Advanced rank/select data structures: succinctness, bounds, and applications

Alessio Orlandi

Supervisor
Roberto Grossi

Referee
Veli Mäkinen
Gonzalo Navarro
Kunihiko Sadakane

December 19, 2012
Acknowledgements

This thesis is the end of a long, 4 year journey full of wonders, obstacles, aspirations and setbacks. From the person who cautiously set foot in Pisa to the guy who left his own country, none of these steps would have been possible without the right people at the right moment, supporting me through the dark moments of my life and being with me enjoying the bright ones.

First of all, I need to thank Prof. Roberto Grossi, who decided that he could accept me as a Ph.D. Student. Little he knew of what was ahead :). Still, he bailed me out of trouble multiple times, provided invaluable support and insight, helping me grow both as a person and as a scientist. I also owe a lot to Prof. Rajeev Raman, who has been a valuable, patient and welcoming co-author. I also want to thank Proff. Sebastiano Vigna and Paolo Boldi, who nurtured my passion for computer science and introduced me to their branch of research. Also Prof. Degano, as head of the Ph.D. course, provided me with non-trivial help during these years.

In Pisa, I met a lot of interesting people who were crucial for my happiness, was it when studying on a Sunday, on a boring the-proof-does-not-work-again day, in moments of real need, or on a Saturday night. Hence, this goes to Matteo S., Claudio G., Rachele, Petr, Rui, Anna and Valentina M., Maddalena, Dario F., Andra, Carlo de S., Rossano and Igor. And of course, I cannot forget my office mates, Giovanni and Giuseppe, who provided valuable lessons of Sicilian and managed not to kill me because of the open door.

Also, the last year has been full of changes and important steps in my life. I doubt this thesis would never have been printed if it weren’t for you, Cristina, Niamh, Mihajlo, Nicolas, Fred, Claudio, Andrea, Beatrice, Ivo. You guys know what I am talking about.

Beyond this, I need to thank my family, whose love and support provided me with the help one need when facing the unknowns of almost 4 years as a grad student.

I want to end by saying that it is hard to find people who have the bravery to believe in you, to enjoy your life with, to trust you, to share the little triumphs, the big losses, the hard lessons as well a laugh on the sunset on the beach, the kindness
of a tear, or the inspiration and the passion that drives you through. And I find it even harder to let go those people the way I knew them. So, Mauriana, Fabrizio, this “conclusion”, this thesis, is dedicated to you.
Contents

1 Introduction 9
   1.1 Succinct data structures 10
       1.1.1 Systematic vs. non systematic 11
   1.2 Thesis structure 12
       1.2.1 Focus 12
       1.2.2 Motivation 13
       1.2.3 Contents 14

2 Basic concepts 19
   2.1 Time complexities 19
   2.2 Space occupancy 21
       2.2.1 Empirical entropy 21
   2.3 Information-theoretic lower bounds 22
   2.4 Succinct bounds 24
   2.5 On binary rank/select 25
       2.5.1 Elias-Fano scheme 26
       2.5.2 Relations with predecessor problem 28
       2.5.3 Weaker versions 30
       2.5.4 Basic employment and usage 32
   2.6 Operating on texts 38
       2.6.1 Generic rank/select 38
       2.6.2 Compressed text indexing 39
       2.6.3 Burrows-Wheeler Transform 40
       2.6.4 Backward search 42
       2.6.5 A more generic framework 43

3 Improving binary rank/select 45
   3.1 Our results 45
   3.2 Relationship with predecessor search 46
   3.3 Searching the integers 49
CHAPTER 0. CONTENTS

3.3.1 Previous work: searching in $O(\log \log u)$ time .............................................. 49
3.3.2 Previous work: searching in $o(\log n)$ time ....................................................... 50
3.3.3 New result: searching systematically ................................................................. 51
3.4 An improved data structure .................................................................................... 53
  3.4.1 Overview of our recursive dictionary ................................................................. 53
  3.4.2 Multiranking: Oracle $\psi$ .................................................................................. 55
  3.4.3 Completing the puzzle ....................................................................................... 57
  3.4.4 Proof of main theorems .................................................................................... 58
3.5 Density sensitive indexes ....................................................................................... 60
  3.5.1 Preliminary discussion ....................................................................................... 60
  3.5.2 A succinct index for rank/selector ..................................................................... 61
  3.5.3 A succinct index for select$_d$ ........................................................................... 62

