}$$

The space used by all the structures S_i is $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits. In total the space used is $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits.

Theorem 50. *There exist a dynamic succinct $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$-bit data structure that supports one-dimensional range reporting queries in $O(k + 1)$ time where k is the number of points within the query, and updates in amortized expected $O(\lg^{\varepsilon} m)$ time.*

7.4.2 Fully-Dynamic Structure with Worst Case Updates

Next, we present a fully-dynamic succinct one-Dimensional range reporting structure that supports queries in $O(k)$ time and insertions and deletions in expected $O(\lg^{\varepsilon} m)$ time. Our data structure uses techniques similar to the ones presented in [51, 71, 70].

Data Structure We define a parameter $n_f = \Theta(n)$; the value of n_f changes as n becomes too large or too small. We divide m into $(n_f/\lg^2 n_f)$ chunks each of size $((m \lg^2 n_f)/n_f)$ and we store a dynamic range reporting structure B with a universe of size $2(n_f/\lg^2 n_f)$ on the endpoints of the non-empty chunks. For each chunk b where $1 \le b \le (n_f/\lg^2 n_f)$ we store the following:

k_f^b an estimate of k the number of points in the chunk. $k_f^b = \Theta(k)$, the value of k_f^b changes as k becomes too large or too small.

Data Structures $\mathcal{C}_1^b, \ldots, \mathcal{C}_{\lg^\varepsilon n_f}^b$. These structures are the succinct semi-dynamic structures described in the previous section. They partition the chunk into sub-chunks of possibly different sizes, each containing $\Theta(k_f^b / \lg^\varepsilon n_f)$ points.

Data Structures $\mathcal{D}_1^b, \ldots, \mathcal{D}_{\lg^\varepsilon n_f}^b$. These structures are the fully dynamic structures described in [68].

$\mathcal{F}^b$ a fusion tree on the endpoints of the $\mathcal{C}_i^b$ data structures.

Queries are answered in a manner similar to the previous subsection. To report all the points within an interval $[a, b]$ we query B to get the non-empty chunks. Whenever a non-empty chunk (say the b^{th} chunk) is reported we query $\mathcal{F}^b$ to get the sub-chunks it spans. For each sub-chunk (say the s^{th} sub-chunk) we query both $\mathcal{C}_s^b$ and $\mathcal{D}_s^b$.

Insertions To insert the new point p we compute the chunk $b = \lfloor (p \lg^2 n_f)/n_f \rfloor$ that p belongs to. If the b^{th} chunk is empty we insert its endpoints into B. Next, we check if any structure in the $\mathcal{C}^b$ collection is being rebuilt. In that case we spend $\Theta(\lg^{3\varepsilon} n_f)$ time rebuilding it. Then we determine the s^{th} sub-chunk that p belongs to using $\mathcal{F}^b$. Finally, we insert p into $\mathcal{D}_s^b$.

In each chunk we run the following background process. After each series of $\delta = k_f^b / (\lg^{2\varepsilon} n_f \lg \lg n_f)$ insertions we identify the s^{th} sub-chunk with the largest number of inserted points and rebuild $\mathcal{C}_s^b$ during the next δ updates in that chunk. The re-building works as follows. We construct a semi-dynamic data structure $\overline{\mathcal{C}}_s^b = \mathcal{C}_s^b \cup \mathcal{D}_s^b$. If a point is inserted into this sub-chunk, we store it in the additional data structure $\overline{\mathcal{D}}^b$. When $\overline{\mathcal{C}}_s^b$ is completed we set $\mathcal{C}_s^b := \overline{\mathcal{C}}_s^b$ and $\mathcal{D}_s^b := \overline{\mathcal{D}}^b$. Thus at any time only one sub-chunk of a chunk is re-built. This method guarantees that the number of inserted elements into $\mathcal{D}^b$ does not exceed $k_f^b / \lg^\varepsilon n$ as follows from a Theorem of Dietz and Sleator:

Lemma 51 ([26], Theorem 5)**.** *Suppose that $x_1, \ldots, x_g$ are variables that are initially zero. Suppose that the following two steps are iterated:*

(i) we add a non-negative real value a_i to each x_i such that $\sum a_i = 1$

(ii) set the largest x_i to 0.

