Upper and Lower Bounds for Text Indexing Data Structures

by

Alexander Golynski

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, 2007

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.

(Alexander Golynski)

Abstract

The main goal of this thesis is to investigate the complexity of a variety of problems related to text indexing and text searching. We present new data structures that can be used as building blocks for full-text indices which occupies minute space (FM-indexes) and wavelet trees. These data structures also can be used to represent labeled trees and posting lists. Labeled trees are applied in XML documents, and posting lists in search engines.

The main emphasis of this thesis is on lower bounds for time-space tradeoffs for the following problems: the rank/select problem, the problem of representing a string of balanced parentheses, the text retrieval problem, the problem of computing a permutation and its inverse, and the problem of representing a binary relation. These results are divided in two groups: lower bounds in the cell probe model and lower bounds in the indexing model.

The cell probe model is the most natural and widely accepted framework for studying data structures. In this model, we are concerned with the total space used by a data structure and the total number of accesses (probes) it performs to memory, while computation is free of charge. The indexing model imposes an additional restriction on the storage: the object in question must be stored in its raw form together with a small index that facilitates an efficient implementation of a given set of queries, e.g. finding rank, select, matching parenthesis, or an occurrence of a given pattern in a given text (for the text retrieval problem).

We propose a new technique for proving lower bounds in the indexing model and use it to obtain lower bounds for the rank/select problem and the balanced parentheses problem. We also improve the existing techniques of Demaine and López-Ortiz using compression and present stronger lower bounds for the text retrieval problem in the indexing model.

The most important result of this thesis is a new technique for cell probe lower bounds. We demonstrate its strength by proving new lower bounds for the problem of representing permutations, the text retrieval problem, and the problem of representing binary relations. (Previously, there were no non-trivial results known for these problems.) In addition, we note that the lower bounds for the permutations problem and the binary relations problem are tight for a wide range of parameters, e.g. the running time of queries, the size and density of the relation.

Acknowledgments

I am deeply thankful to my supervisors Ian Munro and Prabhakar Ragde for their valuable guidance, encouragement, patience, and for providing an advice whenever I was seeking it. They allowed me a great deal of academic freedom in pursuing the topics of my interest, which made my academic life in Waterloo enjoyable. Professor Alejandro López-Ortiz supervised me during the first part of my program and his support is greatly appreciated, discussions with him gave me a broad view of many research fields and topics. I am also thankful to my coauthors without whom research would be much less fun.

Many thanks to my committee members Faith Ellen, Jérémy Barbay, Timothy Chan, Ashwin Nayak, Prabhakar Ragde, and Ian Munro for valuable feedback that greatly helped to improve this thesis. Faith Ellen did tremendous job in thorough reading and providing criticism that not only led to a better presentation of the existing results, but also inspired new ones. I am much obliged to Ian Munro, Prabhakar Ragde, and especially to Jérémy Barbay for their time and dedication to proof-reading drafts of this thesis. I would like to acknowledge the administrative stuff of the school, and especially Margaret Towell.

My research was supported by NSERC grants of Prabhakar Ragde, Ian Munro, and Alejandro López-Ortiz; also by scholarships and awards of David R. Cheriton School of Computer Science, University of Waterloo, and Ontario Graduate Scholarship program. Thank you!

I also thank my friends Pranab, Marina, Sonia, Michael, Oleg, Denis, Aleh, and others who made my time in Waterloo wonderful.

Finally, I am very grateful to my mother for her support.

Contents

List of Figures

Nomenclature

$\texttt{access}_T(i)$ p consecutive characters of the text T starting at position i, page 15

β forward coverage factor, page 111

β' inverse coverage factor, page 111

β, β^* coverage factor, page 111

$\texttt{bin\_acc}_B(i)$, $B[i]$ the value of the bit in position i of B, page 19

$\texttt{bin\_rank}_B(b, i)$ the number of b bits up to and including position i in B, page 13

$\texttt{bin\_sel}_B(b, x)$ the position of the x-th b-bit in B, page 14

$\mathcal{A}[i]$ indicates whether q_i is present (Section 4.3), page 95

$\mathcal{C}_{\|}$ set of remaining cells after step k (Chapter 5), page 110

$\mathcal{C}'[i]$ indicates whether the i-th chunk is present (Section 4.3), page 96

$\mathcal{D}$ positions of deleted cells (Chapter 5), page 110

$\mathcal{F}_k$ set of remaining forward queries after step k (Chapter 5), page 110

$\mathcal{I}_k$ set of remaining inverse queries after step k (Chapter 5), page 110

$\mathcal{L}$ unary separator sequence of $\mathcal{P}$ (Chapter 4), page 89

$\mathcal{M}$ array of encoded small permutations (Chapter 5), page 110

$\mathcal{P}$ sequence of probed bits or cells (Chapter 4), page 87

$\mathcal{Q}, \mathcal{Q}^*$ sets of queries, page 12

$\mathcal{R}$ contents of the remaining cells (Chapter 5), page 110

$\mathcal{R}$ contents of the undiscovered bits (Section 4.3), page 95

$\mathcal{R}'$ sequence of the bits in absent chunks (Section 4.3), page 96

$\texttt{card}(B)$ cardinality of B, page 51

$\texttt{col\_nb}_R(j)$ the number of 1-bits (cardinality) in the j-th column, page 31

$\texttt{col\_sel}_R(x, j)$ the row of the x-th occurrence of 1-bit in the j-th column of R, page 31

`COUNT` count index, page 22

$\texttt{findmatch}(i)$ the location of the matching parenthesis to the parenthesis at the position i, page 12

γ number of queries in $\mathcal{Q}^*$, page 51

$\mathcal{H}$ set of hard instances, page 52

μ_{d_k} permutation stored in $\mathcal{M}$ on the k-th step, defines a map between $F'(d_k)$ and $I'(d_k)$ (Chapter 5), page 111

$\texttt{occ}_T(c)$ the number of occurrences of character c in T, page 29

π permutation, page 4

π^{-1} inverse of a permutation, page 4

`RANK` binary rank index, page 25

ρ the inverse density of a binary matrix, $\rho = nm/f$, page 31

$\texttt{row\_nb}_R(i)$ the number of 1-bits (cardinality) in the i-th row, page 31

$\texttt{row\_rank}_R(i, j)$ the number of 1-bits in the i-th row of R up to and including column j, page 30

$\texttt{row\_rank}_R(i, j)$ the number of 1-bits in the j-th column of R up to and including row i, page 31

$\texttt{row\_sel}_R(i, x)$ the column of the x-th occurrence of 1-bit in the i-th row of R, page 30

$\texttt{search}_T(X, j)$ the j-th occurrence of pattern X of length p in T, page 15

SELECT binary select index, page 25

Σ alphabet, page 5

σ size of alphabet, page 5

$\texttt{str\_acc}_T(i)$, $T[i]$ the i-th character of T, page 29

$\texttt{str\_rank}_T(c, i)$ the number of occurrences of character c before and including the position i, page 29

$\texttt{str\_sel}_T(c, x)$ the position of x-th occurrence of character c in T, page 29

$\texttt{tab\_acc}_R(i, j)$ the element on the intersection of the i-th row and the j-th column, page 31

Υ the information-theoretic minimum to represent an object in question, page 8

a the number of present simulations (Section 4.3), page 96

B binary vector, page 19

b-probe a bit probe that returns value b, where $b \in \{0, 1\}$, page 51

BW Burrows-Wheeler transform, page 27

$C(x)$ set of hard instances compatible with a leaf x (Chapter 3), page 54

d_k deleted cell on step k (Chapter 5), page 110

f the number of 1 entries in a binary matrix (cardinality), page 31

$F'(l)$ set of forward queries q so that q and q^{-1} use cell l (Chapter 5), page 111

$F(l)$ set of forward queries that use the l-th cell (Chapter 5), page 110

$G, G(\mathcal{Q}^*)$ choices tree, page 51

h number of levels in the count index (Section 2.1), page 22

H_0 empirical 0-th order entropy, page 29

H_k empirical k-th order entropy, page 29

$I'(l)$ set of inverse queries q so that q and q^{-1} use cell l (Chapter 5), page 111

$I(l)$ set of inverse queries that use the l-th cell (Chapter 5), page 110

I, I_B index, page 13

k threshold on the number of discovered bits in a chunk (Section 4.3), page 94

L length of text, page 5

l_k length of a chunk of level k (Section 2.1), page 22

l_k length of the i-th slice in the split lemma (Section 2.2), page 36

m cardinality of a bit vector (Section 2.1 and Chapter 3), page 25

m number of rows in a binary matrix (Section 2.2), page 27

m_i cardinality of the i-th block (Chapter 3), page 55

n length of bit vector (Section 2.1 and Chapter 3), page 21

n number of columns in a binary matrix (Section 2.2), page 27

n_i length of the i-th block (Chapter 3), page 55

p length of pattern, page 10

$P(d_k)$ set of protected cells on step k (Chapter 5), page 110

q, q_i query, page 51

q^{-1} reciprocal query to q (Chapter 5), page 110

r size of the index (space cost) for the indexing model (Chapter 3 and Chapter 4), redundancy for the non-indexing model (Chapter 5), page 8

S array of bits or cells of a data structure, page 8

SA Suffix array, page 27

T text (binary and general alphabet), page 5

t, t' running time for upper bounds (Chapter 2), time cost for lower bounds (Chapter 3, Chapter 4, and Chapter 5), page 8

t_a Run time of `str_acc`, page 34

t_r Run time of `str_rank`, page 34

t_s Run time of `str_sel`, page 34

U $U = n - t\gamma$, the total number of unprobed bits (Chapter 3), page 55

u_i the number of unprobed bits in the i-th block (Chapter 3), page 55

V the total number of unprobed 1-bits (Chapter 3), page 55

v the number of overlap bits (Section 4.3), page 97

v_i the number of unprobed 1-bits in the i-th block (Chapter 3), page 55

w word size, page 12

X pattern, page 10

Y^{-1} set reciprocal queries to Y (Chapter 5), page 110

y_i the number of 1-probes performed on the i-th block (Chapter 3), page 55

z number of removal steps (Chapter 5), page 110

$Z(x)$ set of instances compatible with a leaf x (Chapter 3), page 52

`BINREL` binary relations problem, page 15

`PARENTHESES` balanced parentheses problem, page 14

`PERMS` permutations problem, page 14

`TEXTSEARCH` binary text search problem, page 15

Chapter 1

Introduction

The goal of this thesis is to investigate efficient representations of text data, and the complexity of a variety of problems related to text indexing and text searching. Text retrieval problems have gained importance in recent years. For example, Google's index expanded by a factor of 1000 from 1998 to 2005 (according to the Anna Patterson's blog dated June 2007). Their coverage is still growing rapidly: currently (in 2007) they are estimated to index approximately 50 billion pages, which is a factor of 6 increase over the 8 billion pages in the year 2005. There has been a large body of work devoted to text retrieval problems. For example, the algorithm of Knuth, Morris, and Pratt [35] (e.g. the grep utility in UNIX) has complexity $O(|T| + |X|)$ on a text T and a pattern X. This algorithm uses some precomputation on the pattern, but not on the text. This is, of course, as well as we can do on an unpreprocessed text. If we expect to perform searches frequently, it is desirable on reasonable sized texts and essential on large ones to preprocess the text to permit faster searches. The large scale applications, e.g. Google, take advantage of the precomputation, e.g. search engines can precompute inverted lists that for each keyword store the list of web pages that contain this keyword. Common data structures that are used for text searching problem include suffix trees [11, 22, 30, 39, 52, 54], suffix arrays [28, 29, 34, 38, 49, 50], FM-indices [14, 15, 16], and wavelet trees [26]. Recent developments include the results by Golynski et al. [25] and Barbay et al. [3] that can be used as building blocks in these data structures (see [46] for an up-to-date survey of text indexing data structures). Such data structures provide various tradeoffs (upper bounds for

time-space tradeoffs) between the size of the storage and the time to perform text retrieval operations, such as searching for a pattern. To facilitate further progress in this direction, it is important to understand the intrinsic limitations (lower bounds for time-space tradeoffs) imposed on any type of data structure for the problem.

In the first part of this thesis, we will develop several data structures that address the problem of representing binary vectors, strings, binary relations, and the problem of indexing and searching in textual data (e.g. XML documents). Our concern is with space-efficient data structures that allow fast run times for some specified set of data retrieval operations. More specifically, we are interested in so-called *succinct data structures*. By the term "succinct" we informally mean a data structure that uses close to the minimum amount of space required to represent the combinatorial objects in question while performing the required operations quickly. The ideal would be to represent the objects in the information-theoretic minimum space and to perform the operations in constant time. (This, of course, may not be possible.) Our focus is on representing *static* objects; that is, the efficient implementation of update operations is not considered. In the second part, perhaps more significantly, we prove a number of lower bounds on the tradeoffs between time and space for the problems related to text searching in preprocessed data.

1.1 Indexing Data Structures

We first consider an interesting class of *indexing data structures*. An indexing data structure explicitly stores a given object, such as a text string, plus a small *index* to facilitate the implementation of retrieval operations. Gál and Miltersen [19] call them *systematic data structures*, and call explicit representation *verbatim*; however we use the term *raw form* in this thesis. The notion of raw form is not well defined for some combinatorial objects, e.g. trees (binary and non-binary) and graphs; however, for others, e.g. texts (binary and non-binary) and strings of parentheses, there is a canonical way to represent them as binary vectors.

The first example of such a data structure was given by Jacobson [32], who represented a binary vector B such that the rank and select operations can be performed efficiently: the details are given in Jacobson's PhD thesis [33]. The $\texttt{bin\_rank}_B(1, i)$ operation determines

the number of 1-bits up to and including the given position i in B. The $\texttt{bin\_sel}_B(1, x)$ operation determines the position of the x-th 1-bit in B. ($\texttt{bin\_rank}(0, i)$ and $\texttt{bin\_sel}(0, x)$ are defined similarly.) Let us denote the value of the bit in the position i by $B[i]$, the number of 1-bits in B by $\texttt{occ}_B(1)$, and the number of 0-bits by $\texttt{occ}_B(0)$. B can also be considered as the characteristic vector of a set $S \subset [|B|]$[1], such that $i \in S$ if $B[i] = 1$. The number of 1-bits of B equals to the number of elements (cardinality) in S and will be referred to as the *cardinality* of B as well. The operations `bin_rank` and `bin_sel` are one sided inverses of each other in that if $x \leq \texttt{occ}_B(1)$ then $\texttt{bin\_rank}_B(1, \texttt{bin\_sel}_B(1, x)) = x$, however $\texttt{bin\_sel}_B(1, \texttt{bin\_rank}_B(1, i)) = i$ for $i \leq |B|$ holds only in the case if $B[i] = 1$. A similar condition holds for 0-bits. Jacobson's representation stored B directly together with some relatively small auxiliary structures to make these operations efficient. Such structures are generally called *directories* or *indices*. The time complexities of his implementations of `bin_rank` and `bin_sel` are both $O(\lg n)$. As the output of `bin_rank` and `bin_sel` queries are $\lg n$ bits long, these bounds are optimal to within a constant factor, if we count output operations as part of time complexity.

Clearly, the rank and select operations can be defined over any alphabet Σ. We call them `str_rank` and `str_sel`, where $\texttt{str\_rank}_T(c, i)$ returns the number of characters $c \in \Sigma$ in T up to and including the position i, and the operation $\texttt{str\_sel}_T(c, x)$ returns the position of the x-th character c in T. To these operation, we add the operation $\texttt{str\_acc}_T(i)$, which returns the character in position i of T, also denoted by $T[i]$.

There are three reasons that we use different notation in the binary and non-binary cases. First, the binary case is particularly important, since it is a building block in many existing succinct data structures. The case of larger alphabets has a smaller range of applications that are mostly in the text indexing area. The second reason is that the existing implementations (including the ones in this thesis) in the binary and the non-binary case use substantially different techniques. And the third reason is that the implementation of the operation `str_acc` in the binary case can be efficiently simulated using just two `str_rank` queries. We can compute $\texttt{str\_rank}_T(0, i+1) - \texttt{str\_rank}_T(0, i)$ and if this difference is 1 then $\texttt{str\_acc}_T(i) = 0$, otherwise $\texttt{str\_acc}_T(i) = 1$. However, for the non-binary case, to implement $\texttt{str\_acc}_T(i)$, we might need to compute $\texttt{str\_rank}_T(c, i +$

[1] We define $[n]$ to be $\{1, 2, \ldots, n\}$.

$1) - \texttt{str\_rank}_T(c, i)$ for $|\Sigma| - 1$ characters.

Raman et al. [48] called a data structure that implements `bin_rank` and `bin_sel` a *fully indexable dictionary* (FID).

1.2 Non-Indexing Data Structures

For some problems, it is more advantageous to represent the data without the indexing restriction. Such representations are called *non-indexing* [23] or *non-systematic* [19].

A simple and interesting example of such a data structure based on Benes networks [36] was developed by Munro et al. [44] for representing permutations. The problem of inverting permutations, `PERMS`, can be described as follows. We are to represent a permutation on n elements, and to answer queries $\pi(i)$ and $\pi^{-1}(i)$ for given i, $1 \leq i \leq n$. A natural raw form representation would be to store n cells of memory of size $\lg n$ bits each, so that the value in the i-th cell is $\pi(i)$. Hence, determining $\pi(i)$ is trivial, but finding $\pi^{-1}(i)$ requires a scan of the array. An indexing data structure stores this raw form plus some additional information to facilitate the π^{-1} queries. (One possibility would be to store both $\pi(i)$ and $\pi^{-1}(i)$ considering the part of the memory that stores $\pi^{-1}(i)$ as the index. However, there are much more succinct data structures.) Indeed, such representations do exist, as was shown by Munro et al. [44]; moreover, they are optimal up to a constant factor under the indexing model restriction as was proven by Demaine and López-Ortiz [10], Munro et al. [44], and in Chapter 4 of this thesis. These results can be stated as follows: if t is the time cost of the π^{-1} operation, then the size of extra storage should be at least $\Omega((n \lg n)/t)$ bits (we state the corresponding theorems more precisely in Chapter 4). However, the non-indexing data structure based on the Benes networks from Munro et al. [44] performs better than this lower bound would suggest. Namely, it uses only $\lg n! + O(n(\lg \lg n)^2 / \lg n)$ bits of storage (i.e. $O(n(\lg \lg n)^2 / \lg n)$ extra bits as compared to the information-theoretic minimum). It implements both π and π^{-1} operations in $O(\lg n / \lg \lg n)$ time. For such running times, the best known indexing representation uses a factor of $\Theta(\lg n / \lg \lg n)$ extra bits (i.e. $\Omega(n \lg \lg n)$ bits) than the lower bound suggests.

Based on the representation of permutations, Golynski et al. [25] proposed two encodings of texts on large alphabets: *select encoding*, and *access encoding*. We represent a

Name	Row	Column	Benes	Label	Object
`row_rank`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$	$\lg\lg\xi$	$\lg\lg\xi\lg\lg\lg\xi$
`row_sel`	1	$\frac{\lg\lg\rho}{\lg\lg\lg\rho}$	$\frac{\lg\rho}{\lg w}$	1	$\lg\lg\xi$
`col_rank`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$	$\lg\lg\xi\lg\lg\lg\xi$	$\lg\lg\xi$
`col_sel`	$\frac{\lg\lg\rho}{\lg\lg\lg\rho}$	1	$\frac{\lg\rho}{\lg w}$	$\lg\lg\xi$	1
Space	$\Upsilon+O\left(\frac{\Upsilon\lg\lg\lg\rho}{\lg\lg\rho}\right)$	$\Upsilon+O\left(\frac{\Upsilon\lg\lg\lg\rho}{\lg\lg\rho}\right)$	$\Upsilon+O(f)$	$f(\lg\xi+o(\lg\xi))$	$f(\lg\xi+o(\lg\xi))$

$\Upsilon = f\lg(nm/f) - O(f)$ is the information-theoretic minimum space to encode an $m \times n$ matrix with f 1-bits in it, $\rho = nm/f$ is the *inverse density* of R, and $\xi = \min\{m, n\}$.

Figure 1.1: Upper bounds for the `BINREL` problem (all running times are asymptotic)

text T of length $L = |T|$ on an alphabet Σ of size $\sigma = |\Sigma|$ using $L(\lg\sigma + o(\lg\sigma))$ bits, and perform the operations `str_rank`, `str_sel` and `str_acc` in the following asymptotic running times:

	select encoding	access encoding
`str_acc`	$\lg\lg\sigma$	1
`str_sel`	1	$\lg\lg\sigma$
`str_rank`	$\lg\lg\sigma$	$\lg\lg\sigma\lg\lg\lg\sigma$

Later, our encoding was generalized by Barbay et al. [2] to binary relations. We consider the `BINREL` problem: represent an $m \times n$ binary matrix R containing f 1-bits and implement queries `row_sel`, `col_sel`, `row_rank`, and `col_rank`. $\texttt{row\_sel}_R(i, x)$ (respectively, $\texttt{col\_sel}_R(x, j)$) returns the position of the x-th 1-bit in the i-th row (resp., j-th column) of R, it returns -1 if the i-th row (respectively, the j-th column) contains fewer than x 1-bits. $\texttt{row\_rank}_R(i, j)$ (resp., $\texttt{col\_rank}_R(i, j)$) returns the number of 1-bits in the i-th row (resp., j-th column) up to and including the position j (resp., i). We propose two encodings called *label* encoding and *object* encoding. In Section 2.2, we improve the these results by showing three new representations called *row*, *column* and *benes* encodings. Figure 1.1 shows the comparison between the results of Barbay et al. [2] and the new results. The new encoding achieves the succinct space bound $\Upsilon + o(\Upsilon)$ for all values of n, m, and f, while the encoding of Barbay et al. [3] only achieves this bound in the case where $f = o(\max\{n, m\})^{o(1)}$.

1.3 Lower Bounds

In this section, we discuss the limitation of indexing and non-indexing representations in more detail, and review some related results. We start with the usual cell probe model, and then introduce the indexing cell probe model and the indexing bit probe model.

1.3.1 Cell Probe Model

In general, little is known about lower bounds for data structures in the cell probe model. In his influential paper "Should Tables be Sorted?" Yao [57] proposed a computation model called the *cell probe model* which is now widely accepted as a framework for proving lower bounds. In this model, we have an array S of cells, each consisting of w bits. An algorithm in this model can be viewed as a decision tree. The nodes of the tree are labeled with "S[i] = ?", where i is an index into S, i.e. a value from between 1 and $|S|$ inclusively. The edges are labeled with numbers from 0 to $2^w - 1$. The time cost of the data structure is defined as the depth of this tree, and the space cost is defined as the size of the storage, $|S|$. This means that the cell probe model is only concerned with the number of cells accessed and the number of cells stored, while the computation is free, and there are no restrictions on the way the data is represented. Thus, the lower bounds obtained in this model will also apply to any other reasonable model of computation, e.g. the RAM model (but the converse is not true). This model provides an information-theoretic insight into a problem ignoring the computation issues. Still, proving meaningful lower bounds in this model is hard even for simple data structural problems.

Yao was the first to consider the *static membership problem* (i.e. where update operations are not considered), which can be stated as follows: given a set B of m keys, represent it so that the queries of the form "Is $i \in B$" can be implemented efficiently. Another important problem is the *static predecessor problem*. In this problem, we are to store a set S of m numbers from the universe $[n]$[2], and answer queries of the form "what is the largest number from S that is smaller than a given value?". Both the membership and

[2]We choose the notation n to denote the size of a universe as it was used in some early papers by Jacobson [32, 45]. We are also aware that other (more recent) papers [24, 48] use the notation m to denote the size of the universe and n to denote the size of the set.

the predecessor problems can also be studied in the dynamic setting where it is possible to update the set in question, and one wants to implement these updates efficiently. However, throughout the thesis, we only discuss the static versions, and indeed omit the word static.

The predecessor problem has been studied for over 30 years and is one of the most interesting problems for studying the lower bounds of classical static data structures. This problem can be solved by storing the elements of the set in increasing order and using binary search. The space cost is m and the time cost is $\lg m$. The (now classic) improvement was given by van Emde Boas et al. [53] in 1977, who showed how to implement queries in $O(\lg \lg n)$ time using $O(n)$ space. Ajtai [1] was the first to show that if the storage size is at most polynomial in the size of the set (i.e. $m^{O(1)}$ words), then any deterministic algorithm to answer such queries has to perform $t = \omega(1)$ word probes in the RAM model with $\lg n$ bit word size. Later, his results were improved: by Miltersen [40] who showed that $t = \Omega(\sqrt{\lg \lg n})$; by Miltersen, Nisan, Safra and Widgerson [42] who showed that $t = \Omega(\lg m^{1/3})$; and by Beame and Fich [4] who showed that

$$t = \Omega\left(\min\left\{\frac{\lg \lg n}{\lg \lg \lg n}, \sqrt{\frac{\lg m}{\lg \lg m}}\right\}\right).$$

Sen [51] showed that the last bound holds also for randomized data structures. These bounds are based on communication complexity techniques, and due to their limitations do not give precise bounds on the space requirements of a data structure. The results of Beame and Fich [4] and Sen [51] are applicable in the case of polynomial size storage (they do not make a distinction between polynomial and linear storage size as mentioned in [47]). The results of Pătraşcu and Thorup [47] essentially close the problem by showing bounds that are optimal up to constant factors for all choices of the parameters: the size of the set, the size of the universe, and the word size in the RAM model. They are able to separate between storages of size $m^{1+o(1)}$ and $m^{1+\varepsilon}$ for any positive constant ε. Defining $a = \lg \frac{|S|}{m} + \lg w$ and $\ell = \lg n$, they showed that the optimal search time (up to constant

factors) is:

$$\min \begin{cases} \log_w n \\ \lg \frac{\ell - \lg n}{a} \\ \frac{\lg \frac{\ell}{a}}{\lg\left(\frac{a}{\lg n} \cdot \lg \frac{\ell}{a}\right)} \\ \frac{\lg \frac{\ell}{a}}{\lg\left(\lg \frac{\ell}{a} \;\middle/\; \lg \frac{\lg n}{a}\right)} \end{cases}$$

However, even the techniques from Pătraşcu and Thorup do not give precise *space* bounds in the case where we need to store a set of $m = \Theta(n)$ elements; they cannot distinguish between storages of size smaller than $m^{1+o(1)}$, e.g. they cannot separate storages of sizes $n+O((n \lg\lg n)/\lg n)$, $2n$, and $n \lg n$. Using a simple bit vector with `bin_rank` and `bin_sel` functionality, we can solve this problem using $n + O(n \lg n \lg n/\lg n)$ bits (that is, $n/\lg n$ cells of memory) in constant time (e.g. using the data structures described in Section 2.1), while the naive solution that stores all predecessors explicitly uses $O(m \lg m)$ words of space.

Our interest here is in showing lower bounds that are more sensitive, e.g. that can distinguish between storage sizes below the $n^{1-o(1)}$ barrier. Let Υ be the information-theoretic minimum to represent an object in question (e.g. $\Upsilon = \lg \binom{n}{m}$ if we are to represent sets of size m from a universe of size n). Let S be the storage, and let $|S| = \Upsilon + r$ be the size of the storage, where r is called the *redundancy*. Let t be the running time to perform the required operations. We are interested in the tradeoff between t and r up to constant factors, rather than the tradeoff between t and $|S|$. Intuitively, when $r = o(\Upsilon)$, the former type of tradeoff is more meaningful. Consider an example of a data structure that stores a permutation π on n elements, and implements queries $\pi(i)$ and $\pi^{-1}(i)$ for $i \in [n]$. If we store all the answers to these queries explicitly using $2n$ cells of memory, we obtain constant running times. However, if we use the data structure of Munro et al. [44], the space is $n + O(n(\lg\lg n)^2/(\lg n)^2$, and the time is $O(\lg n/\lg\lg n)$. Hence, we cannot obtain a lower bound on $|S|$ of the form $|S| = \Omega(f(n,t))$ for some function f other than the trivial $|S| = \Omega(n)$, but we can try to obtain a tradeoff of the form $r = \Omega(g(n,t))$ for some function g.

We are aware of only one cell probe lower bound of this type: Gál and Miltersen [19]

showed a lower bound for the problem of evaluating polynomials. In this problem, we are to represent a polynomial f of given degree d over the field $GF(2^k)$, and to answer queries of type $f(i)$ for given $i \in GF(2^k)$. Their techniques are "based on the fact that the problem of polynomial evaluation hides an error correcting code: The strings of query answers for each possible data (i.e. each polynomial) form the Reed-Solomon code." [19]. However, these methods do not extend to the problems that are of interest in this thesis. In fact, in [19], they stated "We do not know how to prove similar lower bounds for natural storage and retrieval problems such as Substring Search. However, we get a natural restriction of the cell probe model by looking at the case of systematic or index structures." To tackle with this difficulty, they used the indexing model, which differs from Yao's cell probe model in that it restricts the data structure under investigation to be an indexing data structure. We consider this model and relevant lower bounds in Section 1.3.2.

In Chapter 5, we propose a new technique for proving space sensitive lower bound for the problem that possess what we call the *reciprocal* property. To describe this property, consider the `PERMS` problem where we have to implement two types of queries: the forward queries $\pi(i)$ and the inverse queries $\pi^{-1}(i)$. Each type describes the object (i.e. permutation π) completely, and each query is responsible for "its own part" of the object (i.e. a directed link between i and j for each $\pi(i) = j$, if we visualize π as a directed bipartite graph). Examples of such problems are: the `TEXTSEARCH`, the problem of implementing the `str_sel` and `str_acc` operations for texts on arbitrary alphabets (we will define this problem formally later), the `PERMS` problem, and the `BINREL` problem. For the `PERMS` problem, using the new technique, we prove lower bounds that are tight up to constant factors: we show that both the "back pointer" data structure and the data structure based on the Benes networks given in Munro et al. [44] are optimal even in the non-indexing model. As was shown in Golynski et al. [25], a lower bound for the `PERMS` problem implies a lower bound for the `str_sel`/`str_acc` problem and vice versa. Hence our bounds are tight for certain parameters of the latter problem as well. We also show a lower bound for the text search and retrieval problem that can be formulated as follows. Represent a text T of length L on alphabet Σ of size σ, and answer the queries

- $\texttt{access}_T(i)$, which retrieves a substring of the text of length p at the position i, and
- $\texttt{search}_T(X)$, which finds the location of an occurrence of the given substring X

(called *pattern*) of length $p \leq (\lg L)/(\lg \sigma)$ in T.

Denote the running time of `access` by t, and the running time of `search` by t'. In the cell probe model with the cell size $\lg L$ and small enough running times t and t', we show a lower bound of the form $r \geq \Omega(\Upsilon/(tt'))$. In particular, if we wish to implement `access` and `search` in constant time (i.e. $t = t' = O(1)$), then the redundancy has to be at least $\Omega(\Upsilon)$.

1.3.2 Indexing Lower Bounds

The indexing model differs from Yao's cell probe model in that it restricts the data structure under investigation to be an indexing data structure. Informally, it can be described as follows. We are given a raw form representation of a combinatorial object free of charge, that is, it is stored in external memory (e.g. a large database) or provided by the outside world (e.g. the web graph [55]). We are to develop a small index (e.g. a road map of the web graph) that helps us to perform a given set of operations (e.g. shortest paths queries between sites in this graph). Presumably the index is stored "locally" in a relatively fast, expensive and/or limited memory. Thus, we are charged 1 unit of space for each bit that is occupied by the index. An algorithm that implements one of the retrieval operations is charged 1 unit of time for each access to the external raw data, e.g. I/O operation, and is not charged to access the local index. Finally, we will allow the algorithm to perform unlimited computation. If we allowed to access the raw data bit by bit, then we call the model, the *indexing bit probe model*; and if we can only access the raw data cell by cell (w bits at once), then we call the model, the *the indexing cell probe model*. Clearly, any algorithm with time cost t in the cell probe model can be turned into an algorithm with time cost tw in the bit probe model. The converse is not true: in general, we can only claim that an algorithm in the indexing bit probe model with time cost t can be turned into an algorithm in the cell probe model with time cost t (hence, losing a factor of w).

In this model, we can also raise the question of time versus space tradeoffs. Intuitively, the more space we allow for the index (up to some limit), the more efficient the retrieval operations should be. Although this model is unreasonable for any practical implementation, we will show that certain data structure problems have limitations even in this

model, and hence the same limitations hold in more realistic models (e.g. the traditionally accepted RAM model). In particular, we can prove lower bounds under this very "liberal" model, while the matching upper bounds do not abuse the freedom.

Such lower bounds have been investigated by Yao [58], Demaine and López-Ortiz [10], Miltersen [41], Golynski [25], and Golynski et al. [24]. Demaine and López-Ortiz considered the problem of representing a binary text T of length L so that we can find an occurrence of a given pattern X of length $p = \lg L - o(\lg L)$. We call this problem and its variations the `TEXTSEARCH` problem. In the indexing bit probe model, they showed that if the space cost t is at most $t = o((\lg L)^2/\lg\lg L)$, then the time cost is at least $r = \Omega((L \lg L)/t)$. We consider their result in more detail in Chapter 3. Gál and Miltersen considered the following problems:

- the *substring search* problem, a variation of the `TEXTSEARCH` problem where we do not have to report a location where a given pattern occurred in the text (if any), but only output YES or NO depending on whether the pattern occurred. They only considered the case where the patterns are of length $p = \Theta(\lg L)$. They showed that for a given time cost t, the space cost must be at least $r = (L/(t \lg L))$.

- the *prefix sum* problem, which is a variation of the problem of implementing `bin_rank`, where we are only required to report the parity of `bin_rank`$(1, \cdot)$. They showed that $\Theta(n/r)$ bit probes are necessary and sufficient for this problem.

In [41], Miltersen considered lower bounds for the problems of implementing `bin_rank` and `bin_sel`. For the problem of implementing the `bin_sel` operation, he showed that the space cost must be $r = \Omega(n/t)$ in the indexing bit probe model. For the problem of implementing the `bin_rank` operation, he showed that $r = \Omega(n \lg\lg n/(t \lg n))$ in the indexing cell probe model if the cell size is $w = \lg n$. Later, Golynski [25] improved his results to the best possible up to constant factors showing that, for implementing either `bin_rank` or `bin_sel` operation, the index size must be $r = \Omega((n \lg t)/t)$. This is an improvement by a factor of $\lg t$ (for constant time RAM algorithms, $t = O(\lg n)$) for the `bin_sel` problem, and a generalization of Miltersen's result for the `bin_rank` problem from the more restrictive cell probe model to the less restrictive bit probe model. We also provide a more sensitive analysis of these problems, namely we consider the case where the

bit vectors have a fixed number of 1-bits in them, i.e. fixed cardinality m. A more detailed comparison, and the proofs of these results are presented in Chapter 3.

In Golynski et al. [24], we considered the *balanced parentheses problem*: represent a string of balanced parentheses so that the query `findmatch`(i) can be implemented for $1 \leq i \leq n$. The operation `findmatch`(i) returns the position of the parenthesis matching the parenthesis at the position i. A raw form representation of a balanced parentheses string is a bit vector with 0 indicating an opening parenthesis, and 1 indicating a closing parenthesis. We showed a lower bound for this problem that matches the upper bound by Geary et al. [20]. Namely, if the time cost of an algorithm is $t = O(\lg n)$ then the space cost must be $r = \Omega((n \lg \lg n)/ \lg n)$. We present these results in Section 3.7.

1.4 Models and Problems

Throughout the thesis, we assume that all algorithms are deterministic.

For the implementations in this thesis, we use the word RAM model. This model allows conditional jumps, indirect addressing, and a set of basic instructions, such as left and right bit shifts, additions, multiplications, divisions (MOD and DIV operations) on registers. The size of registers, w, is also called *cell size*. Typically, we consider the registers to hold $w = \lg n$ bits, where n is the size of the data structure (e.g. the length of a bit vector). w is chosen so that we can fit a pointer in a register and perform indirect addressing.

For studying the lower bounds we consider three models in this thesis: the *indexing bit probe model*, the *indexing cell probe model*, and the *cell probe model*. Let us introduce some common notation for all the definitions. Let $\mathcal{H}$ be the set of combinatorial objects. Let $g(B, x_1, x_2, \ldots, x_k)$ be a function, where $B \in \mathcal{H}$ and the tuple of parameters $(x_1, x_2, \ldots, x_k)$ belongs to some given domain $\mathcal{X}$. Let $\mathcal{Y}$ denote the range of g. The function g defines the queries that we are to implement on the set of objects $\mathcal{H}$. Namely, we define the set of queries as

$$\mathcal{Q} = \{\text{query } (g, B, x_1, x_2, \ldots, x_k) | B \in \mathcal{H}, (x_1, x_2, \ldots, x_k) \in \mathcal{X}\}.$$

To make the notation more intuitive, we will use query $g(B, x_1, x_2, \ldots, x_k)$ instead of

query $(g, B, x_1, x_2, \ldots, x_k)$. We give examples of g, $\mathcal{Y}$ and $\mathcal{X}$ later in this section together with the formal problem definitions.

We start by defining the indexing models first.

Definition 1. *In the indexing model, we store the object B in raw form together with an index I_B of size r (r is called the* space cost*). Let us denote by A the algorithm that takes the input parameters $x_1, x_2, \ldots, x_k$ and computes the value of $g(B, x_1, x_2, \ldots, x_k)$. We say that the algorithm A* implements *the queries from $\mathcal{Q}$. A has access to B, I_B, the parameters $x_1, x_2, \ldots, x_k$, and to unlimited storage for its internal computations. The* time cost*, t, of A is defined as the number of bit (or cell) probes A performs on the raw data B in the worst case. The cell size is denoted by w. A is not charged for any computation it performs, nor for accessing any of I_B, x_1, $x_2, \ldots, x_k$.*

In the *non-indexing cell probe model*, we define

$$\Upsilon = \left\lceil \frac{\log |\mathcal{H}|}{w} \right\rceil$$

to be the information-theoretic minimum space required to represent an object from $\mathcal{H}$. Let S_B denote the storage. S_B is an array of $|S| = \Upsilon + r$ cells. We call r the *redundancy*. Let A be the algorithm that implements $\mathcal{Q}$ on $\mathcal{H}$. In this model, we define the *time cost* of A to be the number of cell probes A performs on the storage S_B. The *space cost* of A is defined to be r.

We now define the problems and clarify the meaning of g, $\mathcal{X}$ and $\mathcal{Y}$.

The `bin_rank` problem. The set of objects is

$$\mathcal{H} = \{B |\ B \text{ is a bit vector of length } n\ \}.$$

The set of queries is given by a function g that has two parameters: degree $b \in \{0, 1\}$, and element $i \in [n]$, so that $\mathcal{X} = \{0, 1\} \times [n]$. The range of this function is $\mathcal{Y} = [n]$. The answer to the "query $g(B, b, i)$" is the number of b-bits in B up to and including the position i. We also denote "$g(B, b, i)$" by "query $\texttt{bin\_rank}_B(b, i)$".

The density sensitive `bin_rank` problem. The set of objects is given by

$$\mathcal{H} = \{B|\ B \text{ is a bit vector of length } n \text{ with } m \text{ 1-bits in it}\},$$

where m is some fixed parameter. The set of queries is the same as for the `bin_rank` problem.