4 Rank and select on sequences ................................................................................... 65
  4.1 Our results ............................................................................................................... 65
  4.2 Related work .......................................................................................................... 67
  4.3 Extending previous lower bound work ................................................................... 67
  4.4 A general lower bound technique ........................................................................ 72
  4.5 Proof of Theorem 4.2 ............................................................................................ 77
    4.5.1 Encoding ........................................................................................................... 79
  4.6 Upper bound .......................................................................................................... 83
    4.6.1 Bootstrapping ................................................................................................... 83
    4.6.2 Supporting rank and select ............................................................................. 84
  4.7 Subsequent results ................................................................................................. 86

5 Approximate substring selectivity estimation ........................................................... 89
  5.1 Scenario ................................................................................................................... 89
  5.2 Preliminaries .......................................................................................................... 91
    5.2.1 Suffix trees ....................................................................................................... 91
    5.2.2 Pruned suffix trees ......................................................................................... 92
    5.2.3 Naive solutions for occurrence estimation .................................................... 93
    5.2.4 Previous work .................................................................................................. 94
  5.3 Uniform error range ............................................................................................... 95
  5.4 Lower-side error range ......................................................................................... 100
    5.4.1 Computing counts ........................................................................................... 101
    5.4.2 Finding the correct node ................................................................................ 102
  5.5 Lower bounds ......................................................................................................... 106
  5.6 Experiments ........................................................................................................... 107
0.0. CONTENTS

6 Future directions 113

Bibliography 115
Chapter 1

Introduction

Analysis and manipulation of large data sets is a driving force in today’s development of computer science: information retrieval experts are pushing against the limits of computation capabilities and data processing every day, both in academic and industrial environment. The data to process can be expressed in different formats: unstructured text, semi-structured data like HTML or XML, fully structured data like MIDI files, JPEG images, etc.. Data can be simply stored in some format, to be retrieved afterward, processed by a huge amount of machines in parallel that simply read it, manipulated to be altered, or searched, . . . Searching is, indeed, one of the most fascinating challenges: from simply retrieving to semantic interpretation of documents current computer science is employing massive data set computations to provide universal access to information (see e.g. [http://www.wikipedia.org or http://www.yahoo.com]).

At a design level, this translates into a set of less or more complex analyses and re-encoding of the collection documents (see e.g. [WMB99, BYRN11]) in a more suitable fashion so that retrieval can be eased. There are, then, multiple scenarios having a number of fixed points in common:

- Data is heterogeneous, but is usually naturally unstructured or semi-structured text (searching on images is a completely different area).

- Data is large, but is highly compressible: (un)structured text is still a predominant category of data sources and there exist a lot of compressors explicitly designed for it

- Data is static: when dealing with massive data sets, keeping data static is a great simplification both for compression, handling and testing reasons.

Compression is then a tool with multiple effects: by occupying less space, it is more economical to store data; moreover, it helps transforming network- and I/O-
bound applications into CPU-bound ones. Compressed data can be read and fed to the CPU faster and, if it the compression ratio is particularly effective, can lead to entire important chunks of data to be entirely in cache, speeding up computations and outweighing the processing penalty of decompression. As an extreme nontrivial example one can think of the LZO1x algorithm \[LZO\] that has been used to speed up transmission between Earth and the NASA Spirit and Opportunity rovers. They claim performance of $3 \times$ speed slowdown w.r.t. in-memory-copy for decompression at an average text compression of 2.94 bits/byte, i.e. bringing the file size to almost $1/3$ of its original size. Compressed data was then used in soft and hard real-time computations on the rovers, proving almost flawlessly transparent to underlying computation, also in terms of memory and CPU consumption.