Then at any time $x_i \leq 1 + h_{g-1}$ for all i, $1 \leq i \leq g$, where h_i denotes the i-th harmonic number.

Let m_s be the number of inserted elements into $\mathcal{D}_s^b$ and $x_s = m_s/\delta$. Every iteration of the background process sets the largest x_s to 0 and during each iteration $\sum x_s$ increases by 1. Hence the value of x_s can be bounded from above by: $x_s \leq 1 + h_{\lg^\varepsilon n_f}$ for all s at all times. Thus $m_s = O((k_f^b/\lg^{2\varepsilon} n_f \lg\lg n_f)\lg\lg n_f) = O(k_f^b/\lg^{2\varepsilon} n_f)$ for all i because $h_i = O(\lg i)$, and the total size of the $\mathcal{D}^b$ collection is $O((k_f^b/\lg^{2\varepsilon} n_f)\lg^\varepsilon n_f) = O(k_f^b/\lg^\varepsilon n_f)$.

Once the value of k_f^b becomes too big or too small we rebuild the whole chunk during the next $k_f^b/\lg^{3\varepsilon} n_f$ updates (spending $O(\lg^{4\varepsilon} n_f)$ time per update). The old chunk is locked such that only deletions are allowed. We rebuild the chunk with an updated value of k_f^b and as points are inserted into the new chunk we delete them from the old one to preserve space. If the size of the sub-chunk becomes too big we split it into two and update $\mathcal{F}^b$ accordingly.

Deletions Deletions are similar to insertions. To delete a point p we compute the chunk $b = \lfloor (p \lg^2 n_f)/n_f \rfloor$ that p belongs to. Then we check if any structure in the $\mathcal{C}^b$ collection is being rebuilt. In that case we spend $\Theta(\lg^{3\varepsilon} n_f)$ time rebuilding it. Next, we determine the sub-chunk s that p belongs to using $\mathcal{F}^b$. Finally, we delete p from $\mathcal{C}_s^b$ and $\mathcal{D}_s^b$.

In each chunk we run a background process similar to the process run for insertions. After each series of δ deletions, we identify the s^{th} sub-chunk with the largest number of deletions and rebuild $\mathcal{C}_s^b$ during the next δ updates in that chunk. This method guarantees that the number of deleted elements in the $\mathcal{C}^b$ collection does not exceed $k_f^b/\lg^\varepsilon n$. If the size of a sub-chunk becomes too small we merge it with the neighboring sub-chunk and update $\mathcal{F}^b$ accordingly. Moreover if a chunk becomes empty we delete its endpoints from B.

Space Analysis The space used by B is $O(n/\lg n)$. The space used by all the $\mathcal{C}_i$ structures in all chunks is $\mathcal{B}(n,m) + o(\mathcal{B}(n,m))$ bits. The total size of all the $\mathcal{D}$ structures is $O(n_f/\lg^\varepsilon n_f)$ so they use at most:

$$\begin{aligned} O(n \lg(m \lg^2 n/n)/\lg^\varepsilon n) &= O((n \lg(m/n)/\lg^\varepsilon n) + (n \lg\lg n/\lg^\varepsilon n)) \\ &= o(\mathcal{B}(n,m)). \end{aligned}$$

The space used by the fusion trees in all chunks is:

$$\begin{aligned} O(n \lg^\varepsilon n \lg(m \lg^2 n/n)/\lg^2 n) &= O((n \lg(m/n)/\lg^{2-\varepsilon} n) + (n \lg\lg n/\lg^{2-\varepsilon} n)) \\ &= o(\mathcal{B}(n,m)). \end{aligned}$$

Thus the total space is $\mathcal{B}(n,m) + o(\mathcal{B}(n,m))$ bits.

Once the value of n_f becomes too big or too small, we rebuild the whole data structure in the background during the next $n_f / \lg^{3\varepsilon} n_f$ updates (spending $O(\lg^{4\varepsilon} n_f)$ time per update). We replace the chunks from left to right. The chunk being replaced is locked such that only deletions are allowed. We rebuild that chunk with an updated value and as points are inserted into the new chunk we delete them from the old one to preserve space.