The `bin_sel` problem. The set of objects is

$$\mathcal{H} = \{B|\ B \text{ is a bit vector of length } n\ \}.$$

The set of queries is given by a function g that has two parameters: degree $b \in \{0,1\}$, and element $i \in [n]$, so that $\mathcal{X} = \{0,1\} \times [n]$. The range of this function is $\mathcal{Y} = [n]$. The answer to the "query $g(B,b,i)$" is the i-th b-bits in B, and -1 if B contains less than i of b-bits. We also denote "$g(B,b,i)$" by "query $\texttt{bin\_sel}_B(b,i)$".

The density sensitive `bin_sel` problem The set of objects is given by

$$\mathcal{H} = \{B|\ B \text{ is a bit vector of length } n \text{ with } m \text{ 1-bits in it}\},$$

where m is some fixed parameter. The set of queries is the same as for the `bin_sel` problem.

The `PARENTHESES` problem. The set of objects is

$$\mathcal{H} = \{B|\ B \text{ is a sequence of balanced parentheses of length } n\ \},$$

where n is an even integer. The set of queries is given by a function g with one parameter i, so that $\mathcal{X} = [n]$. The range of this function is $\mathcal{Y} = [n]$. The answer to the "query $g(B,i)$" is the position of the parenthesis that matches the parenthesis at position i. We also denote "$g(B,i)$" by "query $\texttt{findmatch}_B(i)$".

The `PERMS` problem. The set of objects is

$$\mathcal{H} = \{\pi|\ \pi \text{ is a permutation on } n \text{ elements }\}.$$

The set of queries is given by a function g that has two parameters: degree $d \in \{+1,-1\}$, and element $i \in [n]$, so that $\mathcal{X} = \{+1,-1\} \times [n]$. The range of this function is $\mathcal{Y} = [n]$. The answer to the "query $g(\pi,d,i)$" is $\pi^d(i)$. We denote "$g(\pi,1,i)$" by "query $\texttt{forw\_perm}_\pi(i)$", and "$g(\pi,-1,i)$" by "query $\texttt{inv\_perm}_\pi(i)$".

The `TEXTSEARCH` problem. The set of objects is

$$\mathcal{H} = \{T|\ T \text{ is a text of length } L \text{ on an alphabet } \Sigma \text{ of size } \sigma\ \}.$$

The set of queries is given by a function g that has three parameters $x_1 \in \{0,1\}$, x_2 and x_3, where $x_1 = 0$ indicates an access query and $x_1 = 1$ indicates a search query.

Access query: We are to return the substring of length p that we are to access that starts at position $x_2 \in [L - p + 1]$. The value x_3 is not used.

Search query: We are to search for the x_3-th occurrence ($x_3 \in [L]$) of the pattern $x_2 \in \Sigma^p$ in the text T and output its position if it exists (-1 otherwise).

The range of $g(T, x_1, x_2, x_3)$ is therefore $\mathcal{Y} = \Sigma^p \cup [L-p+1] \cup \{-1\}$. We denote "$g(T, 0, x_2)$" by "query $\texttt{access}_T(x_2)$", and "$g(T, 1, x_2, x_3)$" by "query $\texttt{search}_T(x_2, x_3)$.

The substring report problem for binary alphabets. The set of objects is

$$\mathcal{H} = \{B|\ B \text{ is a bit vector of length } L\ \}.$$

The set of queries is given by a function g that has one parameter X, which is a binary vector of length p, so that $\mathcal{X} = \{0,1\}^p$. The range of this function is $\mathcal{Y} = [L-p+1] \cup \{-1\}$. The answer to the "query $g(B, X)$" is the position of any of the occurrences of X in B, and -1 if X does not occur in B. We denote "$g(B, X)$" by "query $\texttt{search}_T(X)$.

The `str_acc/str_sel` problem. This is a special case of the `TEXTSEARCH` problem, where the pattern length p equals to 1.

The `BINREL` problem. The set of objects is

$$\mathcal{H} = \{R|\ R \text{ is an } m \times n \text{ binary matrix containing } f \text{ 1-bits}\ \}.$$

The set of queries is given by a function g with three parameters: $x_1 \in \{0,1\}$, x_2 and x_3, where $x_1 = 0$ indicates that it is a row query and $x_1 = 1$ indicates that it is a column query.

Row query: We are to search the x_2-th row for the x_3-th 1-bit ($x_2 \in [m]$ and $x_3 \in [n]$) and output its position if it exists (-1 otherwise).

Column query: We are to search the x_3-th column for the x_2-th 1-bit ($x_2 \in [m]$ and $x_3 \in [n]$) and output its position if it exists (-1 otherwise).

We denote $g(R, 0, x_2, x_3)$ by "query $\texttt{row\_sel}_R(x_2, x_3)$", and $g(T, 1, x_2, x_3)$ by "query $\texttt{col\_sel}_R(1, x_2, x_3)$". The range of $g(R, x_1, x_2, x_3)$ is $\mathcal{Y} = [\max\{n, m\}] \cup \{-1\}$.

1.5 Contributions and Outline

The thesis is organized as follows.

In Chapter 2, we present new upper bounds. Section 2.1 describes a new data structure to implement the `bin_rank` and `bin_sel` operations. For any given parameter $t \geq 1$, we use $t + 1$ consecutive cell probes to the raw form representation of the bit vector, where the space used for such representation is only

$$r = \frac{n \lg(t \lg n)}{t \lg n} + O\left(\frac{n(\lg(t \lg n))^2}{t^2 (\lg n)^2}\right).$$

This data structure is based on the *count index* (we will define it later in Section 2.1) that uses $(n \lg(t \lg n))/(t \lg n)$ bits. The remaining parts of the `bin_rank` and `bin_sel` indices use a factor of $(t \lg n)/\lg(t \lg n)$ less space than the count index. (We would like to note that such implementations are also possible for `PARENTHESES` problem; however, we leave this discussion outside the scope of this thesis. We refer the reader to [24], where we proposed a non-indexing implementation of balanced parentheses strings that uses less extra space than in Section 2.1). Earlier data structures implemented `bin_rank` and `bin_sel` separately and used different methods for the two implementation, each occupying $\Theta(n \lg \lg n / \lg n)$ bits. An advantage of our data structure is that it combines the most space-consuming parts of the indices for `bin_rank` and `bin_sel` in the *count index* that is shared between the two. In addition, our data structures can be used for with any number of bit probes t. This provides greater flexibility, and can be advantageous when we can access the bit vector in chunks of data larger than one cell, such as blocks on modern hard disk drives, i.e. for large values of t. We believe that this implementation can be useful for practitioners due

to the simplified structure and improved constant of the size of the index as compared to the work of Raman et al. [48]. (In the indexing model, these data structures are optimal up to constant factors as shown in Sections 3.2 and 3.3.) Although constants are not explicit in the representation of [48], they build two separate indexes for `bin_rank` and `bin_sel`, while we combine and reuse the most space consuming part of these indices, called the count index (we also discuss this aspect in Chapter 6).

In Section 2.2, we generalize this problem and give a data structure for implementing `bin_rank` and `bin_sel` operations on the rows and columns of a given binary matrix. This result subsumes both the data structures by Golynski et al. [25] and Barbay et al. [2]. It also has applications to text searching [25] (we them in Section 2.2.5 in more detail), intersection algorithms for conjunctive queries [2], and representation of labeled objects [2], such as labeled trees and XML data. Another advantage of this result is that it compresses the binary matrix to its information-theoretic bound minimum space (the previous results could not achieve this compression), and allows constant running times for the selection operations on rows and columns of matrices with a high density of 1-bits. For more detailed comparisons, see Figure 1.1.

Chapter 3 is devoted to lower bounds for representing binary vectors in the indexing model. In Sections 3.2 and 3.3, we show lower bounds for implementing the `bin_rank` and `bin_sel` operations. These lower bounds match the upper bounds given in Section 2.1. In Sections 3.5 and 3.6, we generalize the results from Sections 3.2 and 3.3 to the case where the cardinality m of the bit vector is a parameter, and express our lower bounds in terms of m. In Section 3.7, we consider the balanced parentheses problem, and show a lower bound that matches the upper bound of [20].

In Chapter 4, we present an indexing lower bound for the `TEXTSEARCH` problem. In Section 4.2, we consider the `PERMS` problem in the indexing cell probe model. This result uses ideas similar to those of Demaine and López-Ortiz [10]; however, it introduces a new compression technique that allows the restrictions of their results to be relaxed significantly. More specifically, we are able to prove lower bounds for any running time $t \leq n/2$ while the results from [10] and Munro et al. [44] only yield meaningful lower bounds for the case $t = o(\lg n / \lg \lg n)$. In Section 4.3, we generalize the results from Section 4.2, and show a new lower bound on text indexing that is stronger than in [10].

In Chapter 5, we prove the main result of this thesis. In Section 5.3.1, we introduce a new technique for proving lower bounds in the non-indexing cell probe model. Using this technique, we are able to show tight lower bounds for the `PERMS` problem. These lower bounds match the known upper bounds developed by Munro et al. [44] up to constant factors. Section 5.3.2 shows how to extend these results and prove a lower bound for the `TEXTSEARCH` problem in the non-indexing cell probe model. In Section 5.3.4, we show lower bounds for representing binary relations.

Chapter 2

Upper Bounds

This chapter deals with the problem of the representation of binary vectors, strings and binary relations. We construct several data structures and illustrate their applications.

2.1 Fully Indexable Dictionaries

In this section, we consider the problem of implementing rank and select operations on binary vectors. We start with the notion of a fully indexable dictionary:

Definition 2. *A* fully indexable dictionary *(FID) for a vector B is a data structure that supports the following operations [48]:*

- $\texttt{bin\_acc}_B(i)$*: gives the value of the bit in position i of B;*
- $\texttt{bin\_rank}_B(b, i)$*: gives the number of b-bits up to and including position i in B. We define* $\texttt{bin\_rank}_B(b, 0) = 0$ *for convenience, and*
- $\texttt{bin\_sel}_B(b, x)$*: gives the position of the x-th b-bit in B. We define* $\texttt{bin\_sel}(b, 0) = 0$ *for convenience.*

Fully Indexable Dictionary

These operations can also be defined for general alphabets Σ (that is, for $b \in \Sigma$); we consider this case in Section 2.2 in more detail. The techniques in the binary case

are substantially different from the techniques used in Section 2.2. Also, in the binary case, $\texttt{bin\_acc}_B(i) = \texttt{bin\_rank}_B(1, i) - \texttt{bin\_rank}_B(1, i-1)$. Therefore, we do not consider `bin_acc` to be a separate operation for binary vectors and we focus only on implementing the `bin_rank` and `bin_sel` operations. In addition, $\texttt{bin\_rank}_B(0, i) = i - \texttt{bin\_rank}_B(1, i)$ so it is enough to implement the `bin_rank` operation for $b = 1$.

2.1.1 Related Work

These operations were first proposed in the work of Jacobson [32], and implemented in his PhD thesis [33]. He used the following two-level data structure. He divided B into pieces of length $(\lg n)^2/\lg\lg n$. For each piece, he stored the number of 1-bits prior to the start of the piece using $\lg n$ bits. The size of the first level is therefore $n \lg n \lg\lg n/(\lg n)^2 = n \lg\lg n/\lg n$. Then, he further subdivided each piece into sub-pieces of length $(\lg n)/2$. For each sub-piece inside a given piece, he stored the number of 1-bits inside the piece that precede the starting position of the sub-piece using $2 \lg\lg n$ bits. Therefore, the size of the second level is $2n \lg\lg n/\lg n$. The `bin_rank` of a given position i can be found as follows. First, find the piece and sub-piece to which the given position belongs to. Then sum the following three values: the number of 1-bits preceding the piece, the number of 1-bits preceding the sub-piece, and the number of bits from the beginning of the sub-piece to the given position. The latter number can be found using a table that tabulates the answers for all possible sub-pieces and positions. The size of this table is $2^{(\lg n)/2}(\lg n)/2 \lg\lg n = O(\sqrt{n} \lg n \lg\lg n)$. Hence, Jacobson's data structure occupies $O(n \lg\lg n/\lg n)$ bits in addition to storing the bit vector B. The running time for the computation of $\texttt{bin\_rank}_B(1, i)$ is clearly a constant in the word RAM model, though Jacobson's concern was that it required $O(\lg n)$ bit accesses. A data structure that stores an object (e.g., in this example, a bit vector) in its raw form plus some extra information (typically, of size that is lower order term of the original storage) to facilitate the implementation of some given queries (e.g. `bin_rank`) is called an index. In particular, the data structure by Jacobson described above is a *rank index*.

Jacobson also proposed an implementation of `bin_sel` using $O(\lg n)$ bit accesses of B with an index of size $o(n)$, however this accesses were scattered among several words. Clark and Munro in [8, 43] focused on the $\lg n$-bit word RAM and offered a data structure with

only $O(1)$ cell accesses to implement `bin_sel`.

An efficient implementation for the `bin_sel` operation was first proposed by . Their data structure (called *select index*) is similar to Jacobson's; however, it has three levels, occupies $O(n/\lg\lg n)$ bits, and allows implementation of `bin_sel` in constant time. Later the space bound was improved by Raman et al. [48] to $O((n\lg\lg n)/\lg n)$. They offered a data structure that compresses the bit vector and is able to implement the FID operations in constant time. It occupies $\mathcal{B}(n,m) + O(n\lg\lg n/\lg n)$ bit of space, where m is the number of 1-bits in the bit vector, and $\mathcal{B}(n,m) = \lceil \lg \binom{n}{m} \rceil$.

2.1.2 Contributions

In Theorem 1, we propose an index that allows us to implement both `bin_rank` and `bin_sel` operations in $O(t)$ time with only $t+1$ cell probes to B, where t is an arbitrary parameter. Here, we assume that B is stored in memory directly, i.e. the first $\lg n$ bits of B are stored in the first cell, the next $\lg n$ bits in the second and so on. A cell probe is an access to one of these cells, and it is performed in constant time. The size of this index in bits is

$$r = \frac{n\lg(t\lg n)}{t\lg n} + O\left(\frac{n(\lg(t\lg n))^2}{t^2(\lg n)^2}\right).$$

This tradeoff is particularly interesting as in Chapter 3 we will show that the space used by this data structure is optimal up to a constant factor. That is, the $\Omega((n\lg(t\lg n))/(t\lg n))$ term is unavoidable, though potentially the constant in front of it can be lower than 1.

An advantage of this data structure is that it combines and reuses the most space consuming part of both rank and select indexes of size of order $\Theta((n\lg(t\lg n))/(t\lg n))$. Later in the proof we call this part the *count index*. It is designed so that it can be shared by both rank and select indexes, and the space occupied by both rank and select indexes apart from the count index is a lower order term: $O(n(\lg(t\lg n))^2/(t\lg n)^2)$.

2.1.3 Construction of the Count Index

We proceed by constructing the count index, the part that is shared between both rank and select indexes. The count index is the asymptotically largest part of both of these

indexes. It uses $\frac{n \lg(t \lg n)}{t \lg n} + O\left(\frac{n(\lg(t \lg n))^2}{t^2 (\lg n)^2}\right)$ bits, while the remaining parts of the indices for `bin_rank` and `bin_sel` only occupy $O\left(\frac{n(\lg(t \lg n))^2}{t^2 (\lg n)^2}\right)$ bits.

The count index consists of a hierarchy of chunks. At the bottom level, we partition B into *level* 1 *chunks* of equal length $l_1 = t \lg n$. The number of 1-bits in a chunk is called its *cardinality*. We store the cardinalities of chunks of level 1 in equally spaced fields of size $\lg(l_1)$ in a table $\texttt{COUNT}_1$ of size $(n \lg l_1)/l_1$. The bit vector is padded with extra 0-bits as necessary if the last level 1 chunk is of length less than l_1. At the next level, we group $(t \lg n)/\lg l_1$ consecutive level 1 chunks into a *level* 2 *chunk*, so that its length is $l_2 = l_1(t \lg n)/\lg l_1$, We store the cardinalities of the level 2 chunks in equally spaced fields of size $\lg l_2$ in a table $\texttt{COUNT}_2$ of size $(n \lg l_2)/l_2$. In general, we can define the level k chunks recursively as the concatenation of $(t \lg n)/\lg l_{k-1}$ consecutive chunks of level $k-1$. The length of level k chunks is denoted by l_k. Note that $l_1 \leq l_k \leq (t \lg n)^k$ so that $\lg l_1 \leq \lg l_k \leq k \lg l_1$. Also,

$$l_k \;=\; l_{k-1}\frac{t \lg n}{\lg l_{k-1}} = \Theta\left(l_{k-1}\frac{t \lg n}{\lg l_1}\right) = \Theta\left(l_{k-1} f\right) = \cdots = \Theta\left(l_1 f^{k-1}\right), \tag{2.1}$$

where $f = l_1/\lg l_1$. We store the cardinalities of the level k chunks, $k > 1$, in a table $\texttt{COUNT}_k$ of size

$$\frac{n \lg l_k}{l_k} = O\left(\frac{n \lg l_1}{f^{k-1} l_1}\right) = O\left(\frac{r}{f^{k-1}}\right).$$

This construction is illustrated in Figure 2.1. We store h levels of the count index for some constant h which we will choose later. This construction can also be viewed as a forest, where the root nodes are the level h chunks, the nodes of depth 2 are the level $h-1$ chunks and so on. The leaves are the level 1 chunks. The children of a level k chunk are all the level $k-1$ chunks that are contained in this chunk. The space used by the count index is dominated by the table $\texttt{COUNT}_1$ of size $(n \lg l_1)/l_1$; the rest of the tables $\texttt{COUNT}_2, \texttt{COUNT}_3, \ldots, \texttt{COUNT}_h$ occupy $O(n(\lg l_1)^2/l_1^2)$ bits. There are $l_k/l_{k-1} = (t \lg n)/\lg l_{k-1}$ chunks of level $k-1$ that constitute a given chunk C_k of level k, and for each of those chunks, the table $\texttt{COUNT}_{k-1}$ stores $\lg l_{k-1}$ bits. The lengths are chosen so that the total number of bits that are occupied by C_{k+1} in $\texttt{COUNT}_k$ is $t \lg n$, so that we can read all those bits using at most $t+1$ consecutive cell probes to the table $\texttt{COUNT}_k$. We might have to use

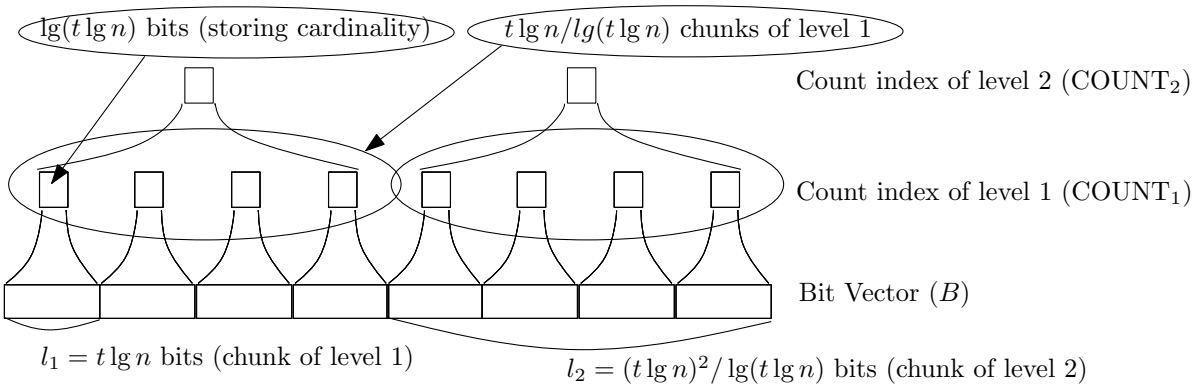


Figure 2.1: Structure of the the first two levels of the count index

$t+1$ cell probes (not just t), since the bits that we require ($t \lg n$ consecutive bits) are not necessary aligned with the word boundaries and can be spread across $t+1$ cells.

2.1.4 Implementing Rank and Select on Chunks

In this section, we show how to implement `bin_rank` and `bin_sel` operations on a level k chunk in $O(kt)$ time using the tables $\texttt{COUNT}_1, \texttt{COUNT}_2, \ldots, \texttt{COUNT}_h$ and word parallelism. To implement the `bin_rank` operation, we introduce tables $\texttt{RANK}_k[z, i]$ where z is a value at most $2^{\lfloor (\lg n)/2 \rfloor}$, and i is such that $0 \leq i \leq \lfloor (\lg n)/2 \rfloor$. For $k > 1$, we treat z as a bit string composed of y_k equally spaced bit fields of size $\lg l_k$, where we define

$$y_k = \left\lceil \frac{\lg n}{2 \lg l_k} \right\rceil .$$

The value $\texttt{RANK}_k[z, i]$ is defined as the sum of the first i fields of z. For $k = 1$, we define $\texttt{RANK}_1[z, i]$ as the sum of the first i bits in z. To implement $\texttt{bin\_rank}_{C_{k+1}}(1, i)$ on a level $k+1$ chunk, C_{k+1}, for $k > 1$ we perform the following procedure at most $4t$ times. Let C_k be the level k chunk that contains the given position i of C_{k+1}, and let j be its starting position, so that

$$\texttt{bin\_rank}_{C_{k+1}}(1, i) = \texttt{bin\_rank}_{C_{k+1}}(1, j-1) + \texttt{bin\_rank}_{C_k}(1, i-j+1).$$

The operation $\texttt{bin\_rank}_{C_{k+1}}(1, j-1)$ for $j-1$ that is divisible by l_k and for $k > 1$ can be implemented using tables $\texttt{COUNT}_k$ and $\texttt{RANK}_k$ as follows. We retrieve bit strings $w_1, w_2, \ldots, w_p$

from $\texttt{COUNT}_k$, each of length at most $y_k \lg l_k$, that correspond to the cardinalities of all the level k chunks that precede the position j inside C_{k+1}. Namely, w_1 contains the cardinalities of the first y_k chunks, w_2 contains the cardinalities of the next y_k chunks, and so on. The last bit string w_p contains the cardinalities of the remaining q chunks, $0 \leq q < y$. Then,

$$\begin{aligned} \texttt{bin\_rank}_{C_{k+1}}(1, j-1) &= \texttt{RANK}_{k+1}[w_1, y] + \texttt{RANK}_{k+1}[w_2, y] + \ldots + \texttt{RANK}_{k+1}[w_{p-1}, y] \\ &+ \texttt{RANK}_k[w_p, q] \end{aligned}$$

The operation $\texttt{bin\_rank}_{C_1}(1, i)$ can be performed similarly. We retrieve $t+1$ consecutive bits from the bit vector B, and split them into bit strings $w_1, w_2, \ldots, w_p$ of length at most $\lfloor (\lg n)/2 \rfloor$. Then,

$$\begin{aligned} \texttt{bin\_rank}_{C_1}(1, i) &= \texttt{RANK}_1[w_1, \lfloor (\lg n)/2 \rfloor] + \texttt{RANK}_1[w_2, \lfloor (\lg n)/2 \rfloor] + \ldots \\ &+ \texttt{RANK}_1[w_{p-1}, \lfloor (\lg n)/2 \rfloor] + \texttt{RANK}_1[w_p, q]. \end{aligned}$$

We can implement the $\texttt{bin\_sel}$ operation on level $k+1$ chunks in a similar fashion. First, let us introduce the tables $\texttt{SELECT}_k[z, x]$, where $0 \leq z \leq 2^{\lfloor (\lg n)/2 \rfloor} - 1$ and $0 \leq x \leq \lfloor (\lg n)/2 \rfloor$. The value z is treated as a concatenation of y_k fields of $\lg l_k$ bits each. The element $\texttt{SELECT}_k[z, x]$ stores the minimum value, such that the sum of the first $\texttt{SELECT}_k[z, x]$ fields is at least x, or -1 in the case when the sum of all the fields is smaller than x. In other words, the x-th 1-bit is contained in the $\texttt{SELECT}[z, x]$-th level k chunk among the chunks whose cardinalities are stored in z. Using this table, the operation $\texttt{bin\_sel}_{C_{k+1}}(1, x)$ can be performed as follows. We first read the string w_1 from $\texttt{COUNT}_k$ that contains the cardinalities of the first y_k level k chunks inside C_{k+1}. If the total cardinality of these chunks, which equals $\texttt{RANK}_k[w_1, y_k]$, is greater than x, then the x-th 1-bit is located inside one of these chunks, and the number of this chunk is $q = \texttt{SELECT}_k[w_1, x]$. Therefore,

$$\texttt{bin\_sel}_{C_{k+1}}(1, x) = \texttt{bin\_sel}_{C_k}(1, x - \texttt{RANK}_k[w_1, q-1]) + q l_k,$$

where C_k denotes the q-th level k chunk of C_{k+1}. If the total cardinality of the first y_k chunks is smaller than x, then we retrieve the cardinalities of the $y_k + 1$-st, $y_k + 2$-nd, ..., $2y_k$-th level k chunks from $\texttt{COUNT}_k$, store them in w_2, and verify if the required 1-bit is

located inside one of them by the same operation. We keep performing this operation until we find the required chunk. If the chunk is not found, we return -1.

Note that the sizes of the tables $\texttt{RANK}_k$ and $\texttt{SELECT}_k$ are small compared to B, i.e. at most $O(\sqrt{n}(\lg n)^2)$ bits.

2.1.5 Rank and Select Indices

Now we can implement the `bin_rank` and `bin_sel` operations using the count index that we constructed in the previous section and prove the main theorem of Section 2.1.

Theorem 1. *Given a binary vector B of size n, the operations* `bin_rank` *and* `bin_sel` *can be supported in the word RAM model with word size $\lg n$ bits in $O(t)$ time probing only $t+1$ consecutive words of B and using an index of size*

$$r = \frac{n \lg(t \lg n)}{t \lg n} + O\left(\frac{n(\lg(t \lg n))^2}{t^2 (\lg n)^2}\right)$$

bits.

Proof. We will first construct a relatively simple rank index, and then, construct a select index. To construct the rank index, we use the count index of level $h = 3$. For the j-th chunk of level 3, we store the value $\texttt{bin\_rank}_B(1, l_3 j)$ in a field of size $\lg n$ in a table $\texttt{RANK}^*[j]$. Thus,

$$\texttt{bin\_rank}_B(1, i) = \texttt{bin\_rank}_{C_3}(1, i') + \texttt{RANK}^*[j]$$

where the position i in B corresponds to the position i' of C_3, and C_3 is the j-th chunk of level 3 in B, $j = \lfloor i/l_3 \rfloor$. The constant $h = 3$ is chosen so that the table $RANK^*$ occupies

$$|\texttt{RANK}^*| = \frac{n \lg n}{l_3} = \Theta\left(\frac{n \lg n (\lg l_1)^2}{l_1^3}\right) = O\left(\frac{n(\lg l_1)^2}{l_1^2}\right) \tag{2.2}$$

bits. That is, the size of $\texttt{RANK}^*$ is at most the size of $\texttt{COUNT}_2$.

We construct the select index using the count index of level $h = 7$. We store the positions of every l_3-th occurrence of a 1-bit explicitly, that is, for each $1 \leq j \leq m/l_3$, $\texttt{SELECT}^*[j] = \texttt{bin\_sel}_B(1, j l_3)$, where m denotes the cardinality of B. The table $\texttt{SELECT}^*$

occupies $(m \lg n)/l_3 \leq (n \lg n)/l_3 = O(n(\lg l_1)^2/l_1^2)$ bits by inequality (2.2)[1]. The region of B from position $\texttt{bin\_sel}_B(1, (j-1)l_3) + 1$ to position $\texttt{bin\_sel}_B(1, jl_3)$ is called the j-th *block*. The operation $\texttt{bin\_sel}_B(1, i)$ can be implemented as follows. Let $j = \lfloor i/l_3 \rfloor$, and let X be the j-th block. By construction X contains the i-th 1-bit of B, so that

$$\texttt{bin\_sel}_B(1, i) = \texttt{bin\_sel}_B(1, jl_3) + \texttt{bin\_sel}_X(1, i - jl_3).$$

It remains to implement $\texttt{bin\_sel}_X(1, i - jl_3)$. Let us call a block *sparse* if its length is at least l_7. For sparse blocks, we store the positions of all the 1-bits explicitly in a table $\texttt{SELECT}^s$. There are at most n/l_7 sparse blocks, so that the size of $\texttt{SELECT}^s$ is at most[2]

$$\frac{n}{l_7} l_3 \lg n = \Theta\left(\frac{n \lg n (\lg l_1)^6}{l_1^7} \frac{l_1^3}{(\lg l_1)^2}\right) = \Theta\left(\frac{n(\lg l_1)^4}{l_1^3} \frac{\lg n}{l_1}\right) = o\left(\frac{n}{l_1^2}\right)$$

bits. We used the facts that $l_7 = \Theta(l_1^7/(\lg l_1)^6)$, $l_3 = \Theta(l_1^3/(\lg l_1)^2)$, $\lg n = O(l_1)$, and the fact that $(\lg l_1)^4/l_1 = O(1)$. Note that choosing $h = 6$ is not sufficient, since

$$\frac{n}{l_6} l_3 \lg n = \Theta\left(\frac{n \lg n (\lg l_1)^5}{l_1^6} \frac{l_1^3}{(\lg l_1)^2}\right) = \Theta\left(\frac{n(\lg l_1)^3}{l_1^2} \frac{\lg n}{l_1}\right) = O\left(\frac{n(\lg l_1)^3}{l_1^2}\right)$$

is by a factor of $\lg l_1$ bigger than the requirements of the theorem. To implement the select operation $\texttt{bin\_sel}_Y(1, x)$ on the blocks that are not sparse, we will employ the count index of level 7. Since the length of Y is l_7, it is covered by two consecutive level 7 chunks. The answer to the select operation lies either in the first chunk C_7 that contains the starting position of Y, $y = \texttt{bin\_sel}_B(1, (j-1)l_3) + 1$, or the next chunk C_7'. To see which is the case, we can compute the cardinality of the intersection of Y and C_7, $x' = \texttt{bin\_rank}_{C_7}(1, l_7) - \texttt{bin\_rank}_{C_7}(1, y')$, and compare it with x, where y' denotes the starting position of Y relative to the starting position of C_7. If $x > x'$, then $\texttt{bin\_sel}_Y(1, x) = \texttt{bin\_sel}_{C_7'}(1, x - x')$, otherwise $\texttt{bin\_sel}_Y(1, x) = \texttt{bin\_sel}_{C_7}(1, x + x'')$, where x'' denotes the cardinality of $C_7 \setminus Y$, $x'' = \texttt{bin\_rank}_{C_7}(1, y')$. We described earlier how to support rank and select operations on chunks of level $h = 7$, so the theorem follows. □

[1]We chose to store the position of every l_3-th occurrence of 1-bits, so that the space occupied by the $\texttt{SELECT}^*$ table satisfied the requirements of Theorem 1.

[2]The length of the sparse block is chosen such that the size of the table $\texttt{SELECT}^s$ satisfies the requirements of Theorem 1.

2.2 Strings and Binary Relations

In this section, we consider generalizations of the rank/select problem to the case of binary matrices of size $m \times n$ and strings of length L over alphabet Σ. We first present some basic notions (the Burrows-Wheeler transform and empirical entropy) in Section 2.2.1. We then describe some related results in Section 2.2.2 and Section 2.2.3, and present a data structure for the rank/select problem on binary matrices in Section 2.2.4. In Section 2.2.5, we show applications of our data structures to the FM-index, and to the problem of representing and navigating labeled trees.

2.2.1 Preliminaries

Let T be a text of length L over alphabet Σ of size σ. The Burrows-Wheeler transform [6] reorders characters in a given string to facilitate compression by the move to front (MTF) [5] and similar techniques. It can be described as follows. We first append the special character "#" to the end of the text T. This character is defined to be lexicographically smaller than any character in Σ. We consider all the suffixes of T ($T[1..L+1], T[2..L+1], \ldots, T[L+1..L+1]$), and sort them in alphabetical order. Suppose $T[i_1..L+1] < T[i_2..L+1] < \ldots < T[i_{L+1}..L+1]$. Consider the string $BW = T[i_1 - 1] \cdot T[i_2 - 1] \cdot \ldots \cdot T[i_{L+1} - 1]$, where "$\cdot$" denotes concatenation, and $T[0]$ is defined to be $T[0] = T[L+1] =$ "#" for convenience. The string BW is called the *Burrows-Wheeler transform* of T. This transform is reversible; that is, given BW, we can reconstruct T (as shown in [6]).

The *suffix array*, SA, of the text T is defined so that $SA[j] = i_j$ for $j \in [L+1]$. In other words, the suffixes $SA[1], SA[2], \ldots, SA[L+1]$ are sorted lexicographically. In particular, all the suffixes that start with "#", "a", "b", ... "z" appear in consecutive chunks in SA. We call the group of suffixes that start with a given character c the *c-zone*. For an example, see Figure 2.2.

We define the empirical entropy as follows. Let $\mathtt{occ}_T(c)$ be the number of occurrences of character c in T. We define the *empirical probability* of c, p_c, to be $\mathtt{occ}_T(c)/L$. Then the *empirical 0-th order entropy* is

$$H_0(T) = \sum_{c \in \Sigma} p_c \lg \frac{1}{p_c}.$$

Text:

pos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$T =$	a	c	b	a	a	c	c	a	c	b	c	b	b	b	#

Sorted suffixes:

$$
\begin{aligned}
T[10..15] &= \#\\
T[4..15] &= aaccacbcbbb\#\\
T[1..15] &= acbaaccacbcbbb\#\\
T[8..15] &= acbcbbb\#\\
T[5..15] &= accacbcbbb\#\\
T[14..15] &= b\#\\
T[3..15] &= baaccacbcbbb\#\\
T[13..15] &= bb\#\\
T[12..15] &= bbb\#\\
T[10..15] &= bcbbb\#\\
T[7..15] &= cacbcbbb\#\\
T[2..15] &= cbaaccacbcbbb\#\\
T[11..15] &= cbbb\#\\
T[9..15] &= cbcbbb\#\\
T[6..15] &= ccacbcbbb\#
\end{aligned}
$$

Suffix array:

$SA =$ 15 | 4 1 8 5 | 14 3 13 12 10 | 7 2 11 9 6

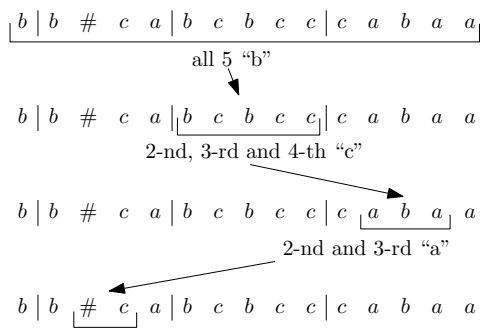
\# zone | a-zone | b-zone | c-zone

Burrows-Wheeler Transform:

$BW =$ b | b # c a | b c b c c | c a b a a

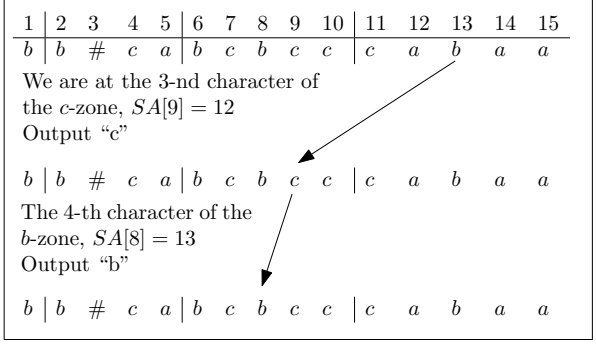
\# zone | a-zone | b-zone | c-zone

Backward search algorithm. Pattern $X = acb$.



We have 2 occurrences corresponding to $SA[3..4]$ at positions $SA[3] = 1$ and $SA[4] = 8$

Decoding text after an occurrence. Let $i = 13$ be a given index in SA, $SA[13] = 11$.



Decoding text before an occurrence. Let $i = 4$ be a given index in SA, $SA[4] = 8$.

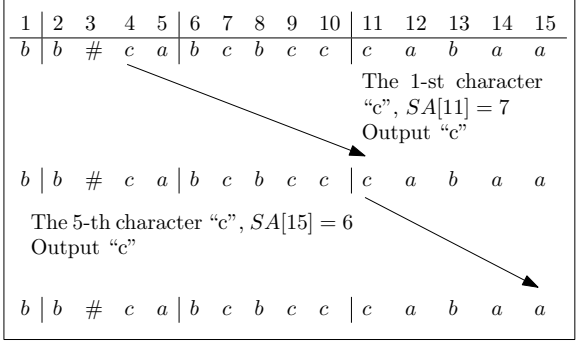


Figure 2.2: The Burrows-Wheeler transform, the backward search algorithm, and the decoding algorithms.

The empirical 0-th order entropy can be generalized to the empirical k-th order entropy as follows. Consider a pattern X of length k. For the i-th occurrence of X in T, let x_i be the character that follows this occurrence. Let w_X be the string $x_1 x_2 \cdot x_z$, where z is the number of occurrences of X in T. The *empirical k-th order entropy*, $H_k(T)$, is

$$H_k(T) = \frac{1}{n} \sum_{X \in \Sigma^k} |w_X| H_0(w_X).$$

The value $|T|H_k(T)$ is a lower bound on the output size of any compressor that encodes each symbol with a code that only depends on the symbol itself and on the k immediately preceding symbols [18].

2.2.2 Introduction and Related Work

Motivated by text retrieval problems, Grossi, Gupta and Vitter [26] first considered the rank/select problem for strings. Given a text T of length L over an alphabet Σ of size σ, their wavelet tree structure supports the following operations in $O(\lg \sigma)$ time, using $LH_0 + o(L)$ bits:

- $\texttt{str\_rank}_T(c, i)$: finds the number of occurrences of character c up to and including position i in T,
- $\texttt{str\_sel}_T(c, x)$: finds the position of the x-th occurrence of character c in T, and
- $\texttt{str\_acc}_T(i)$: returns the i-th character of T.

We also define $\texttt{occ}_T(c)$ to be the number of occurrences of $c \in \Sigma$ in the text T, so that $\texttt{occ}_T(c) = \texttt{str\_rank}_T(c, L)$.

A wavelet tree is a binary tree at each node v of which, the alphabet is split into two parts of equal size Σ_1 and Σ_2. The node stores a bit vector B_v of length L, such that $B_v[i] = 0$ if $T[i] \in \Sigma_1$ and $B_v[i] = 1$ otherwise. For example, if $T = abadbcdab$, $\Sigma_1 = \{a, b\}$ and $\Sigma_2 = \{c, d\}$, then $B_v = 000101100$, where v is the root node of the wavelet tree. The left subtree of r encodes the string $T_l = ababab$, and the right subtree encodes $T_r = dcd$ recursively. The `str_rank` operation corresponds to traversing this tree in a top-down

fashion and performing `bin_rank` operations along the way, e.g.

$$\texttt{str\_rank}_T(a,5) = \texttt{str\_rank}_{T_l}(a, \texttt{bin\_rank}_{B_v}(0,5)) = \texttt{str\_rank}_{T_l}(a,4) = 2.$$

The `str_sel` operation starts at the leaf that corresponds to a given character and traverses the tree in a bottom-up fashion performing `bin_sel` operations along the way. In our example,

$$\begin{aligned} \texttt{str\_sel}_T(a,3) &= \texttt{bin\_sel}_{B_v}(0, \texttt{str\_sel}_{T_l}(a,3)) \\ &= \texttt{bin\_sel}_{B_v}(0,5) = 8. \end{aligned}$$

The `str_acc` operation corresponds to traversing the tree in a top-down fashion as follows:

$$\texttt{str\_acc}_T(6) = \texttt{str\_acc}_{T_r}(\texttt{bin\_rank}_{B_v}(B_v[6],6)) = \texttt{str\_acc}_{T_r}(2) = \text{“c”}$$

Since the tree depth is $O(\lg \sigma)$, and `bin_rank`, `bin_sel`, and `bin_acc` can be performed in constant time, the run-times for `str_rank`, `str_sel` and `str_acc` operations are $O(\lg \sigma)$.