Back to the original example, after an initial warehousing phase, analysis and serving systems, or just databases, need to access the underlying data. Traditionally this involves locating the chunk of data to load, decompressing it and operating on it, implying that computations repeated intensively on compressed data can cumulatively bring the balance of enforcing compression on the negative side. A solution to the dilemma is compressed data structures, and algorithms running on those: when the set of operations to be performed on the compressed data is stated in advance, compression may be tweaked so that operations can be performed on the data without actually decompressing the whole stream but just the areas of data that are interested by the specific query. In a sense, these data structure operate directly on the compressed data. Compressed data structures evolved into an entire branch of research (see bibliography), especially targeting the most basic data structures – the rationale behind it being that the more basic the data structures the more likely they are involved in any computation – and mainly binary and textual data.

### 1.1 Succinct data structures

Compressed data structures usually work as follows: the data is initially compressed with an adhoc compression mechanism. Then, a little amount of additional data must be introduced to perform the necessary data structure operation on the underlying data. Hence, the space complexity of compressed data structures is divided in a first, data-only, measure, plus a redundancy additional factor, which usually depends on the time requested to perform the data structure operation. A specific and well studied class of compressed data structures is called succinct data structures, the term succinct meaning that, under a certain data model, the redundancy footprint is a lower-order term of the data-only measure. As a matter of fact if a string requires $b$ bits to be represented at least, then a succinct data structure will
use $o(b)$ additional bits to perform the requested operations. A more thorough explanation of succinct bounds, being the principal focus of this thesis, can be found in Section 2.2.

1.1.1 Systematic vs. non systematic

Data structures are divided into two main categories: systematic and non-systematic ones [GM07]. Many classical data structures are non-systematic: given an object $c$, the data structure can choose any way to store $c$, possibly choosing a representation helping the realization of a set of operations $\Pi$ with low time complexity. Systematic data structures are different since the representation of the data cannot be altered; it is hypothesized that access to the data is given by an access operation that the succinct data structure uses to probe the data. The semantics of access depends on the current data domain. For example, considering strings in the character RAM model, access$(i)$ refers to the $i$th character of the input string. The additional data that systematic data structures employ to perform their task is referred to as a succinct index, as it is used to index the original data and at the same time must not exceed the succinct bound, namely the index must be lower order term of the data size. A data structure that receives a string and encodes it as-is, providing search for any substring $P$ into the string by just scanning, is a systematic data structure. A data structure encoding the input string by just rearranging the characters so as to ease search, is a non-systematic one.

Given their traits, comparing systematic and non-systematic succinct data structures is intrinsically difficult: data for systematic ones can be stored in any format, hence the only space complexity to be held into account is the redundancy, i.e. the amount of redundancy used to support operations, whereas non-systematic ones operate differently. In the latter case, there can be no conceptual separation between data encoding and index size. For example, let $f(n, \sigma)$ be the minimum information-theoretic space needed to store a string of length $n$ over an alphabet of size $\sigma$. Let also $\pi$ denote a set of operations to be supported on strings. A non-systematic data structure that can store strings in $f(n, \sigma) + O(n/\log \sigma)$ bits while providing $\pi$ can be, overall, smaller or larger than a systematic data structure with the same redundancy of $O(n/\log \sigma)$ bits: depending on how $S$ is kept encoded, an $O(f(n, \sigma))$ data structure providing access to the string instead of a plain $n \log \sigma$ bits encoding would pose a difference.

Both approaches have advantages and drawbacks: a non-systematic data structure is in control of every aspect of the data, and by exploiting its best representation w.r.t. the operations to be performed, can as such outperform a systematic one in terms of space complexity. Clearly, this solution can be employed only if the data
can be re-encoded by the data structure for its own purposes, discarding the original representation. On the other hand, consider two sets of operations \( \Pi_1 \) and \( \Pi_2 \) to be implemented on the same object \( c \), so that no single non-systematic data structure implementing both exists: the solution with non-systematic data structures is to store the data twice, whereas systematic ones can exploit their indirect relation with data to provide, in some scenario, a better result.