Theorem 52. *There exist a dynamic succinct $\mathcal{B}(n,m)+o(\mathcal{B}(n,m))$-bit data structure that supports one-dimensional range reporting queries in $O(k+1)$ time where k is the number of points within the query, and updates in expected $O(\lg^{\varepsilon} m)$ time.*

7.5 Succinct Static One-Dimensional Point Reporting With Fast Construction Time

In this section we prove Theorem 48. Denote by T the classic binary tree with 2^w leaves where all leaves have depth w as described in subsection 7.2.1. Let P be the set of nodes in T with non-empty subtrees and V the set of branching nodes in T union the leaves of T and its root. Let T_V be the tree formed from T by deleting all vertices in $T - P$ then contracting all vertices in $P - V$. Given a node $x \in T_V$ denote by $T(x)$ its corresponding node in T, conversely, given a node $x \in V$ denote by $T_V(x)$ its corresponding node in T_V. We fix a constant $\varepsilon = 1/k$, and let $H_i = \lg^{(k-i)/k} m$ where $1 \leq i < k$. Finally, given a node u in T we define $\pi_i(u)$ to be the nearest ancestor of u whose depth is a multiple of H_i.

Data Structure We store the coordinates of the points in $\mathcal{B}(n,m) + o(\mathcal{B}(n,m))$ bits. Also we store T_V using $4n + o(n)$ bits using the tree representation of Navarro and Sadakane [80] which allows the following operations in constant time:

lmost-leaf(i) / rmost-leaf(i): given the preorder number of a node return the preorder number of the leftmost(rightmost) leaf of node i.

leaf-rank(i): given the preorder number of a leaf i returns the number of leafs to the left of i.

In addition we store in $o(n)$ bits the index described in [44] that enables conversion between τ-names of the nodes in T_V and their preorder numbers.

To maintain the mapping between the labels of the branching nodes in T with their preorder numbers in T_V we store the following tables using Corollary 47:

M_1: for each node $x \in V$ with root $(T_V(x)) = T_V(x)$ we store the value $\tau_1(T_V(x))$ in a table M_1. Since T_V is a binary tree, it is possible that $T_V(x)$ belongs to two different micro trees μ_0 and μ_1. In that case we store both $p(M_0)$ and $p(M_1)$.

M_2: for each node $x \in V$ we store in a table M_2 the values $\tau_2(T_V(x))$, $\tau_3(T_V(x))$, and a bit that indicates to which micro tree does $T_V(x)$ belongs to if root $(T_V(x))$ belongs to two different micro trees.

M_3: for each node $x \in V$ we store the distance from x to $T(\text{root}\,(T_V(x)))$ in a table M_3.

Finally, given a node in P we need to compute its nearest branching ancestor. To achieve this we use the same technique as in [4] but with bootstrapping. We store $k-1$ tables $D_1, \ldots, D_{(k-1)}$ using Corollary 47. D_1 contains the distances to the nearest branching ancestor for all nodes u in P satisfying $\pi_1(u) = u$. D_i $(2 \leq i < k-1)$ contains the distances to the nearest branching ancestor for all nodes u in P satisfying the conditions $\pi_{(i-1)}(u)$ is closer to u than the nearest branching ancestor of u and $\pi_i(u) = u$. Finally, $D_{(k-1)}$ contains the distances to the nearest branching ancestor for all nodes u in P satisfying the conditions: $\pi_{(k-2)}(u)$ is closer to u than the nearest branching ancestor of u and $\pi_{(k-1)}(u) = u$, or $\pi_{(k-1)}(u)$ and $\pi_{(k-2)}(u)$ are closer to u than the nearest branching ancestor of u. More formally we define:

B_1: $B_1(z) = 1$ if $\pi_1(z) = z$ and $\exists u \in V$ such that $\pi_1(u) = z$, otherwise $B_1(z) = 0$.

$B_i(1 < i < k)$: $B_i(z) = 1$ if $B_{(i-1)}(\pi_{(i-1)}(z)) = 1$, $\pi_i(z) = z$, and $\exists u \in V$ such that $\pi_i(u) = z$, otherwise $B_i(z) = 0$

and store the following tables using Corollary 47:

D_1: which contain the distance to the nearest branching ancestor for all nodes u in P satisfying $\pi_1(u) = u$.