Ferragina et al. [17] improved the time to $O(1)$ (using essentially the same space) for the case of small alphabets, i.e., when $\sigma = \text{polylog}(L)$ using $LH_0 + O(L \lg\lg L / \lg_\sigma L)$ bits. The idea is to increase the branching factor of wavelet trees from 2 to $\sqrt{\lg \sigma}$, so that in the case of $\sigma = \text{polylog}(L)$, the tree is of constant depth. Grossi and Sadakane [27] improved the space for this structure to $LH_k + O(L \lg\lg L / \lg_\sigma L)$ using Lempel-Ziv tries. Their ideas apply to the more general class of data structures where the text is stored in the raw form.

Our approach here is to represent a given text T on an alphabet of size σ of length L as a binary matrix R of size $\sigma \times L$, where $R[i,j] = 1$ if and only if the j-th position of T contains the i-th character of the alphabet (so that each column contains only one 1-bit). The `BINREL` problem for binary matrices can be defined as follows. Let R be a binary matrix with m rows and n columns. We say that the i-th row is *associated* with the j-th column if $R[i,j] = 1$. We consider the following operations on R:

- $\texttt{row\_rank}_R(i,j)$: finds the number of 1-bits in the i-th row of R up to and including column j;
- $\texttt{row\_sel}_R(i,x)$: finds the column of the x-th occurrence of a 1-bit in the i-th row of R; if there are fewer than x occurrences this operation returns ∞.

- $\texttt{row\_nb}_R(i)$: returns the number of 1-bits in the i-th row (also called the *cardinality* of the i-th row, $\texttt{row\_nb}_R(i) = \texttt{row\_rank}(i, n)$;
- $\texttt{col\_rank}_R(i, j)$: finds the number of 1-bits in the j-th column of R up to and including row i;
- $\texttt{col\_sel}_R(x, j)$: finds the row of the x-th occurrence of a 1-bit in the j-th column; if there are fewer than x occurrences this operation returns ∞.
- $\texttt{col\_nb}_R(j)$: returns the number of 1-bits in the j-th column (also called the *cardinality* of the j-th column, $\texttt{col\_nb}(j) = \texttt{col\_rank}(m, j)$;
- $\texttt{tab\_acc}_R(i, j)$: returns the element $R[i, j]$. Typically, we do not represent the table explicitly, so that this operation is non-trivial.

Clearly, the three operations for strings `str_rank`, `str_sel`, and `str_acc` can be implemented in a straightforward manner using the operations `row_rank`, `row_sel`, and `col_sel` respectively. Similarly, the operation `occ` can be implemented using `row_nb`. Typically, in the representations that are given in this section, the operations `row_nb` and `col_nb` can be implemented more efficiently than more general operations `row_rank` and `col_rank` respectively.

Let f denote the *cardinality* of R (i.e. the number of 1-bits in R), let $\rho = nm/f$ be the *inverse density* of R, and let w be the word size in the RAM model. We make the assumption that $w \geq \max\{\lg n, \lg m\}$, so that we can store each row and column index in one register of the RAM. We present three data structures for this problem in Theorem 4.

The *row structure* supports `row_sel` in $O(1)$ time, `col_sel` in $O(\lg\lg\rho/\lg\lg\lg\rho)$ time; and `row_rank`, `col_rank`, and `tab_acc` in $O(\lg\lg\rho)$ time.

The *column structure* supports operations within the same time bounds, but with rows and columns switched, that is, `col_sel` is supported in $O(1)$ time, and `row_sel` is supported in $O(\lg\lg\rho\lg\lg\lg\rho)$ time.

The *Benes structure* supports both `row_sel` and `col_sel` in $O(\lg\rho/\lg w)$ time; and `row_rank`, `col_rank`, and `tab_acc` in $O(\lg\lg\rho(\lg\rho/\lg w + 1))$ time. This data struc-

ture is based on the Benes networks [36, 44] which will be described later in this section.

The space required by the row and column structures is at most $f \lg \rho + O\left(f \lg \rho \frac{\lg\lg\lg \rho}{\lg\lg \rho}\right)$, while the space for the Benes structure is a little less, $f \lg \rho + O(f)$. The Benes structure has the advantage that, for small values of ρ as compared to w, the run times for the operations are much less than in the other two data structures: if $\rho = w^{O(1)}$ then both `row_sel` and `col_sel` can be implemented in $O(1)$ time. The operations `row_nb` and `col_nb` come "for free" with our encoding, that is, they do not require any additional space, and the run times for both of them is $O(1)$ for all three data structures. The operation `tab_acc` is implemented as the difference of the corresponding `row_rank` or `col_rank` operations, i.e. as either of

$$\texttt{tab\_acc}(i,j) = \texttt{row\_rank}(i,j+1) - \texttt{row\_rank}(i,j) \tag{2.3}$$

$$\texttt{tab\_acc}(i,j) = \texttt{col\_rank}(i+1,j) - \texttt{col\_rank}(i,j). \tag{2.4}$$

Note that in the case of strings (i.e. one 1-bit per column), we can implement `tab_acc` using $\texttt{col\_sel}(i,1)$ and comparing the result to j.

A Benes network is a way of representing a permutation using communication switches. A switch has two inputs x_0 and x_1 and two outputs y_0 and y_1, and can be configured in two possible ways: (i) to connect x_0 with y_0 and x_1 with y_1 or (ii) to connect x_0 with y_1 and x_1 with y_0. Such a switch can represent any of the two permutations on two elements. The Benes network $Ben(2^x)$ is a network with 2^x inputs and 2^x outputs composed of $(2x-1)2^{x-1}$ switches. For $x = 1$, the Benes network is just one switch. For $x > 1$, the Benes network $Ben(2^x)$ is constructed recursively using two networks $Ben(2^{x-1})$ and 2^x switches. The main property of this network is that any possible one-to-one connection between inputs and outputs (in other words, a bijection between them) can be realized by configuring the switches. Munro et al. [44] applied Benes networks for the problem of representing permutations on n elements. They considered implementing the queries $\pi(i)$ and $\pi^{-1}(i)$ efficiently for $i \in [n]$, and showed the following result.

Theorem 2. *[44] The evaluation of an arbitrary permutation π and its inverse can be supported in the times listed below using $\lceil \lg n! \rceil + r$ bits.*

Name of data structure	π	π^{-1}	r
Forward representation	$O(1)$	t'	$O((n \lg n)/t' + (n \lg \lg n)/\lg n)$
Inverse representation	t	$O(1)$	$O((n \lg n)/t + (n \lg \lg n)/\lg n)$
Benes representation	$O(\lg n/\lg w)$	$O(\lg n/\lg w)$	$O(n(\lg \lg n)^2/\lg n)$

where w is the word size (in bits), and t and t' are arbitrary positive integer values.

Note that if $w = \lg n$, then the Benes approach takes time $\lg n/\lg \lg n$.

We also use the following data structure developed by Willard [56]. He considered the problem of representing a set of m numbers from a universe of size n, and showed the following result.

Theorem 3. *[56] For a binary vector B of length n and cardinality m, there exists a data structure (called the y-*fast trie*) that uses $\Theta(m \lg n)$ bits and allows one to perform* `bin_rank` *queries in $O(\lg \lg n)$ time in the RAM model.*

2.2.3 The FM-index and the xbw Transform

The data structure for storing binary relations has applications to text indexing, e.g. to FM-index [14, 15, 16], and also can be used to improve the running times of wavelet trees [26]. Typically, in text indexing applications, the following query types are discussed:

- *Existential*: Does the pattern occur in the text? (This type corresponds to the substring search problem.)
- *Cardinality*: How many times does the pattern occur?
- *Listing*: Give the positions of each occurrence. (This type corresponds to the substring report problem.)

To this we add another useful operation:

- *Context*: give the characters immediately after or before a specific occurrence (similar to what is done in search engines such as Google).

We use the notations t_r (respectively, t_s and t_a) for the worst case time complexity of the `str_rank` (respectively, `str_sel` and `str_acc`) operation. As usual, r denotes the redundancy of the data structure.

Ferragina et al. [16] showed how to support cardinality queries for a given pattern X of length p in $O(pt_r)$ time, listing queries in $O((t_a + t_r)\lg^{1+\epsilon} L)$ time per occurrence, and retrieving any text substring of length l in $O((l + \lg^{1+\epsilon} L)(t_a + t_r))$ time. They store the Burrows-Wheeler transform BW of a given text T, and perform `str_rank`, `str_sel`, and `str_acc` queries on BW directly. The text T is not stored explicitly. Using techniques and the representation of BW similar to the one described by Ferragina et al. [16], Golynski et al. [25] showed how to support the context queries.

We first describe the backward search of Ferragina and Manzini [14]. Let X be a given pattern of length p. They look at pattern X backwards starting at the last character. For each i, $1 \leq i \leq p$, they find the maximum interval in the suffix array $SA[\mathtt{sp}_i + 1..\mathtt{ep}]$, such that all the suffixes $SA[\mathtt{sp}_i + 1], SA[\mathtt{sp}_i + 2], \ldots, SA[\mathtt{ep}_i]$ start with the string $X[i..p]$. Recall that, SA is an array of sorted suffixes so that occurrences of $X[i..p]$ (if there are any) are grouped together in SA. If there are no occurrences of $X[i..p]$, then they define $\mathtt{sp} = \mathtt{ep}$. The transition from i to $i - 1$ can be explained as follows. Let $c = X[i - 1]$ be the next character of X. Consider all the characters c in BW from left to right. By the definition of the Burrows-Wheeler transform, the j-th character c (denote its position by x_j) precedes the suffix that starts at $SA[x_j]$. On the other hand, we have determined that the range of suffixes that start with $X[i + 1..p]$ is $SA[\mathtt{sp}_i + 1..\mathtt{ep}_i]$. Combining these two conditions gives us that $x_j \in [\mathtt{sp}_i + 1..\mathtt{ep}_i]$. In other words, we are only interested in the characters c that lie inside the interval $[\mathtt{sp}_i + 1..\mathtt{ep}_i]$. Let $l = \mathtt{str\_rank}_{BW}(c, \mathtt{sp}_i)$ and $u = \mathtt{str\_rank}_{BW}(c, \mathtt{ep}_i)$, so that there are l characters c that precede the interval, and $u - l$ characters that are inside it. Hence, the condition $x_j \in [\mathtt{sp}_i + 1..\mathtt{ep}_i]$ is equivalent to $j \in [l + 1..u]$. Finally observe that the j-th lexicographically smallest suffix that starts with c is located in position $SA[z + j]$, where $z + 1$ is the starting location of the c-zone. Hence, our new interval $\mathtt{sp}_{i-1} + 1..\mathtt{ep}_{i-1}$ is $z + l + 1..z + u$.

More formally, let

$$\mathtt{Occur} := 0\ 1^{\mathtt{occ}_T(c_1)}\ 0\ 1^{\mathtt{occ}_T(c_2)}\ 0 \ldots 1^{\mathtt{occ}_T(c_\sigma)},$$

where $c_1, c_2, \ldots, c_\sigma$ are the characters from Σ in lexicographical order. (For simplicity, assume that $\Sigma = [\sigma]$.) Store $\texttt{Occur}$ with a fully indexable dictionary functionality supported on it. Their algorithm is as follows:

Algorithm 1 BackwardSearch

$sp \leftarrow 0$
$ep \leftarrow L$ { the current set of suffixes is $SA[\texttt{sp}+1..\texttt{ep}]$ }
for $i = p$ to 1 by -1 **do**
  $c \leftarrow X[i]$
  $z \leftarrow \texttt{bin\_rank}_{\texttt{Occur}}(1, \texttt{bin\_sel}_{\texttt{Occur}}(0, c))$ { $z+1$ is the start of the c-zone in the suffix array }
  $u \leftarrow \texttt{str\_rank}_{BW}(c, \texttt{ep})$
  $l \leftarrow \texttt{str\_rank}_{BW}(c, \texttt{sp})$
  $\texttt{sp} \leftarrow z + l$
  $\texttt{ep} \leftarrow z + u$

At the end of the algorithm, $\texttt{ep} - \texttt{sp}$ is the number of occurrences of X, and the pattern occurs at positions $SA[\texttt{sp}+1], SA[\texttt{sp}+2], \ldots, SA[\texttt{ep}]$. We extend this algorithm in Section 2.2.5.

Ferragina et al. [13] generalized the Burrows-Wheeler transform for text strings to the case of labeled trees. The xbw transform of a labeled tree $\mathcal{T}$ is a pair $\texttt{xbw}(\mathcal{T}) =< S_{\texttt{last}}, S_\alpha >$, such that $S_{\texttt{last}}$ is a binary vector of length $|\mathcal{T}|$ and S_α is a string of length $|\mathcal{T}|$, where $|\mathcal{T}|$ is the sum of the numbers of labels of all the nodes of $\mathcal{T}$. Using this transform, they support two navigation operations: GetChildren (which lists the children of a given node) and GetParent (that returns the parent of a given node), and one search query called SubPathSearch (that searches for a given pattern in all root-to-leaf paths of the tree). As a building block, they require a data structure that supports the operations $\texttt{bin\_rank}$ and $\texttt{bin\_sel}$ on $S_{\texttt{last}}$ and the operations $\texttt{str\_rank}$, $\texttt{str\_sel}$ and $\texttt{str\_acc}$ on S_α as follows. The operation GetChildren uses one $\texttt{str\_acc}$ query and one $\texttt{str\_rank}$ query, the operation GetParent uses one $\texttt{str\_sel}$ query, and the operation SubPathSearch uses two $\texttt{str\_rank}$ queries. There are two tradeoffs that are mentioned: either (1) one doubles the information-theoretic minimum space and achieves optimal time for those three operations

[13, Theorem 3] (this might not be desirable for large trees); or (2) one uses wavelet trees [26] with space of $nH_0 + o(n)$ bits, but increasing run times of all three operations by a factor of $O(\lg \sigma)$. We apply our data structures to improve the running times of these operations in Section 2.2.5.

2.2.4 Technical Results

In this section, we prove four lemmas that will be used to prove the main result of this section. The first lemma allows us to split the binary matrix into smaller pieces horizontally and vertically. The second lemma allows to reduce the problem to the rank space. That is, we can skip the empty rows and columns in binary matrix for a small extra space cost. The third lemma reduces the problem of implementing `row_sel` and `col_sel` to the problem of representing permutations, so that we can apply the result of Munro et al. [44]. The fourth lemma allows to implement `row_rank` and `col_rank` operations using `row_sel` and `col_sel` for the small extra space cost. The main ingredient of this lemma is the result of Willard, Theorem 3.

We start by showing the first lemma.

Lemma 1 (Split Lemma)**.** *Let R be a binary matrix of size $m \times n$ of cardinality f with $m \leq n$. We can decompose R horizontally into q matrices $R_1, R_2, \ldots R_q$, such that $q \leq 4f/m + 1$, the cardinality f_k of R_k is at most m, and the size of R_k is $m \times l_k$, $l_k \leq nm/f$. The operations* `row_rank`*,* `row_sel`*,* `col_rank`*, and* `col_sel` *on R can be reduced to the operations* `row_rank`*,* `row_sel`*,* `col_rank`*, and* `col_sel` *on $R_1, R_2, \ldots, R_q$ respectively. This reduction requires $O(1)$ time overhead per operation, and with space overhead of at most $O(n + f)$ bits.*

Proof. We start by showing that the required decomposition of R exists. We first show that it is possible to split R into matrices of cardinality at most $2m - 1$, then we show that it is possible to decompose such a matrix into at most 3 matrices of cardinality m, and finally we show how to reduce the operations `row_rank`, `row_sel`, `col_rank`, and `col_sel`.

We split the columns of R into q *slices* in a greedy fashion from left to right, and traverse the columns starting from the first column, add the current column to the previously started slice, if (i) the number of columns of the slice would not exceed $\rho = nm/f$, (ii) the

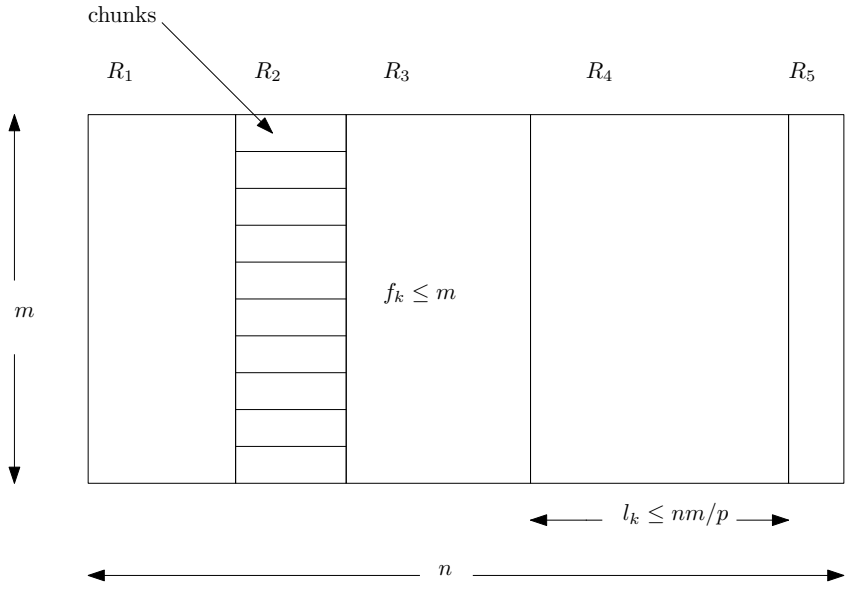


Figure 2.3: Illustration of the split lemma

cardinality of the slice would not exceed $2m-1$. If one of the two conditions is not satisfied, then we finalize the current slice, and start a new slice with the current column. The number of slices g_1 that we finalized due to condition (i) is at most $g_1 \leq n/(nm/f) = f/m$. The cardinality of each column is at most m, so that the slices that are finalized due to constraint (ii) satisfies $m \leq f_k < 2m$, and the number of such slices, g_2, is at most $g_2 \leq f/m$. We can further split such slices into at most three slices, each of cardinality at most m. For a slice R', select the largest possible slice R'_l (respectively, R'_r) starting from on the leftmost (respectively, rightmost) column of R' which cardinality does not exceed m in a greedy fashion. If $R'_l \cup R'_r = R'$, then R' can be split into R'_l and $R'_r \setminus R'_l$. Otherwise, the cardinality of R'_l and the next column to left, x, after R'_l exceeds m, and similarly for R_r and its preceding column y. If $x \neq y$, then the cardinality of R' is at least $2m$, which is not possible. Therefore $x = y$, and R' can be split into three slices satisfying the requirements: $R' = R'_l \cup \{x\} \cup R'_r$. Hence the total number of slices q is at most $g_1 + 3g_2 + 1 \leq f/m + 3f/m + 1 = 4f/m + 1$.

Every matrix R_k is further split horizontally into m sub-matrices of size $1 \times l_k$ called *slivers*, see Figure 2.3. Recall that l_k is the number of columns in the k-th slice. Let `Card`

be a bit vector that stores the cardinalities of slivers in unary form in the row-major order, namely:

$$\texttt{Card} = 0\ 1^{u_{11}}\ 0\ 1^{u_{12}}\ 0 \ldots 1^{u_{1q}}\ 0\ 1^{u_{21}}\ 0\ 1^{u_{22}}\ 0 \ldots 1^{u_{2q}}\ \ldots 0\ 1^{u_{m1}}\ 0\ 1^{u_{m2}}\ 0 \ldots 1^{u_{mq}}, \qquad (2.5)$$

where u_{ik} is the cardinality of the i-th sliver of the k-th slice. Let `Slices` be a bit vector that stores the widths of each slice in unary form from left to right:

$$\texttt{Slices} = 0\ 1^{l_1}\ 0\ 1^{l_2}\ 0 \ldots 1^{l_q}.$$

The 1-bits correspond to the columns of R, and the 0-bits correspond to the starting positions of the slices. Namely, if j is the column number in R then

$$k = \texttt{bin\_rank}_{\texttt{Slices}}(0, \texttt{bin\_sel}_{\texttt{Slices}}(1, j)) \qquad (2.6)$$

is the number of the slice where the j-th column belongs to (the number of 0-bits that precede the j-th 1-bit). Also

$$j' = j - \texttt{bin\_rank}_{\texttt{Slices}}(1, \texttt{bin\_sel}_{\texttt{Slices}}(0, k)) \qquad (2.7)$$

is the column number in R_k corresponding to the j-th column in R (the number of 1-bits between the staring position of the k-th slice and the j-th 1-bit). We can reduce the column operations by "re-addressing" the query to the corresponding slice:

$$\begin{aligned} \texttt{col\_rank}_R(i, j) &= \texttt{col\_rank}_{R_k}(i, j') \\ \texttt{col\_sel}_R(x, j) &= \texttt{col\_rank}_{R_k}(x, j') \end{aligned}$$

To implement $\texttt{row\_rank}(i, j)$, we first find the position y_1 in `Card` where the description of the i-th row begins, y_1 is the position of the $q(i-1)+1$-st 0-bit separator. Similarly, we can find y_2, the position in `Card` where the description of the k-th sliver of the i-th row begins, i.e. y_2 is the position of the $q(i-1)+k$-th 0-bit separator. Finally, the number of 1-bits in the i-th row up to position j equals to the number of 1-bits in the first $k-1$ slivers of the i-th row plus the number of 1-bits in the k-th sliver that precede position j'. Hence, the following algorithm implements `row_rank`.

Algorithm 2 Row Rank

$y_1 \leftarrow \texttt{bin\_sel}_{\texttt{Card}}(0, (i-1)q)$ { start of the row }
$y_2 \leftarrow \texttt{bin\_sel}_{\texttt{Card}}(0, (i-1)q + k - 1)$ { start of the slice }
return $\texttt{bin\_rank}_{\texttt{Card}}(1, y_2) - \texttt{bin\_rank}_{\texttt{Card}}(1, y_1) + \texttt{row\_rank}_{R_k}(i, j')$

The query $\texttt{row\_sel}(i, x)$ can be implemented as follows. We start by finding the position, y_1, in `Card` where the description of the i-th row begins. Let y_3 be the position of the 1-bit in `Card` that corresponds to the x-th 1-bit in i-th row. We can find it by skipping over the 1-bits of `Card` that correspond to the first $i-1$ rows. We can find k, the slice number to which the x-th 1-bit of i-th row belongs to by subtracting the number of 0-bit separators in the first $i-1$ rows ($q(i-1)$ of 0-bits total) from the number of separators prior to the y_3-th position. Then we can find position y_2 where the description of this block begins in a fashion similar to the case of the `bin_rank` operation. Next, we subtract the number of 1-bits in the first $k-1$ slivers from x and obtain x'. Thus, the position of the x-th 1-bit in the i-th row is the position of the x'-th 1-bit in the k-th sliver of the i-th row. The following algorithm implements `row_sel`.

Algorithm 3 Row Select

$y_1 \leftarrow \texttt{bin\_sel}_{\texttt{Card}}(0, (i-1)q)$ { start of the row }
$y_3 \leftarrow \texttt{bin\_sel}_{\texttt{Card}}(1, x + \texttt{bin\_rank}_{\texttt{Card}}(1, y_1))$ { x-th 1-bit in the row }
$k \leftarrow \texttt{bin\_rank}_{\texttt{Card}}(0, y_3) - (i-1)q$ { number of the slice }
$y_2 \leftarrow \texttt{bin\_sel}_{\texttt{Card}}(0, (i-1)q + k - 1)$ { start of the slice }
$x' \leftarrow x - \texttt{bin\_rank}_{\texttt{Card}}(1, y_3) - \texttt{bin\_rank}_{\texttt{Card}}(1, y_2)$ { the rank inside the slice }
return $\texttt{row\_sel}_{R_k}(i, x')$

The total space for the bit vector `Card` is at most $f + mq \leq 5f$. The total space for the bit vector `Slices` is at most $n + q \leq 2n$. The extra time is at most $O(1)$ if we use a data structure implementing FID (e.g. from Section 2.1 with parameter $t = O(1)$), so the lemma follows. □

Definition 3. *We call a row, a column of a binary matrix empty if it contains only* 0 *entries.*

The following lemma is quite straightforward and allows to reduce the row and column operations to the rank space. In another words, it allows to delete all empty rows and columns from the binary matrix, and consider the operations with row and column indices i' (respectively, j') in the range $[m']$ (respectively, $[n']$), where m' (respectively, n') is the number of non-empty rows (respectively, columns).

Lemma 2 (Reduction to Rank Space Lemma)**.** *Let R be a binary matrix of size $m \times n$, and let R' be R with empty rows and columns removed. Then we can reduce the operations* `row_rank`*,* `row_sel`*,* `col_rank`*, and* `col_sel` *on R to the corresponding operations on R' with $O(1)$ time overhead per operation, and with space overhead of at most $O(n+m)$.*

Proof. We can store two binary indicator vectors `ER`, and `EC` of the form

$$\begin{aligned} \texttt{ER} &= I_1^r I_2^r \dots I_m^r \\ \texttt{EC} &= I_1^c I_2^c \dots I_n^c, \end{aligned}$$

where $I_i^r = 1$ (respectively, $I_j^c = 1$) if the i-th row (respectively, the j-th column) of R is not empty, otherwise $I_i^r = 0$ (respectively, $I_j^c = 0$). The i-th row of R corresponds to i'-th row of R' if $\texttt{ER}[i] = 1$ and $i' = \texttt{bin\_rank}_{\texttt{ER}}(1, i)$. If the i-th row of R is empty, then it falls in between the i'-th and $i'+1$-st rows of R'. In a similar fashion, we can introduce $j' = \texttt{bin\_rank}_{\texttt{EC}}(1, j)$.

The row operations can be implemented as follows.

$$\texttt{row\_rank}_R(i,j) = \begin{cases} 0 & \text{, if } \texttt{ER}[i] = 0 \\ \texttt{row\_rank}_{R'}(i', j') & \text{, otherwise} \end{cases}$$

$$\texttt{row\_sel}_R(i,x) = \begin{cases} 0 & \text{, if } x = 0 \\ +\infty & \text{, if } x > 0 \text{ and } \texttt{ER}[i] = 0 \\ \texttt{bin\_sel}_{\texttt{EC}}(1, \texttt{row\_sel}_{R'}(i', x)) & \text{, otherwise.} \end{cases}$$

The operations `col_rank` and `col_sel` can be implemented similarly. □

Lemma 3 (Permutation Lemma)**.** *Let R be an $m \times n$ binary matrix of cardinality f. We can reduce the operations* `row_sel` *and* `col_sel` *on R to the operations π and π^{-1} respectively on a permutation on f elements with $O(1)$ time overhead per operation and with space overhead of at most $O(f+n+m)$.*

Proof. Recall that $\texttt{row\_nb}(i)$ (respectively, $\texttt{col\_nb}(i)$) is the number of 1-bits in the i-th row (column, respectively). Let `Rows` and `Columns` be bit vectors that store the values $\texttt{row\_nb}(i)$ and respectively $\texttt{col\_nb}(i)$ in unary:

$$\begin{aligned} \texttt{Rows} &= 0\,1^{\texttt{row\_nb}(1)}\,0\,1^{\texttt{row\_nb}(2)}\,0\ldots1^{\texttt{row\_nb}(n)}\text{, and} \\ \texttt{Columns} &= 0\,1^{\texttt{col\_nb}(1)}\,0\,1^{\texttt{col\_nb}(2)}\,0\ldots1^{\texttt{col\_nb}(n)}. \end{aligned}$$

Hence, the 1-bits of R are in one-to-one correspondence with 1-bits in `Rows` and with 1-bits in `Columns`. We use two standard traversal orders of R, *row major* and *column major*: in row major order, we traverse R starting at the top row from left to right, then the second top row, and so on; in column major order, we start at the leftmost column from top to bottom, and then traverse the second left column and so on. If we encounter a given 1-bit at a position (i, j), k-th (respectively, l-th) among all the 1-bits in the row (respectively, column) major order, then k (respectively, l) is called the *row position* (respectively, the *column position*) of a given occurrence. We define $\pi(k) = l$ for this occurrence. It is not hard to see that π is a permutation on f elements; see Figure 2.4 for an example. We store this permutation using one of the data structures given by Theorem 2. We can find the position y_1 in `Rows`, where the description of the i-th row of R starts as $y_1 = \texttt{bin\_sel}(0, i)$ so that $k = y_1 + x$-th position of `Rows` corresponds to the x-th 1-bit in the i-th row of R. In turn, the k-th 1-bit in `Rows` and the $l = \pi(k)$-th 1-bit in `Columns` correspond to the same 1-bit in R by the definition of π. It remains to find the y_2-th column of R, description of which in `Columns` contains the l-th 1-bit, it is not hard, i.e. $y_2 = \texttt{bin\_rank}_{\texttt{Columns}}(0, \texttt{bin\_sel}_{\texttt{Columns}}(1, l))$. Thus, the operation $\texttt{row\_sel}(i, x)$ can be supported as follows:

Algorithm 4 Row Select

{ Precondition: $1 \le x \le \texttt{row\_nb}(i)$ }
$k \leftarrow \texttt{bin\_rank}_{\texttt{Rows}}(1, \texttt{bin\_sel}_{\texttt{Rows}}(0, i) + x)$ { k is the row position of the occurrence }
$l \leftarrow \pi(k)$ { l is the column position of the occurrence }
return $\texttt{bin\_rank}_{\texttt{Columns}}(0, \texttt{bin\_sel}_{\texttt{Columns}}(1, l))$

The operation $\texttt{col\_sel}(x, j)$ can be implemented in a similar fashion to `row_sel`, but with rows and columns exchanged, and the mapping π replaced by the mapping π^{-1}:

Example with $m = 6$, $n = 8$, and $f = 10$:

		$(1,5)$					
	$(2,2)$				$(3,7)$		
	$(4,3)$						
$(5,1)$			$(6,6)$				$(7,9)$
	$(8,4)$						
						$(9,8)$	

Empty cells correspond to 0 entries in R, a pair (k,l) correspond to a 1 entry in R, it also corresponds to the k-th 1-bit in `Rows` and the l-th 1-bit in `Columns`.

i	1	2	3	4	5	6	7	8	9
$\pi(i)$	5	2	7	3	1	6	9	4	8

$$\texttt{Rows} = 0\,1\,0\,1\,1\,0\,1\,0\,1\,1\,1\,0\,1\,0\,1$$
$$\texttt{Columns} = 0\,1\,0\,1\,1\,1\,0\,1\,0\,1\,0\,0\,1\,0\,1\,0\,1$$

$\texttt{col\_sel}(2,2)$ first computes $l = \texttt{bin\_rank}_{\texttt{Columns}}(1, \texttt{bin\_sel}_{\texttt{Columns}}(0,1) + 2) = 3$, then $k = \pi^{-1}(3) = 4$, and finally returns $\texttt{bin\_rank}_{\texttt{Rows}}(0, \texttt{bin\_sel}_{\texttt{Rows}}(1,4)) = 2$, so that $\texttt{col\_sel}(2,2) = 3$.

Figure 2.4: An illustration of the permutation lemma

Algorithm 5 Column Select

{ Precondition: $1 \leq x \leq \texttt{col\_nb}(j)$ }
$l \leftarrow \texttt{bin\_rank}_{\texttt{Columns}}(1, \texttt{bin\_sel}_{\texttt{Columns}}(0, j) + x)$ { l is the column position of the occurrence }
$k \leftarrow \pi^{-1}(k)$ { k is the row position of the occurrence }
return $\texttt{bin\_rank}_{\texttt{Rows}}(0, \texttt{bin\_sel}_{\texttt{Rows}}(1, l))$

The lemma follows. □

Lemma 4 (Rank Lemma)**.** *Let R be a binary matrix of size $m \times n$ of cardinality f. Assume that it is stored in a data structure that supports* `row_sel` *and* `col_rank` *in times t, and t' respectively. Then we can also support the operations* `row_rank` *and* `row_sel` *in times $O(\lg\lg n + t \lg z)$ and $O(\lg\lg m + t' \lg z')$ respectively and extra space*

$$O\left(\frac{f \lg n}{z} + \frac{f \lg m}{z'}\right)$$

for arbitrary positive parameters $z, z' > 0$.

Proof. Let us choose parameter $z > 0$. For the i-th row of R, let F_i be the set of column indices of all the occurrences of 1-bits in the row. Let F_i' be the set of every z-th element of F. We call F_i' a *sparsified* bit vector, and store it using the y-fast trie data structure of Willard (Theorem 3) using $O(|F_i'| \lg n) = O((|F_i| \lg n)/z)$ bits. `row_rank` on the i-th row can be implemented as follows: we first compute an approximate rank using Willard's y-fast trie for F', $y = \texttt{bin\_rank}_{F_i'}(j)$, so that $yz \leq \texttt{bin\_rank}_{F_i}(j) < (y+1)z$; and then we employ binary search on this interval using at most $\lg z$ $\texttt{row\_sel}(i, \cdot)$ queries and comparing their results to j. The time complexity of `row_rank` is $O(\lg\lg n + t \lg z)$. The extra space we used is $(f \lg n)/z$ (summing over all the rows of R). We can support the `col_rank` operation in a similar fashion. □

Now we can prove the main theorem in this section.

Theorem 4. *Let R be an $m \times n$ binary matrix of cardinality f. There are three encodings of R such that the operations* `row_rank`*,* `row_sel`*,* `col_rank`*, and* `col_sel` *can be supported*

with the following time and space bounds:

Name	*Row*	*Column*	*Benes*
`row_rank`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$
`row_sel`	1	$\frac{\lg\lg\rho}{\lg\lg\lg\rho}$	$\frac{\lg\rho}{\lg w}$
`col_rank`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$
`col_sel`	$\frac{\lg\lg\rho}{\lg\lg\lg\rho}$	1	$\frac{\lg\rho}{\lg w}$
`row_nb`	1	1	1
`col_nb`	1	1	1
`tab_acc`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$
Redundancy	$O\left(\frac{f\lg\rho\lg\lg\lg\rho}{\lg\lg\rho}\right)$	$O\left(\frac{f\lg\rho\lg\lg\lg\rho}{\lg\lg\rho}\right)$	$O(f)$

where $\rho = nm/f$ *is the* inverse density *of* R*, and* redundancy *is the space used by the corresponding data structure (measured in bits) minus the information-theoretic minimum*

$$\lg\binom{nm}{f} = f\lg\left(\frac{nm}{f}\right) - \Theta(f)$$

for representing a set of cardinality f *in a universe of size* nm*.*

Proof. We use Lemma 1 (the Split Lemma) twice: to split our matrix vertically first and then each of the resulting matrices R_k horizontally. The resulting matrices R_{ks} have sizes $l_{ks} \times l_k$, $l_k \le \rho = nm/f$, and cardinalities $f_{ks} \le l_k$. For an example, see Figure 2.5. Note that the sum of all vertical dimensions of R_{ks} is $\sum_{k,s} l_{ks} = \sum_k m = mO(f/m+1) = O(f+m)$, since the number of slices in Lemma 1 is at most $4f/m+1$. The sum of all horizontal dimensions is $\sum l_i O(f_i/l_i+1) = O(\sum f_i) + O(n) = O(f+n)$. We use lemma 2 to eliminate empty rows and columns from matrices R_{ks}, so that the vertical sizes of new matrices, l'_{ks}, are at most $l'_{ks} \le f_{ks} \le l_k \le \rho$. The extra space used by Lemma 2 is the sum of cardinalities of R_{ks} plus the sum of horizontal dimensions of R_{ks} plus the sum of vertical dimensions of R_{ks}, so it is $O(f+n+m)$.

For the rest of the theorem, we assume that R_{ks} has no empty rows and columns, its size is at most $\rho \times \rho$, and its cardinality is at most ρ. We then implement operations `row_sel`, `col_sel`, `row_rank`, and `col_rank` using Lemma 3 (the Permutation Lemma) and

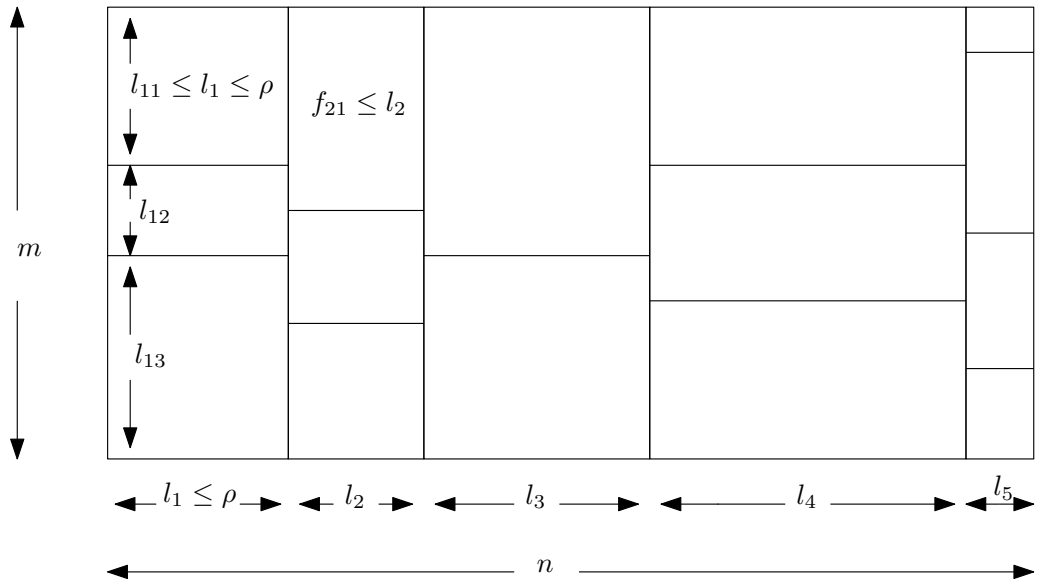


Figure 2.5: Two splits

Lemma 4 (the Rank Lemma). Summing over all indices k and s, the total extra space for the Permutation Lemma is at most $O(\sum_{k,s} f_{ks} + l_k + l_{ks}) = O(f) + O(f+n) + O(f+m)$. The extra space for the Rank Lemma is at most $f \lg \rho(1/z + 1/z')$. The total extra space is

$$f \lg \rho \left(\underbrace{1 + \frac{1}{\xi}}_{\text{Theorem 2}} + \underbrace{\frac{1}{z} + \frac{1}{z'}}_{\text{Lemma 4}} \right) + \underbrace{O(f+m+n)}_{\text{Lemma 1 and Lemma 3}},$$

where ξ is such that the representation of a permutation using Theorem 2 uses $(1+\xi)$ times the information theoretic minimum space, and $\rho = (nm)/f$. Therefore, the extra space in Lemma 1 and Lemma 3 is at most $O(n+f)$. The run times for our operations are as follows.