Finally, separation of storage and operations can help in more complex environments. It is not uncommon that the basic object representation of \( c \) to be multiple times larger than the index itself, so that in some scenarios storage of the data itself onto the main memory is prohibitive, even in a transparently compressed format that the data structure does not control. Whereas the index can be loaded into main memory, the data must be accessed through on-disk calls. The modularity of systematic succinct data structures comes into help: data structures can be designed so as to minimize usage of the access primitive and work very well in such models.

1.2 Thesis structure

1.2.1 Focus

In this thesis, we focus our attention on studying, creating and applying one of the most prominent subclasses of succinct data structures, both in systematic and non-systematic fashion, called rank and select data structures. Operations are performed on strings of the alphabet \([\sigma] = 0, 1, 2, \ldots, \sigma - 1\), where \( \sigma \geq 2 \). For a string \( S = [0, 1, \ldots, |S| - 1] \) of length \( n \), the data structure answers to two queries:

- rank\(_c\)(\( p \)), for \( c \in [\sigma], p \in [n + 1] \), which returns \(|\{x < p|S[x] = c\}|\), i.e. the number of characters of value \( c \) present in the prefix of \( p \) characters of \( S \).

- select\(_c\)(\( i \)), for \( c \in [\sigma], i \in [1, n] \) which returns the position of the \( i \)th-most occurrence of \( c \) in \( S \) from the left from the left, namely returns \( y \) such that \( S[y] = c \) and rank\(_c\)(\( y \)) = \( i - 1 \), or \( -1 \) if such position is nonexistent.

For example for \( S = abcbba \), \( \text{rank}_b(3) = 1 \) and \( \text{select}_c(1) = 2 \), while \( \text{select}_b(3) = -1 \). Furthermore, for \( S = 0110101 \) we have \( \text{rank}_1(3) = 2 \) and \( \text{select}_0(1) = 0 \).

\(^1\)In literature, sometimes the definition \( \{x \leq p|S[x] = c\} \) is preferred, although it easily create inconsistencies unless the string is indexed from 1.

\(^2\)To avoid confusion, we use 0 and 1 to denote characters in binary strings.
1.2.2 Motivation

There are a number of high level motivations for this thesis to focus on \texttt{rank} and \texttt{select} data structures. We now give a less technical overview and motivation for the whole thesis. Chapter \ref{chap:overview} gives a more detailed and technical overview. Section \ref{sec:motivation-chapter} gives a per-chapter motivation to the results in the thesis.

First, they are basic operations in computer science. If one considers the binary case, where $\sigma = 2$, performing \texttt{select}_1(\texttt{rank}_1(x)) for some $x$ recovers the \textit{predecessor} of $x$, i.e. $\max_y \{ y < x, S[y] = 1 \}$. To grasp the importance of this operation, let us consider a set of IPv4 net addresses, say 1.1.2.0, 1.1.4.0, etc. The basic operation in a router is to perform routing lookup: given a destination IP address, say 1.1.2.7, find the network address that matches it, by finding the IP address with the longest common prefix with it. Not surprisingly, this can be reduced to a set of predecessor searches: let us interpret IPv4 address as 1s in a string of 2^{32} bits. Finding the predecessor of the 32-bit representation of the destination IP address, gives the possible next hop in routing. In a sense, predecessor problem is one of the most diffused and executed on the planet. Since storing routing tables compactly while keeping fast predecessor lookup is important \cite{DBCP97}, study of \texttt{rank} and \texttt{select} operations is well justified, both in theory and in practice.

Still in the binary case, \texttt{rank} and \texttt{select} are relevant without the need to be combined into predecessor search. For \texttt{select} a simple idea is given by the following scenario. Let us consider a static set of variable length records $R_0, R_1, \ldots, R_{n-1}$. Those records can be stored contiguously, with the caveat that it would make difficult to have random access to each record. Assuming the representation of all records requires $u$ bits, one can store a bitvector of $u$ bits with $n$ 1 inside, marking the bit that denotes the start of the record. Performing \texttt{select}_1(k) for some $k$ gives the address at which the $k$th record begins \cite{Eli74a, Fan71a}. This would require $n \log(u/n) + O(n)$ bits to be encoded by means of simple schemes, and \texttt{select} can be solved in $O(1)$ time. Considering that the trivial $O(1)$-time solution of array of fixed-size pointers would require $nw$ bits (where $w$ is the size of the pointer), and we can assume w.l.o.g. that $u \leq nw$, the difference is self-evident: $n \log(u/n) \leq n \log w \ll nw$, for $w$ larger than some constant.