D_i $(2 \leq i < k-1)$: which contain the distance to the nearest branching ancestor for all nodes u in P satisfying: $B_{(i-1)}(\pi_{(i-1)}(u)) = 1$ and $\pi_i(u) = u$.

$D_{(k-1)}$: which contain the distance to the nearest branching ancestor for all nodes u in P satisfying: $B_{(k-2)}(\pi_{(k-2)}(u)) = 1$ and $(\pi_{(k-1)}(u) = u$ or $B_{(k-1)}(\pi_{(k-1)}(u)) = 1)$.

Query Given a query FindAny(a, b) we first find the nearest common ancestor p of a and b. Then we get $k-1$ candidate nearest branching ancestor $v_1, \ldots, v_{(k-1)}$ of p using $D_1, \ldots, D_{(k-1)}$. Afterwards for each v_i we need to compute the preorder number of v_i in T_V. To achieve this goal we get $\tau_2(T_V(v_i)), \tau_3(T_V(v_i))$, and the bit b indicating which micro tree v_i belongs to from M_2. Next, we compute $u_i = T(\text{root}\,(T_V(v)))$ after obtaining its distance from v_i using M_3. Afterwards we query M_1 for $\tau_1(T_V(u_i)) = p(\mu_b)$. After obtaining the τ-name of $T_V(v_i)$ we get its preorder number, and then we check the ranks of the leftmost and rightmost leaves of v_i's left and right child. If one of them is within $[a, b]$ we return its value. If for all v_i no element was found within $[a, b]$ we return that $S \cap [a, b]$ is empty.

Space Analysis Storing the points coordinates uses $\mathcal{B}(n, m)$ bits. The tree T_V uses $4n + o(n)$ bits. The tables M_2, M_3 contain $O(n)$ entries each of size $O(\lg\lg m)$ so they use $O(n \lg\lg m)$ bits. The table M_1 contains $O(n/\lg n)$ entries each of size $O(\lg n)$ so it uses $O(n)$ bits. The table D_1 contains $O(n \lg m / \lg^{(k-1)/k} m) = O(n \lg^{\varepsilon} m)$ entries of size $O(\lg\lg m)$ bits each so it uses $O(n \lg^{\varepsilon} m \lg\lg m)$ bits. Moreover each table D_i $(1 < i < k-1)$ contains $O(n(H_{(i-1)}/H_i)) = O(n \lg^{\varepsilon} m)$ entries each of size $O(\lg\lg m)$ bits so they use a total of $O(n \lg^{\varepsilon} m \lg\lg m)$ bits. Finally, we need to bound the size of D_{k-1}. The number of entries due to $\pi_{k-1}(u) = u$ is $O(n(H_{(k-1)}/H_k)) = O(n \lg^{\varepsilon} m)$. To bound the entries due to $B_{k-1}(\pi_{k-1}(u)) = 1$ notice that the subtree T_z of height $H_{(k-1)}$ rooted at $z = \pi_{(k-1)}(u)$ will contain $s > 1$ entries, and will have at most $s + 1 < 2s$ leaves that are nodes in P. Thus it will contribute at most $(2H_{(k-1)}s)$ entries. Since there are at most $n-1$ branching nodes the total number of entries due to $B_{(k-1)}(\pi_{(k-1)}(u)) = 1$ is $2H_{(k-1)}n = O(n \lg^{\varepsilon} m)$. D_{k-1} uses $O(n \lg^{\varepsilon} m \lg\lg m)$ bits because each entry in $D_{(k-1)}$ is of size $O(\lg\lg m)$ bits. In total the space used is $\mathcal{B}(n, m) + O(n) + O(n \lg^{\varepsilon} m \lg\lg m)$ bits.

Construction Time In a manner similar to [4] we can identify V in $O(n)$ time, and then construct T_V also in $O(n)$ time. The tables M_1, M_2,and M_3 can be constructed in expected $O(n(\lg\lg m))$ time. Finally, the tables B_i where $1 \leq i < k$ can be constructed in expected $O(n \lg^{\varepsilon} m)$ time by identifying the $O(n \lg^{\varepsilon} m)$ entries and building the tables. The workspace is $O(n)$ words.