`row_rank`	`row_sel`	`col_rank`	`col_sel`
$\lg\lg\rho + t\lg z$	t	$\lg\lg\rho + t'\lg z'$	t'

where t and t' are the times to implement π and π^{-1} respectively using Theorem 2. In particular, if we use the forward or inverse encoding and choose $z = z' = \max\{t, t'\}$, so that $\xi = z$; and choose z such that $z \lg z = \lg\lg\rho$, so that $z = \Theta(\lg\lg\rho / \lg\lg\lg\rho)$, then we

obtain the asymptotic run-times as:

Name	`row_rank`	`row_sel`	`col_rank`	`col_sel`
Row	$\lg\lg\rho$	1	$\lg\lg\rho$	$\lg\lg\rho/\lg\lg\lg\rho$
Column	$\lg\lg\rho$	$\lg\lg\rho/\lg\lg\lg\rho$	$\lg\lg\rho$	1

The space for both the encodings is $f\lg\rho + O((f\lg\rho\lg\lg\lg\rho)/\lg\lg\rho)$.

For $z = z' = \lg\rho$, the Benes encoding gives us:

Name	`row_rank`	`row_sel`	`col_rank`	`col_sel`
Benes	$\lg\lg\rho(\lg\rho/\lg\lg n + 1)$	$\lg\rho/\lg\lg n$	$\lg\lg\rho(\lg\rho/\lg\lg n + 1)$	$\lg\rho/\lg\lg n$

and the total space is $f\lg\rho + O(f)$ bits.

The operations `row_nb` and `col_nb` can be implemented using the bit vector `Card` that was constructed in Lemma 1. To implement `row_nb`(i), we can find the positions y_1 and y_2 where the descriptions of the i-th and i+1-st rows respectively begins in `Card`, and then find the number of 1-bits between these positions as follows:

$y_1 \leftarrow$ `bin_sel`$_{\texttt{Card}}(0, (i-1)q+1)$ { start of the i-th row }
$y_2 \leftarrow$ `bin_sel`$_{\texttt{Card}}(0, iq+1)$ { start of the $i+1$-st row }
return `bin_rank`$_{\texttt{Card}}(1, y_2) -$ `bin_rank`$_{\texttt{Card}}(1, y_1)$

The operation `col_nb`(j) can be implemented similarly using the bit vector `Card`$_k$ that was constructed by Lemma 1 for the slice that contains the j-th column. The index of the slice, k, and the index of the corresponding column inside it, j, can be computed using (2.6) and (2.7). The operation `tab_acc` can be implemented using either (2.3) or (2.4). □

2.2.5 Applications to Text Indexing and Labeled Trees

In this section, we consider applications of data structures supporting the operations `str_rank`, `str_sel` and `str_acc` on a given text T. As earlier, let us denote their running times by t_r, t_s and t_a respectively. Let us encode T using a binary matrix R with $m = \sigma$ rows and $n = L$ columns with n 1-bits in it: $R[c, i] = 1$ if $T[i] = c$. Note that

$$\begin{aligned} \texttt{str\_rank}_T(c, i) &= \texttt{row\_rank}_R(c, i) \\ \texttt{str\_sel}_T(c, x) &= \texttt{row\_sel}_R(c, x) \\ \texttt{str\_acc}_T(i) &= \texttt{col\_sel}_R(i, 1) \end{aligned}$$

Hence, we can use Theorem 4 to implement `str_rank`, `str_sel` and `str_acc`.

In our first application, we add more functionality to the backward search algorithm that we described in Section 2.2.3. For a given occurrence of X in T, we show how to retrieve the text before and after it. Namely, the goal is: given a position i in SA, find the position i' such that $SA[i'] = SA[i] \pm 1$ ("+" sign is for the forward direction in the text, and "-" is for the backward). The main observation to solve this problem is as follows.

Lemma 5. *If* $BW[i] = c$ *and* $\texttt{str\_rank}(c, i) = j$*, then* $SA[z + j] = SA[i] - 1$*, where* $z + 1$ *is the starting location of the* c*-zone in* BW*.*

Proof. Recall that (by the definition of the suffix array) the j-th lexicographically smallest suffix that starts with c is located in position $SA[z + j]$, where $z + 1$ is the starting location of the c-zone. Let $i' = \texttt{str\_sel}_{BW}(c, j)$ be the position of the j-th character c in BW. Since all the suffixes are sorted lexicographically in SA, all suffixes that preceded by a character c are also sorted lexicographically. Hence, the suffix $SA[i']$ is the j-th lexicographically smallest out of those that preceded by a character c. It follows that $SA[i'] = SA[z + j] - 1$. □

To determine the starting locations of the c-zones, we will use the following bit vector

$$\texttt{Occur} := 0\ 1^{\texttt{occ}_T(c_1)}\ 0\ 1^{\texttt{occ}_T(c_2)}\ 0 \ldots 1^{\texttt{occ}_T(c_\sigma)},$$

where $c_1, c_2, \ldots, c_\sigma$ are the characters from Σ in lexicographical order.

Decoding Text Before an Occurrence The following algorithm shows how to decode the text to the left of $SA[i]$ using $\texttt{str\_acc}_{BW}$ and $\texttt{str\_rank}_{BW}$ operations only at a cost of $t_a + t_r$ per character (this algorithm is similar to the backward search algorithm).

Algorithm 6 DecodeBefore

$c \leftarrow \texttt{str\_acc}_{BW}(i)$
$j \leftarrow \texttt{str\_rank}_{BW}(c, i)$
$z \leftarrow \texttt{bin\_rank}_{\texttt{Occur}}(1, \texttt{bin\_sel}_{\texttt{Occur}}(0, c)) + 1$ { the c-zone in SA starts at the position $z + 1$ }
$i' \leftarrow z + j$ { by Lemma 5, $SA[i'] = SA[i] - 1$ }
output c, i' { decoded character and the location i' }

For an example, see Figure 2.2.

Decoding Text After an Occurrence To decode the text before a given occurrence, we can invert the previous algorithm. We first represent a given position i' in the form $i' = z+j$, where z is a starting location of some c-zone. Then, we find the position (denote it by i) of j-th character c in BW using `str_sel`. Since $BW[i] = c$ and $\texttt{str\_rank}_{BW}(c, i) = j$, by Lemma 5, we have that $SA[z + j] = SA[i] - 1$. Therefore, $SA[i] = SA[i'] + 1$. Using `Occur`, the values of c and j can be found as follows.

$$\begin{aligned} c &= \texttt{bin\_rank}_{\texttt{Occur}}(0, \texttt{bin\_sel}_{\texttt{Occur}}(1, i' - 1)) \\ j &= \texttt{bin\_sel}_{\texttt{Occur}}(1, i - 1) - \texttt{bin\_sel}_{\texttt{Occur}}(0, c) \end{aligned}$$

The following algorithm shows how to decode the text to the right of $SA[i]$ using $\texttt{str\_sel}_{BW}$ operation only at a cost of t_s per character.

Algorithm 7 DecodeAfter

$c \leftarrow \texttt{bin\_rank}_{\texttt{Occur}}(0, \texttt{bin\_sel}_{\texttt{Occur}}(1, i' - 1))$
$j \leftarrow \texttt{bin\_sel}_{\texttt{Occur}}(1, i' - 1) - \texttt{bin\_sel}_{\texttt{Occur}}(0, c)$
$i \leftarrow \texttt{str\_sel}_{BW}(c, j)$
output c, i { decoded character and the location i }

For an example, see Figure 2.2.

We summarize these algorithms by the following.

Theorem 5. *Let T be the text of length L on an alphabet of size σ, and ϵ be an arbitrary positive constant. Let r be the redundancy, and t_r, t_s, and t_a be the times to perform* `row_rank`, `row_sel`, *and* `col_sel` *operations for a binary matrix of size $\sigma \times L$ of cardinality L as defined by Theorem 4. There is a data structure that allows to perform Counting and Listing operations on T in times $O(pt_r)$, $O((t_a+t_r)\lg^{1+\epsilon} L)$, and allows to perform Context operations in time t_s (respectively, $t_a + t_r$) for retrieving a character after (respectively, before) an occurrence.*

The xbw Transform As a second application of our data structures, we improve the running times of the navigation and search operations of the xbw transform [13]. Note that the operations GetChildren, GetParent and SubPathSearch are quite similar to the algorithms DecodeBefore, DecodeAfter, and BackwardSearch respectively as the following table shows.

xbw operation	text operation	string operations
GetChildren	DecodeBefore	one `str_rank` and one `str_acc`
GetParent	DecodeAfter	one `str_sel`
SubPathSearch	BackwardSearch	two `str_rank`

Hence, using the data structure from Section 2.2.4, we can improve the running times of the two navigation of one search operations of Ferragina et al. [13] by a factor of $\lg \sigma / \lg \lg \sigma$ comparing to the wavelet trees of Grossi [26], while keeping the space close to the information theoretic minimum.

Chapter 3

Lower Bounds for Binary Vectors in the Indexing Model

In this chapter, we consider some intrinsic limitations of static indexing data structures. More specifically, we consider problems of the following form: given a bit vector B of length n, we are to represent it so that a given set of queries $\mathcal{Q}$ can be supported efficiently. We consider the indexing model of computation in this chapter (in Chapter 1, we gave a formal definition of the indexing model). Recall that the main restriction of this model is that the representation consists of two parts: the vector B itself in its *raw* form plus a small *index* I the purpose of which is to facilitate an efficient implementation of the queries in $\mathcal{Q}$. We investigate the question of the relationship between time t and space r for any data structure that implements rank or select queries under the model described in Definition 1. Earlier, in Section 2.1, we discussed data structures that implement a fully indexable dictionary, and presented a data structure that supports `bin_rank` and `bin_sel` queries in constant time using the original bit vector plus extra $r = (n \lg t)/t + O(n(\lg t)^2/t^2)$ bits of space, where t is the number of bit probes performed on raw data. This result gives a sufficient condition for the existence of a rank/select data structure. In Sections 3.2, 3.3, 3.5, and 3.6, we consider this problem from a different point of view and give some necessary conditions for the existence of a rank/select data structure. In particular, we show that an index of size $\Omega((n \lg \lg n)/\lg n)$ bits is necessary to implement rank or select queries on a bit vector in constant time. Moreover, we show that it is necessary even if

we are allowed to use $O(\lg n)$ bit probes to raw data, unlimited access to the index, and unlimited computation resources. In Section 3.7, we show a necessary condition for the balanced parentheses problem. We show that an index of size $r = \Omega((n \lg t)/t)$ is necessary if we require that the operation of finding the matching parenthesis uses at most t probes to raw data.

The proofs of all these results have the following framework. First, we fix a mapping between the bit vectors B and the possible indices I, and an algorithm A that implements a given set of queries $\mathcal{Q}$. We choose a subset of queries $\mathcal{Q}^*(n, m, t) \subseteq \mathcal{Q}$, where n is the length of B, $m = \texttt{card}(B)$ is the *cardinality* of B (the number of 1-bits in B), and t is the worst case running time of the algorithm A. We use γ to denote the number of queries in $\mathcal{Q}^*$.

We construct the *choices tree* G for $\mathcal{Q}^*$. The choices tree is essentially the decision tree for the specific set of queries $\mathcal{Q}^*$ that examines the index first. More precisely, the nodes of the choices tree are labeled with either "$I[l]$=?" (respectively, "$B[l]$=?"), where $1 \leq l \leq r$ (respectively, $1 \leq l \leq n$) the location being inspected in I (respectively, B), and the two outgoing edges of a node are labeled by 0 or 1 depending on the outcome of the probe. We call the probe a 0-*probe* or a 1-*probe* correspondingly. If the algorithm A during a simulation performs a probe to the location prescribed by the label of the current node x and the outcome of this probe is 0, then it moves to x's left child, and otherwise, it moves to x's right child. The first r levels correspond to all possible choices of index: all the nodes at level d, $1 \leq d \leq r$, are labeled with "$I[d]$=?". We informally say that any node at the level $r + 1$ of the choices tree "knows" the contents of the index, I_B.

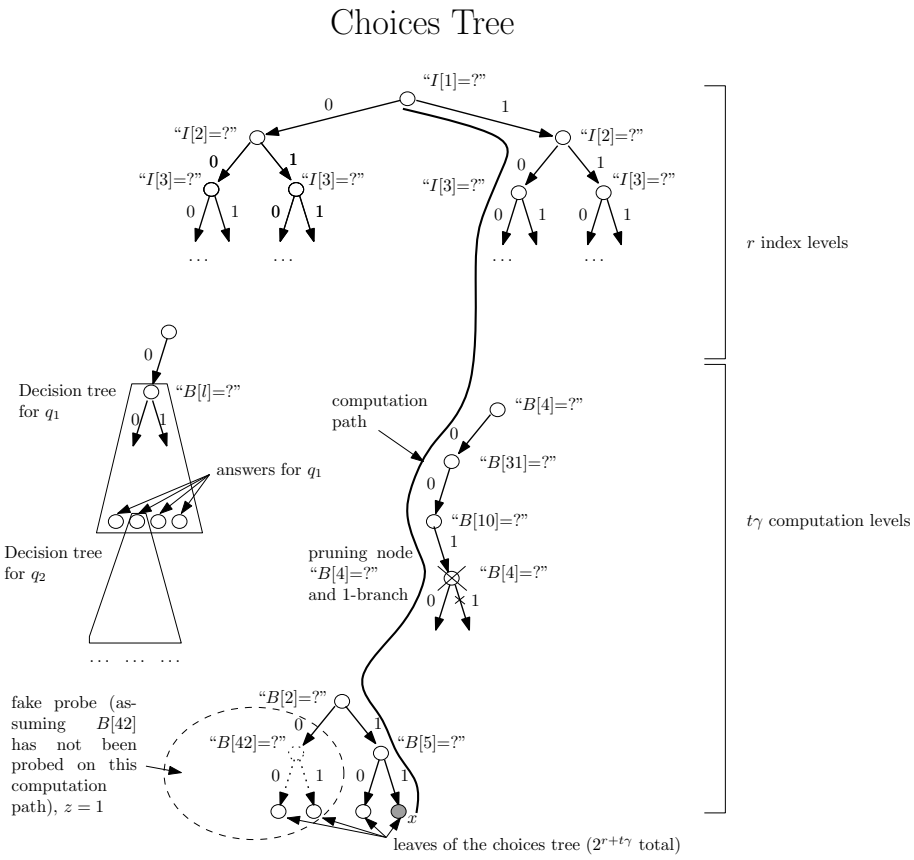
We consider queries from $\mathcal{Q}^*$ one by one in some fixed order, $\mathcal{Q}^* = \{q_1, q_2, \ldots, q_\gamma\}$. To each node at depth r of the tree constructed so far, we attach the decision tree of the computation that A performs for the query q_1. If for query q_1 and fixed index I (a node at the level $r + 1$ corresponds to a choice of the index), the first location of B that A probes is l, then at level $r + 1$ we create a node labeled "$B[l]$=?" and two outgoing edges labeled 0 and 1 depending on the outcome of the probe. At each of the two nodes at level $r + 2$, we perform the same procedure, and so on. We proceed until all possible branches of the computation are terminated with an answer for query q_1. We informally say that at the leaves of this tree, the index and the result of the query q_1 is "known". Once the

construction is completed for query q_1, we proceed with the queries $q_2, q_3, \ldots, q_\gamma$ in that order. Namely, at each leaf of the resulting tree, we attach the decision tree for query q_2, and so on.

The root-to-leaf paths in this tree are called *computation paths*. All the bit vectors that follow the computation path leading to a given leaf x have the property that they correspond to the same index I, the locations of probed bits and their contents, and also the results of the queries $q_1, q_2, \ldots, q_\gamma$ are the same for all of these bits vectors. For a node x, we call a bit vector B *compatible* with x if: (1) the labels of the first r edges on the root to the leaf path to x correspond to the index I of B; and (2) the remaining nodes on the root to the leaf path correspond to the choices made by the computation described above. In other words, B has the bits corresponding to the labels of the edges on the computation path to x in the respective positions that correspond to the labels of the nodes on this computation path. The set of instances compatible with with leaf x is denoted by $Z(x)$. Informally, we say that at the leaves of the choices tree, the results of all the queries in $\mathcal{Q}^*$ are "known".

The height of the choices tree is at most $r + t\gamma$. To simplify the calculations, we adjust the tree so that: (i) for every leaf x, the nodes on the computation path to x have distinct labels (in other words, the computation does not probe the same location twice), and (ii) all the leaves are at the same depth $r + t\gamma$. If we have a tree that violates condition (i), then consider a location l that is probed twice on the same path. We can remove the nodes that correspond to the second and later probes on l (i.e. all nodes labeled "$B[l]$=?" on the path except the first one), for all the removed nodes, only keeping the branch that corresponds to the outcome of the first probe to l on the path. We can keep repeating this procedure until the tree satisfies condition (i). If the tree violates the condition (ii), then consider a leaf x that has depth less than $r + t\gamma$, say $r + t\gamma - z$. We perform z arbitrary probes to B, by picking z unprobed locations, probing them one by one, and creating 2^z leaves at the depth $r + t\gamma$. The resulting tree is called a choices tree $G(\mathcal{Q}^*)$ for $\mathcal{Q}^*$. See Figure 3.1 for an example.

Once the choices tree is constructed, we pick a subset $\mathcal{H} \subseteq \{0,1\}^n$ of *hard* bit vectors. Informally, $\mathcal{H}$ is the set of instances of B for which implementing queries $\mathcal{Q}$ is hard. The notion of hardness only depends on the problem in question, but does not depend on a



	n_i	m_i	u_i	v_i	y_i
Block 1	7	4	3	2	2
Block 2	6	4	5	3	1
Block 3	6	4	4	3	1
Block 4	8	3	2	1	2
Block 5	5	1	3	0	1

Example of $B \in C(x)$ with $n = 32$, $m = 16$, $\gamma = 5$, $t = 3$, and $U = 17$

Figure 3.1: Choices tree and bit vectors

particular data structure (i.e. a mapping between the problem instances B and the indices I_B), and the algorithm A that implements the queries in $\mathcal{Q}$. We will explicitly define the set $\mathcal{H}$ for each problem in this chapter. For a given leaf x, let $C(x) = \mathcal{H} \cap Z(x)$ be the set of hard bit vectors that are compatible with x. Since every bit vector is compatible with exactly one leaf,

$$\sum_{x \text{ is a leaf}} |C(x)| = |\mathcal{H}|.$$

On the other hand, for each $B \in C(x)$,

1. the locations and the contents of probed bits are defined by the labels of nodes and edges respectively on the computation path to x;
2. the answers of the queries of $\mathcal{Q}$ are fixed by the outcome of the computation defined by the computation path to x; and
3. B is hard ($B \in \mathcal{H}$).

Based on these three facts only, we will compute an upper bound $C^*(x)$ on the number of different bit vectors that satisfy them, so that $|C(x)| \leq C^*(x)$. The method to compute this upper bound also depends on the problem in question and we will explain it later. The total number of instances that are compatible with all the leaves is $\sum_x C(x) \leq \sum_x C^*(x)$, since each hard instance is compatible with exactly one leaf. It follows that

$$|\mathcal{H}| = \sum_x |C(x)| \leq \sum_x C^*(x).$$

For simple cases, we will use

$$\begin{aligned} \lg |\mathcal{H}| &\leq \lg \sum_x |C^*(x)| \\ &\leq \lg(\text{number of leaves in } G) + \lg(\max_x\{C^*(x)\}) \\ &\leq (r + t\gamma) + \lg(\max_x\{C^*(x)\}), \end{aligned}$$

and derive a lower bound on the size of the index from this inequality.

The general approach we use to derive an upper bound on $C^*(x)$ is as follows. Consider a bit vector $B \in C(x)$ and split it into γ consecutive *blocks* of lengths $n_1(B), n_2(B), \ldots, n_\gamma(B)$.

This split is such that the values $n_1(B), n_2(B), \ldots, n_\gamma(B)$ only depend on the leaf x, but not on a particular B in question, so that we can denote them by $n_1(x), n_2(x), \ldots, n_\gamma(x)$. For the i-th block, we denote by $m_i(B)$ its cardinality (i.e. the number of 1-bits in it), by $u_i(B)$ the number of unprobed bits in it, by $v_i(B)$ the number of unprobed 1-bits in it, and by $y_i(B)$ the number of 1-probes performed on it (recall, that a 1-probe is a probe that returns value 1). The values of $u_i(B)$ and $y_i(B)$ depend only on the set and the contents of the probed locations. In particular, they do not depend on the choice of B in $C(x)$, that is, for $B_1, B_2 \in C(x)$, we have that $u_i(B_1) = u_i(B_2)$, and $y_i(B_1) = y_i(B_2)$ for $i \in [\gamma]$. We will use the notation $u_i(x)$ and $y_i(x)$. Note that, in general, the cardinalities of blocks $m_i(B)$ are not determined by the choice of block positions, and clearly they do not depend on the set of probed locations and their contents. It may happen that $m_i(B_1) \neq m_i(B_2)$ for $B_1, B_2 \in C(x)$ for some x. However, for the particular problems that we consider in this chapter and for the choices of block locations that we use in the proofs, it turns out that m_i is always a function of a leaf only. In this case, we can use the notation $m_i(x)$ instead of $m_i(B)$. Observe that $v_i(B) = m_i(x) - y_i(x)$ so that v_i is also a function of the leaf x only. Later, in this chapter, we fix some leaf x of G, and shorten the notation to n_i, u_i, y_i, m_i, v_i (it is understood that they are a function of the leaf x in question). We denote by $U(x) = \sum_{i=1}^{\gamma} u_i$ the total number of unprobed bits, and $V(x) = \sum_{i=1}^{\gamma} v_i$ the total number of unprobed 1-bits. By the construction of the choices tree, $U(x) = n - t\gamma$. For an example, see Figure 3.1.

3.1 Related Work

The problem of deriving the necessary conditions for the existence of the rank/select indexing data structure was first considered by Miltersen [41] who showed that:

- any indexing data structure that implements `bin_rank` queries using t cell probes requires

  $$2(2r + \lg(w+1))tw \geq n \lg(w+1),$$

  where w denotes the cell size, t denotes the number of cell probes, and r is the size of an index (in bits); and

- any indexing data structure that implements `bin_sel` queries using t bit probes requires
$$3(r+2)(tw+1) \geq n.$$

We briefly analyze his techniques. To prove the results for the rank data structure, Miltersen splits the bit vector B into n/w blocks of the same size. He only considers the bit vectors, such that all the blocks are of the form 0^j1^{w-j} for some $j \in [w]$ (that is, a block encodes the value j in unary). He further splits blocks into $n/(2tw)$ groups of blocks (that is, $2t$ blocks per group), and considers $n/(2tw)$ independent problems. The i-th problem is to compute the cardinality of the group modulo $w+1$. Observe that this set of problems is equivalent to computing the sum of $2t$ vectors with $n/(2tw)$ components where each component is considered modulo $w+1$ and encoded in unary in a block. A cell probe corresponds to revealing one of the components of one of the $2t$ vectors. Then, he reduces the latter problem to the problem of computing $\texttt{bin\_rank}_B(1, p_i)$ for positions $p_i = 2twi$ for all i, $0 \leq i \leq n/(2tw)$.

For the case $(wt)/n = o(1)$, his bound is

$$r = \Omega\left(\frac{n \lg(w+1)}{tw}\right). \tag{3.1}$$

His method gives the optimal trade-off $r = \Omega(n \lg\lg n / \lg n)$ for the case where $w = \Theta(\lg n)$ and $t = \Theta(1)$. Also, for the case where $\lg w = \Omega(\lg t)$, his bound coincide with the bound that we obtain in Section 3.2, which is optimal. However, for the case of small w, for example, for the case $w = 1$ that corresponds to the indexing bit probe model with t bit probes, his method gives a bound of $\Omega(n/t)$, which is a factor of $\lg t$ worse than our bounds given by Theorem 6.

One can try to generalize Miltersen's approach to allow $O(\lg n)$ bit probes instead of $O(1)$ word probes. The difficulty is that in the bit probe model, a number $j \in [w]$ represented as 0^j1^{w-j} can be recognized using binary search in $\lg w$ bit probes, so that each independent problem of (i) can be solved in $O(\lg w)$ bit probes without using an index. One can also try to "shuffle" bits in this representation to disallow such binary searches; however, it is not clear whether this method can give a better bound. Another difficulty in generalizing Miltersen's approach lies in his reduction from the problem of

computing the sum of $2t$ vectors modulo $w+1$. A naive approach to this problem that stores the resulting vector modulo $w+1$ using $n \lg(w+1)/(2tw)$ bits shows that Miltersen's techniques cannot surpass the barrier given by (3.1) asymptotically. We conclude that the analysis of techniques in [41] is tight, and the bottleneck is in the reduction from the vector sum problem to the rank problem.

Next, consider the select problem. For this problem, Miltersen's bound (3.2) is

$$r = \Omega(n/(tw)). \tag{3.2}$$

We give an argument that techniques from Miltersen [41] cannot be improved from $r = \Omega(n/(tw))$ to the optimal $r = \Omega(n \lg(tw)/(tw))$. In his proof, Miltersen only considered the bit vectors that have cardinality $m = \Theta(n/(tw))$. However, for such vectors, we can construct an index of size $O(n/(tw))$ that lets us to implement `bin_sel` queries in constant time. Let us divide B into $\gamma = n/(tw)$ blocks of the same size tw. For the i-th block, we store its cardinality m_i in unary representation in a bit vector `Card`:

$$\texttt{Card} = 0\ 1^{m_1}\ 0\ 1^{m_2}\ 0\ \ldots\ 0\ 1^{m_\gamma}$$

of length $p + \Theta(n/(tw)) = O(n/(tw))$. We can implement $\texttt{bin\_sel}_B(1, x)$ on B as follows. First, find in which block the x-th 1-bit is located by:

- selecting the j-th 1-bit in `Card` that corresponds to the j-th bit in B, denote its position by z, and
- counting the number of 0-bits before position z will give us the required block number, denote it by k.

Let z' be the position of the k-th 0-bit, then there are $z - z'$ 1-bits between the beginning of the k-th block and the z-th position. Hence, we can scan the k-th block from left to right looking for the $(z-z')$-th 1-bit in B, and it is the bit in question, the required j-th 1-bit in B. As it was shown in Chapter 2, we can store bit vector `Card` using $O(|\texttt{Card}| + o(|\texttt{Card}|) = O(n/(tw))$ bits. Hence, an index of size $O(n/(tw))$ suffices, and it is not possible to show an asymptotically better bound than (3.2) using Miltersen's techniques. Also, it follows that for such bit vectors B, select indexes of size $O(n/(tw))$ are optimal.

Finally, we consider the background of the problem of representing balanced parentheses. Jacobson [33] was the first to consider this problem. His approach was to store the position of the matching parenthesis for every $n/\lg n$-th parenthesis, and hence, the extra space that he used for the index is $\Theta(n)$ bits. Later, Munro and Raman [45] employed a more sophisticated three level scheme using blocks of sizes $\Theta((\lg n)^2)$, $\Theta((\lg\lg n)^2)$ and $\Theta(\lg\lg n)$. The extra space that they used is $\Theta(n/\lg\lg n) = o(n)$, so it was the first succinct data structure for the balanced parentheses problem. Later, Geary et al. [20] used a two level recursive scheme with blocks of sizes $\Theta((\lg n)^2)$ and $\Theta(\lg n)$. They were able to reduce the extra space to $\Theta(n\lg\lg n/\lg n)$ bits. In Section 3.7, we show a lower bound that matches the bound of Geary et al. up to a constant factor.

3.2 Rank Index

In this subsection, we apply the general framework to the case of `bin_rank` queries, namely determining the number of 1-bits up to a given position.

Theorem 6. *Let B be a bit vector of length n, and*

$$\mathcal{Q} = \{\text{“query } \texttt{bin\_rank}_B(1,i)''\mid 1 \le i \le n\}$$

Let t be the time cost and r be the space cost of implementing $\mathcal{Q}$ in the indexing bit probe model. Then

$$r = \Omega\left(\frac{n\lg t}{t}\right).$$

Proof. Let the set of simulated queries be $\mathcal{Q}^* = \{\text{“query } \texttt{bin\_rank}_B(1,3ti)''\mid 1 \le i \le \gamma\}$, where $\gamma = \lfloor n/(3t)\rfloor$. Let $\mathcal{H} = \{0,1\}^n$ be the set of all possible bit vectors.

Fix a leaf x of the choices tree $G(\mathcal{Q}^*)$. An upper bound on the number $|C(x)|$ of bit vectors compatible with x can be derived as follows. We choose the block lengths to be $n_i = 3t$, so that the i-th block starts at position $1 + 3t(i-1)$ and ends at position $3ti$. Clearly, this split does not depend on a choice of $B \in C(x)$. Since

$$\{\text{“query } \texttt{bin\_rank}_B(1,3t(i-1))'', \text{“query } \texttt{bin\_rank}_B(1,3ti)''\} \subseteq \mathcal{Q}^*,$$

the results of these queries are "known" at the leaf x: namely for any $B_1, B_2 \in C(x)$, we have

$$\begin{aligned} \texttt{bin\_rank}_{B_1}(1, 3ti) &= \texttt{bin\_rank}_{B_2}(1, 3ti) \text{ , and} \\ \texttt{bin\_rank}_{B_1}(1, 3t(i-1)) &= \texttt{bin\_rank}_{B_2}(1, 3t(i-1)). \end{aligned}$$

Therefore, the cardinality of the i-th block,

$$m_i = \texttt{bin\_rank}_B(1, 3ti) - \texttt{bin\_rank}_B(1, 3t(i-1)),$$

depends on x only, but not on a particular choice $B \in C(x)$ (we define $\texttt{bin\_rank}_B(1, 0) = 0$ for convenience). Hence, the v_i also depend on x only. Thus, the number of bit vectors compatible with x is

$$|C(x)| = \binom{u_1}{v_1}\binom{u_2}{v_2}\dots\binom{u_\gamma}{v_\gamma}. \tag{3.3}$$

We divide both parts of (3.3) by 2^U, and obtain

$$\frac{|C(x)|}{2^U} = \frac{\binom{u_1}{v_1}}{2^{u_1}}\frac{\binom{u_2}{v_2}}{2^{u_2}}\dots\frac{\binom{u_\gamma}{v_\gamma}}{2^{u_\gamma}}. \tag{3.4}$$

To bound the latter expression, we use the estimation for central binomial coefficients, $\binom{2x}{x} = O(2^x/\sqrt{x})$ that follows from Stirling's approximation formula. We obtain,

$$\frac{\binom{u_i}{v_i}}{2^{u_i}} \le \frac{\binom{u_i}{u_i/2}}{2^{u_i}} \le \frac{c}{\sqrt{u_i}} \tag{3.5}$$

for some constant $c > 0$.

We call the i-th block *undetermined* if it has more than t unprobed bits in it, more precisely, if $u_i \ge t$, and *determined* otherwise. If the i-th block is determined, then $n_i - u_i > 3t - t \ge 2t$ probes have been performed on it. Since the total number of bit probes is $t\gamma$, the total number of determined blocks is at most $\gamma/2$. Thus, the number of undetermined blocks, is at least $\gamma/2$. Using the inequality $\binom{u_i}{v_i}/2^{u_i} \le 1$ for determined blocks, and (3.5) for undetermined blocks, we obtain that the product in (3.4) can be bounded by

$$\frac{|C(x)|}{2^U} \le \left(\frac{c}{\sqrt{t}}\right)^{\gamma/2},$$

where $U = n - t\gamma$. Summing this bound over all of the $2^{r+t\gamma}$ leaves in $G(\mathcal{Q}^*)$ yields

$$2^n = |\mathcal{H}| = \sum_{x \text{ is a leaf}} |C(x)| \leq 2^{r+t\gamma} 2^{n-t\gamma} c^{\gamma/2} t^{-\frac{\gamma}{4}} \leq 2^n 2^r c^{\gamma/2} t^{-\frac{\gamma}{4}}. \tag{3.6}$$

Hence, the theorem follows:

$$r \geq \frac{\gamma}{4} \lg t - \frac{\gamma}{2} \lg c = \Omega\left(\frac{n \lg t}{t}\right).$$

□

3.3 Select Index

In this section, we apply a similar technique to show an optimal lower bound for the problem of finding the position of the i-th 1-bit.

Theorem 7. *Let B be a bit vector of length n, and*

$$\mathcal{Q} = \{\text{“query } \texttt{bin\_sel}_B(1, i)\text{”} \mid 1 \leq i \leq m\}$$

Let t be the time cost and r be the space cost of implementing $\mathcal{Q}$ in the indexing bit probe model. Then $r = \Omega\left(\frac{n \lg t}{t}\right)$.

Proof. Essentially, the techniques in this proof are different from the proof of Theorem 6 in that (i) we have to choose a different set of queries $\mathcal{Q}^*$ to simulate, (ii) it is more convenient to choose a slightly smaller set hard instances $\mathcal{H}$, so that the queries from $\mathcal{Q}^*$ do not return -1. More formally, let us choose

$$\begin{aligned} \mathcal{H} &= \{B \in \{0,1\}^n \mid \texttt{card}(B) = \lceil n/2 \rceil\} \\ \mathcal{Q}^* &= \{\text{“query } \texttt{bin\_sel}_B(1, 3ti)\text{”} \mid 1 \leq i \leq \gamma\}, \end{aligned}$$

where $\gamma = \lfloor n/(6t) \rfloor$. We construct the choices tree $G(\mathcal{Q}^*)$, and fix a leaf x of G. It remains to compute the number of compatible bit vectors $C(x)$. We split B into γ *blocks*, such that the i-th block starts at the position $\texttt{bin\_sel}_B(1, 3t(i-1)) + 1$ and ends at the position $\texttt{bin\_sel}_B(1, 3ti)$ (we define $\texttt{bin\_sel}_B(1, 0) = 0$ for convenience). This split does not depend

on a choice of B in $C(x)$, since the results of the queries "query $\texttt{bin\_sel}_B(1, 3t(i-1))$" and "query $\texttt{bin\_sel}_B(1, 3ti)$" are in $\mathcal{Q}^*$. The cardinality m_i does not depend on a choice of B in $C(x)$, since $m_i = 3t$. As in the proof of Theorem 6, we say that the i-th block is *determined* if $u_i < t$ and *undetermined* otherwise. If the i-th block is determined, then $n_i - u_i > m_i - t = 2t$ probes have been performed on it. Since the total number of bit probes is $t\gamma$, then the number of determined blocks is at most $\gamma/2$. We use the same bounds for determined and undetermined blocks, and as with the inequality (3.6) in the proof of Theorem 6, we obtain

$$\binom{n}{\lceil n/2 \rceil} = \sum_{x \text{ is a leaf}} |C(x)| \leq 2^{r+t\gamma} c^{\gamma/2} 2^{n-t\gamma} t^{-\frac{\gamma}{4}} \leq 2^n 2^r c^{\gamma/2} t^{-\frac{\gamma}{4}},$$

since the number of bit vectors that are compatible with all the leaves is $|\mathcal{H}| = \binom{n}{\lceil n/2 \rceil}$. Thus,

$$r \geq \lg \binom{n}{n/2} - n + \frac{\gamma}{4} \lg t - \frac{\gamma}{4} \lg c = \Omega\left(\frac{n \lg t}{t}\right).$$

□

3.4 Bounding Lemmas

In this section, we consider the products of the following form:

$$X = \binom{u_1}{v_1}\binom{u_2}{v_2} \ldots \binom{u_\gamma}{v_\gamma}. \tag{3.7}$$

The main goal is to obtain good upper bounds for X. Later, the results developed in this section will be used to derive lower bounds for the size of the rank/select indices in Section 3.5 and Section 3.6. Define $U = \sum_i u_i$, and $V = \sum_i v_i$. Let $u_* \leq \min_i u_i$, and $v_* \leq \min_i v_i$ be some fixed parameters. In this section, we use the notation u_i, v_i, U and V which is similar to Section 3.2 and Section 3.3, however, it is not necessary that, for example, u_i should be the number of unprobed bits in the i-th block for some leaf of the choices tree. The results that we obtain here hold for abstract values of u_i and v_i, and the notational similarity is for convenience of the later use of these results (in Section 10 and Section 3.6). We prove the results of this section under the assumption that $V \leq U/2$, which will be reflected later in our results in Section 10 and Section 3.6.

This section is organized as follows. We first give some basic inequalities in Section 3.4.1. In Section 3.4.2, we develop some tools for bounding single binomial coefficients. In Section 3.4.3, we consider the problem of bounding X, and prove the main results which will be used later in this chapter. Namely, we consider special cases of this problem

- the case where $u_i \geq u_*$ for some given parameter $u_* > 0$, and
- the case where $v_i \geq v_*$ for given $v_* > 0$.

3.4.1 Related Work and Tools

In this section, we present the necessary tools which we use for bounding binomial coefficients.

Lemma 6 (Stirling approximation [12])**.** *For $n \geq 1$, we have*

$$\sqrt{2\pi} < \frac{n!}{\sqrt{n}(n/e)^n} \leq e.$$

One of the main tools used in Section 3.4.3 is the inequality between arithmetic and geometric means.

Theorem 8 (AM-GM inequality [7])**.** *For non-negative values $x_1, x_2, \ldots, x_n$, we have*

$$\prod_i x_i \leq \left(\frac{\sum_i x_i}{n}\right)^n.$$

We use this inequality to show the following lemma.

Lemma 7. *Let $v_1, v_2, \ldots, v_\gamma$ and $u_1, u_2, \ldots, u_\gamma$ be integers such that $0 < v_i < u_i$. Then*

$$\prod_i \left(\frac{u_i}{v_i}\right)^{v_i} \left(\frac{u_i}{u_i - v_i}\right)^{u_i - v_i} \leq \left(\frac{U}{V}\right)^V \left(\frac{U}{U-V}\right)^{U-V},$$

where $U = \sum_i u_i$ and $V = \sum_i v_i$.

Proof. We use Theorem 8 for the values

$$\underbrace{\frac{u_1}{v_1},\ldots,\frac{u_1}{v_1}}_{v_1 \text{ times}},\underbrace{\frac{u_2}{v_2},\ldots,\frac{u_2}{v_2}}_{v_2 \text{ times}},\cdots\underbrace{\frac{u_\gamma}{v_\gamma},\ldots,\frac{u_\gamma}{v_\gamma}}_{v_\gamma \text{ times}},$$

so that

$$\prod_i \left(\frac{u_i}{v_i}\right)^{v_i} \leq \left(\frac{\sum_i v_i \cdot \frac{u_i}{v_i}}{\sum_i v_i}\right)^{\sum_i v_i} = \left(\frac{\sum_i u_i}{\sum_i v_i}\right)^{\sum_i v_i} = \left(\frac{U}{V}\right)^V.$$

In a similar fashion, we use Theorem 8 for the values $u_i/(u_i - v_i)$ each taken $u_i - v_i$ times, and obtain

$$\prod_i \left(\frac{u_i}{u_i - v_i}\right)^{u_i - v_i} \leq \left(\frac{\sum_i u_i}{\sum_i u_i - v_i}\right)^{\sum_i u_i - v_i} = \left(\frac{U}{U-V}\right)^{U-V}.$$

Hence, the statement of the lemma follows. □

3.4.2 Bounds for a Binomial Coefficient

To estimate binomial coefficients, we use the following theorem.