For \texttt{rank} operation, consider again the set of IPv4 addresses as 32 bits values. Assume that for a subset of those IP addresses, some satellite data has to be stored. To be practical, assume that the physical network interface to reach that address is to be added (so that we have a small integer). Hence, we assume that we have a universe of $u = 2^{32}$ values out of which $n$ values are selected to be decorated with the satellite data. One can encode a bitvector $B$ of $u$ bits with $n$ ones, setting $B[i] = 1$ iff the $i$th IP address has some satellite data attached. Then, one can encode $B$
and lay down the satellite data for each record in a consecutive array $C$ so that the $C[\text{rank}_1(i)]$ is the data attached to the $i$th IP, since $\text{rank}_1(i)$ over $B$ will retrieve the rank (in the mathematical sense) of the $i$th IP among those selected.

$\text{rank}$ and $\text{select}$ data structure are bread and butter of other succinct data structures. For the binary case, it turns out [Jac89, MR01] that they are the basic tool to encode static binary trees: given a tree of $n$ nodes, $\text{rank}$ and $\text{select}$ data structures are the basic piece to obtain a representation of $2n + o(n)$ bits which supports a wide variety of navigation operations (up to LCA and Level ancestors) in constant time.

For the generic alphabet case, they are at the basis of compressed text indexing [GV05, Sad03, ?]: given a text $T$, it is possible to compress it and at the same time search for occurrences of arbitrary patterns in $T$. The data structures have necessarily to spend more space than the simple compression, but the resulting advantage is substantial: instead of having to decompress the whole text to perform substring searching, the text can be decompressed only in “local” regions which are functional to the substring search.

$\text{rank}$ and $\text{select}$ have also a complex history. Being intertwined with predecessor data structures, they received indirect and direct attention in terms of lower bounds. This is especially true for the binary case. Various studies [Gol07b, Ajt88, BF02, PT06b, PV10] proved that there exist separation barriers on the minimum time complexity for predecessor search as a function of the space used by the data structure, the universe size and the word size of the machine we operate on. As a result, multiple paths have been taken: some researchers insisted on $O(1)$ time complexity and small spaces when both are possible; others tried to focus on fast systematic data structures irrespective of the space complexities; others explored small, although non-optimal, space complexities giving up on the constant time.

Finally, $\text{rank}$ and $\text{select}$ data structures have been studied in practice, proving that some schemes are actually interesting, or can be engineered to be, in practical scenarios [GGMN05, GHSV06, OS07, Vig08, CN08, OG11, NP12].

### 1.2.3 Contents

The thesis is structured as follows.

**Basic concepts**

Chapter [2] contains a more thorough introduction to succinct data structures in general, providing an overview of related work, applications and past relevant results in the world of such data structures. The chapter also serves as an introduction
to many of the data structures that will use as comparison or as pieces in our constructions in the chapters to follow.

### Improving binary rank/select

Chapter 3 contains the first contributions about binary rank/select data structure, introducing a data structure able to operate in little space where it was not possible before. The results were originally presented in [GORR09] and [GORR12] (preprint available as [GORR11]).

Given a universe $U$ of size $u = |U|$ and a subset $X \subseteq U$ of size $n = |X|$, we denote by $B(n, u)$ the information-theoretic space necessary to store $X$.