Reducing Space To further reduce the space we use a well known trick and split the universe $[m]$ into n ranges $r_1, \ldots, r_n$ each of size m/n. We construct a bit vector B of size $2n$ bits with rank and select queries. B stores a zero for each range r_i followed by n_i ones where n_i is the number of points in the range r_i. To count the number of points before a range r_i we use a select query to get the position of the i^{th} zero in B, and then use a rank query to count the number of ones before that position. We store a separate data structure for each range. To locate the data structures for any range r_i within A we count the number of points in the ranges r_j for $j < i$, and then scale that number. Given a query FindAny(a, b) we check if $[a, b]$ spans a non-empty range as follows. We use a rank query to get the number of ones k before the $\lfloor (an/m) \rfloor$ zero. Then we check if the $(k+1)^{\text{th}}$ element is within $[a, b]$ and return it in that case. Otherwise we query the data structure corresponding to the $(\lceil (an/m) \rceil)^{\text{th}}$ range. The total space used is $\mathcal{B}(n, m) + O(n) + O(n(\lg (m/n))^{\varepsilon} \lg \lg (m/n)) = \mathcal{B}(n, m) + o(\mathcal{B}(n, m)) + O(n)$ bits.

If $O(n)$ is not a lower order term then $n > m/c$ for some constant c. In that case we adopt a different approach and store the points in a compressed bit vector of size m. To answer a query FindAny(a, b) we use a rank query to get the number of ones k before position a, and then we use a select query to get the position of the $(k+1)^{\text{th}}$ one. If that position is within $[a, b]$ we return it otherwise $S \cap [a, b]$ is empty. The space used is now $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits.

Reducing Construction Workspace To further improve the construction workspace we divide n into $\lg^2 m$ ranges each containing $n/\lg^2 m$ points and build a separate data structure for each of them. We note that the universe size in each range may vary. Additionally we store a fusion tree F on the endpoints of each range. Given a query FindAny(a, b), we check if the successor of a in F is within $[a, b]$ and return it in that case. Otherwise we query the range containing the successor of a.

Chapter 8

Conclusion

In conclusion, we studied various combinatorial objects from the perspective of succinct and compact data structures.

We started this thesis in Chapter 3 by presenting compact representations for unlabeled permutations. Given an arbitrary unlabeled permutation π, we store it compactly such that $\pi^k(i)$ can be computed quickly for any i and any integer power k. We considered the problem in several scenarios.

In the first scenario we assigned labels to elements so that queries are answered by just examining the labels of the queried elements. We showed that a label space of $\sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \cdot i$ is necessary and sufficient. In other words, $2 \lg n$ bits of space are necessary and sufficient for representing each of the labels. In the second scenario we assigned labels to the n elements from the label set $\{1, \ldots, cn\}$ where $c \geq 1$ is a constant. We showed that $\Theta(\sqrt{n})$ bits are necessary and sufficient to represent the permutation. Moreover, we supported queries in such a structure in $O(1)$ time. Finally, in the third scenario we assigned labels to the n elements from the label set $\{1, \ldots, cn^{1+\varepsilon}\}$ where c is a constant and $0 < \varepsilon < 1$. We showed that $\Theta(n^{(1-\varepsilon)/2})$ bits are necessary and sufficient to represent the permutation. We also supported queries in such a structure in $O(1)$ time.

Then in Chapter 4 we covered the problem of powering permutations in place. Given a permutation of n elements, stored as an array, we addressed the problem of replacing the permutation by its k^{th} power while using $o(n)$ bits of extra storage. We presented an algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\varepsilon}\})$ additional bits.

Afterwards, we covered a bunch of data structures for range reporting problems. In Chapter 5, we present a data structure for multi-dimensional range mode queries that that uses $O(s_n + (n/\Delta)^{2d}/w)$ words and answers queries in $O(\Delta \cdot t_n)$ time, where s_n and t_n are the space and query time of a data structure that supports orthogonal range counting queries, thus improving a result in [20].

In Chapter 6, we presented a succinct data structure for static one-dimensional approximate color counting that uses $\mathcal{B}(n, m) + O(n) + o(\mathcal{B}(n, m))$ bits, Thus we showed, somewhat counter-intuitively, that it is not necessary to store colors of the points in order to answer approximate color counting queries. Moreover, our structure answers queries in constant time, thus, solving an open problem from [86].