Theorem 9. *For values u and v, such that $0 < v \leq u/2$, we have*

$$\frac{1}{e} < \frac{\binom{u}{v}}{\frac{1}{\sqrt{v}}\left(\frac{u}{v}\right)^v \left(\frac{u}{u-v}\right)^{u-v}} < \frac{4}{5}.$$

Proof. We start by estimating the value of $u!/(u-v)!$. To do so, we first show that the sequence $a_n = n!/(n/e)^n$ is increasing and $b_n = n!/(n/e)^{n+1}$ is decreasing for integers n, $n > 0$. Consider

$$\frac{a_{n+1}}{a_n} = \frac{(n+1)!e^{n+1}}{(n+1)^{n+1}} \frac{n^n}{n!e^n} = \frac{e}{\left(1+\frac{1}{n}\right)^n} = e^{1-n\lg(1+1/n)} > 1,$$

since $\lg(1+x) < x$ for $x > -1$. In a similar fashion, consider

$$\frac{b_{n+1}}{b_n} = \frac{(n+1)!e^{n+2}}{(n+1)^{n+2}} \frac{n^{n+1}}{n!e^{n+1}} = e\left(1 - \frac{1}{n+1}\right)^{n+1} = e^{1+(n+1)\lg(1-1/(n+1))} < 1,$$

since $\lg(1-x) < -x$ for $x < 1$. Since $a_u/a_{u-v} > 1$ and $b_u/b_{u-v} < 1$, we have

$$\begin{aligned}\frac{u!}{(u-v)!} &= \frac{\frac{a_u u^u}{e^u}}{\frac{a_{u-v}(u-v)^{u-v}}{e^{u-v}}} > \frac{(u/e)^u}{((u-v)/e)^{u-v}} = \left(\frac{u}{e}\right)^v \left(\frac{u}{u-v}\right)^{u-v}, \text{ and} \\ \frac{u!}{(u-v)!} &= \frac{\frac{b_u u^{u+1}}{e^{u+1}}}{\frac{b_{u-v}(u-v)^{u-v+1}}{e^{u-v+1}}} < \frac{u}{u-v}\frac{(u/e)^u}{((u-v)/e)^{u-v}} = \frac{u}{u-v}\left(\frac{u}{e}\right)^v \left(\frac{u}{u-v}\right)^{u-v}.\end{aligned}$$

We divide both of these inequalities by $v!$, and use Lemma 6. We obtain

$$\frac{1}{e}\frac{1}{\sqrt{v}}\left(\frac{e}{v}\right)^v \left(\frac{u}{e}\right)^v \left(\frac{u}{u-v}\right)^{u-v} < \binom{u}{v} < \frac{1}{\sqrt{2\pi}}\frac{1}{\sqrt{v}}\frac{u}{u-v}\left(\frac{e}{v}\right)^v \left(\frac{u}{e}\right)^v \left(\frac{u}{u-v}\right)^{u-v}$$

By the precondition of the theorem, $v \leq u/2$, so that $u/(u-v) \leq 2$. Also, $\sqrt{2\pi} > 5/2$, the statement of the theorem follows. □

For the case where we expect many binomial coefficients with $v_i = 0$, we will use the following simple lemma

Lemma 8. *For $0 \leq v \leq u$, we have*

$$u^v < \binom{u}{v} < \left(\frac{u}{v}\right)^v$$

Proof. The first inequality is obvious, and the second one is folklore, and also easy to derive:

$$\binom{u}{v} = \frac{u!}{(u-v)!v!} = \frac{u}{v}\frac{u-1}{v-1}\cdots\frac{u-v+1}{1} \geq \left(\frac{u}{v}\right)^v.$$

Here we used the fact that $(u-i)/(v-i) \geq u/v$ for $0 < i < v$, since $(u-i)v = uv - iv \geq uv - iu = u(v-i)$. □

3.4.3 Bounds for a Product of Binomial Coefficients

In this section, we present the results that allow bounding the value of X from (3.7) defined as

$$X = \binom{u_1}{v_1}\binom{u_2}{v_2}\cdots\binom{u_\gamma}{v_\gamma}.$$

Recall that $U = \sum_i u_i$, $V = \sum_i v_i$, $u_* \leq \min_i u_i$, and $v_* \leq \min_i v_i$, and $V \leq U/2$. First, we present a simple lemma that follows from Theorem 9 and Lemma 7.

Lemma 9. *Assume that $u_* \geq 2$ and $v_* \geq 1$, then*

$$X \leq 2^{-(\gamma/2)\lg v_* - 0.3\gamma + (\lg V)/2} \binom{U}{V}.$$

Proof. We bound each individual binomial coefficient $\binom{u_i}{v_i}$ in product X using the right part of the inequality of Theorem 9. Then, the resulting product is bounded using Lemma 7. Finally using the left part of the inequality of Theorem 9, we obtain a bound involving the binomial coefficient $\binom{U}{V}$.

In the case where $v_i \leq u_i/2$, we apply Theorem 9 directly. For the case $v_i > u_i/2$, we first decrease the value of v_i to $v_i' = \lfloor u_i/2 \rfloor$, and bound $\binom{u_i}{v_i} \leq \binom{u_i}{v_i'}$ and then apply the lemma. Note that $v_i' \geq u_i/2 \geq v_i/2$, and thus, $V' = \sum_i v_i'$ satisfies $V/2 \leq V' \leq V \leq U/2$. We obtain

$$\binom{u_i}{v_i} \leq \binom{u_i}{v_i'} \leq \frac{4}{5}\frac{1}{\sqrt{v_i'}} \left(\frac{u_i}{v_i'}\right)^{v_i'} \left(\frac{u_i}{u_i - v_i'}\right)^{u_i - v_i'}$$

We then take the product of these inequalities and apply Lemma 7 to bound the products of $(u_i/v_i')^{v_i'}$ and $(u_i/(u_i - v_i'))^{u_i - v_i'}$. We get

$$X < \prod_i \frac{4}{5\sqrt{v_i'}} \left(\frac{U}{V'}\right)^{V'} \left(\frac{U}{U - V'}\right)^{U - V'}$$

Then, we apply Theorem 9 to bound X in terms of $\binom{U}{V'}$,

$$X < e\sqrt{V'}\binom{U}{V'} \prod_i \frac{4}{5\sqrt{v_i}} \leq 2^{-\gamma/2 \lg v_* - 0.3\gamma + (\lg V)/2} \binom{U}{V},$$

since $\binom{U}{V'} \leq \binom{U}{V}$ as $V' \leq V \leq U/2$. We estimated $\lg(4/5) \leq -0.3$ to simplify the formulas. □

Lemma 9 gives a good bound when v_* can be chosen large enough. This is the case when all v_i are "evenly distributed" among the binomial coefficients. However, in the case when $v_i = 1$ for some i, we can only choose $v_* = 1$, and the bound is weak. To alleviate this difficulty, we prove two facts that help reduce our problem to the case where all v_i are of the same order. We start by considering the following problem: given that the values

of u_i are fixed such that $u_i \geq u_*$, where u_* is a given parameter such that $u_* \leq \min_i\{u_i\}$, we wish to maximize the product X. Intuitively, if we vary the values v_i, the ones that maximize X should be proportional to the respective values u_i, that is, v_i/u_i should be approximately the same (and hence, close to the value of V/U). The next lemma and corollary formalize this fact.

We call the tuple $(v_1, v_2, \ldots, v_\gamma)$ a *local maximum* if we cannot increase some v_i by 1 and decrease some other v_j by 1, so that the right part of (3.10) increases. The following simple lemma characterizes the local maxima

Lemma 10. *At a local maximum,*

$$\frac{v_j + 1}{u_j + 1} \geq \frac{v_i}{u_i + 1}$$

is satisfied for each pair (i, j), $i \neq j$.

Proof. At a local maximum, we have the following inequality

$$\binom{u_i}{v_i}\binom{u_j}{v_j} \geq \binom{u_i}{v_i - 1}\binom{u_j}{v_j + 1}$$

Dividing both parts by $\frac{(u_i)!}{(v_i-1)!(u_i-v_i)!}\frac{(u_j)!}{(v_j)!(u_j-v_j-1)!}$, we obtain

$$\frac{1}{v_i}\frac{1}{(u_j - v_j)} \geq \frac{1}{(u_i - v_i + 1)}\frac{1}{(v_j + 1)}$$

From this, we get $u_i v_j + u_i - v_i + v_j + 1 \geq v_i u_j$ and the lemma follows. □

Corollary 1. *At a local maximum, for all i, $1 \leq i \leq \gamma$, we have*

$$\left|\frac{v_i}{u_i} - \frac{V}{U}\right| < \frac{2}{u_*}.$$

Proof. Fix $i \neq j$, and apply Lemma 10 for the pair (i, j) and for the pair (j, i). It follows that

$$\frac{v_j}{u_j + 1} + \frac{1}{u_j + 1} \geq \frac{v_i}{u_i + 1} \geq \frac{v_j}{u_j + 1} - \frac{1}{u_i + 1}$$

Since u_i and u_j are at least u_*, we have

$$\left|\frac{v_i}{u_i + 1} - \frac{v_j}{u_j + 1}\right| \leq \frac{1}{u_* + 1} < \frac{1}{u_*}$$

Since v_i/u_i and v_j/u_j are at most 1,

$$\left|\left(\frac{v_i}{u_i+1}-\frac{v_j}{u_j+1}\right)-\left(\frac{v_i}{u_i}-\frac{v_j}{u_j}\right)\right| = \left|\frac{v_i/u_i}{u_i+1}-\frac{v_j/u_j}{u_j+1}\right| < \frac{1}{u_*}$$

Therefore,

$$\left|\frac{v_i}{u_i}-\frac{v_j}{u_j}\right| < \frac{2}{u_*}.$$

Finally, we observe that

$$\min\left\{\frac{v_1}{u_1},\ldots,\frac{v_\gamma}{u_\gamma}\right\} \le \frac{V}{U} \le \max\left\{\frac{v_1}{u_1},\ldots,\frac{v_\gamma}{u_\gamma}\right\}$$

and the corollary follows. □

Using these facts, we now can establish the following lemma for the case where $u_*V/U \ge 3$.

Lemma 11. *If $u_*V/U \ge 3$, then*

$$X \le 2^{-(\gamma/2)\lg(u_*V/U)-0.3\gamma+(\lg V)/2}\binom{U}{V}.$$

Proof. We first maximize X with respect to v_i's for fixed u_i's. At a local maximum, Corollary 1 gives us the bound

$$\frac{v_i}{u_i} > \frac{V}{U}-\frac{2}{u_*} > \frac{V}{U}-\frac{2V}{3U} = \frac{V}{3U},$$

so that $v_i > u_iV/(3U)$. Hence, we can apply Lemma 9 with $v_* = u_iV/(3U) \ge u_*V(3U) \ge 1$. The result follows. □

3.5 Density-Sensitive Rank Index

In this section, we consider the problem of implementing the rank queries in the case where the cardinality m of the bit vector B is fixed. The bounds are then expressed in terms of both parameters m and n, the length of B. We will use techniques similar to Section 3.2; however, the calculations are slightly more involved in this case.

Theorem 10. *Let $\mathcal{H}$ be the set of all bit vectors of length n containing $m \leq n/2$ 1-bits. To support queries*

$$\mathcal{Q} = \{\text{“query } \texttt{bin\_rank}_B(1, i)\text{”} \mid 1 \leq i \leq n\}$$

on vectors from $\mathcal{B}$ with the time cost t, we must incur a space cost of

$$r = \begin{cases} \Omega\left(\frac{n}{t} \lg\left(\frac{mt}{n}\right)\right) & , \text{ if } \frac{mt}{n} = \omega(1) \\ \Omega(m) & , \text{ if } \frac{mt}{n} = \Theta(1) \\ \Omega\left(m \lg\left(\frac{n}{mt}\right)\right) & , \text{ if } \frac{mt}{n} = o(1) \end{cases}$$

Proof. This proof follows the lines similar to the proof of Theorem 6; however, it is a bit more technically involved. We choose an integer parameter $\gamma \leq n/(3t)$, and simulate the set of queries

$$\mathcal{Q}^* = \{\text{“query } \texttt{bin\_rank}_B(1, ik)\text{”} \mid 1 \leq i \leq \gamma\},$$

where $k = \lfloor n/\gamma \rfloor$. Accordingly, we split bit vectors B into γ blocks of equal lengths $n_1 = n_2 = \ldots = n_\gamma = k$. We define $\mathcal{H} = \{B \in \{0,1\}^n \mid \texttt{card}(B) = m\}$. The exact value of parameter γ is chosen depending on the relationship between mt and n as follows.

1. In the case where $mt = \omega(n)$, we will choose $\gamma = n/(3t)$.
2. For the case $mt = \Theta(n)$, we need an additional requirement that $\gamma \leq m/3$, so that we will choose $\gamma = \min\{n/(3t), m/3\}$, we will clarify this requirement later in the proof.
3. Finally, for the case $mt = o(n)$, we will choose $\gamma = \sqrt{nm/t}$.

We can verify that, by the definition of γ, the number of probed bits is at most $t\gamma \leq n/3$: for the first case the number of probed bits is exactly $n/3$, for the second case it is also at most $n/3$, and for the third case it is $\sqrt{nmt} = o(n) < n/3$ as well.

As compared with the proof of Theorem 6, to get a meaningful result, we must be more careful when bounding $C(x)$. To do so, let us partition all the leaves of the choices tree $G(\mathcal{Q}^*)$ (see the construction of this tree in the introduction to this chapter) into m groups depending on the total number of 1-probes performed on B on the corresponding computation path (that is, excluding the first r index levels). Denote by L_y the group of

leaves for which we performed exactly y 1-probes on B. We have $|L_y| = 2^r\binom{t\gamma}{y}$. For each leaf $x \in L_y$, we bound the number of compatible bit vectors, $C(x)$, by

$$|C(x)| \leq \binom{u_1}{v_1}\binom{u_2}{v_2}\ldots\binom{u_\gamma}{v_\gamma}, \tag{3.8}$$

where u_i is the number of unprobed bits in the i-th blocks, and v_i is the number of unprobed 1-bits in the i-th block. By definition, the values u_i and v_i satisfy

$$u_1 + u_2 + \ldots + u_\gamma = U \tag{3.9a}$$

$$v_1 + v_2 + \ldots + v_\gamma = V, \tag{3.9b}$$

where $U = n - t\gamma$ (respectively, $V = m - y$) is the total number of unprobed bits (respectively, unprobed 1-bits). We will employ the inequalities from Section 3.4 to bound this product.

We start by combining blocks into larger *superblocks* as follows. We traverse all the blocks consecutively from left to right in a greedy fashion. We add blocks to a superblock, until the total number of unprobed bits in the current superblock is at least k, at which point we finalize the superblock and start the next one. Since the total number of unprobed bits u_i in the $i-th$ block is less than $u_i \leq n_i = k$, by the construction, the total number of unprobed bits u_i^* in the i-th superblock satisfies $k \leq u_i^* < 2k$. More formally, the i-th superblock (except, perhaps, the last one) will contain the blocks $z_{i-1}+1, \ldots, z_i$ so that

$$k \leq u_i^* = u_{z_{i-1}+1} + u_{z_{i-1}+2} + \ldots + u_{z_i} < 2k,$$

where $z_0 = 0$. Let γ_s be the number of superblocks, then $\gamma_s \geq U/(2k)$, since the number of unprobed bits in a superblock is at most $2k$. Also, $U = n - t\gamma \geq 2n/3$. Thus, $\gamma_s \geq n/(3k) = \gamma/3$, so that

$$\gamma/3 \leq \gamma_s \leq \gamma.$$

For each superblock, we use the inequality

$$\binom{u_{z_{i-1}+1}}{v_{z_{i-1}+1}}\binom{u_{z_{i-1}+2}}{v_{z_{i-1}+2}}\ldots\binom{u_{z_i}}{v_{z_i}} \leq \binom{u_i^*}{v_i^*}$$

where $v_i^* = v_{z_{i-1}+1} + v_{z_{i-1}+2} + \ldots + v_{z_i}$ is the total number of unprobed 1-bits in the i-th superblock. Then

$$\binom{u_1}{v_1}\binom{u_2}{v_2}\ldots\binom{u_\gamma}{v_\gamma} \leq \binom{u_1^*}{v_1^*}\binom{u_2^*}{v_2^*}\ldots\binom{u_{\gamma_s}^*}{v_{\gamma_s}^*}$$

Thus, the total number of bit vectors that are compatible with all the leaves, $\sum_x |C(x)|$, is at most

$$P := 2^r \sum_{y=0}^{\min\{t\gamma,m\}} \binom{t\gamma}{y}\binom{u_1^*}{v_1^*}\binom{u_2^*}{v_2^*}\cdots\binom{u_{\gamma_s}^*}{v_{\gamma_s}^*}$$

We define

$$X := \max_{y,v_1^*,,\ldots v_{\gamma_s}^*} \binom{t\gamma}{y}\binom{u_1^*}{v_1^*}\binom{u_2^*}{v_2^*}\cdots\binom{u_{\gamma_s}^*}{v_{\gamma_s}^*} \tag{3.10}$$

Hence, $P \leq 2^r mX$. Lemma 11 can be used to bound (3.10). However, on the other hand, $\sum_x |C(x)| = |\mathcal{H}| = \binom{n}{m} \leq P$. Therefore,

$$r \geq \lg\left(\frac{\binom{n}{m}}{X}\right) - \lg m$$

gives a lower bound on the size of the rank index.

- First, consider the case where $mt/n = \omega(1)$. Thus,

  $$\frac{m\min\{t\gamma, u_1^*, u_2^*, \ldots, u_{\gamma_s}^*\}}{n} \geq \frac{mk}{n} = \frac{3mt}{n} = \omega(1),$$

  so that the precondition of Lemma 11 for the product X is satisfied. It follows that,

  $$\begin{aligned} P &\leq m2^r 2^{-\gamma_s/2\lg(mt/n)}\binom{n}{m}, \text{ and} \\ r &\geq -\frac{\gamma_s}{2}\lg\left(\frac{mt}{n}\right) - \Theta(\gamma_s) + \Theta(\lg m) = \Theta\left(\frac{n}{t}\lg\left(\frac{mt}{n}\right)\right). \end{aligned}$$

- Now consider the case $mt/n = \Theta(1)$. We have,

  $$\frac{m\min\{t\gamma, u_1^*, u_2^*, \ldots, u_{\gamma_s}^*\}}{n} \geq \frac{mk}{n} = \frac{m}{\gamma_s} \geq 3,$$

  since we chose $\gamma_s \leq \gamma \leq m/3$ as we chose $\gamma = \min\{n/(3t), m/3\}$. Thus, we can apply Lemma 11 and derive

  $$\begin{aligned} P &\leq m2^r 2^{-\Theta(\gamma_s)}\binom{n}{m}, \text{ and} \\ r &= \Omega(\gamma_s) + \Theta(\lg m) = \Omega(m). \end{aligned}$$

- Finally, consider the case where $mt/n = o(1)$. We use a slightly different idea than in the previous two cases. We bound the product $\prod_i \binom{u_i}{v_i}$ directly by using Lemma 8. Namely, since

$$\max_i u_i \;\leq\; k = \frac{n}{\gamma}, \text{ we have } \prod_i \binom{u_i}{v_i} \leq \binom{n}{\gamma}^V \leq \binom{\frac{nV}{\gamma}}{V},$$

using Lemma 8 again. Thus,

$$\binom{t\gamma}{y}\prod_i \binom{u_i}{v_i} \leq \binom{t\gamma}{y}\binom{\frac{nV}{\gamma}}{m-y} \leq \binom{t\gamma+\frac{nV}{\gamma}}{m} \leq \binom{t\gamma+\frac{nm}{\gamma}}{m},$$

since $V = m - y \leq m$, and $t\gamma + \frac{nV}{\gamma} \geq \sqrt{nmt} = \omega(mt)$ is at least $2m$. Finally, we can bound r by

$$r \geq \lg\left(\frac{\binom{n}{m}}{\binom{\sqrt{nmt}}{m}}\right) \geq \lg\left(\frac{\left(\frac{n}{m}\right)^m}{\left(\frac{\sqrt{nmt}}{m}\right)^m}\right) \geq m\lg\sqrt{\frac{n}{mt}} = \frac{m}{2}\lg\left(\frac{n}{mt}\right)$$

The parameter $\gamma = \sqrt{nm/t}$ was chosen such that the upper part of the last binomial coefficient is minimized, namely $t\gamma + \frac{nm}{\gamma} = 2\sqrt{nmt}$. Also note that a naive choice of $\gamma = \Theta(n/t)$ does not give the desired lower bound, since the product of binomial coefficients

$$\binom{t\gamma}{m}\prod_i \binom{u_1}{0} \geq \left(\frac{\Theta(n)}{m}\right)^m$$

alone differs from $\binom{n}{m}$ by a factor of $2^{\Theta(m)}$ only. Thus, the best lower bound we can hope for in this case is only $r = \Omega(m)$. Similarly, if we choose $\gamma = \Theta(m)$, then

$$X \leq \binom{t\gamma}{m}\prod_i \binom{u_i}{v_i} \leq \frac{\binom{n}{m}}{2^{\Theta(m)}},$$

since each of the binomial coefficients in the product $\binom{u_i}{v_i}$ "generates" a factor of $\Theta(1)$ in front of $\binom{n}{m}$ according to Theorem 9 and Lemma 9. So we can obtain $r = \Omega(m)$ bound only for this choice of γ as well. It turns out that we need to choose parameter γ such that $\gamma = \omega(m)$ and $\gamma = o(n/t)$.

□

3.6 Density-Sensitive Select Index

In this section, we combine the techniques from Section 3.3 and the bounding techniques of Section 3.5 to show a lower bound for the size of the index required to implement the select queries in the case where the cardinality m of the bit vector B is fixed. The bounds are then expressed in terms of both parameters m and n, the length of B.

Theorem 11. *Let $\mathcal{H}$ be the set of all bit vectors of length n containing $m \leq n/2$ 1-bits. To support queries*

$$\mathcal{Q} = \{\text{“query } \texttt{bin\_sel}_B(1, i)\text{”} \mid 1 \leq i \leq m\}$$

on vector from $\mathcal{B}$ with the time cost t, we must incur a space cost of

$$r = \begin{cases} \Omega\left(\frac{n}{t} \lg\left(\frac{mt}{n}\right)\right) & , \text{ if } \frac{mt}{n} = \omega(1) \\ \Omega(m) & , \text{ if } \frac{mt}{n} = \Theta(1) \\ \Omega\left(m \lg\left(\frac{n}{mt}\right)\right) & , \text{ if } \frac{mt}{n} = o(1) \end{cases}$$

Proof. This proof builds upon proofs of Theorem 7 and Theorem 10, and the bounding techniques from Section 3.4. As with the proof of Theorem 10, we consider three cases: where $mt = \omega(n)$, $mt = \Theta(n)$, and where $mt = o(n)$.

We simulate the set of queries

$$\mathcal{Q}^* = \{\text{“query } \texttt{bin\_sel}_B(1, ik)\text{”} \mid 1 \leq i \leq \gamma\},$$

where $k = \lfloor m/\gamma \rfloor$. Accordingly, we split bit vectors B into γ *blocks* of equal cardinalities $m_1 = m_2 = \ldots = m_\gamma = k$. We define blocks similarly to the proof of Theorem 7. Namely, the i-th block starts at position $\texttt{bin\_sel}_B(1, (i-1)k) + 1$ and ends at position $\texttt{bin\_sel}_B(1, ik)$, so that the *cardinality* of each block (the number 1-bits in it) is exactly k (recall, that we defined $\texttt{bin\_sel}_B(b, 0) = 0$ for $b \in \{0, 1\}$ for convenience). We set $\mathcal{H} = \{B \in \{0,1\}^n \mid \texttt{card}(B) = m\}$. We choose parameter γ depending on the relationship between mt and n. In the case where $mt = \omega(n)$, we will choose $\gamma = n/(3t)$. For the case $mt = \Theta(n)$, we need an additional requirement that $\gamma \leq m/3$, so that we will choose $\gamma = \min\{n/(3t), m/3\}$, we will clarify this requirement later in the proof. Finally, for the case $mt = o(n)$, we will choose $\gamma = \sqrt{nm/t}$. Note that in all cases, the number of

unprobed bits U is $n - t\gamma \geq 2n/3$, so that the average number of unprobed bits per block is at least $(2/3)n/\gamma$ (we expect most of the blocks to have at least constant fraction of unprobed bits).

Also, in a similar fashion to the proof of Theorem 10, we define *superblocks*. The i-th superblock (except, perhaps, for the last one) will contain consecutive blocks $z_{i-1}+1, \ldots, z_i$, such that the number of unprobed 1-bits in the i-th superblock

$$v_i^* = v_{z_{i-1}+1} + v_{z_{i-1}+2} + \ldots + v_{z_i}$$

satisfies $k \leq v_i^* < 2k$. Note that this is always possible, since $v_i \leq m_i = k$. And hence, γ_s, the number of superblocks, is at least $V/(2k)$. The number of unprobed bits of i-th superblock is given by

$$u_i^* = u_{z_{i-1}+1} + u_{z_{i-1}+2} + \ldots + u_{z_i}$$

We use inequality

$$\binom{u_1}{v_1}\binom{u_2}{v_2}\ldots\binom{u_\gamma}{v_\gamma} \leq \binom{u_1^*}{v_1^*}\binom{u_2^*}{v_2^*}\ldots\binom{u_{\gamma_s}^*}{v_{\gamma_s}^*}$$

to bound the number of bit vectors compatible with a given leaf. The total number of bit vectors compatible with all the leaves is then

$$P = 2^r \sum_{y=0}^{\min\{t\gamma, m\}} \binom{t\gamma}{y}\binom{u_1^*}{v_1^*}\binom{u_2^*}{v_2^*}\ldots\binom{u_{\gamma_s}^*}{v_{\gamma_s}^*} \tag{3.11}$$

We can derive a bound on P, $P \leq m2^r X$, where X is the biggest product of binomial coefficients in this sum, let us denote it by X. To derive a bound on X, we can, for example, use Lemma 9. A difference with the proof of Theorem 10 is that we do not need to "redistribute" the weight of V between v_i's uniformly as it was done in Lemma 10 and Corollary 1, since we have bounds $k \leq v_i^* < 2k$ already. To derive a lower bound for r, we observe that $\sum_x |C(x)| = |\mathcal{H}| = \binom{n}{m} \leq P$. Therefore,

$$r \geq \lg\left(\frac{\binom{n}{m}}{X}\right) - \lg m.$$

- First, we consider the case where $mt/n = \Omega(1)$. Recall that we chose the parameter $\gamma = \min\{n/(3t), m/3\} = \Theta(n/t)$. The goal is to derive a bound on

$$X = \binom{t\gamma}{y}\binom{u_1^*}{v_1^*}\binom{u_2^*}{v_2^*}\ldots\binom{u_{\gamma_s}^*}{v_{\gamma_s}^*} \tag{3.12}$$

subject to constraints

$$\begin{aligned} t\gamma + \sum_i u_i^* &= n, \\ y + \sum_i v_i^* &= m, \text{ and} \\ t\gamma &\leq \frac{n}{3} \end{aligned}$$

Since there is a bound on v_i^*'s, namely, $v_i^* \geq k$, it seems that we can apply Lemma 9 directly and obtain a bound on X. The caveat is that, if V is too small, then the number of superblocks γ_s is small as well, and the bound will turn out to be weak. This problem did not arise in the proof of Theorem 10, since the bound on γ_s was based on the fact that $\sum u_i \geq 2n/3$, and we were grouping blocks into superblocks based on values of u_i. However, in this proof, we form superblocks based on v_i's, so that we need to bound their sum, $V = \sum_i v_i$, from below.

For this purpose, we use the idea that is similar to an idea in the proof of Theorem 10. Let us vary u_i's and v_i's in order to maximize

$$\binom{t\gamma}{y} \prod_i \binom{u_i}{v_i}.$$

As a very rough estimation, we can state the following: since $t\gamma \leq n/3$, we expect that y will be at most $m/3$ as well, and so that $V = m - y \geq 2m/3$ which is sufficient for our purposes. More formally, Lemma 10 gives us the following conditions at a local maximum:

$$\frac{v_i + 1}{u_i + 1} \geq \frac{y}{t\gamma + 1} \geq \frac{y}{n/3 + 1}$$

Thus, for any $i \in [\gamma]$, we have

$$\left(\frac{n}{3} + 1\right)(v_i + 1) \geq y(u_i + 1).$$

Summing them up, we obtain

$$V \geq y\frac{U + \gamma}{n/3 + 1} - \gamma \geq (m - V)\frac{2n/3 + 2}{n/3 + 1} - \frac{m}{3} = \frac{5m}{3} - 2V,$$

since $2 \leq \gamma \leq m/3$. Thus,

$$V \geq \frac{5m}{9}.$$

Now, it is easy to derive a bound on the number of superblocks, γ_s,

$$\gamma_s \geq \frac{V}{2k} \geq \frac{5m}{18}\frac{\gamma}{m} \geq \frac{5\gamma}{18} = \Theta\left(\frac{n}{t}\right),$$

which is sufficient to derive our bounds.

Let us apply Lemma 9 to (3.12). We obtain,

$$\begin{aligned} X &\leq \binom{t\gamma}{m} 2^{-(\gamma_s/2)\lg k - \Theta(\gamma_s)} \binom{U}{V} \leq 2^{-(\gamma_s/2)\lg k - \Theta(\gamma_s)} \binom{n}{m} \\ &\leq 2^{-\Theta(n/t)\lg k - \Theta(n/t) + \Theta(\lg m)} \binom{n}{m}. \end{aligned}$$

So that, in the case where $mt = \omega(n)$, we obtain

$$r = \Theta\left(\frac{n}{t}\lg\left(\frac{mt}{n}\right)\right) - \Theta\left(\frac{n}{t}\right),$$

since $k = m/\gamma = 3mt/n$. In the case where $mt = \Theta(n)$, we obtain

$$r = \Theta(m),$$

since $k = m/\gamma \geq 3$, $k = \Theta(1)$.

- It remains to consider the case where $mt = o(n)$. It turns out that this is the easiest case in the proof of Theorem 10 and Theorem 11. Recall that we defined $\gamma = m$ in this case, so that we select all the 1-bits in the bit vector using our queries. Hence, the number of compatible bit vectors $|C(x)|$ with any leaf x is exactly 1. We can bound the sum

$$\begin{aligned} \sum_x |C(x)| &\leq 2^r \sum_{y=0}^{m} \binom{t\gamma}{y} \leq m2^r \binom{tm}{m} \leq m2^r \left(\frac{etm}{m}\right)^m = m2^r(et)^m \\ &\leq m2^r(et)^m \frac{\binom{n}{m}}{\left(\frac{n}{m}\right)^m} \leq 2^r 2^{-m\lg(n/(mt)) + \Theta(m)} \binom{n}{m}, \end{aligned}$$

here we used Lemma 8. Thus,

$$r = \Theta\left(m\lg\frac{n}{mt}\right) - \Theta(m)$$

Thus, we considered all cases, and obtained the required results. □

3.7 Balanced Parentheses

In this section, we consider the problem PARENTHESES. We consider the set of objects B is all the balanced parentheses of a given length n. We are to implement a set of queries

$$\mathcal{Q} = \{\text{“query } \texttt{findmatch}_B(i)\text{”}\},$$

where the query $\texttt{findmatch}_B(i)$ returns the position of the matching parenthesis for the parenthesis at the position i. Since we work in the indexing model, without loss of generality, we can assume that B is stored as a binary vector of length n, where a 0-bit corresponds to an opening parenthesis and a 1-bit to a closing parenthesis. For a sequence of parentheses (not necessarily balanced), we define $\texttt{excess}_B(i)$ as the difference between the numbers of opening and closing parentheses in B up to (and including) the given position i. Define $\texttt{excess}_B(0) = 0$ for convenience. We can represent B via its excess function.

We start by giving some preliminary results. Let us consider a special class $N(n, f, l)$ of sequences of parentheses of length n that are not necessarily balanced. This class depends on integer parameters n, f, and l, such that $l < \min\{f, 0\}$, $-n \leq f \leq n$, and $f + n$ is even. Intuitively, this class can be described as follows: the excess function starts at 0 and finishes at f and it never crosses the value l. More formally, sequence B in $N(n, f, l)$ if and only if

- $\texttt{excess}_B(n) = f$, and
- $\texttt{excess}_B(i) > l$ for $0 \leq i \leq n$.

The conditions $-n \leq f \leq n$ and $f + n \equiv 0 \bmod 2$ are necessary for the class $N(n, f, l)$ to be not empty, since the function $\texttt{excess}_B(i)$ is 0 for $i = 0$ and it changes exactly by 1 as i grows by 1. The condition $l < \min\{f, 0\}$ is necessary, since the excess function has to take values 0 and f.

These sequences will play an important role later in the proof of the theorem that given a lower bound on the index size of the PARENTHESES problem. We show the following lemma.

Lemma 12 (due to Andre (1887), e.g. see the proof in [12]). *For f and l, such that $l < \min\{f, 0\}$, $-n \le f \le n$, and $(f+n)/2$ is even,*

$$|N(n, f, l)| = \binom{n}{(n+f)/2} - \binom{n}{(n+f)/2 - l}.$$

Proof. The number of sequences in $N(n, f, l)$ is the number of sequences with the property $\texttt{excess}(n) = f$ and no restriction on the intermediate points, $N(n, f, -\infty)$, minus the number of sequences with the property that $\texttt{excess}(n) = f$ and $\texttt{excess}(i) = l$ at some intermediate point i, $0 < i < n$. Let $M(n, f, l)$ be the set of sequences that have the latter property. In other words, $N(n, f, l) = N(n, f, -\infty) \setminus M(n, f, l)$. Consider a sequence $B \in M(n, f, l)$ and let i be the first index such that $\texttt{excess}(i) = l$. Construct a sequence $B' \in N(n, 2l - f, -\infty)$ as follows, $B'[k] = B[k]$ for $k \le i$, and $B'[k] = 1 - B[k]$ (i.e. the opposite bracket to $B[k]$) for $k > i$. Also, for every sequence $B' \in N(n, 2l-f, -\infty)$, we can find the first index j, such that $\texttt{excess}_{B'}(j) = l$, since $0 > l > 2l - f$. We can construct a sequence $B'' \in M(n, f, l)$ similarly: $B''[k] = B'[k]$ for $k \le j$, and $B''[k] = 1 - B'[k]$ (the opposite parenthesis of $B'[k]$) for $k > i$. Note that $B'' = B$, and so there is a one-to-one correspondence between sequences $M(n, f, l)$ and $N(n, 2l - f, -\infty)$. Thus, $|N(n, f, l)| = |N(n, f, -\infty)| - |N(n, 2l - f, -\infty)|$. Note that $|N(n, f, -\infty)| = \binom{n}{(n+f)/2}$, and $|N(n, 2l - f, -\infty)| = \binom{n}{(n-f)/2-l}$. □

Let us consider another class $N(n, h)$ of sequences of length n, where $h > 0$ is an integer parameter. This class is similar to $N(n, -h, -h-1)$ with an exception that the $\texttt{excess}_B(i)$ function is allowed to reach value $-h$ only when $i = n$. Clearly, the last position of such sequences is a closing parenthesis, so that a sequence from $N(n, h)$ is a sequence from $N(n-1, -h+1, -h)$ that is appended with a closing parenthesis. More formally, $N(n, h)$ is a set of sequences B of length n such that $\texttt{excess}_B(n) = -h$, and $\texttt{excess}_B(i) > -h$ for i, $0 \le i < n$. We say that $B \in N(n, h)$ satisfies the *block condition*. We will use the sequences from $N(n, h)$ to build the set of hard instances for the PARENTHESES problem. We first show the following corollary that gives a lower bound on the number of sequences in $N(n, h)$.

Corollary 2. *If $h = O(\sqrt{n})$ and $n + h$ is even, then the number $|N(n, h)|$ of sequences*

that satisfy the block condition is

$$|N(n,h)| = \Omega\left(\frac{h2^n}{n^{3/2}}\right).$$

Proof. Clearly, the last position of B is a closing parenthesis, so that we can apply lemma 12.

$$\begin{aligned} |N(n,h)| &= |N(n-1,-h+1,-h)| = \binom{n-1}{(n-h)/2} - \binom{n-1}{(n+h)/2} \\ &= \frac{(n-1)!}{\left(\frac{n-h}{2}\right)!\left(\frac{n+h}{2}-1\right)!} - \frac{(n-1)!}{\left(\frac{n-h}{2}-1\right)!\left(\frac{n+h}{2}\right)!} = \frac{h}{n}\binom{n}{(n+h)/2} \end{aligned} \tag{3.13}$$

Equation 3.13 is also known as Bertrand's ballot theorem [12]. For even values of n, we can estimate

$$\begin{aligned} \frac{\binom{n}{(n+h)/2}}{\binom{n}{n/2}} &= \frac{\left(\frac{n}{2}\right)\ldots\left(\frac{n}{2}-\frac{h}{2}+1\right)}{\left(\frac{n}{2}+\frac{h}{2}\right)\ldots\left(\frac{n}{2}+1\right)} = \left(1-\frac{h}{n+h}\right)\ldots\left(1-\frac{h}{n+2}\right) \\ &\geq \left(1-\frac{h}{n}\right)^{h/2} = \left(1-\frac{h}{n}\right)^{(n/h)(h^2/2n)} = (1/e-o(1))^{h^2/2n}. \end{aligned}$$

Since $h = O(\sqrt{n})$, we can further bound (3.13) by

$$\frac{h}{n}(1/e-o(1))^{h^2/2n}\binom{n}{n/2} = \Omega\left(\frac{h2^n}{n^{3/2}}\right)(1/e-o(1))^{O(1)} = \Omega\left(\frac{h2^n}{n^{3/2}}\right),$$

where we used the bound for the central binomial coefficient $\binom{n}{n/2} = \Omega(2^n/\sqrt{n})$. □

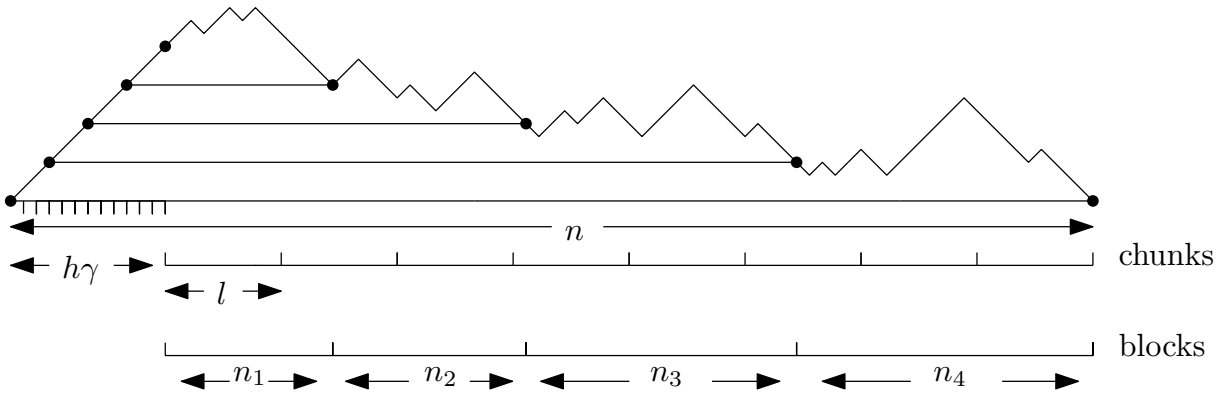
Now we can show the following theorem.

Theorem 12. *Let B be a balanced sequence of parentheses of length n. To support $\mathcal{Q} = \{\texttt{findmatch}(i) | \, 1 \leq i \leq n\}$ queries with time cost t, we must incur space cost r at least*

$$r = \Omega((n \lg t)/t).$$

Proof. Let us consider balanced parenthesis strings B of the following form:

- the string starts with $h\gamma$ opening parentheses;



Upward segments correspond to opening parentheses, downward segments correspond to closing parentheses, matching parentheses correspond to horizontal lines.

Figure 3.2: An example of `excess` function for $n = 84$, $l = 9$, $\gamma = 4$, and $h = 3$

- the rest of the string is divided into 2γ *chunks* of length l, $l = (n - h\gamma)/(2\gamma)$; and
- the matching parenthesis $\gamma_i = \texttt{findmatch}(ih + 1)$ is located in the $2(\gamma - i)$-th chunk for $0 < i < \gamma$, and $\texttt{findmatch}(1) = n$.

$$B = \underbrace{((\ldots(}_{h\gamma}\underbrace{\underbrace{**\ldots*}_{l}\underbrace{**\ldots)}_{l}\ldots*\ldots\underbrace{**\ldots)}_{l}\ldots*\underbrace{**\ldots*}_{l}\underbrace{**\ldots*)}_{l}}_{2\gamma \text{ chunks}}$$

For an example, see Figure 3.2. Let $\mathcal{H}$ denote the corresponding set of bit vectors. Let $\mathcal{Q}^* = \{\texttt{findmatch}(ih + 1) |\ 0 \leq i < \gamma\}$. We construct the choices tree $G(\mathcal{Q}^*)$. There is an important difference in the construction of G: if the query algorithm probes one of the first $h\gamma$ locations, then we do not count it as a probe, and do not create a separate node for it in G. The additional bit probes in the construction of the choices tree (that might be necessary to ensure that all leaves are at the same depth) can only be performed on the last $n - h\gamma$ bits of B, that is, no probes are performed on the first $h\gamma$ bits (they are fixed). The result of such probes is always 0 (corresponding to an opening parenthesis) for the bit vectors in $\mathcal{H}$.

We divide B into *blocks*, so that the first block starts at position $h\gamma + 1$ and ends at position $\texttt{findmatch}((\gamma - 1)h + 1)$, and the $(\gamma + 1 - i)$-th block starts at position $\texttt{findmatch}(ih + 1) + 1$ and ends at position $\texttt{findmatch}((i - 1)h + 1)$ for i, $1 \leq i \leq \gamma - 1$. Recall that $n_1, n_2, \ldots, n_\gamma$ denote the lengths of these blocks, and $\sum_i n_i = n - h\gamma$. The problem instances were constructed such that we can guarantee that $l \leq n_i \leq 3l$ for $1 \leq i \leq \gamma$. Namely positions $\texttt{findmatch}(ih + 1)$ and $\texttt{findmatch}((i + 1)h + 1)$ belong to chunks that are 1 chunk away from each other (also recall that chunks are of length l). Another property of our construction is that the i-th blocks satisfies the block condition $N(n_i, h)$, since the i-th block starts at height $(\gamma - i + 1)h + 1$ (see Figure 3.2) and ends at the height $(\gamma - i)h + 1$, and also the `excess` function never takes the value $(\gamma - i)h + 1$ in between these positions (for otherwise, the answer to the query $\texttt{findmatch}(\gamma - i)h + 1)$ would lie inside the block, which is impossible by the definition of blocks).

Let $\mathcal{Z}$ denote the set of all possible block configurations, that is, $\mathcal{Z}$ is the set of tuples $(n_1, n_2, \ldots, n_\gamma)$, such that there at least one hard problem instance from $\mathcal{H}$, such that $n_1, n_2, \ldots, n_\gamma$ are the 1-st 2-nd, ..., γ-th block length respectively for this instance. For the first block there is at least $\lfloor l/2 \rfloor$ choices for its length n_1, since the block has to end inside the second chunk, and we also have to satisfy the parity condition, i.e. $n_1 + h$ has to be even. In a similar fashion, we can show that for every $1 \leq i \leq \gamma$, there are $l/2$ choices for n_i. Thus,

$$|\mathcal{Z}| \geq \left(\left\lfloor \frac{l}{2} \right\rfloor \right)^\gamma$$

Using Corollary 2, we can estimate the number of such sequences

$$\begin{aligned} |\mathcal{H}| &= \sum_{\mathcal{Z}} \prod_{i=1}^{\gamma} |N(n_i, h)| = \sum_{\mathcal{Z}} \prod_{i=1}^{\gamma} \Omega\left(\frac{2^{n_i} h}{n_i^{3/2}} \right) = \sum_{\mathcal{Z}} \Omega(1)^\gamma \frac{2^{n-h\gamma} h^\gamma}{l^{3\gamma/2}} \\ &= 2^{\Theta(\gamma)} \left(\frac{l}{2} \right)^\gamma \frac{2^{n-h\gamma} h^\gamma}{l^{3\gamma/2}} = 2^{\Theta(\gamma)} 2^{n-h\gamma} \left(\frac{h}{\sqrt{l}} \right)^\gamma \end{aligned}$$

We can choose $h = \sqrt{l}$ so that the precondition of Corollary 2 is satisfied. Hence, $|\mathcal{H}| = 2^{\Theta(\gamma)} 2^{n-h\gamma}$.

Now we need to show an upper bound $C^*(x)$ on the number of sequences $C(x)$ compatible with any given leaf x of the choices tree $G(\mathcal{Q}^*)$. Recall that for the i-th block,

u_i denotes the number of unprobed locations in it. Also note that the number of closing parentheses in the i-th block is $m_i = (n_i + h)/2$. Note that the values n_i and m_i for a given sequence $B \in C(x)$ only depend on the leaf x, but not on a particular choice of $B \in C(x)$, so that they are well-defined as a function of leaf x. We can bound the number of compatible sequences by

$$C(x) \leq \prod_i \binom{u_i}{v_i} \leq \prod_i \binom{u_i}{u_i/2}$$

We say that the i-th block is *undetermined* if $u_i \geq l/2$, and *determined* otherwise. For a determined block, the total number of probes performed on it is at least $l/2$, since $n_i \geq l$, so that the total number of such blocks is at most $t\gamma/(l/2)$, since $t\gamma$ is the total number of probed bits. We choose $l = 4t$, so that the total number of undetermined blocks is at least $\gamma/2$.

We bound $\binom{u_i}{u_i/2}/2^{u_i} \leq 1$ for determined blocks, and we use Theorem 9 for undetermined blocks, namely

$$\binom{u_i}{u_i/2} \leq \frac{4}{5}\frac{2^{u_i}}{\sqrt{u_i/2}} \leq \frac{8}{5}\frac{2^{u_i}}{\sqrt{l}}.$$

Thus, the total number of compatible sequences with x is at most

$$\left(\frac{8}{5}\right)^{\gamma/2} \frac{2^{\sum_i u_i}}{(\sqrt{l})^{\gamma/2}} = \left(\frac{8}{10}\right)^{\gamma/2} \frac{2^{n-h\gamma-t\gamma}}{(\sqrt{t})^{\gamma/2}}$$

since $\sum_i u_i = n - h\gamma - t\gamma$. The total number of leaves in the choices tree is $2^r 2^{t\gamma}$, so that

$$2^{n-h\gamma-\Theta(\gamma)} \leq |\mathcal{H}| \leq 2^{r+t\gamma}\left(\frac{8}{10}\right)^{\gamma/2} 2^{n-h\gamma-t\gamma} t^{-\frac{\gamma}{4}} \leq 2^{n-h\gamma} 2^r 2^{\Theta(\gamma)} t^{-\frac{\gamma}{4}}$$

and hence

$$r \geq \frac{\gamma}{4}\lg t - \Theta(\gamma).$$

Since we chose $l = 4t$, and $h = 2\sqrt{t}$, we have to choose γ to satisfy the condition $l = (n - h\gamma)/(2\gamma)$. Namely, we have $\gamma = n/(8t + 2\sqrt{t}) \geq n/(10t)$, and thus,

$$r \geq \frac{n}{40t}\lg t - \Theta\left(\frac{n}{t}\right).$$

□

Chapter 4

The Text Searching Problem in the Indexing Bit Probe Model

In this chapter, we consider lower bounds regarding the time-space tradeoff for the problem of inverting permutations (PERMS) and the problem of searching in binary text strings (TEXTSEARCH). We first focus on the problem PERMS described as follows: represent a permutation such that queries $\{\pi(i)|1 \leq i \leq n\} \cup \{\pi^{-1}(i)|1 \leq i \leq n\}$ can be implemented efficiently. Our second problem, binary TEXTSEARCH, is as follows: Given a binary text T, we are required to support two sets of queries: $\texttt{access}_T(i)$ returns the contents of p consecutive characters of the text T starting at position i, and $\texttt{search}_T(X, j)$ returns the j-th occurrence of a given pattern X of length p in T if it exists (and -1 otherwise). We use L to denote the length of the text.

We study the PERMS problem in the indexing cell probe model first, and then improve and generalize the results to the binary TEXTSEARCH problem in the indexing bit probe model. The indexing bit probe model that we use for the binary TEXTSEARCH problem is the same as in Chapter 3; however, the techniques are quite different.

This chapter is organized as follows. In Section 4.1, we discuss the related work, contrast it with our contributions, state the main results of this chapter, and outline the techniques that are used in the proofs. In Section 4.2, we prove a theorem regarding the PERMS problem in the indexing cell probe model. In Section 4.3, we prove show a lower bound for the TEXTSEARCH problem in the indexing bit probe model.

4.1 Introduction

The first lower bound for the binary TEXTSEARCH problem in the indexing bit probe model was shown by Demaine and López-Ortiz [9, 10].

Theorem 13 (Demaine and López-Ortiz, Corollary 3.1 of [10]). *For the substring report problem in the indexing model, if* $p = \lg L + o(\lg L)$ *and* $t = o((\lg L)^2/\lg\lg L)$*, then*

$$r = \Omega\left(\frac{L \lg L}{t}\right)$$

Gal and Miltersen [19] considered a weaker *substring search problem* that can be formulated as follows: given a binary vector T of length L, and a pattern X of length p, determine whether X occurs in T. They showed a different tradeoff:

Theorem 14 (Gal and Miltersen, Theorem 3 of [19]). *For the substring search problem in the indexing model, if* $p = \Theta(\lg n)$*, then*

$$r = \Omega\left(\frac{L}{t \lg L}\right).$$

For small values of t, the bound of Theorem 13 is always bigger than in Theorem 14. However, as t reaches $\Omega((\lg L)^2/\lg\lg L)$, the techniques from [10] are not strong enough to yield any meaningful lower bound on the index size, while Theorem 14 still gives a non-trivial lower bound. Gal and Miltersen posed an interesting question [19]: "Can the two techniques be combined to yield a better lower bound?". More precisely, is the limitation $t = o((\lg L)^2/\lg\lg L)$ essential for the substring search/report problem? We develop a new compression technique and answer this question affirmatively by showing the following result:

Theorem 15. *For the substring report problem in the indexing bit probe model, if* $p = \lg L - o(\lg L)$*,* $t = o(\sqrt{L/\lg L})$*, then*

$$r \geq \begin{cases} \frac{L}{\lg L}(D-t)\left(1-\Theta\left(\frac{1}{t}\right)\right) & \text{, for } t \leq \frac{D}{2} \\ \frac{LD^2}{4t\lg L}\left(1-\Theta\left(\frac{1}{D}\right)\right) & \text{, otherwise,} \end{cases}$$

where $D = \lg L - \lg\lg L - 2\lg t$.

In comparison with Theorem 13, our bound applies to a much wider range of parameters t, namely

$$t \leq \sqrt{\frac{L}{\lg L}},$$

versus

$$t \leq \frac{\lg L}{\lg \lg L}.$$

In the range of applicability of Theorem 13, our bound is stronger. For example, in the interesting case where $t = c \lg L$, for a constant c, we obtain the bound

$$r \geq \frac{L}{4c} - \Theta\left(\frac{L}{\lg L}\right),$$

while the bound of Theorem 13 is

$$r \geq L\left(1 + 2c - 2\sqrt{c(1+c)}\right) - \Theta\left(\frac{L \lg \lg L}{\lg L}\right).$$

Our bound is stronger, since for $c > 0$,

$$\begin{aligned} 1 + 2c - 2\sqrt{c(1+c)} &= 1 - 2c(\sqrt{1+1/c} - 1) = 1 - 2c\left(\frac{1/c}{1+\sqrt{1+1/c}}\right) \\ &= \frac{\sqrt{1+1/c} - 1}{1+\sqrt{1+1/c}} = \frac{1}{c}\frac{1}{\left(1+\sqrt{1+1/c}\right)^2} < \frac{1}{4c}. \end{aligned}$$

For the PERMS problems, the first lower bound was shown by Munro, Raman, Raman, and Rao [44].

Theorem 16. *[44, Theorem 6] For the* PERMS *problem in the indexing cell probe model with time cost $t = o(\lg n / \lg \lg n)$, we have the following bound for the space cost:*

$$r \geq \Omega\left(\frac{n \lg n}{t}\right).$$

In fact, they considered a more general *permutation powers problem* that can be defined as follows. We are given a permutation on n elements π which must be represented so that $\{\pi^k(i)|1 \leq i \leq n, k \in \mathbb{Z}\}$ can be computed efficiently. In [44, Theorem 6], they reduced it to the PERMS problem (a subcase in which $k = \pm 1$) with only $n + o(n)$ extra bits of space. Thus, the PERMS problem is almost as hard as the original permutation powers problem. To show Theorem 16, they used the result of Demaine and López-Ortiz (Theorem 13) by converting cell probes into bit probes. A disadvantage of these results is that the range of applicability is quite small, $t = o(\lg n / \lg \lg n)$. It is not unreasonable to have running times higher than this. For example, an interesting data structure based on Benes networks proposed by Munro et al. [44] offers very little redundant space in exchange for running time $t = O(\lg n / \lg \lg n)$. However, this is not an indexing data structure.

In the following theorem, we show an improvement over [44, Theorem 6], where the constraint $t = o(\lg n / \lg \lg n)$ is relaxed to $t < n/2$.

Theorem 17. *For the* PERMS *problem in the indexing cell probe model with time cost $t < n/2$, we have the following bound for the space cost:*

$$r = \Omega\left(\frac{n}{t} \lg\left(\frac{n}{t}\right)\right).$$

Similar techniques were used in cryptography by Gennaro and Trevisan [21, Lemma 1]. Although not explicitly claimed in [21], using their techniques it is possible to prove the following bound on r:

$$r = \Omega\left(\frac{n}{t} \lg\left(\frac{n}{t^2}\right)\right).$$

for $t = o(\sqrt{n})$, which is weaker than in Theorem 17. Their techniques can be described as follows. Their idea was to build a directed graph G on n vertices, such that there is an edge from i to i' if and only if "query $\pi^{-1}(i)$" probes a location that contains i'. Without loss of generality, we can assume that after the algorithm probes a cell j that contains the value i, it terminates with answer j. Also, without loss of generality, we assume that the algorithms probes the location j to "make sure" that the answer is indeed j. This can only increase our running time by 1. They employ a greedy algorithm that starts out with the sets $I = [n]$ and $Y = \emptyset$. At each step, it removes the smallest element from I and includes it into Y. Then it removes all elements i' from I such that there is an edge from i to i'

and repeats the procedure. They argue that it takes at least $n/(t+1)$ steps before the greedy terminates. They also show that the permutation π can be encoded using the set Y, the set $X = \pi^{-1}(Y)$ and the sequence of probes. Our technique are slightly different: we encode the sequence of probes $\mathcal{P}$, and the markers $\mathcal{L}$ that denote how many probes were used in the sequence of probes by the current "query $\pi^{-1}(i)$". Advantages of our encoding compared to Gennaro and Trevisan are:

- we store only one set $\mathcal{L}$ in addition to $\mathcal{P}$ instead of two sets X and Y, and
- a new technique that allows us to compress $\mathcal{P}$ and $\mathcal{L}$.

Yao [58, Theorem 2] also stated a similar lower bound for inverting 1-cycle permutations and functions $[n] \mapsto [n]$ in the randomized case. The proof was omitted from [58] (extended abstract) and to the best of our knowledge has not appeared. He considered the following problem in the indexing bit probe model:

"Let N, m be positive integers. Consider the following game to be played by A and B. There are N boxes with lids $\texttt{BOX}_1, \texttt{BOX}_2, \ldots, \texttt{BOX}_N$ each containing a boolean bit. In the preprocessing stage, Player A will inspect the bits and take notes using an m-bit pad. Afterwards, Player B will ask Player A a question of the form "What is in $\texttt{BOX}_i$?". Before answering the question, Player A is allowed to consult the m bit pad and take off the lids of an adaptively chosen sequence of boxes not including $\texttt{BOX}_i$. The puzzle is, what is the minimum number of boxes A needs to examine in order to find the answer?"

To show a lower bound for this problem, he used an encoding scheme similar to Gennaro and Trevisan, namely, he simulated queries "what is in $\texttt{BOX}_i$" for i from 1 to n each time choosing a smallest index of the box that has not been opened by the previous queries. The difference with the case of permutations is that when we are answering the query "what is the value of $\pi^{-1}(i)$?", we do not restrict ourselves to probe any of the locations in memory. In particular, we allow to probe the location j that contains the value i. Allowing such probes (as it will be seen later in the proof) requires to introduce an additional bit vector $\mathcal{L}$ that is responsible for keeping track of how many probes we performed before such location j was accessed (if it was probed by the query).

Another contribution is a new technique that allows these results to be applied to the substring report problem and that in conjunction with ideas from Demaine and López-Ortiz [10] yields a stronger lower bound for the substring report problem, Theorem 15. The main idea of their techniques is also similar to Yao [58] and Gennaro and Trevisan [21]. They simulate queries that compute $\pi^{-1}(i)$, each time choosing the smallest i that has not been discovered so far. The main differences with Gennaro and Trevisan is that they adapt this scheme for the indexing bit probe model using the following two ideas:

- They encode permutation π using the following bit vector

$$T = \underbrace{0\,\mathtt{binary}_{\lceil \lg \mathtt{n} \rceil}(\pi(\mathtt{1}) - \mathtt{1})\,0}_{\lceil \lg n \rceil + 2 \text{ bits}} \quad \underbrace{11\ldots1}_{\lceil \lg n \rceil + 1 \text{ bits}} \quad \ldots \quad \underbrace{0\,\mathtt{binary}_{\lceil \lg \mathtt{n} \rceil}(\pi(\mathtt{n}) - \mathtt{1})\,0}_{\lceil \lg n \rceil + 2 \text{ bits}} \quad \underbrace{11\ldots1}_{\lceil \lg n \rceil + 1 \text{ bits}} .$$

- They treat the value $\pi(i)$ as probed if there are at least k bits that are probed out of the bits that encode $\pi(i)$, for some integer parameter $0 \leq k \leq \lg n$.

Our contribution is a new compression technique that allows us combine the techniques of Theorem 17 and these two ideas to obtain Theorem 15.

The general approach of the proofs of Theorem 17 and Theorem 15 is as follows. We fix a representation of the index I and an algorithm A that computes $\pi^{-1}(i)$ for given i, $1 \leq i \leq n$. Based on A and I, we will give an alternative encoding of permutations. For any fixed π, this new encoding can be described as follows. We "simulate" the algorithm A on queries q, $q \in \mathcal{Q} = \{\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)\}$ in that order. For a query q, we store the transcript of probes that A performed to the storage S. Namely, when A probes a bit or a cell (depending on the problem in question), the result of which cannot be derived from the probes or simulations performed earlier, we store the contents of the probed bit or cell in $\mathcal{P}$. We allow two deviations from this general scheme:

- It might happen that it is not "beneficial" for the encoding procedure to perform the simulation for a query q, in which case we skip it, and call the query q *absent*. The other queries are called *present*. We will clarify the notion of "beneficial" later in the proof, here it is given for the proof overview purposes.

- While performing the simulation of a present query q, it might happen that it is not beneficial to continue running it, in which case we *terminate* this simulation

immediately. However we do not remove any probes from $\mathcal{P}$ that A has made prior to its termination.

At the end of all the simulations, we record information that will allow us:

- to decode which queries are present and which are absent; and
- for every bit (or cell, depending on the model) of the vector $\mathcal{P}$, to determine which of the present queries has performed the corresponding probe. By construction, every present query q corresponds to a subsequence of consecutive bits (or cells, respectively), possibly empty, in $\mathcal{P}$.

We also store some extra information, to be described later, that allows us to decode π entirely. Finally, we calculate the total size of all the parts of this new encoding. The inequality on the size of index is derived by considering a permutation π that is incompressible in the sense of Kolmogorov [37]. Namely, any encoding of π must use at least $\lg n! - O(1)$ bits. Hence, in particular, our encoding that includes I as a part must use at least $\lg n! - O(1)$ bits. This inequality will imply a lower bound on the size of the index.

4.2 Permutations

In this section, we present a proof of Theorem 17.

Proof. Let us fix the representation of index I, and the query algorithm A. Let t be the time cost of A, and r be the space cost in bits (size of the index I). For every permutation π, we perform the simulations of the queries in $\mathcal{Q}$ in the order $\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)$. In the process of these simulations, we mark some of the cells as *discovered*.

In particular, (i) if a simulation probes an undiscovered cell, we mark it as discovered; (ii) if the simulation $\pi^{-1}(i)$ successfully terminates with answer j, then we mark the cell $S[j]$ discovered. Recall, that $S[j]$ stores the value of $\pi(j)$, so that $S[j] = i$, since $\pi^{-1}(i) = j$. We call $S[j]$ the *target cell* of $\pi^{-1}(i)$. Before running the next simulation in the list, say $q = \pi^{-1}(i)$, we first check whether its target cell was discovered earlier; if so, then we call the simulation q *absent* and skip it (since its result is already known), otherwise we call it *present*, and proceed with the execution of the algorithm A on the query q. If the

simulation $\pi^{-1}(i)$ probes its target cell, it is *terminated* just before such a probe is made, and we do not store the content of the target cell in $\mathcal{P}$. We still call such a simulation present and mark the target cell as discovered.

Let g_q be the number of cells that were appended to $\mathcal{P}$ during the simulation of the query q. We append $1^{g_q}0$ to the bit vector L to indicate that the last g_q cells were appended to $\mathcal{P}$ by the simulation of q.

The encoding algorithm can be described as follows:

Algorithm 8 Encode Permutation

$\mathcal{P}$ is a sequence of cells, initially empty
$\mathcal{L}$ is a sequence of bits, initially empty
for all $i = 1, 2, \ldots, n$ **do**
  if location $\pi^{-1}(i)$ is not marked as discovered **then**
    Call simulation i *present*
    Execute A on the query $\pi^{-1}(i)$
    Let $l_1, l_2, \ldots, l_z$ denote the locations of the cells that A inspected in order of its execution ($z \leq t$)
    Let j be the target cell ($j = \pi^{-1}(i)$)
    for all $f = 1, 2, \ldots, z$ **do**
      if location l_f is *not* marked as discovered **then**
        if $l_f = j$ **then**
          { Probing our target cell. Do not store $S[l_f]$ in $\mathcal{P}$ }
          Break the loop on f { Proceed to the "Append 0 to $\mathcal{L}$" statement }
        Append $S[l_f]$ to the end of $\mathcal{P}$
        Append 1 to the end of $\mathcal{L}$
        Mark l_f as discovered { Same location is never recorded twice}
    Append 0 to the bit vector $\mathcal{L}$
    Mark j as discovered

At the end of execution of this encoding algorithm, we compress $\mathcal{P}$ and $\mathcal{L}$ down to their information-theoretic minimum. The total length of $\mathcal{L}$ is n: whenever we discover a

cell, we also append 0 or 1 to $\mathcal{L}$; and at the end all of the simulations, all of the cells are discovered. The number of 0-bits in $\mathcal{L}$ is a, where a is the number of present simulations. There are at most t probes per a present simulation, and hence, there are at most $t+1$ discovered cells for each iteration of the loop on i in the encoding algorithm. Thus, the number of present simulations, a, is at least $n/(t+1)$.

Let us count the total possible number of possible sequences $\mathcal{P}$. First note, that the same value does not appear twice in $\mathcal{P}$, since the encoding algorithm cannot discover the same cell twice. Let $q_1, q_2, \ldots, q_a$ be all the present simulations, where q_j corresponds to the "query $\pi^{-1}(i_j)$". Let $g_1, g_2, \ldots, g_a$ be the numbers of cells that were used by $q_1, q_2, \ldots, q_a$ respectively, that is,

$$L = 1^{g_1}01^{g_2}0 \ldots 1^{g_a}0,$$

where $|P| = \sum g_i = n - a$. Note that by the construction, the value i_1 cannot occur in the first g_1 cells of $\mathcal{P}$, since if the encoding algorithm probes the value q_1 it immediately exits the simulation. The total number of possible subsequences $\mathcal{P}[1..g_1]$ that correspond to the probes made by the first query q_1 is $(n-1)(n-2)\ldots(n-g_1)$, since we can choose the first element, $\mathcal{P}[1]$, from the set $[n] \setminus \{i_1\}$; the second element, $\mathcal{P}[1]$, can be chosen from the set $[n] \setminus \{i_1, \mathcal{P}[1]\}$; and so on. The total number of subsequences $\mathcal{P}[g_1+1..g_1+g_2]$ that correspond to the second query q_2 is $(n-g_1-2)(n-g_1-3)\ldots(n-g_1-g_2-1)$, since $\mathcal{P}[g_1+1]$ can be chosen from the set $[n] \setminus \{i_1, i_2, \mathcal{P}[1], \mathcal{P}[2] \ldots \mathcal{P}[g_1]\}$, the element $\mathcal{P}[g_1+1]$ can be chosen from the set $[n] \setminus \{i_1, i_2, \mathcal{P}[1], \mathcal{P}[2] \ldots \mathcal{P}[g_1+1]\}$; and so on. Thus, the total possible number of sequences $\mathcal{P}$ is

$$\begin{aligned} M \;=\; & (n-1)(n-2)\ldots(n-g_1) \\ & (n-g_1-2)(n-g_1-3)\ldots(n-g_1-g_2-1) \\ & (n-g_1-g_2-3)(n-g_1-g_2-4)\ldots(n-g_1-g_2-g_3-2)\ldots \end{aligned}$$

Note that this product consists of $n-a$ distinct factors chosen from the set $[n]$. In other words,

$$M = \frac{n!}{\prod_{k=1}^{a} x_k},$$

where we denote

$$x_k = n - (k-1) - \sum_{j=1}^{k-1} g_k,$$

for $k \in [a]$.

Let us describe the compression scheme for sequences $\mathcal{P}$ and $\mathcal{L}$. Intuitively, the idea of this encoding is as follows. Since $\mathcal{P}$ can be encoded using $\lg M = \lg n! - \sum_k \lg x_k$ bits, we can pack $\mathcal{L}$ in the remaining bits as follows. If x_k is larger than t, then we can afford to encode g_k using $\lg t$ bits, and still "save" the remaining $\lg x_k - \lg t$ bits. However, if x_k is smaller than t, we cannot allow to spend $\lg t$ bits for g_k. However, recall that $g_k = x_k - x_{k-1} - 1$, so that $g_k \leq x_k - 1$, and we can encode g_k using $\lg x_k$ bits. We did not manage to save any bits in the former case, but at least we do not lose any.

More formally, for k such that $x_k > t$, we encode g_k as a value in the range from 0 to t, for other k, we encode g_k as a value in the range from 0 to $x_k - 1$. The length of the information-theoretic encoding of $\mathcal{L}$ is then

$$\left\lceil \lg \left((t+1)^{\alpha} \prod_{k=\alpha+1}^{a} x_k \right) \right\rceil.$$

The length of the information-theoretic encoding of $\mathcal{P}$ is $\lceil \lg M \rceil$. The total length of the encodings of $\mathcal{L}$ and $\mathcal{M}$ together is at most

$$\lg n! - \lg \left(\prod_{k=1} \alpha \frac{x_k}{t+1} \right) + 2.$$

Since $x_k \geq n - (k-1) - (k-1)t = n - (k-1)(t+1)$, we have that $x_k \geq n/2$ for $k \leq n/(2(t+1))$. Therefore,

$$\lg \prod_{k=1}^{\alpha} \frac{x_k}{t+1} \geq \lg \prod_{k=1}^{n/(2(t+1))} \frac{x_k}{t+1} \geq \frac{n}{2(t+1)} \lg \left(\frac{n}{2(t+1)} \right).$$

We conclude that the length of encoding of $\mathcal{P}$ and $\mathcal{L}$ is at most

$$\lg n! - \Theta \left(\frac{n}{t} \lg \left(\frac{n}{t} \right) \right).$$

Now let us show that using A, I, $\mathcal{P}$, and $\mathcal{L}$ we can decode π. We run the encoding and decoding algorithms in parallel in order to justify the correctness of the decoding algorithm. The decoding algorithm starts with an uninitialized array $\mathcal{S}$ of n elements, at the end of the decoding algorithm, we should obtain $\mathcal{S}[i] = \pi(i)$.

In the loop of the decoding algorithm, we maintain the invariant that $\mathcal{S}[j]$ is initialized with $\pi(j)$ if and only if $\pi(j)$ is discovered. In other words, value i is written somewhere in array $\mathcal{S}$ if and only if $\pi^{-1}(i)$ is marked as discovered. At each step, the decoding algorithm chooses the smallest value i that has not been written to $\mathcal{S}$. Clearly, this step corresponds to the step of the encoding algorithm where we choose the first i such that $\pi^{-1}(i)$ is not marked as discovered. Next, the decoding algorithm performs the simulation on $q =$ "query $\pi^{-1}(i)$". It starts by reading g, the number of cells probes that q needs to perform in order to obtain the answer to q or before the simulation of q probes the target cell with value i. Then it decodes the sequence $\mathcal{N}$ of next g values from the sequence $\mathcal{P}$ that will be fed to the simulation of q. More formally, we decode the sequence of g distinct values that are chosen from the subset of the values that are currently not written to $\mathcal{S}$ and are different from i. (Recall that, i is never probed during the simulation of q.) Next, it proceeds with the simulation retrieving the probes from $\mathcal{S}$ if possible, and if not, retrieving the probes from the decoded sequence $\mathcal{N}$. The simulation terminates in two cases: (i) naturally, if the algorithm A terminates after reading the last probe, or (ii) forcefully, if the algorithm tries to obtain $g + 1$-st probe from $\mathcal{N}$. In the latter case, the encoding algorithm has just made the probe to the target cell and terminated the simulation. Hence, the decoding algorithm writes the value i in the address of $\mathcal{S}$ that is currently requested, and proceeds to the next simulation. In the former case, the algorithm returned j as the result of $\pi^{-1}(i)$, so that the decoding algorithm writes $\mathcal{S}[j] = i$ and proceeds to the next simulation. The decoding algorithm can be also described in pseudo-code:

Algorithm 9 Decode Permutation

```
Start with an uninitialized storage S
for all i = 1, 2, ..., n do
  if the value i does not occur in S then
    { The simulation i is present }
```

Read g, the number of discovered cells, from $\mathcal{L}$.
Decode the next g cells from the sequence $\mathcal{P}$.
Start simulating A on the input i
When A needs to probe the cell at a location l
if $S[l]$ is initialized **then**
  Provide the value $S[l]$ to the algorithm A
else
  Read the next bit b from $\mathcal{L}$
  if $b = 0$ **then**
    { This probe is made to the target cell}
    Initialize $S[l]$ with i
    Terminate the simulation and put control back to the beginning of the loop on i
  Initialize $S[l]$ with the next value in the sequence $\mathcal{P}$
  Provide the value $S[l]$ to the algorithm A
Keep running the simulation until it stops naturally with an answer j
Initialize $S[j]$ with i

The length of our encoding is $|I|$ plus the size of the encoding of $\mathcal{P}$ plus the size of the encoding of L. There are at least a constant fraction of permutations that have high Kolmogorov complexity [37] of $K = \lg n! - O(1)$, and therefore cannot be described using less than K bits. Consider such a permutation π. In particular, the length of our encoding must be

$$r + \lg n! - \Theta\left(\frac{n}{t}\lg\left(\frac{n}{t}\right)\right) \geq \lg n! - O(1),$$

It follows that

$$r = \Omega\left(\frac{n}{t}\lg\left(\frac{n}{t}\right)\right).$$

□

4.3 Text Searching

In this section, we present a proof of Theorem 15. It combines the techniques of the proof of Theorem 17 and of Demaine and López-Ortiz [10].

Proof. Let A be an algorithm that implements a substring report query on a string of length L with time cost t and space cost r. We set $L = n(2\lg n + 3)$. Let π be a permutation on n elements. We will encode π as a text T of length L:

$$T = \underbrace{0\,\mathtt{binary}_{\lceil \lg \mathtt{n} \rceil}(\pi(\mathtt{1}) - \mathtt{1})\,0}_{\lceil \lg n \rceil + 2 \text{ bits}} \quad \underbrace{11\ldots1}_{\lceil \lg n \rceil + 1 \text{ bits}} \quad \ldots \quad \underbrace{0\,\mathtt{binary}_{\lceil \lg \mathtt{n} \rceil}(\pi(\mathtt{n}) - \mathtt{1})\,0}_{\lceil \lg n \rceil + 2 \text{ bits}} \quad \underbrace{11\ldots1}_{\lceil \lg n \rceil + 1 \text{ bits}} \quad . \quad (4.1)$$

The part of T that encodes $\pi(i) - 1$ is called the *i-th chunk*. The bits of T that do not depend on π are called *separators*. For each $i = 1, 2, \ldots, n$ in that order, we simulate the following query

$$q_i = \mathtt{search}(0\ \mathtt{binary}_{\lceil \lg \mathtt{n} \rceil}(\mathtt{i} - \mathtt{1})\ 0)$$

that searches for a pattern X of length $p = \lceil \lg n \rceil + 2$. The separators of the form $0\ 1^{\lceil \lg n \rceil + 1}\ 0$ ensure that the answer to the query q_i is the $\pi^{-1}(i)$-th chunk. The pattern X cannot overlap with two consecutive chunks since $\mathtt{binary}_{\lceil \lg \mathtt{n} \rceil}(\mathtt{i} - \mathtt{1})$ does not contain $1^{\lceil \lg n \rceil + 1}$.

In the process of these simulations, we mark some of the bits in T as *discovered*. Initially, all the separator bits are marked as discovered. The bits are also marked as discovered in the following two cases:

- if a simulation probes an undiscovered bit, we mark it as discovered;
- if the simulation q_i successfully terminates with an answer $(2\lceil \lg n \rceil + 3)j + 1$, then we mark all the bits in the $j + 1$-th chunk as discovered. Recall that, in this case, the $j + 1$-th chunk encodes $i - 1$ in binary, so $j + 1 = \pi^{-1}(i)$.

We call the $j + 1$-th chunk, the *target chunk* of q_i. Let $k \leq \lceil \lg n \rceil$ be a parameter. We will choose k later in the proof. Before running the next simulation in the list, say q_i, we first check whether its target chunk has more than k discovered bits; if so, then we call the simulation q_i *absent* and skip it. We know more than k bits of its target chunk, so it is not "beneficial" to run q_i. Otherwise we call it *present*, and proceed with the execution

of the algorithm A on the query q_i. Let $\mathcal{A}$ be a binary vector of length n, $\mathcal{A}[i] = 1$ if the query q_i is present. Similarly to the proof of Theorem 15, as the algorithm probes the locations in the bit vector, we immediately store them in the bit sequence $\mathcal{P}$. However, if the simulation q_i probes a location from its target chunk, it is *terminated* just before such probe is made, and we do not store in $\mathcal{P}$ the content of the bit at this location. We still call such a simulation present and mark all the bits in its target chunk as discovered; we can say (informally) that the last location that A tried to probe while simulating q_i just before it was terminated uniquely identifies the target chunk of q_i. Note the bit at a location l might be marked as discovered twice: once, probed by the simulation of a query q_i, and the second time, when it is a part of the target chunk of another query q_j; we call such locations *overlaps*. By the construction, there are at most k such bits per query q_j (otherwise, q_j is called absent and skipped during simulations).

Let q be the current query, and g_q be the number of bits we appended to $\mathcal{P}$ when simulating q (in other words, the number of discovered bits). Then we append $1^{g_q}0$ to the bit vector $\mathcal{L}$ to indicate that the previous g_q bits in $\mathcal{L}$ belong to the simulation of the query q. Finally, we store the contents of all the undiscovered bits from left to right in the bit vector $\mathcal{R}$.

Algorithm 10 Encode Bit String

$\mathcal{P}$, $\mathcal{L}$, and $\mathcal{R}$ are sequences of bits, initially empty
$\mathcal{A}$ is a binary vector of length n
for all $i = 1, 2, \ldots, n$ **do**
  if the target chunk of q_i has at most k probed bits **then**
    { Simulation i is called *present* }
    Set $\mathcal{A}[i]$ to 1
    Simulate A on the query q_i
    Let $l_1, l_2, \ldots, l_z$ denote the locations of the bits that A probed in order of its execution ($q \leq t$)
    Let $(2\lceil \lg n \rceil + 3)j + 1$ be the output of A { the first position of the target chunk of q_i }
    for all $f = 1, 2, \ldots, z$ **do**

if location l_f is not marked as discovered **then**
if l_f belongs to the target chunk **then**
Terminate the loop on f
Append $T[l_f]$ to the end of $\mathcal{P}$
Append 1 to the end of $\mathcal{L}$
Mark l_f as probed {The same location is never recorded twice in P}
Append 0 to the end of $\mathcal{L}$
Mark all the bits in the target chunk as discovered { Some of the bits in the target chunk might have already been marked as discovered, recall that in this case we call those bits *overlaps*, and there are at most k such bits }
else
{ Simulation i is called *absent* }
Set $\mathcal{A}[i]$ to 0

We will now compress the bit vectors $\mathcal{P}$, $\mathcal{R}$, $\mathcal{L}$, and $\mathcal{A}$. Let us denote the number of present simulations by a. We call a chunk *present* if it is the target chunk of a present simulation, otherwise call it *absent*. We introduce an additional bit vector $\mathcal{C}'$ of length n, $\mathcal{C}'[i] = 1$ if the chunk i is present, and $\mathcal{C}'[i] = 1$ otherwise. In other words, the chunk j is present if and only if it had at most k discovered bits at the moment when the simulation for the query q_i was about to execute, where $i = \pi(j)$. We encode all the bits in all the absent chunks in the bit vector $\mathcal{R}'$ from left to right. Note that these bits are of two types: all the bits recorded in $\mathcal{R}$, and some of the bits from $\mathcal{P}$. The bit vector $\mathcal{C}'$ can be encoded using $\lceil \lg \binom{n}{a} \rceil$ bits of space. The bit vector $\mathcal{R}'$ can be compressed in a similar fashion to the proof of Theorem 17, namely, $\mathcal{R}'$ stores the $n-a$ distinct numbers (the contents of all the chunks are distinct from each other) from $\{0, 1, \ldots, n-1\}$ and hence it can be encoded using $\lceil \lg(n!/a!) \rceil$ bits. Using $\mathcal{C}'$ and $\mathcal{R}'$ we can restore the contents and the locations of all the absent chunks. Note that storing the bit vector $\mathcal{A}$ is no longer necessary, since the i-th simulation is present if and only if the chunk encoding $i-1$ is not among the absent chunks. Also, we can remove all the bits that are located in the absent chunks from $\mathcal{P}$ to obtain $\mathcal{P}'$ (since those bits are encoded in $\mathcal{R}'$); respectively, we remove the corresponding 1-bits from $\mathcal{L}$ to obtain $\mathcal{L}'$.