We then extended the techniques presented to describe a data structure for the one dimensional approximate median reporting problem. We present a data structure that uses $\mathcal{B}(n, m) + O(n) + o(\mathcal{B}(n, m))$ bits and answers queries in constant time. In the special case where the points are in the rank space our data structure uses only $O(n)$ bits, thus improving a result from [16].

Then we turned to succinct data structures for color reporting. We described a data structure that uses $\mathcal{B}(n, m) + nH_d(S) + o(\mathcal{B}(n, m)) + o(n \lg \sigma)$ bits and answers queries in $O(k + 1)$ time, where k is the number of colors in the answer, and $nH_d(S)$ ($d = \log_\sigma n$) is the d-th order empirical entropy of the color sequence. We also presented a succinct dynamic data structure with constrained updates that uses $nH_d(S) + o(n \lg \sigma)$ bits for one-dimensional color reporting, restricted to the case when the points are in the rank space.

Finally, in Chapter 7 we presented a succinct dynamic data structure for the one-dimensional range reporting problem. Our data structure uses $\mathcal{B}(n, m) + o(\mathcal{B}(n, m))$ bits, supports updates in $O(\lg^\varepsilon m)$ time, and answers queries in optimal $O(k)$ time where k is the number of points in the answer and m is the universe size.

As future work, we aim to focus on the succinct and compact representation of other combinatorial objects. We are currently investigating the compact representation of other types of range queries such as approximate range mode and approximate k^{th} selection for arbitrary k.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*, 1974. *Reading: Addison-Wesley*, pages 207–209, 1987.

[2] Stephen Alstrup, Philip Bille, and Theis Rauhe. Labeling schemes for small distances in trees. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 689–698, 2003.

[3] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 198–207, 2000.

[4] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Optimal static range reporting in one dimension. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 476–482, 2001.

[5] Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Reducing the space requirement of lz-index. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, pages 318–329, 2006.

[6] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*, volume 463. ACM Press, New York, 1999.

[7] Jérémy Barbay, Luca Castelli Aleardi, Meng He, and J. Ian Munro. Succinct representation of labeled graphs. In *Algorithms and Computation, 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007, Proceedings*, pages 316–328, 2007.

[8] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comput. Sci.*, 387(3):284–297, 2007.

[9] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):52, 2011.

[10] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002.

[11] Djamal Belazzougui and Gonzalo Navarro. New lower and upper bounds for representing sequences. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 181–192, 2012.

[12] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[13] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, S. Srinivasa Rao, and Oren Weimann. Random access to grammar-compressed strings. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 373–389. Society for Industrial and Applied Mathematics, 2011.

[14] Peter Van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science, FOCS 1975, Berkeley, California, USA, October 13-15, 1975*, pages 75–84, 1975.

[15] Prosenjit Bose, Eric Y. Chen, Meng He, Anil Maheshwari, and Pat Morin. Succinct geometric indexes supporting point location queries. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 635–644, 2009.

[16] Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *22nd Annual Symposium on Theoretical Aspects of Computer Science, STACS 2005, Stuttgart, Germany, February 24-26, 2005, Proceedings*, pages 377–388, 2005.

[17] Panayiotis Bozanis, Nectarios Kitsios, Christos Makris, and Athanasios Tsakalidis. New upper bounds for generalized intersection searching problems. In *Proceedings of the 22nd International Colloquium on Automata, Languages, and Programming*, pages 464–474. Springer, 1995.

[18] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer System Sciences*, 60(3):630–659, 2000.

[19] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

[20] Timothy M. Chan, Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Bryan T. Wilkinson. Linear-space data structures for range mode query in arrays. *Theory of Computing Systems*, pages 1–23, 2013.

[21] Timothy M. Chan, Stephane Durocher, Matthew Skala, and Bryan T. Wilkinson. Linear-space data structures for range minority query in arrays. In *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings*, pages 295–306, 2012.

[22] Timothy M. Chan and Bryan T Wilkinson. Adaptive and approximate orthogonal range counting. *ACM Transactions on Algorithms (TALG)*, 12(4):45, 2016.

[23] David Clark. *Compact Pat trees.* PhD thesis, University of Waterloo, 1997.