Since we removed from $\mathcal{P}$ all the bits that are in absent chunks, the sequence $\mathcal{P}'$ consists of overlap bits only. Denote the number of overlap bits by $v = |\mathcal{P}'|$. Since in each present chunk we have at most k overlap bits, it follows that $|\mathcal{P}'| \leq ak$. The bit vector $\mathcal{L}'$ consists of the concatenation of bit vectors $1^{f'_i}0$ for all present queries q_i in increasing order of i. The value f'_i denotes the number of bits that corresponds to q_i in $\mathcal{P}'$, that is, the bits that were recorded in $\mathcal{P}$ during the simulation of q_i, and were not removed from $\mathcal{P}$ later (i.e., the overlap bits). The length of $\mathcal{L}'$ is $|\mathcal{P}'| + a$, and a of them are 0-bits. Thus, we can encode $\mathcal{L}'$ using only $\lceil \lg \binom{ak+a}{a} \rceil$ bits of space, we pad $\mathcal{L}'$ with 1-bits as necessary so that its length is exactly $ak + a$. Using the index I, the algorithm A, and the bit vectors $\mathcal{P}'$, $\mathcal{R}'$, $\mathcal{L}'$, and $\mathcal{C}'$, we can decode the permutation π. The decoding can be described in a fashion similar to Algorithm 9 as follows:

Algorithm 11 Decode Bit String

Let T be a bit vector of length $L = n(2 \lg n + 3)$.
Initialize all the separator positions with corresponding separators
Using $\mathcal{R}'$ and $\mathcal{C}'$, initialize all the absent chunks in T
for all $i = 1, 2, \ldots, n$ **do**
  if $(i-1)$ does not occur as the content of an absent chunk **then**
    { q_i is present }
    Simulate A on the query q_i
    When A needs to probe the bit at a location l in T
    if $T[l]$ is initialized **then**
      Provide $T[l]$ to the algorithm A
    else
      Read the next bit g from $\mathcal{L}'$
      if $g = 0$ **then**
        {This probe is made to an undiscovered location in the target chunk}
        Let j be the chunk where l is located
        Initialize the bits of the j-th chunk of T with $\texttt{binary}_{\lceil \lg \texttt{n} \rceil}(\texttt{i} - \texttt{1})$
        Terminate the simulation and put control back to the beginning of the loop on i

Initialize $T[l]$ with the next bit in the sequence $\mathcal{P}'$
Provide the value $T[l]$ to the algorithm A
Keep running the simulation until it stops naturally with an answer $(2\lceil \lg n \rceil + 3)j + 1$

Initialize the $j+1$-st chunk of T with $\texttt{binary}_{\lceil \lg \mathtt{n} \rceil}(\mathtt{i}-\mathtt{1})$

Consider permutations that are hard in Kolmogorov's sense, i.e. the minimum length of their descriptions is at least $\lg n! - O(1)$. The length of our encoding for such permutations is then

$$|I| + |\mathcal{R}'| + |\mathcal{C}'| + |\mathcal{L}'| + |\mathcal{P}'| \geq \lg(n!) - O(1). \tag{4.2}$$

Recall that $\mathcal{R}'$ can be encoded using $\lceil \lg(n!/a!) \rceil$ bits, $\mathcal{C}'$ can be encoded using $\lceil \lg \binom{n}{a} \rceil$ bits, $\mathcal{L}'$ can be encoded using $\lceil \lg \binom{ak+a}{a} \rceil$ bits, and the length of $\mathcal{P}'$ is at most v. We estimate all these quantities up to an $O(a)$ additive term:

$$\begin{aligned} |I| &\geq \lceil \lg(n!) \rceil - \lceil \lg(n!/a!) \rceil - \left\lceil \lg \binom{n}{a} \right\rceil - \left\lceil \lg \binom{ak+a}{a} \right\rceil - v && (4.3) \\ &\geq a \lg a - a \lg \frac{n}{a} - a \lg k - v - \Theta(a) = a \lg \frac{a^2}{nk} - v - \Theta(a). && (4.4) \end{aligned}$$

(4.5)

The total number of bits A probed during a present simulations is at most ta; v of these bits were probed in the present chunks and hence at most $ta - v$ in the absent ones. However, there are at least $(k+1)(n-a)$ bits probed in the absent chunks by definition of the absent chunks. Therefore, $ta - v \geq (k+1)(n-a) \geq k(n-a)$.

To bound expression (4.3), we prove the following technical lemma.

Lemma 13. *Let n and t be some fixed parameters, such that $t = \omega(1)$ and $n/t^2 = \omega(1)$. Denote*

$$f(k, a, v) = a \lg \frac{a^2}{nk} - v - ca,$$

where parameters k, a and v satisfy the constraint

$$v \leq \min\{ak, a(t+k) - nk\},$$

and c is a constant. Then

$$\max_k \min_{a,v} f(k,a,v) \geq \begin{cases} n\left(\lg\left(\frac{n}{t^2}\right) - t\right)\left(1 - O\left(\frac{1}{t}\right)\right) & , \text{ if } t < \frac{\lg(n/t^2)}{2} \\ \frac{n}{4t}\left(\lg\left(\frac{n}{t^2}\right)\right)^2\left(1 - O\left(\frac{1}{\lg(n/t^2)}\right)\right) & , \text{ otherwise.} \end{cases}$$

Proof. Function f is not hard to minimize. The minimum over v for fixed a is achieved when $v = \min\{ak, a(t+k) - nk\}$.

First consider the case where $ak \leq a(t+k) - nk$, i.e. where $a \geq nk/t$. Then,

$$f(k, a, ak) = a \lg\left(\frac{a^2}{2^c nk 2^k}\right)$$

is an increasing function for values of a where f is positive. Since we can control parameter k, we will choose it such that $2^c nk 2^k \leq a^2$ for all $a \geq nk/t$. It is equivalent $2^k/k \leq 2^{-c} n/t^2$. It suffices to choose $k = \lg(n/t^2)$, since $2^k/k < 2^{-c}2^k = 2^{-c}n/t^2$, since $k = \omega(1) > 2^c$. Then, the minimum equals to

$$g_1(k) = \frac{nk}{t} \lg\left(\frac{nk}{t^2 2^k}\right) - \Theta\left(\frac{nk}{t}\right).$$

Next, consider the case where $a < nk/t$, and so $v = a(t+k) - nk$. We have

$$f(k, a, a(t+k) - nk) = nk + a\left(\lg\left(\frac{a^2}{nk}\right) - (k+t) - c\right).$$

A local minimum is achieved when

$$f'_a(k, a, a(t+k) - nk) = \lg\frac{a^2}{nk} + 2\lg e - (k+t) - c = 0.$$

By substituting $\lg(a^2/(nk)) = k + t - 2\lg e - c$, we obtain that the value of the minimum is

$$g_2(k) = nk - 2a\lg e < nk - O\left(\frac{nk}{t}\right).$$

It remains to find

$$\max_{k \in [0, \lg(n/t^2)]} \min\{g_1(k), g_2(k)\}.$$

This maximum can be achieved in the following cases: (i) at a point where $g_1(k) = g_2(k)$, (ii) at a local maximum of $g_1(k)$, (iii) at a local maximum of $g_2(k)$, or (iv) at a boundary points $k = 0$ or $k = \lg(n/t^2)$.

(i) $g_1(k) = g_2(k)$ when $t = \lg \frac{nk}{t^2 2^k}$. In this case, we have $k - \lg k = \lg(n/t^2) - t$, and

$$g_1(k) = g_2(k) = nk - O\left(\frac{nk}{t}\right) \geq n\left(\lg\left(\frac{n}{t^2}\right) - t\right) - O\left(\frac{nk}{t}\right).$$

Note that this case only applies when $t < \lg(n/t^2)$. For the case $t > \lg(n/t^2)$, we have that $g_1(k) < g_2(k)$ for all k.

(ii) Local maxima of $g_1(k)$ are given by the following condition

$$g_1'(k) = \frac{n}{t}\left(\lg\left(\frac{nk}{t^2 2^k}\right) + \lg e - k\right) = 0.$$

Thus $2k = \lg(enk/t^2) \geq \lg(n/t^2)$. Substituting $\lg\left(\frac{nk}{t^2 2^k}\right) = k - \lg e$, we obtain

$$g_1(k) = \frac{nk^2}{t} - O\left(\frac{nk}{t}\right) \geq \frac{n}{4t}\left(\lg\left(\frac{n}{t^2}\right)\right)^2 - O\left(\frac{nk}{t}\right).$$

This case applies if $g_1(k) \leq g_2(k) \Leftrightarrow k \leq t \Leftrightarrow \lg(n/t^2) \leq 2t$.

(iii) The function $g_2(k)$ does not have local maxima.

(iv) At a point $k = \lg(n/t^2)$, we have that

$$\begin{aligned} \min\{g_1(k), g_2(k)\} &= \min\left\{nk, \frac{nk \lg k}{t}\right\} \leq \frac{n \lg(n/t^2) \lg\lg(n/t^2)}{t} \\ &= o\left(\frac{n}{t}\left(\lg\left(\frac{n}{t^2}\right)\right)^2\right), \end{aligned}$$

so that it is smaller than the value in (ii) and should not be considered. In the case $k = 0$, we have $g_1(k) = g_2(k) = 0$.

Let us compare the values in the case (i) and case (ii). We have

$$\left(\lg\left(\frac{n}{t^2}\right) - t\right) \leq \frac{\left(\lg\left(\frac{n}{t^2}\right)\right)^2}{4t}, \text{ since } \left(\lg\left(\frac{n}{t^2}\right) - 2t\right)^2 \geq 0.$$

Hence the value of (ii) is always greater than the value of (i), and they are equal when $t = (\lg(n/t^2))/2$. However, the case (ii) only applies for values $t \geq (\lg(n/t^2))/2$, hence, if $t < (\lg(n/t^2))/2$, we must use the case (i). The lemma follows. □

We can also reduce the length of the bit vector T by using separators of the form 01^z0 instead of $01^{\lceil \lg n \rceil + 1}0$ for some parameter z. In the following, for simplicity of the presentation, we assume that n is a power of 2. We call a number $i \in [n]$ *valid* if its binary encoding does not contain 1^z anywhere in it. Let M_n^z be the set of all valid numbers. We choose $z = 2 \lg \lg n$, and bound

$$n' = |M_n^z| \geq n - (\lceil \lg n \rceil - z + 1) 2^{\lceil \lg n \rceil - z} \geq n - \frac{2n \lg n}{(\lg n)^2} = n - o(n)$$

by excluding all the numbers that have a substring 1^z starting at position j for all the positions $j = 1, 2, \ldots, \lceil \lg n \rceil - z + 1$. We will encode a permutation on $[n']$ elements instead of n, and simulate queries for valid numbers only; note the set $M_n^{2 \lg \lg n}$ depends on n only, so we do not need to encode it explicitly. Inequality (4.2) becomes

$$|I| \geq \lg(n'!) - \lg(n'!/a'!) - \lg \binom{n'}{a'} - \lg \binom{a'k' + a'}{a'} - v',$$

We apply Lemma 13 to this expression, and obtain

$$|I| \geq a' \lg \left(\frac{a'^2}{n'k'} \right) - v' - \Theta(a') \geq \begin{cases} n'(D-t)\left(1 - O\left(\frac{1}{t}\right)\right) & \text{, if } t < \frac{D}{2} \\ \frac{n}{4t} D^2 \left(1 - O\left(\frac{1}{D}\right)\right) & \text{, otherwise.} \end{cases} \tag{4.6}$$

Where we denoted $D = \lg(n'/t^2)$.

The length of the bit vector B' with short separators is

$$L' = n' \lg n + 2n' \lg \lg n + 2n' < n' \lg n' + 2n' \lg \lg n' + 3n'.$$

Hence, $n' > L' / \lg L'$ and $D > \lg L' - \lg \lg L' - 2 \lg t$. Substituting this into (4.6), we obtain

$$|I| \geq \begin{cases} \frac{L'}{\lg L'}(D-t)\left(1 - \Theta\left(\frac{1}{t}\right)\right) & \text{, for } t \leq \frac{D}{2} \\ \frac{L' D^2}{4t \lg L'} \left(1 - \Theta\left(\frac{1}{D}\right)\right) & \text{, otherwise.} \end{cases}$$

The statement of the theorem follows.

□

Chapter 5

Lower Bounds in the Non-Indexing Model

In this chapter, we consider lower bounds in the cell probe model. In contrast with the lower bounds of Munro et al. [44] and Demaine and López-Ortiz [10] and their extensions in Chapter 4, we make no indexing model assumption about the representation. For example, in the case of the `PERMS` problem, we do not assume that the permutation is stored in its raw form in slow memory together with some extra indexing information in fast memory.

5.1 Introduction

In this section, we define the problems and the notation that will be used throughout this chapter. We start by introducing notation. Let $\mathcal{H}$ be the set of combinatorial objects. Let $\Upsilon = \left\lceil \frac{\log |\mathcal{H}|}{w} \right\rceil$ be the information-theoretic minimum space to represent an object from $\mathcal{H}$. That is, we need at least Υ cells each of size w bits to identify an object from $\mathcal{H}$ uniquely. For example, in this chapter, we consider several types of combinatorial objects: permutations π on n elements ($|\mathcal{H}| = n!$), texts T of length L over an alphabet of size σ ($|\mathcal{H}| = \sigma^L$), and binary matrices R of size $m \times n$ with f 1-bits ($|\mathcal{H}| = \binom{mn}{f}$).

Let S denote the storage. S is an array of $|S| = \Upsilon + r$ cells. We call r the *redundancy*. A *storage scheme* is an injective mapping `Rep` from objects $\mathcal{H}$ to arrays S. That is, we only consider deterministic storage schemes `Rep` (the storage is determined by the object).

Let $\mathcal{Q}$ be the set of queries that are to be implemented on objects $\mathcal{H}$. This set is given through a function g as follows:

$$\mathcal{Q} = \{\text{“query } (g, B, x_1, x_2, \ldots, x_k)\text{”} | B \in \mathcal{H}, (x_1, x_2, \ldots, x_k) \in \mathcal{X}\},$$

where g is a function $g\colon \mathcal{H} \times \mathcal{X} \rightarrow \mathcal{Y}$. That is, g is a function that is defined for any object $B \in \mathcal{H}$ and for any set of parameters that belong to some given domain $\mathcal{X}$, the range of this function is some given set $\mathcal{Y}$. From now on, to ease the notation, we will use “query $g(B, x_1, x_2, \ldots, x_k)$” instead of “query $(g, B, x_1, x_2, \ldots, x_k)$”.

For example, consider the case of the `PERMS` problem. The set of objects is $\mathcal{H} = \{\pi |\ \pi$ is a permutation on n elements $\}$. The set of queries is given by a function g that has two parameters: degree $d \in \{+1, -1\}$, and element $i \in [n]$, so that $\mathcal{X} = \{+1, -1\} \times [n]$. The range of this function is $\mathcal{Y} = [n]$. The answer to the query $g(\pi, d, i)$ is $\pi^d(i)$. We denote $g(\pi, 1, i)$ by “query $\texttt{forw\_perm}_\pi(i)$”, and $g(\pi, -1, i)$ by “query $\texttt{inv\_perm}_\pi(i)$”.

Consider the case of the `TEXTSEARCH` problem. The set of objects

$$\mathcal{H} = \{T |\ T \text{ is a text of length } L \text{ on an alphabet } \Sigma \text{ of size } \sigma\ \}.$$

The set of queries is given by a function g that has three parameters $x_1 \in \{0, 1\}$, x_2 and x_3, where $x_1 = 0$ indicates an access query and $x_1 = 1$ indicates a search query.

- In the case of an access query, $x_2 \in [L - p + 1]$ is the position in the substring of length p that we are to access, and x_3 is not used. The answer to the query $g(T, 0, x_2)$ is the substring of T of length p that starts at position x_2.
- In the case of a search query, we are to search for the x_3-th occurrence of the pattern x_2 in the text T and output its position if it exists and output -1 if it does not exist, for $x_2 \in \Sigma^p$ and $x_3 \in [L]$.

The range of $g(T, x_1, x_2, x_3)$ is therefore $\mathcal{Y} = \Sigma^p \cup [L - p + 1] \cup \{-1\}$. We denote $g(T, 0, x_2)$ by “query $\texttt{access}_T(x_2)$”, and $g(T, 1, x_2, x_3)$ by “query $\texttt{search}_T(x_2, x_3)$. In the rest of the chapter, we denote the search pattern by X instead of x_2.

Consider the binary relation problem. The set of objects is

$$\mathcal{H} = \{R |\ R \text{ is an } m \times n \text{ binary matrix containing } f \text{ 1-bits}\ \}.$$

The set of queries is given by a function g with three parameters: $x_1 \in \{0, 1\}$, x_2 and x_3, where $x_1 = 0$ indicates that it is a row query and $x_1 = 1$ indicates that it is a column query.

- In the case of a row query, we are to search the x_2-th row for the x_3-th 1-bit and output its position if it exists, and -1 if it does not exist, where $x_2 \in [m]$ and $x_3 \in [n]$.
- In the case of a column query, we are to search the x_3-th column for the x_2-th 1-bit and output its position if it exists, and -1 if it does not exist, where $x_2 \in [m]$ and $x_3 \in [n]$.

We denote $g(R, 0, x_2, x_3)$ by "query $\texttt{row\_sel}_R(x_2, x_3)$", and $g(T, 1, x_2, x_3)$ by "query $\texttt{col\_sel}_R(1, x_2, x_3)$". The range of $g(R, x_1, x_2, x_3)$ is $\mathcal{Y} = [\max\{n, m\}] \cup \{-1\}$.

In these specific examples, and in the general approach to our proof technique, we have two types of queries: *forward queries* and *inverse queries*. This division depends on the problem in question and is essential for the proofs. For example, for the `PERMS` problem, we say that "queries $\texttt{forw\_perm}_\pi(i)$" are forward, and "queries $\texttt{inv\_perm}_\pi(i)$" are inverse for $i \in [n]$. For the `TEXTSEARCH` problem, we say that "queries $\texttt{access}_T(i)$" are forward and "queries $\texttt{search}_T(X, j)$" are inverse. For the binary relation problem, we call "queries $\texttt{row\_sel}_R(i, x)$" forward and "queries $\texttt{col\_sel}_R(x, j)$" inverse. Intuitively, the meaning of forward and inverse queries is that we can reconstruct a given object $B \in \mathcal{H}$ knowing only answers to all the forward or all the inverse queries. The forward and inverse queries describe a combinatorial object from different points of view. For example, a permutation π on n elements is determined by the values of

$$\texttt{forw\_perm}_\pi(1), \texttt{forw\_perm}_\pi(2), \ldots, \texttt{forw\_perm}_\pi(n)$$

as well as by the values

$$\texttt{inv\_perm}_\pi(1), \texttt{inv\_perm}_\pi(2), \ldots, \texttt{inv\_perm}_\pi(n)$$

a text T can be reconstructed using only `access` or only `search` queries; and a binary relation R can be described uniquely by using only `row_sel` or only `col_sel` queries.

Let us fix a combinatorial object B. Depending on B, we fix two sets of queries: a subset of the forward queries $\mathcal{F}_B \subset \mathcal{Q}$ and a subset of the inverse queries $\mathcal{I}_B \subset \mathcal{Q}$ so that

$|\mathcal{F}_B| = |\mathcal{I}_B|$. We also fix a bijection η_B between these sets. The sets $\mathcal{F}_B$ and $\mathcal{I}_B$ and the bijection η_B will be chosen later depending on the problem in question. We start by giving some intuition about them. The first property (i) is that the object B is uniquely determined by either of the two: the set $\mathcal{F}_B$ (namely, an encoding of the parameters of every query in $\mathcal{F}_B$) and (an encoding of) the values returned by all the queries in $\mathcal{F}_B$, or the set $\mathcal{I}_B$ and the values returned by all the queries in $\mathcal{I}_B$. The correspondence η_B between forward and inverse queries is such that a forward query $q \in \mathcal{F}_B$ and its counterpart $q' = \eta(q) \in \mathcal{I}_B$ are "responsible for the same part" of the object B. Such a pair q, q' is called a *reciprocal pair*, and q and q' are called *reciprocal* (with respect to the object B) to each other. The second property (ii) is that the object B is uniquely determined by the sets $\mathcal{F}_B$, $\mathcal{I}_B$ and the mapping η_B.

For example, consider the case of the `PERMS` problem with a permutation π on n elements. Define

$$\begin{aligned} \mathcal{F}_\pi &= \{\text{“query } \texttt{forw\_perm}(i)\text{”} \mid i \in [n]\}, \\ \mathcal{I}_\pi &= \{\text{“query } \texttt{inv\_perm}(i)\text{”} \mid i \in [n]\}, \\ \eta_\pi(\text{“query } \texttt{forw\_perm}(i)\text{”}) &= \text{“query } \texttt{inv\_perm}(j)\text{”, where } j = \pi(i). \end{aligned}$$

The queries $\texttt{forw\_perm}(i)$ and $\texttt{inv\_perm}(j)$ are reciprocal for π if $\pi(i) = j$ (or $\pi^{-1}(j) = i$). In the example in Figure 5.1, we have

query	$\texttt{forw\_perm}(1)$	$\texttt{forw\_perm}(2)$	$\texttt{forw\_perm}(3)$	$\texttt{forw\_perm}(4)$
η(query)	$\texttt{inv\_perm}(7)$	$\texttt{inv\_perm}(5)$	$\texttt{inv\_perm}(2)$	$\texttt{inv\_perm}(1)$

query	$\texttt{forw\_perm}(5)$	$\texttt{forw\_perm}(6)$	$\texttt{forw\_perm}(7)$
η(query)	$\texttt{inv\_perm}(6)$	$\texttt{inv\_perm}(4)$	$\texttt{inv\_perm}(3)$

Next, consider the case of the `TEXTSEARCH` problem.

$$\begin{aligned} \mathcal{F}_T &= \{\text{“query } \texttt{access}(ip+1)\text{”} | 0 \le i < L/p\}, \\ \mathcal{I}_T &= \{\text{“query } \texttt{search}(X, j)\text{”} | \texttt{search}_T(X, j) = ip + 1 \\ &\quad \text{for some integer } i,\ 0 \le i < L/p\ \}, \\ \eta_T(\text{“query } \texttt{access}(ip+1)\text{”}) &= \text{“query } \texttt{search}(X, j)\text{”}, \\ &\quad \text{where } \texttt{search}_T(X, j) = ip + 1. \end{aligned}$$

Let π be the following permutation

i	1	2	3	4	5	6	7
$\pi(i)$	7	5	2	1	6	4	3

The following table shows an example os a data structure with $t = 3$ (number of probes by forward queries), and $t' = 4$ (number of probes by inverse queries), where the cross on the intersection of the row labeled by q and the column labeled by d means that the query q probes the location d.

query	1	2	3	4	5	6	7	8	9	10	11	12	13
$\pi(1)$	×				×				×				
$\pi(2)$		×		×				×					×
$\pi(3)$	×				×				×				
$\pi(4)$				×	×	×							
$\pi(5)$				×					×		×		
$\pi(6)$	×			×		×							
$\pi(7)$	×					×						×	
$\pi^{-1}(1)$		×		×	×					×			
$\pi^{-1}(2)$	×				×				×		×		
$\pi^{-1}(3)$		×			×					×			×
$\pi^{-1}(4)$			×				×			×			×
$\pi^{-1}(5)$		×				×		×					×
$\pi^{-1}(6)$		×				×				×			×
$\pi^{-1}(7)$		×			×				×	×			
$P(d_1)$	×	×		×	del	×			×	×	×	×	

Locations that are used by at least $\beta = \left\lceil \frac{7 \cdot 3}{13/2} \right\rceil = 4$ forward queries or by at least $\beta' = \left\lceil \frac{7 \cdot 4}{13/2} \right\rceil = 5$ inverse queries are discarded (vertical lines).
Let $d_1 = 5$.
The following table describes various sets:

set	consists of queries
$F(d_1)$	$\pi(1), \pi(3), \pi(4)$
$I(d_1)$	$\pi^{-1}(1), \pi^{-1}(2), \pi^{-1}(3), \pi^{-1}(7)$
$(F(d_1))^{-1}$	$\pi^{-1}(7), \pi^{-1}(2), \pi^{-1}(1)$
$(I(d_1))^{-1}$	$\pi(4), \pi(3), \pi(7), \pi(1)$
$F'(d_1) = F(d_1) \cap (I(d_1))^{-1}$	$\pi(1), \pi(3), \pi(4)$
$I'(d_1) = I(d_1) \cap (F(d_1))^{-1}$	$\pi^{-1}(1), \pi^{-1}(2), \pi^{-1}(7)$
$Q(d_1) = (F(d_1))^{-1} \cup (I(d_1))^{-1}$	$\pi(1), \pi(3), \pi(4), \pi(7), \pi^{-1}(1), \pi^{-1}(2), \pi^{-1}(7)$

μ_{d_1} is a mapping between $F'(d_1)$ and $I'(d_1)$, and is as follows

query	$\pi(1)$	$\pi(3)$	$\pi(4)$
μ_{d_1}(query)	$\pi^{-1}(7)$	$\pi^{-1}(2)$	$\pi^{-1}(1)$

or if we enumerate the queries in the lexicographical order (increasing order of paremeter), μ_{d_1} is

lexicographical order of query	1	2	3
μ_{d_1}(lexicographical order of query)	3	2	1

The set of protected cells, $P(d_1)$, is the union of cells that are used by queries in $Q(d_1)$ except for d_1, $P(d_1) = \{1, 9\} \cup \{1, 9\} \cup \{4, 6\} \cup \{1, 6, 12\} \cup \{2, 4, 10\} \cup \{1, 9, 11\} \cup \{2, 9, 10\} = \{1, 2, 4, 6, 9, 10, 11, 12\}$

Figure 5.1: Deleting a cell

The queries $\texttt{access}(ip+1)$ and $\texttt{search}(X, j)$ are reciprocal if they are "responsible" for the substring of T from position $ip+1$ to position $(i+1)p$.

Finally, in the case of the binary relation problem, we define

$$\begin{aligned}
\mathcal{F}_R &= \{\text{"query } \texttt{row_sel}(i,x)\text{"} \mid 1 \le i \le m, 1 \le x \le \texttt{row_nb}(i)\} \\
\mathcal{I}_R &= \{\text{"query } \texttt{col_sel}(x',j)\text{"} \mid 1 \le j \le n, 1 \le x' \le \texttt{col_nb}(i)\} \\
\eta_R(\text{"query } \texttt{row_sel}(i,x)\text{"}) &= \text{"query } \texttt{col_sel}(x',j)\text{"}, \\
&\qquad \text{if } \texttt{row_sel}(i,x) = j \text{ and } \texttt{col_sel}(x',j) = i.
\end{aligned}$$

The queries $\texttt{row\_sel}(i,x)$ and $\texttt{col\_sel}(x',j)$ are reciprocal if they select the same 1-bit in R. It turns out that for the proof of the main lemma, this correspondence encapsulates the important properties of the problems that we are interested in: the PERMS problem, the TEXTSEARCH problem, and the binary relation problem.

Assume that we have encodings of the sets $\mathcal{F}_B$ and $\mathcal{I}_B$. An encoding of $\mathcal{F}_B$ (respectively, $\mathcal{I}_B$) is a description of the parameters of every query from $\mathcal{F}_B$ (respectively, $\mathcal{I}_B$). Also assume that we have answers for queries $\mathcal{F}_B^* \subseteq \mathcal{F}_B$ and $\mathcal{I}_B^* \subseteq \mathcal{I}_B$. And finally assume that for each query $q \notin \mathcal{F}_B^*$ such that $\eta(q) \notin \mathcal{I}_B^*$, we know its reciprocal $q' = \eta(q)$. If we can reconstruct the object B using only this information, then we say that the problem possesses the *reciprocal property*. More formally,

Definition 4. *Consider a problem of representing objects $\mathcal{H}$. We say that the problem possesses the* reciprocal property *if for every object $B \in \mathcal{H}$, we can find subsets $\mathcal{F}_B \subseteq \mathcal{F}$, $\mathcal{I}_B \subseteq CI$, and a bijection η_B between them, such that for any subsets $\mathcal{F}_B^* \subseteq \mathcal{F}_B$ and $\mathcal{I}_B^* \subseteq \mathcal{I}_B$, the object B is uniquely identified by*

- *encodings of $\mathcal{F}_B$ and $\mathcal{I}_B$,*
- *encodings of $\mathcal{F}_B^*$ and $\mathcal{I}_B^*$,*
- *the answers to all the queries $\mathcal{F}_B^*$ and $\mathcal{I}_B^*$, and*
- *an encoding of the bijection η_B restricted to the set $\mathcal{F}_B \setminus \mathcal{F}_B^* \setminus \eta^{-1}(\mathcal{I}_B^*)$ (which maps to the set $\mathcal{I}_B \setminus \mathcal{I}_B^* \setminus \eta(\mathcal{F}_B^*)$).*

Notice that in our three examples, if we know the fact that queries q and q' are reciprocal (i.e. $q = \eta_B(q')$) then we actually know the answers to both q and q' as well. Namely, for the PERMS problem, if q = "query $\texttt{forw\_perm}_\pi(i)$" and q' = "query $\texttt{inv\_perm}_\pi(j)$" then $\texttt{forw\_perm}_\pi(i) = j$ and $\texttt{inv\_perm}_\pi(j) = i$. For the TEXTSEARCH problem, if q = "query $\texttt{access}_T(ip+1)$" and q' = "query $\texttt{search}_T(X, j)$", then $\texttt{access}_T(ip+1) = X$ and $\texttt{search}_T(X, j) = ip + 1$. For the binary relations problem, if q = "query $\texttt{row\_sel}_R(i, x)$" and q' = "query $\texttt{col\_sel}_R(x', j)$", then $\texttt{row\_sel}_R(i, x) = j$ and $\texttt{col\_sel}_R(x', j) = i$. Thus, for our three problems, we can assume that for every reciprocal pair (q, q'), we either know the answer to q or to q' (or to both). In particular, for the PERMS problem, we either know the value of $\texttt{forw\_perm}_\pi(i)$ or we know some j, such that $\texttt{inv\_perm}_\pi(j) = i$, and thus π can be reconstructed. In the TEXTSEARCH problem, we either know the value of $\texttt{access}(ip+1)$ or we know that there is some query q' = "query $\texttt{search}(X, j)$" such that the answer to q is $ip+1$; in both cases we can reconstruct the substring of T from position $ip+1$ to position $(i+1)p$. For the binary relation problem, we start with a matrix R initialized with 0 entries. For each query q that we know an answer to, we write a 1-bit to the corresponding entry in R, e.g. if the answer to the query $\texttt{row\_sel}(i, x)$ is j, then a 1-bit is written to location (i, j) of R. Since for every pair of reciprocal queries we know the answer to at least one, all the f of 1-bits will be written to R, and therefore R is reconstructed correctly. We thus have the following theorem.

Theorem 18. *The* PERMS *problem, the* TEXTSEARCH *problem, and the binary relation problem possess the reciprocal property.*

Definition 5. *We call a reciprocal problem of representing objects $\mathcal{H}$ a* $(\Upsilon, r, t, t', \gamma, w)$-problem*, if*

- *we use the cell probe model with cell size w,*
- *the worst case storage size is $\Upsilon + r$ cells, where $\Upsilon = \left\lceil \frac{\log |\mathcal{H}|}{w} \right\rceil$,*
- *$r \geq 0$,*
- *the forward queries are implemented using algorithm A with worst case cell probe complexity t*

- *the inverse queries are implemented using algorithm A' with worst case cell probe complexity t', and*
- *the parameter γ is such that $\gamma \geq |\mathcal{F}_B| = |\mathcal{I}_B|$ for any object $B \in \mathcal{H}$.*

5.2 Compression Lemma

In this section, we prove the main technical lemma that will be used to prove the results in this chapter.

Lemma 14. *Consider a $(\Upsilon, r, t, t', \gamma, w)$-problem with $r = O(\Upsilon)$. Fix a positive constant ε, $0 < \varepsilon < 1$. If the sets $\mathcal{F}_B$ and $\mathcal{I}_B$ can be encoded using at most*

$$\frac{\varepsilon}{64}\frac{\Upsilon^2 w}{\gamma t t'}$$

bits for any $B \in \mathcal{H}$, and if

$$\lg \max\{t, t'\} < (1-\varepsilon)w, \text{ and} \tag{5.1}$$

$$\min\{t, t'\} < \frac{\varepsilon}{16}\frac{\Upsilon w}{\gamma \lg w} \tag{5.2}$$

then

$$r \geq \frac{\varepsilon}{32}\frac{\Upsilon^2}{\gamma t t'} - O(1). \tag{5.3}$$

Proof. The outline of the proof is as follows: fix a representation `Rep` of the set of all objects $\mathcal{H}$. `Rep` is an injective mapping from the set of all possible objects $\mathcal{H}$ to the set of all possible contents of the memory section S consisting of $\Upsilon + r$ cells. Fix the algorithms A and A' that implement the queries $\mathcal{F}$ and $\mathcal{I}$, respectively, with cell probe complexities of t and t', respectively. In a fashion similar to the proofs of Theorem 15 and Theorem 17, we consider the set $\mathcal{H}^*$ of incompressible objects in the sense of Kolmogorov's [37]. For any fixed object $B \in \mathcal{H}^*$, any of its representations must use at least $\Upsilon - O(1)$ cells of memory.

We first outline a technique for compressing storage S for a given $B \in \mathcal{H}$ using the algorithms A and A'. Fix an object B. We remove some cells from the encoding S of B; to compensate for the lost cells, we add enough information so that B can be recovered. The

recovery procedure does not have to be efficient. The procedure that removes cells from S is iterative. It performs several steps: on the k-th step, it removes the cell d_k (which is called *deleted*) and then it protects some other cells $P(d_k)$ (which are called *protected*) so they cannot be deleted during the later steps. The cells that are not deleted or protected after a given step are called *remaining* and denoted by $\mathcal{C}_k$. We perform the deletion procedure as long as the number of remaining cells $|\mathcal{C}_k|$ is at least $\Upsilon/2$. Let z denote the total number of deletion steps.

At each step, we record some information $\mathcal{M}$ that depends on the deleted cell that will be used later in the decoding phase. The amount of this new information is less than one cell (i.e. w bits), so that, informally, we "save" some space at each step. After the last iteration of this procedure, we record the positions, $\mathcal{D}$, of all the cells that were deleted. Also, we record the contents, $\mathcal{R}$, of all the cells that were not deleted from left to right. Finally, we will describe the decoding procedure, account for the space used to encode $\mathcal{M}, \mathcal{D}$ and $\mathcal{R}$, and conclude with the statement of the lemma as a consequence of this compression scheme.

Before continuing with the details of the proof, we introduce some notation. For a set of queries Y, let $Y^{-1} = \eta_B(Y)$ denote the set of queries reciprocal to Y. In particular, for a query q, we denote by $(q)^{-1} = \eta_B(q)$ the query that is reciprocal to q. We say that a cell at location l is *used* by the query q (respectively, q') if A (respectively, A') probes that location.

Let $\mathcal{C}_k$ be the set of indexes of the remaining cells at the k-th step; let $\mathcal{F}_k$ (respectively, $\mathcal{I}_k$) be the sets of available forward (respectively, inverse) queries at the k-th step, that is, those queries that were not considered at the previous steps. At the first step, $\mathcal{C}_1 = \{1, 2, \ldots, \Upsilon + r\}$, $\mathcal{F}_1 = \mathcal{F}$, and $\mathcal{I}_1 = \mathcal{I}$. We maintain the invariant that $|\mathcal{C}_k| \geq \Upsilon/2$ throughout the loop.

For each k, $1 \leq k \leq z$, we describe the procedure that removes a cell. Let $R(q)$ be the set of remaining cells used by $q \in \mathcal{F}_k \cup \mathcal{I}_k$; and $F(l)$ (respectively, $I(l)$) be the set of queries $q \in \mathcal{F}_k$ (respectively, $q \in \mathcal{I}_k$) that use a given cell $l \in \mathcal{C}_k$. Then,

$$\sum_{l \in \mathcal{C}_k} |F(l)| = \sum_{q \in \mathcal{F}_k} |R(q)| \leq t|\mathcal{F}_k| \leq t\gamma, \text{ and} \tag{5.4}$$

$$\sum_{l \in \mathcal{C}_k} |I(l)| = \sum_{q \in \mathcal{I}_k} |R(q)| \leq t'|\mathcal{I}_k| \leq t'\gamma. \tag{5.5}$$

Therefore, there can be at most $|\mathcal{C}_k|/2$ cells l for which $|F(l)| \geq t\gamma/(|\mathcal{C}_k|/2)$, otherwise $\sum F(l) > (|\mathcal{C}_k|/2)t\gamma/(|\mathcal{C}_k|/2) = t\gamma$, similarly there are at most $|\mathcal{C}_k|/2$ cells l for which $|I(l)| \geq t'\gamma/(|\mathcal{C}_k|/2)$. By the loop invariant, $|\mathcal{C}_k| > \Upsilon/2$, so we can find a remaining cell d_k that is used by at most $\beta = t\gamma/(|\mathcal{C}_k|/2) \leq 4t\gamma/\Upsilon$ forward and $\beta' = t'\gamma/(|\mathcal{C}_k|/2) \leq 4t'\gamma/\Upsilon$ inverse queries, that is, $|F(d_k)| \leq \beta$, and $|I(d_k)| \leq \beta'$. The value β (respectively, β') is called the *forward coverage factor* (respectively, the *inverse coverage factor*).

The cell d_k is deleted on the k-th step. Let $F'(d_k) = F(d_k) \cap (I(d_k))^{-1}$ (respectively, $I'(d_k) = I(d_k) \cap (F(d_k))^{-1}$) denote the set of forward (respectively, inverse) queries q that use the cell d_k and for which the reciprocal q^{-1} also uses cell d_k. Let $\beta^* = \min\{\beta, \beta'\}$. Note that $F'(d_k) = (I'(d_k))^{-1}$, and let

$$g_{d_k} = |F'(d_k)| = |I'(d_k)| \leq \beta^* = \frac{4\gamma \min\{t, t'\}}{\Upsilon}$$

Note that the η_B defines a one-to-one map between $F'(d_k)$ and $I'(d_k)$. We enumerate the elements of $F'(d_k)$ and $I'(d_k)$ in lexicographic order using labels from $[g_{d_k}]$ (the j-th element of $F'(d_k)$ corresponds to the label j), and encode this map as a permutation μ_{d_k} on the set $[g_{d_k}]$. For an example, see Figure 5.1.

Let $Q(d_k) = (F(d_k) \cup I(d_k))^{-1}$ be the set of all reciprocal queries to the queries that use d_k. We protect the following set of cells:

$$P(d_k) = \bigcup_{q \in Q(d_k)} R(q) \setminus \{d_k\}$$

so they cannot be deleted in the forthcoming steps. The number of protected cells $|P(d_k)|$ is at most

$$|P(d_k)| \leq t|I(d_k)| + t'|F(d_k)| \leq t\beta' + t'\beta \leq \frac{8\gamma t t'}{\Upsilon},$$

since $Q(d_k)$ contains $|I(d_k)|$ forward and $|F(d_k)|$ inverse queries. The sets of remaining cells, and available forward and inverse queries on the next step $k+1$ are as follows:

$$\begin{aligned}
\mathcal{C}_{k+1} &= \mathcal{C}_k \setminus (P(d_k) \cup \{d_k\}) \\
\mathcal{F}_{k+1} &= \mathcal{F}_k \setminus (F(d_k) \cup (I(d_k))^{-1}) \\
\mathcal{I}_{k+1} &= \mathcal{I}_k \setminus (I(d_k) \cup (F(d_k))^{-1})
\end{aligned}$$

Hence,

$$\mathcal{C}_{k+1} \geq |\mathcal{C}_0| - \sum_{i=1}^{k} |P(d_i)| - k \geq \Upsilon + r - kt'\beta - kt\beta' \geq \Upsilon - 2kt'\beta,$$

since $t'\beta = t\beta'$. To maintain our loop invariant (i.e. $|\mathcal{C}_z| > \Upsilon/2$) it suffices to restrict the number of iterations z such that $2zt'\beta < \Upsilon/2$, or in other words, we can perform at least

$$z = \frac{\Upsilon}{4t'\beta}$$

steps. After z iterations, the encoding procedure stores the following information:

$\mathcal{D}$ - the locations of all the deleted cells $\mathcal{D} = \cup_k \{d_k\}$ using at most

$$\lg \binom{\Upsilon + r}{z} = \lg z \binom{O(\Upsilon)}{z} = z \lg(O(\Upsilon)/z) = z \lg(t'\beta) + O(z)$$

bits.