[24] Mariano Paulo Consens and Timothy Snider. Maintaining very large indexes supporting efficient relational querying, August 14 2001. US Patent 6,275,822.

[25] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Noy Rotbart. Dynamic and multi-functional labeling schemes. *arXiv preprint arXiv:1404.4982*, 2014.

[26] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth Annual ACM symposium on Theory of Computing*, pages 365–372. ACM, 1987.

[27] Stephane Durocher, Hicham El-Zein, J. Ian Munro, and Sharma V. Thankachan. Low space data structures for geometric range mode query. In *Proceedings of the 26th Canadian Conference on Computational Geometry, CCCG 2014, Halifax, Nova Scotia, Canada, 2014*, 2014.

[28] Stephane Durocher, Hicham El-Zein, J. Ian Munro, and Sharma V. Thankachan. Low space data structures for geometric range mode query. *Theor. Comput. Sci.*, 581:97–101, 2015.

[29] Stephane Durocher, Rahul Shah, Matthew Skala, and Sharma V. Thankachan. Linear-space data structures for range frequency queries on arrays and trees. In *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings*, pages 325–336, 2013.

[30] Hicham El-Zein. On the succinct representation of equivalence classes. Master's thesis, University of Waterloo, 2014.

[31] Hicham El-Zein, J. Ian Munro, and Yakov Nekrich. Succinct color searching in one dimension. In *28th International Symposium on Algorithms and Computation, ISAAC 2017, December 9-12, 2017, Phuket, Thailand*, pages 30:1–30:11, 2017.

[32] Hicham El-Zein, J. Ian Munro, and Yakov Nekrich. Succinct dynamic one-dimensional point reporting. In *16th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2018, June 18-20, 2018, Malmö, Sweden*, 2018.

[33] Hicham El-Zein, J. Ian Munro, and Matthew Robertson. Raising permutations to powers in place. In *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*, pages 29:1–29:12, 2016.

[34] Hicham El-Zein, J. Ian Munro, and Siwei Yang. On the succinct representation of unlabeled permutations. In *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 49–59, 2015.

[35] Arash Farzan and J. Ian Munro. Succinct representations of arbitrary graphs. In *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, pages 393–404, 2008.

[36] Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014.

[37] Arash Farzan, J. Ian Munro, and Rajeev Raman. Succinct indices for range queries with applications to orthogonal range maxima. In *Proceedings of the 39th International Colloquium on Automata, Languages, and Programming, Part I*, pages 327–338, 2012.

[38] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[39] Johannes Fischer. Optimal succinctness for range minimum queries. In *Proceedings of the 9th Latin American Symposium on Theoretical Informatics*, pages 158–169. Springer, 2010.

[40] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[41] Travis Gagie and Juha Kärkkäinen. Counting colours in compressed strings. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching, CPM 2011*, pages 197–207, 2011.

[42] Travis Gagie, Juha Kärkkäinen, Gonzalo Navarro, and Simon J Puglisi. Colored range queries and document retrieval. *Theoretical Computer Science*, 483:36–50, 2013.

[43] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14, 2004*, pages 1–10, 2004.

[44] Richard F. Geary, Rajeev Raman, and Venkatesh Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms (TALG)*, 2(4):510–534, 2006.

[45] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373, 2006.

[46] Mayank Goswami, Allan Grønlund Jørgensen, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 769–775, 2015.

[47] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Cell probe lower bounds and approximations for range mode. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I*, pages 605–616, 2010.

[48] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*, pages 517–528, 2009.

[49] Roberto Grossi, Rajeev Raman, S. Srinivasa Rao, and Rossano Venturini. Dynamic compressed strings with random access. In *Automata, Languages, and Programming -*

40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I, pages 504–515, 2013.

[50] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[51] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. A framework for dynamizing succinct data structures. In *International Colloquium on Automata, Languages, and Programming*, pages 521–532. Springer, 2007.

[52] Prosenjit Gupta, Ravi Janardan, and Michiel H. M. Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *J. Algorithms*, 19(2):282–317, 1995.

[53] Godfrey H. Hardy and Srinivasa Ramanujan. Asymptotic formulae in combinatory analysis. *Proceedings of the London Mathematical Society*, 2(1):75–115, 1918.