$\mathcal{R}$ - the contents of all the cells (in left to right order) that are not deleted using $\Upsilon + r - z$ cells;

$\mathcal{M}$ - the permutations μ_{d_k} for each deleted cell d_k (in left to right order) using $z \lg(\beta^*!) \leq z\beta^* \lg \beta^* - \Theta(z\beta^*)$ bits.

The decoding procedure is as follows. It starts with an uninitialized array S of size $\Upsilon + r$. Next, it writes the corresponding values from $\mathcal{R}$ (in left to right order) to all the locations that are not encoded in $\mathcal{D}$. Then, it simulates all the queries from $\mathcal{F}_B$ and $\mathcal{I}_B$. Note that some of the queries will *fail* due to the fact that the contents of z cells are missing. We call a query q *recoverable* if it fails, but its reciprocal query q^{-1} does not fail. Such queries fall into the set $\mathcal{F}_B^*$ or $\mathcal{I}_B^*$ in Definition 4, and we do not have to worry about them anymore.

For each query q that is not recoverable, we find the first location l where it fails, that is, l is the first deleted cell that the query algorithm (A or A') needs to probe in order of its execution on q. By construction, it is only possible that query q is not recoverable if and only if q and its reciprocal q^{-1} fail at the same location. Recall, that once we cause a

query to fail by deleting a cell l, we immediately protect its reciprocal. In the case where a query and its reciprocal fail at the same cell l, we protect all the other cells that are used by both of them, therefore l can only be the first (and the only) deleted cell that is used by either of the queries.

Also, by construction, for the cell l, there are $g_l \leq \beta^*$ non-recoverable queries. For each deleted cell d (in left to right order), we list all the indices F'_d (respectively, I'_d) of non-recoverable forward (respectively, inverse) queries that fail at d in the same order that we used to encode permutation μ_d (e.g. in increasing order), so that μ_d defines a map between F'_l and I'_l. Therefore, for each deleted cell d, we can restore the mapping between the non-recoverable reciprocal pairs of queries that fail at d using the permutations μ_{d_k} encoded in $\mathcal{M}$. Hence, we can construct the mapping η^*_B between all the non-recoverable forward queries and all the non-recoverable inverse queries.

We also define $\mathcal{F}^*_B$ (respectively, $\mathcal{I}^*_B$) to be the set of all forward (respectively, inverse) queries that do not fail. Since by the assumption the problem possesses the reciprocal property, the decoding procedure is complete.

It remains to account for the space our new representation uses. By the assumption, the encodings for $\mathcal{F}_B$ and $\mathcal{I}_B$ occupy at most

$$\frac{\varepsilon}{64}\frac{\Upsilon^2 w}{\gamma t t'} = \frac{\varepsilon}{4} z w,$$

bits, since

$$z = \frac{\Upsilon}{4t'\beta} = \frac{\Upsilon^2}{16\gamma t t'} \tag{5.6}$$

Assume that $t' \geq t$ so that $\beta^* = \beta$, the other case is symmetric. Storages $\mathcal{D}$ and $\mathcal{M}$ and encodings of $\mathcal{F}_B$ and $\mathcal{I}_B$ together occupy at most

$$\begin{aligned} z\lg(t'\beta) + O(z) + z\beta\lg\beta - \Theta(z\beta) + (\varepsilon/4)zw &= z\lg t' + z(\beta + O(1))\lg\beta + O(z\lg\beta) \\ &\leq (1-\varepsilon)zw + (\varepsilon/4)zw + (\varepsilon/4)zw \\ &= (1-\varepsilon/2)zw \end{aligned} \tag{5.7}$$

bits, if we require that

$$\lg t' < (1-\varepsilon)w, \text{ and} \tag{5.8}$$

$$\beta < (\varepsilon/4)w/\lg w - O(1), \tag{5.9}$$

since in this case

$$(\beta + O(1)) \lg \beta < (\varepsilon/4)(w/\lg w) \lg w = (\varepsilon/4)w.$$

Substituting $\beta = (4\gamma t)/\Upsilon$ into (5.9) we obtain the sufficient condition

$$\frac{4\gamma t}{\Upsilon} < \frac{\varepsilon}{4} \frac{w}{\lg w},$$

which follows from condition (5.2). Condition (5.8) is equivalent to condition (5.1). Hence, the total size of our encoding is bounded by $\Upsilon + r - z + (1 - \varepsilon/2)z$ cells, which must be at least $\Upsilon - O(1)$ for incompressible objects B, and thus

$$r \geq (\varepsilon/2)z - O(1) = \frac{\varepsilon \Upsilon^2}{32\gamma t t'} - O(1) \tag{5.10}$$

using (5.6). The theorem follows. □

5.3 Applications of the Compression Lemma

For the three problems that we considered earlier: the PERMS problem, the TEXTSEARCH problem, and for the binary relations problem, in this section, we show the lower bound trade-offs between r, t and t' as the consequences of the compression lemma.

5.3.1 The PERMS Problem

We start with the simplest of the three problems. In the PERMS problem, we have

$$\begin{aligned} \mathcal{F}_\pi &= \{ \text{“query } \texttt{forw\_perm}_\pi(i)\text{”} | i \in [n] \} \\ \mathcal{I}_\pi &= \{ \text{“query } \texttt{inv\_perm}_\pi(i)\text{”} | i \in [n] \}. \end{aligned}$$

Therefore, we do not have to encode these sets, so that we can use Lemma 14 to derive the following theorem.

Theorem 19. *For the* PERMS *problem, if*

$$\lg \max\{t, t'\} < (1 - \varepsilon)w, \text{ and} \tag{5.11}$$

$$\min\{t, t'\} < \frac{\varepsilon}{16} \frac{\lg n - \Theta(1)}{\lg w} = O\left(\frac{\lg n}{\lg w}\right) \tag{5.12}$$

then

$$rtt' \geq \frac{\varepsilon}{32}\frac{n(\lg n)^2 - O(n \lg n)}{w^2} = \Omega\left(n\left(\frac{\lg n}{w}\right)^2\right), \tag{5.13}$$

where ε is a constant, $0 < \varepsilon < 1$.

Proof. We have

$$\begin{aligned} \Upsilon &= \left\lceil \frac{\lg n!}{w} \right\rceil = \frac{n \lg n - \Theta(n)}{w} \\ \gamma &= n \end{aligned}$$

Thus, inequalities (5.2) and (5.13) transform into

$$\begin{aligned} \min\{t, t'\} &< \frac{\varepsilon}{16}\frac{\lg n - \Theta(1)}{\lg w} \\ rtt' &\geq \frac{\varepsilon}{32}\frac{n(\lg n)^2 - O(n \lg n)}{w^2} \end{aligned}$$

□

In the interesting case where the cell size is $w = \lg n$, we have

Corollary 3. *If $\max\{t, t'\} < n^{1-\varepsilon}$, and $\min\{t, t'\} < (\varepsilon/16) \lg n / \lg \lg n$, then $rtt' \geq (\varepsilon/32)n = \Omega(n)$.*

In particular, if we would like to find an algorithm that performs queries π and π^{-1} in constant time, then we need linear extra space. Also, if we require that our algorithm implements *either* π or π^{-1} in constant time, then we obtain a linear lower bound that matches (up to a constant factor) the upper bound from Munro et al. [44] that uses "back pointers". If we require that our algorithm implements both π and π^{-1} in $\Theta(\lg n / \lg \lg n)$ time, then we get a lower bound that matches (up to a constant factor) the upper bound from Munro et al. [44] that uses Benes networks. Namely, in [44, Theorem 7], they use $O(n(\lg \lg n)^2/(\lg n))$ extra redundancy bits and cell size $w = \lg n$, while our lower bound requires at least $\Omega(n(\lg \lg n)^2/(\lg n)^2)$ extra cells.

5.3.2 The TEXTSEARCH Problem

Recall that in the TEXTSEARCH problem, we are given the text T of length L on an alphabet Σ of size σ. We are required to preprocess and store it such that we can efficiently search for the j-th occurrence of a given pattern X of length p in the text. For the purposes of proving a lower bound on the size of the storage, we only restrict ourselves to the queries of the form

$$\begin{aligned} \mathcal{F}_T &= \{\text{“query } \texttt{access}(ip+1)\text{”} | 0 \le i < L/p\} \\ \mathcal{I}_T &= \{\text{“query } \texttt{search}(X,j)\text{”} | \texttt{search}_T(X,j) = ip+1\} \end{aligned}$$

While the encoding of forward queries is trivial, the encoding of inverse queries is bit involved. For all possible patterns X of length p on the alphabet Σ, we denote J_X to be the bit vector, such that $J_X[j] = 1$ if $\texttt{search}_T(X,j) \equiv 1 \bmod p$, and $J_X[j] = 0$ otherwise. So that what we need to encode is the set

$$\mathcal{J} = \{(X, J_X) | \text{ all possible patterns } X \text{ of length } p\}$$

Let us order possible patterns X in lexicographic order, trim the trailing zeroes in J_X, and concatenate the trimmed bit vectors. The resulting bit vector J has exactly L/p 1-bits in it, since there are L/p positions of the form $i \equiv 1 \bmod p$. The length of J is $L - p + 1$, since there are total $L - p + 1$ positions that can be answers to the search queries. Thus, we can encode this bit vector using $\lg \binom{L}{L/p}$ bits. We also need to encode the sequence of cardinalities C_X (i.e. the number of 1-bits) of J_X for all possible patterns X. It is a sequence of σ^p numbers the sum of which is L/p, and hence can be encoded using $\lg \binom{L/p+\sigma^p}{L/p}$ bits. The original bit vectors J_X can be restored by reading J from left to right and cutting it in a greedy fashion so that the resulting bit vectors have the cardinalities from the sequence of C_X. Thus, the encoding of $\mathcal{J}$ is at most

$$\lg \binom{L}{L/p} + \lg \binom{L/p+\sigma^p}{L/p} \le 2 \lg \binom{L+L/p}{L/p} \le (2L/p) \lg(2ep), \tag{5.14}$$

We assumed that the total possible number of patterns X is not too large, that is

$$\sigma^p \le L$$

Also, we used the fact that $\lg \binom{x}{y} \leq y \lg(ex/y)$. Now we are ready to apply Lemma 14 with parameters

$$\begin{aligned} \Upsilon &= \left\lceil \frac{L \lg \sigma}{w} \right\rceil \\ \gamma &= \frac{L}{p} \end{aligned}$$

The inequalities (5.2) and (5.3) transform into

$$\begin{aligned} \min\{t, t'\} &< \frac{\varepsilon}{16} \frac{p \lg \sigma}{\lg w} \\ rtt' &\geq \frac{\varepsilon}{32} Lp \left(\frac{\lg \sigma}{w} \right)^2 \end{aligned}$$

By the precondition of the Lemma 14, the size of the encoding of the set $\mathcal{I}_T$ should be at most

$$\frac{\varepsilon}{64} \frac{\Upsilon^2 w}{\gamma tt'}.$$

The sufficient condition is that

$$\frac{2L \lg(2ep)}{p} \leq \frac{\varepsilon}{64} \frac{Lp(\lg \sigma)^2}{wtt'},$$

which is satisfied if

$$tt' < \frac{\varepsilon}{128} \frac{(p \lg \sigma)^2}{w \lg(2ep)}. \tag{5.15}$$

Combining these conditions and condition (5.1), we obtain the following theorem.

Theorem 20. *For the* TEXTSEARCH *problem on texts of length L, alphabets of size σ, and patterns of length p, if the following conditions are satisfied*

$$p \lg \sigma \leq \lg L, \tag{5.16}$$

$$\lg \max\{t, t'\} < (1 - \varepsilon) w, \tag{5.17}$$

$$\min\{t, t'\} < \frac{\varepsilon}{16} \frac{p \lg \sigma}{\lg w}, \text{ and} \tag{5.18}$$

$$tt' < \frac{\varepsilon}{128} \frac{(p \lg \sigma)^2}{w \lg(2ep)} \tag{5.19}$$

then

$$rtt' \geq \frac{\varepsilon}{32}\frac{Lp(\lg\sigma)^2}{w^2}$$

The conditions in Theorem 20 look somewhat complicated; however we can simplify them in the case where $w > (2\lg\lg L)/(1-\varepsilon)$, which not too restrictive since the typical choice for w is $\lg L$.

Corollary 4. *Let ε be a constant, $0 < \varepsilon < 1$. For the* TEXTSEARCH *problem, if*

$$p\lg\sigma \leq \lg L, \tag{5.20}$$

$$tt' < \frac{\varepsilon}{128}\frac{(p\lg\sigma)^2}{w\lg(2ep)}, \text{ and} \tag{5.21}$$

$$w > \frac{2}{1-\varepsilon}\lg\lg L, \tag{5.22}$$

then

$$rtt' \geq \frac{\varepsilon}{32}\frac{Lp(\lg\sigma)^2}{w^2}.$$

Proof. Condition (5.21) implies that $\max\{t,t'\} < tt' < \lg((p\lg\sigma)^2)$. Thus, using (5.20) and (5.22), we obtain precondition (5.17) of Theorem 20:

$$\lg\max\{t,t'\} < 2\lg\lg L < (1-\varepsilon)w.$$

Condition (5.21) also implies that

$$\min\{t,t'\} < \sqrt{\frac{\varepsilon}{128\lg(2ep)}}\frac{p\lg\sigma}{\sqrt{w}},$$

which is stronger than precondition (5.18) of Theorem 20, since $w = \Omega(\lg\lg L) = \omega(1)$. □

For the interesting special case of patterns of length $p = \lg L/\lg\sigma$ and word size $w = \lg L$, we obtain

Corollary 5. *Let ε be a constant, $0 < \varepsilon < 1$. For the* TEXTSEARCH *problem, if*

$$p = \lg L/\lg\sigma$$

$$w = \lg L, \text{ and}$$

$$tt' < \frac{\varepsilon}{128}\frac{\lg L}{\lg(2e\lg L)}, \tag{5.23}$$

then

$$rtt' \geq \frac{\varepsilon}{32}\frac{L\lg\sigma}{\lg L} = \Omega(\Upsilon).$$

Proof. Preconditions (5.20) and (5.22) of Corollary 4 are clearly satisfied. Precondition (5.21) transforms into

$$tt' \;<\; \frac{\varepsilon}{128}\frac{(p\lg\sigma)^2}{w\lg(2ep)} = \frac{\varepsilon}{128}\frac{(\lg L)^2}{\lg L(\lg(2e\lg L) - \lg\lg\sigma)},$$

which follows from (5.23) for $\sigma \geq 2$. □

It follows that we need at least $\Omega(\Upsilon)$ cells (i.e., linear in the information-theoretic minimum) of extra space to support both `access` and `search` queries in constant time for patterns of size $p = \lg L/\lg\sigma$ in the cell probe model with word size $\lg L$.

5.3.3 The `str_acc`/`str_sel` Problem

Consider the important special case of $p = 1$. In this case, the `access` operation corresponds to `str_acc`, and the `search` corresponds to `str_sel`. We can improve our bound by encoding $\mathcal{J}$ more compactly. Namely, the encoding in (5.14) can be bounded by $\lg\binom{L+\sigma}{\sigma} \leq \sigma\lg L$ bits in the case $p = 1$. In other words, it suffices to encode the number of occurrences $\mathtt{occ}_T(c)$ of each character $c \in \Sigma$ to be able to obtain $\mathcal{J}$, since $\mathtt{search}_T(c,j) = \mathtt{str\_sel}_T(c,j)$ is in the set $\mathcal{I}_T$ if and only if $j \leq \mathtt{occ}_T(c)$. As before, we require that the size of this encoding is at most

$$\frac{\varepsilon}{64}\frac{\Upsilon^2 w}{\gamma tt'}$$

bits, that is

$$\sigma\lg L \leq \frac{\varepsilon}{64}\frac{L(\lg\sigma)^2}{wtt'},$$

which is satisfied for

$$tt' \leq \frac{\varepsilon}{64}\frac{L(\lg\sigma)^2}{w\sigma\lg L}$$

Thus, as a corollary of Theorem 20, we obtain

Corollary 6. *Consider the* TEXTSEARCH *problem with pattern length* $p = 1$*. If*

$$\lg \max\{t, t'\} < (1-\varepsilon)w, \tag{5.24}$$

$$\min\{t, t'\} < \frac{\varepsilon}{16}\frac{\lg \sigma}{\lg w}, \text{ and} \tag{5.25}$$

$$tt' < \frac{\varepsilon}{64}\frac{L(\lg \sigma)^2}{w\sigma \lg L}, \tag{5.26}$$

then

$$rtt' \geq \frac{\varepsilon}{32}\frac{L(\lg \sigma)^2}{w^2}$$

Proof. Precondition (5.16) is clearly satisfied. The condition (5.15) that is needed to satisfy the precondition of Lemma 14 regarding the encoding of $\mathcal{F}_T$ and $\mathcal{I}_T$ reduces to (5.26). The other preconditions have not changed as compared to Theorem 20. □

The condition (5.26) is quite weak as compared to conditions (5.24) and (5.25), since L is typically much larger than the product $w\sigma$. Namely, in the case where $w = \lg L$, and $\sigma < L^\alpha$ for some constant $0 < \alpha < \varepsilon$, from (5.24) and (5.25) we can show that

$$tt' = \max\{t, t'\} \min\{t, t'\} < L^{1-\varepsilon}\frac{\varepsilon\alpha}{16}\frac{\lg L}{\lg \lg L}.$$

The right hand side of (5.26) is at least

$$\frac{\varepsilon}{64}\frac{L(\lg \sigma)^2}{w\sigma \lg L} > \frac{\varepsilon}{64}\frac{L^{1-\alpha}}{(\lg L)^2} > \frac{\varepsilon\alpha}{16}\frac{L^{1-\varepsilon}\lg L}{\lg \lg L},$$

since $\alpha < \varepsilon$ and the logarithmic in L factors are smaller than $L^{\varepsilon-\alpha}$. We rephrase Corollary 6 in terms of the `str_acc`/`str_sel` problem and obtain the following theorem.

Theorem 21. *Consider the* `str_acc`/`str_sel` *problem with* $w = \lg L$*, and alphabet of size* σ *such that*

$$\frac{16 \lg \lg L}{\varepsilon} \leq \lg \sigma \leq \alpha \lg L$$

for some constants ε *and* α*, such that* $0 < \alpha < \varepsilon < 1$*. If*

$$\max\{t, t'\} < L^{1-\varepsilon}, \text{ and} \tag{5.27}$$

$$\min\{t, t'\} < \frac{\varepsilon}{16}\frac{\lg \sigma}{\lg \lg L}, \tag{5.28}$$

then

$$rtt' \geq \frac{\varepsilon}{32} \frac{L(\lg \sigma)^2}{(\lg L)^2}.$$

For the case of $\sigma = L^\alpha$, $t = 1$, and $t' = \lg\lg\sigma / \lg\lg\lg\sigma$ the lower bound matches the upper bound shown in Theorem 4 (Chapter 2) up to constant factors.

5.3.4 The BINREL Problem

In this section, we apply the general Lemma 14 to the problem of representing binary matrices. Namely, we are to represent $m \times n$ matrices R of a given cardinality f, so that the `row_sel` and the `col_sel` queries can be implemented efficiently. We assume that $m \leq n$ without loss of generality (since we can interchange the roles of the rows and columns). Recall that the sets of forward and inverse queries are as follows

$$\begin{aligned} \mathcal{F}_R &= \{\text{“query } \texttt{row\_sel}(i,x)\text{”} | 1 \leq i \leq m, 1 \leq x \leq \texttt{row\_nb}(i)\} \\ \mathcal{I}_R &= \{\text{“query } \texttt{col\_sel}(x',j)\text{”} | 1 \leq j \leq n, 1 \leq x' \leq \texttt{col\_nb}(i)\} \end{aligned}$$

so that $|\mathcal{F}| = |\mathcal{I}| = f$. Also recall that the queries $q =$ “query $\texttt{row\_sel}(i,x)$” and $q' =$ “query $\texttt{col\_sel}(x',j)$” are reciprocal if $\texttt{row\_sel}(i,x) = j$ and $\texttt{col\_sel}(x',j) = i$, i.e. if they are selecting the same 1-bit entry in R. We showed that this problem possess the reciprocal property, so that we can apply Lemma 14 with parameters

$$\begin{aligned} \gamma &= f, \text{ and} \\ \Upsilon &= \left\lceil \frac{\lg \binom{nm}{f}}{w} \right\rceil > \frac{f}{w} \lg \frac{nm}{f} \end{aligned}$$

To encode the sets $\mathcal{F}_R$ and $\mathcal{I}_R$ we can store the values of $\texttt{row\_nb}(i)$ and $\texttt{col\_nb}(j)$ for each row i and column j. These values of $\texttt{row\_nb}(i)$ form a sequence of m numbers, the sum of which is f, and hence can be stored using $\lg \binom{f+n}{n}$ bits. In a similar fashion, we can store the sequence $\texttt{col\_nb}(j)$. So the size of encoding of $\mathcal{F}_R$ and $\mathcal{I}_R$ is at most

$$\lg \binom{f+n}{n} + \lg \binom{f+m}{m} \leq 2 \lg \binom{f+n}{n} < 2n \lg \frac{ef}{n}$$

bits by the assumption that $m \leq n$. The precondition of Lemma 14 requires that this encoding occupies at most

$$\frac{\varepsilon}{64}\frac{\Upsilon^2 w}{\gamma t t'} > \frac{\varepsilon}{64}\frac{f}{wtt'}\left(\lg\frac{nm}{f}\right)^2$$

bits, so it suffices to require

$$tt' < \frac{\varepsilon}{128}\frac{f(\lg(nm/f))^2}{wn\lg(ef/n)}$$

Combining these conditions, we derive the following theorem.

Theorem 22. *Consider the binary relation problem on $m \times n$ binary matrices R of cardinality f. If parameters t, t', n, m, f, and a constant ε, $0 < \varepsilon < 1$ satisfy*

$$\begin{aligned} \lg\max\{t,t'\} &< (1-\varepsilon)w, \\ \min\{t,t'\} &< \frac{\varepsilon}{16}\frac{\lg(nm/f)}{\lg w}, \text{ and} \\ tt' &< \frac{\varepsilon}{128}\frac{f(\lg(nm/f))^2}{wn\lg(ef/n)}, \end{aligned}$$

then

$$rtt' \geq \frac{\varepsilon}{32}\frac{f(\lg(nm/f))^2}{w^2}$$

In particular, consider the case of matrices of size $m \times m^\alpha$ with cardinalities $f = m^\beta$ for constants α and β such that $1 < \alpha < \beta < 1 + \alpha$. Also assume that $w = \lg m$. Hence,

$$\Upsilon = \frac{(1+\alpha-\beta)m^\beta \lg m + O(m^\beta)}{\lg m}.$$

We can derive the following corollary.

Corollary 7. *For the binary relation problem on $m \times m^\alpha$ matrices with $f = m^\beta$, where α and β are constants such that $1 < \alpha < \beta < 1 + \alpha$. Let ε also be a constant, $0 < \varepsilon < 1$. If*

$$\begin{aligned} \max\{t,t'\} &< m^{1-\varepsilon}, \\ \min\{t,t'\} &< \frac{\varepsilon(1+\alpha-\beta)}{16}\frac{\lg m}{\lg\lg m}, \text{ and} \\ tt' &< \frac{\varepsilon(1+\alpha-\beta)}{128(\beta-\alpha)}m^{\beta-\alpha}, \end{aligned}$$

then

$$rtt' \geq \frac{\varepsilon(1+\alpha-\beta)}{32} m^{\beta} = \Omega(\Upsilon)$$

In particular, for the case where the constant time operations are needed, the extra space required to support them is at least linear in Υ, i.e. $\Omega(\Upsilon)$ extra cells.

Chapter 6

Conclusion

The main contributions of this thesis are as follows:

- A new technique for proving cell probe lower bounds (Chapter 5) that can be applied to any problem that possess the reciprocal property. We applied this technique to the following four data structural problems: `PERMS`, `TEXTSEARCH`, `BINREL`, and `str_sel`/`str_acc`. To the best of our knowledge, there were no cell probe lower bounds known for these problems known prior to this work. For the `PERMS` problem, we showed a cell probe lower bound that matches all the existing upper bounds. (The optimality here and later in this chapter is asymptotic, i.e. up to constant factors in both time and space costs.) For the `TEXTSEARCH` problem, we showed a cell probe lower bound that matches existing lower bounds for indexing data structures (in other words, we removed the indexing assumption). For the `BINREL` problem, we showed a cell probe lower bound that matches an upper bound from Chapter 2, and hence both are optimal.

- A new technique for the indexing bit probe lower bounds (Chapter 3) that is based on the notion of choices tree. This technique improves upon the work of Miltersen [41] and provides a uniform framework for the `bin_rank`, `bin_sel` and `PARENTHESES` problems. The bounds that we show in Section 3.2, Section 3.3, and Section 3.7 are tight. We also believe that our density sensitive bounds in Section 3.5 and Section 3.6 are tight as well, however the matching algorithm exploits the freedom of the indexing

model (e.g. accesses the index for free).

- A new compression technique (Chapter 4) that allows to prove stronger lower bounds for the substring report problem. The ideas in Chapter 4 are inspired by the work of Demaine and Lopez-Ortiz [10]. Similar technique were also presented by Gennaro and Trevisan [21]; however, ours provides better compression and, as a consequence, gives stronger bounds.

- A data structure for representing binary matrices (Section 2.2). This data structure allows to represent the matrix in almost the information-theoretic minimum space, and the running times are also better than in the existing data structures. This work is an extension of the papers by Golynski, Munro and Rao [25] that revealed a connection between `str_sel`/`str_acc` and `PERMS` problem, and the paper of Barbay, Golynski, Munro and Rao [2] that generalized the ideas of [25] to the `BINREL` problem.

- An indexing data structure for `bin_rank` and `bin_sel` problems (Section 2.1). It is an extension of the paper by Golysnki [23]. Similar data structures were proposed by Raman, Raman, and Rao [48]. However, our result generalizes them to an arbitrary number of cell probes (even bit probes), and also combines the most space consuming parts of `bin_rank` and `bin_sel` indices into an index that we call the count index. Although we do not have an experimental evidence, we suspect that the part used by our index should be at most half of the size of the index of [48]. Later, Golynski, Grossi, Gupta, Raman, and Rao [24] proposed a non-indexing data structure with much smaller redundancy (approximately a factor of $\lg n$ smaller than in Section 2.1 and in [48]). It is worth noting that such a data structure is not possible in the indexing model as it would contradict the lower bounds from Section 3.2 and Section 3.3.

For the `bin_rank`, `bin_sel` and the `PARENTHESES` problems, the results of this thesis are summarized in Figure 6.1. Namely, we showed the first matching upper and lower bounds for this problem in the indexing bit probe model for arbitrary costs t. We also believe that our density sensitive lower bound are tight, however the corresponding algorithms use the freedom of the indexing model. For the case of the `bin_rank` problem, the bounds

Operation	Upper	Old Lower	New Lower	New Lower Density Sensitive
`bin_rank`	$\frac{n \lg t}{t}$ *	$\frac{n}{t}$, in [41] †	$\frac{n \lg t}{t}$ ‡	$\max\left\{\frac{n}{t}\lg\left(\frac{mt}{n}\right), m, m\lg\left(\frac{n}{mt}\right)\right\}$ §
`bin_sel`		$\frac{n}{t}$, in [41]		
`findmatch`	$\frac{n \lg\lg n}{\lg n}$, in [20]	N/A		N/A

*Presented in Section 2.1.

† The lower bound for the `bin_rank` problem shown by Miltersen [41] in the indexing cell probe model is $r \geq (n \lg(w+1))/(tw)$. It is tight in the indexing cell probe model, however, in the indexing bit probe model, it only amounts to $r = \Omega(n/t)$.

‡Section 3.2, Section 3.3, and Section 3.7.

§Section 3.5, Section 3.6.

Figure 6.1: Asymptotic Bounds for `bin_rank`, `bin_sel` and the `PARENTHESES` Problems.

of Pătraşcu and Thorup [47] imply that there is no succinct data structure unless $m = n/(\lg n)^{O(1)}$. Another variation of this problem was studied in Golynski et al. [24], they proposed new non-indexing data structures that surpass the barrier given by the lower bounds from Chapter 3. They showed that for the `bin_rank` and `bin_sel` problems, there are two data structures that use r bits in addition to the information-theoretic lower bound of $\lg\binom{n}{m}$ and implement the operations in constant time:

- for $m = n/(\lg n)^{O(1)}$, the use $r = O(m(\lg\lg m)^2/\lg m)$ bits; and
- for m smaller than $n/(\lg n)^{O(1)}$, they achieve the redundancy $r = O((n \lg\lg n)/(\lg n)^2)$.

For the `BINREL` problem, Barbay, Golynski, Munro and Rao [2] proposed two encodings called label encoding and object encoding. In this thesis, we offer three new encodings that improve the results of Barbay et al.: row encoding, column encoding and Benes encoding. These five encodings are compared in Figure 6.2. Note that the reduction to rank space lemma 2 allows to consider the `BINREL` problem on matrices with no empty rows and columns, hence the number of 1-bits is $f \geq \max\{m, n\}$, where the m (respectively, n) is the number of non-empty rows (respectively, columns) in a given binary matrix. Therefore, $\rho = nm/f \leq \min\{m, n\} = \xi$. We also note that for the dense matrices, ρ is much smaller than ξ. For example, in the case $f = \varepsilon mn$ for some constant ε, $0 < \varepsilon < 1$, we have constant running times for operations `row_sel` and `col_sel`, which is impossible to achieve

using the representations of Barbay et al. Another advantage of row, column and Benes encodings is that they are truly succinct data structures for $\rho = \omega(1)$, i.e. the redundancy is $o(\Upsilon)$ cells.

On the other hand, the lower bound from Section 5.3.4 shows us that in the case $\lg \rho = \Omega(w)$, the redundancy must be $r = \Omega(\Upsilon/(wtt'))$ cells. For the row encoding, the running time of `row_sel` is $t = O(1)$, the running time of `col_sel` is $t' = O(\lg \lg \rho / \lg \lg \lg \rho)$, and the redundancy is

$$r = O\left(\frac{\Upsilon}{w} \frac{\lg \lg \lg \rho}{\lg \lg \rho}\right)$$

cells; hence, this encoding is optimal under the condition $\lg \rho = \Omega(w)$. In a similar fashion, we can see that the column encoding is also optimal up to constant factors. However, the Benes encoding does not match the lower bound from Section 5.3.4. For the Benes encoding, $t = t' = O((\lg \rho)/(\lg w))$, and Theorem 22 suggests that

$$r = \Omega\left(\frac{f(\lg \rho)^2}{tt'w^2}\right) = \Omega\left(\frac{f(\lg w)^2}{w^2}\right)$$

cells. However, the redundancy of the Benes encoding is $O(f/w)$ cells, i.e. a factor of $w/(\lg w)^2$ larger than one might desire. The use of Benes encoding is justified in the following cases:

- for very high density matrices, i.e. $\lg \rho = O(\lg w)$, it gives constant running times for both `row_sel` and `col_sel` operations; and
- for the applications that perform roughly equal number of `row_sel` and `col_sel` operations. If the `row_sel` (respectively, `col_sel`) operation is used much more frequently, it is suggested to use the row (respectively, column) encoding.

For the `PERMS` problem, Figure 6.3 summarizes and compares our and previous results. For the substring report problem in the indexing bit probe model, and the `TEXTSEARCH` problem in the non-indexing model, Figure 6.5 compares our results with the previous work. Figure 6.5 summarizes the results of Chapter 5.

Name	Row	Column	Benes	Label	Object
`row_rank`	$\lg\lg\rho$*	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$	$\lg\lg\xi$†	$\lg\lg\xi\lg\lg\lg\xi$
`row_sel`	1	$\frac{\lg\lg\rho}{\lg\lg\lg\rho}$	$\frac{\lg\rho}{\lg w}$	1	$\lg\lg\xi$
`col_rank`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$	$\lg\lg\xi\lg\lg\lg\xi$	$\lg\lg\xi$
`col_sel`	$\frac{\lg\lg\rho}{\lg\lg\lg\rho}$	1	$\frac{\lg\rho}{\lg w}$	$\lg\lg\xi$	1
`row_nb`	1	1	1	1	1
`col_nb`	1	1	1	1	1
`tab_acc`	$\lg\lg\rho$	$\lg\lg\rho$	$\lg\lg\rho\left(\frac{\lg\rho}{\lg w}+1\right)$	$\lg\lg\xi$	$\lg\lg\xi$
Space	Υ‡ $+O\left(\frac{\Upsilon\lg\lg\lg\rho}{\lg\lg\rho}\right)$	$\Upsilon+O\left(\frac{\Upsilon\lg\lg\lg\rho}{\lg\lg\rho}\right)$	$\Upsilon+O(f)$	$f(\lg\xi+o(\lg\xi))$	$f(\lg\xi+o(\lg\xi))$

*$\rho=nm/f$ is the *inverse density* of R.
†$\xi=\min\{m,n\}$.
‡$\Upsilon=f\lg(nm/f)-O(f)$ is the information-theoretic minimum space to encode an $m\times n$ matrix with f 1-bits in it.

Figure 6.2: Asymptotic Upper Bounds for the `BINREL` Problem.

Model	Upper Bounds	Lower Bounds
Indexing	$\frac{n\lg n}{t}$, in [31, 44]	$\frac{n\lg n}{t}$, in [58] *
Non-indexing	$\frac{n\lg n}{t}$, in [31, 44] $\frac{n(\lg\lg n)^2}{(\lg n)}$ for $t=t'=\frac{\lg n}{\lg\lg n}$, in [44]	$\frac{n\lg n}{tt'}$ †

*Also presented in Section 4.2.
†Section 5.3.1.

Figure 6.3: Asymptotic Upper and Lower Bounds for the `PERMS` Problem.

Model	Old Lower Bounds	New Lower Bounds
Indexing (substring report)	$\frac{L\lg L}{t}$ for $t=o\left(\frac{(\lg L)^2}{\lg\lg L}\right)$, in [10]	$\frac{L\lg L}{t}$ for $t=o\left(\sqrt{\frac{L}{\lg L}}\right)$ *
Non-indexing (`TEXTSEARCH`)	N / A	$\frac{L\lg L}{tt'}$ †

*Section 4.3.
†Section 4.3.

Figure 6.4: Asymptotic Lower Bounds for the Substring Report and `TEXTSEARCH` Problems.

Problem	Section	Asymptotic lower bound for rtt'	Υ *
PERMS	5.3.1	$\Upsilon\frac{\lg n}{w}$	$n \lg n - \Theta(n)$
TEXTSEARCH	5.3.2	$\Upsilon\frac{p \lg \sigma}{w}$	$L \lg \sigma$
str_sel/str_acc	5.3.3	$\Upsilon\frac{\lg \sigma}{w}$	$L \lg \sigma$
BINREL	5.3.4	$\Upsilon\frac{\lg \rho}{w}$ †	$f \lg \rho - \Theta(f)$

*Information-theoretic minimum space (in cells).

†$\rho = \frac{mn}{f}$.

Figure 6.5: Cell Probe Lower Bounds from Chapter 5.

Index

Bibliography

[1] Miklós Ajtai. A lower bound for finding predecessors in Yao's cell probe model. *Combinatorica*, 8(3):235–247, 1988.

[2] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Combinatorial Pattern Matching*, pages 24–35, 2006.

[3] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 680–689, 2007.

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

[5] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[6] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[7] Augustin-Louis Cauchy. *Cours d'analyse de l'Ecole Royale Polytechnique, premier partie, Analyse algebrique*, volume 1. Paris, 1821.

[8] David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[9] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 289–294, 2001.

[10] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. *Journal of Algorithms*, 48(1):2–15, 2003.

[11] Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[12] William Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 3rd edition, 1968.

[13] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *IEEE Symposium on Foundations of Computer Science*, pages 184–196, 2005.

[14] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[15] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 2001.

[16] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th Symposium on String Processing and Information Retrieval*, pages 150–160, 2004.

[17] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Succinct representation of sequences. Technical Report TR/DCC-2004-5, Department of Computer Science, University of Chile, August 2004.

[18] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 690–696, 2007.

[19] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. In *International Colloquium on Automata, Languages and Programming*, pages 332–344, 2003.

[20] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[21] Rosario Gennaro and Luca Trevisan. Lower bounds on the efficiency of generic cryptographic constructions. In *IEEE Symposium on Foundations of Computer Science*, pages 305–313, 2000.

[22] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[23] Alexander Golynski. Optimal lower bounds for rank and select indexes. *Theoretical Computer Science*, 387(3):348–359, 2007.

[24] Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and S. Srinivasa Rao. On size of succinct indices. In *European Symposium on Algorithms*, volume 4698 of *Lecture Notes in Computer Science*, pages 371–382, 2007.

[25] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.

[26] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[27] Roberto Grossi and Kunihiko Sadakane. Squeezing succinct data structures into entropy bounds. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1230–1239, 2006.

[28] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *ACM Symposium on Theory of Computing*, pages 397–406, 2000.

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

[30] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. USA: Cambridge University Press, 1997.

[31] M.E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26:401–405, 1980.

[32] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[33] Guy Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, January 1989.

[34] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming*, pages 943–955, 2003.

[35] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[36] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1991.

[37] P. Vitányi M. Li. *An Introduction to Kolmogorov Complexity and its Applications, 2nd edition*. Springer-Verlag, New York, 1997.

[38] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[39] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[40] Peter Bro Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Symposium on Theory of Computing*, pages 625–634, 1994.

[41] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–12, 2005.

[42] Peter Bro Miltersen, Noam Nisan, Shmuel Safra, and Avi Wigderson. On data structures and asymmetric communication complexity. *Journal of Computer and System Sciences*, 57(1):37–49, 1998.

[43] J. Ian Munro. Tables. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, 1996.

[44] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations. In *International Colloquium on Automata, Languages and Programming*, pages 345–356, 2003.

[45] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[46] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[47] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Symposium on Theory of Computing*, pages 232–240, 2006.

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

[49] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *International Conference on Algorithms and Computation*, pages 410–421, 2000.

[50] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.

[51] Pranab Sen. Lower bounds for predecessor searching in the cell probe model. In *IEEE Conference on Computational Complexity*, pages 73–83, 2003.

[52] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[53] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[54] P. Weiner. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[55] Peter Widmayer. Personal communication, 2006.

[56] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space Theta(n). *Information Processing Letters*, 17(2), 1983.

[57] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, 1981.

[58] Andrew Chi-Chih Yao. Coherent functions and program checkers (extended abstract). In *ACM Symposium on Theory of Computing*, pages 84–94, 1990.