[54] Meng He. Succinct and implicit data structures for computational geometry. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 216–235, 2013.

[55] Meng He, J. Ian Munro, and S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 23–32, 2005.

[56] Daniel S. Hirschberg and James Bartlett Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

[57] Christiaan TM. Jacobs and Peter Van Emde Boas. Two results on tables. *Information Processing Letters*, 22(1):43–48, 1986.

[58] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS '89, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 549–554, 1989.

[59] Guy Joseph Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1988.

[60] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*, pages 558–568, 2004.

[61] Ravi Janardan and Mario Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry & Applications*, 3(01):39–69, 1993.

[62] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596–603, 1992.

[63] Michal Katz, Nir A. Katz, Amos Korman, and David Peleg. Labeling schemes for flow and connectivity. *SIAM Journal on Computing*, 34(1):23–40, 2004.

[64] Danny Krizanc, Pat Morin, and Michiel Smid. Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1–17, 2005.

[65] Moshe Lewenstein, J. Ian Munro, and Venkatesh Raman. Succinct data structures for representing equivalence classes. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16-18, 2013, Proceedings*, pages 502–512, 2013.

[66] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of Computing*, pages 103–111. ACM, 1995.

[67] Christian Worm Mortensen. Generalized static orthogonal range searching in less space. Technical report, IT University of Copenhagen, 2003.

[68] Christian Worm Mortensen, Rasmus Pagh, and Mihai Patrascu. On dynamic range reporting in one dimension. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 104–111, 2005.

[69] J. Ian Munro. Tables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.

[70] J. Ian Munro and Yakov Nekrich. Compressed data structures for dynamic sequences. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 891–902, 2015.

[71] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Dynamic data structures for document collections and graphs. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 277–289. ACM, 2015.

[72] J. Ian Munro and Patrick K. Nicholson. Succinct posets. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 743–754, 2012.

[73] J. Ian Munro and Patrick K. Nicholson. Compressed representations of graphs. In *Encyclopedia of Algorithms*, pages 382–386. 2016.

[74] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[75] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 118–126, 1997.

[76] J. Ian Munro and S. Srinivasa Rao. Succinct representation of data structures. In *Handbook of Data Structures and Applications.* 2004.

[77] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666. Society for Industrial and Applied Mathematics, 2002.

[78] Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016.

[79] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2, 2007.

[80] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):16, 2014.

[81] Yakov Nekrich. Efficient range searching for categorical and plain data. *ACM Trans. Database Syst.*, 39(1):9:1–9:21, 2014.

[82] Yakov Nekrich and Jeffrey Scott Vitter. Optimal color range reporting in one dimension. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 743–754, 2013.

[83] Mihai Patrascu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 232–240, 2006.

[84] Holger Petersen. Improved bounds for range mode and range median queries. In *SOFSEM 2008: Theory and Practice of Computer Science, 34th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 19-25, 2008, Proceedings*, pages 418–423, 2008.

[85] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Information Processing Letters*, 109(4):225–228, 2009.

[86] Saladi Rahul. Approximate range counting revisited. In *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, pages 55:1–55:15, 2017.

[87] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[88] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), 2007.

[89] Matthew Robertson. Inverting permutations in place. Master's thesis, University of Waterloo, 2015.

[90] Michael Rubinstein. The distribution of solutions to $xy = n \bmod a$ with an application to factoring integers. *arXiv preprint math/0610612v5*, 2012.

[91] Milan Ruzic. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, pages 84–95, 2008.

[92] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[93] Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys*, 9(2):137–164, 1977.

[94] Matthew Skala. Array range queries. In *Proc. Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *LNCS*, pages 333–350. Springer, 2013.

[95] György Turán. On the succinct representation of graphs. *Discrete Applied Mathematics*, 8(3):289–294, 1984.

[96] Andries Van Dam and David Evans. A compact data structure for storing, retrieving and manipulating line drawings. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, pages 601–610. ACM, 1967.

[97] Peter Van Emde Boas, Robert Kaas, and Erik Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(1):99–127, 1976.

[98] I. Vinogradov. *Selected works. With a biography by K. K. Mardzhanishvili.Translated from the Russian by Naidu Psv. Translation edited by Yu. A.* Springer-Verlag, Berlin, 1985.

[99] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.