Among non-systematic data structures, multiple results been presented in the past. If one assumes that $u \leq n \log^O(1) n$, it is possible store $X$ in $B(n, u) + o(n)$ bits maintaining $O(1)$ rank time complexity. On that line, Pagh [Pag01] obtained $B(n, u) + O(n \log u \log(u/n) \log^2 u)$. Afterwards, Golynski et al. [GRR08] obtained $B(n, u) + O(n \log u \log(u/n) / \log^2 u)$. An optimal solution in this dense case was proved by Pătraşcu [P08], obtaining, for any $t > 1$,

$$B(n, u) + O \left( \frac{nt^t}{\log^t u} \right) + O \left( u^{3/4} \log^O(1) u \right),$$

with $O(t)$ time to execute rank, essentially closing the problem (see [PV10] for a proof). For sparser cases, namely when $u = \omega(n \log^O(1) n)$, $\Omega(n)$ bits of redundancy are necessary [PT06b] [PV10] for $O(1)$ rank. To keep this time complexity Golynski et al. [GGG+07] obtained $B(n, u) + O(u \log u \log^2 u)$ bits. Note that also Pătraşcu’s data structure can be applied to the sparse case, still being a competitive upper bound.

We introduce a novel data structure for the sparser case. Let $s = O(\log \log u)$ be an integer parameter and let $0 < \delta, \varepsilon \leq 1/2$ be two parameters. Also recall that ($B(n, u) = \log \lceil u \rceil / n$). Theorem 3.1 shows how to store $X$ in

$$B(n, u) + O(n^{1+\delta} + n \log \log u + n^{1-s\varepsilon}u^{\varepsilon})$$

bits of space, supporting select and rank operations in $O(\log(1/(\delta^s \varepsilon)))$ time. Similarly, under the additional assumptions that $s = \Theta(1)$ and $u = \Theta(n2^\log n/\log \log n)$ Theorem 3.2 shows how to store $X$ in

$$B(n, u) + O(n^{1+\delta} + n^{1-s\varepsilon}u^{\varepsilon})$$

bits of space supporting rank and select in $O(\log(1/(\delta \varepsilon)))$.

In particular, Theorem 3.2 surpasses Pătraşcu’s solution for $\delta = O(1/\log n)$ range, since his data structure is optimal for the denser cases. Theorem 3.1 has a
different role: it proves that already existing data structures for predecessor search (hence tightly connected to \texttt{rank}_1) can be employed to support \texttt{select}_0 (a non-trivial extension) without the need to actually write the complement of \(X\), which may be quite large.

**Rank and select on sequences**

Chapter 4 contains contributions about systematic data structures, proving an optimal lower bound for \texttt{rank} and \texttt{select} strings. The results were originally presented in \cite{GOR10}.

In this scenario, we assume to have a string \(S\) of length \(n\) over alphabet \([\sigma]\) that we can probe for each character, so that \(\text{access}(i) = S[i]\) for any \(0 \leq i < |S|\). We study systematic data structures for \texttt{rank} and \texttt{select} over \(S\). These data structures where, up to our work, limited to \texttt{rank} in \(\omega(\log \log \sigma)\) time, when \(o(n \log \sigma)\) bits of redundancy were used. Regarding the binary case, a clear lower bound for systematic data structures was proved by Golynski \cite{Gol07a}, so that for any bitvector of length \(u\), in order to execute \texttt{rank} and \texttt{select} with \(O(t)\) calls to \texttt{access} for single bits of the bitvector, an index must have \(\Omega(u \log \log (t \log \log u))\) bits of redundancy. No lower bound for the generic case was available. At first, one could be tempted to just generalize the techniques of the binary case to the generic case. This actually leads to a lower bound of \(\Omega(n \log \log \log \log \sigma)\) bits for \texttt{rank} and \texttt{select}, as proved by our Theorem 4.1 which is extremely weak when \(t = o(\sigma)\). Indeed, a linear trade off for systematic data structures on strings was already proved by Demaine and López-Ortiz \cite{DLO03}. Our Theorem 4.3 and Theorem 4.4 finally prove that for any \(t = o(\log \sigma)\), a redundancy of \(r = \Omega(n \log \sigma / t)\) bits of redundancy are necessary for either \texttt{rank} and \texttt{select}, independently.

Assuming \(t_a\) represents the time to access a character in the representation of the string \(S\) on which the data structure is built, Barbay et al. \cite{BHMR07c} gave a data structure able to perform \texttt{rank} in \(O(\log \log \sigma \log \log \log \log \sigma)\) time with \(O(n \log \log \sigma)\) bits of redundancy. In terms of number of calls to \texttt{access}, the complexity amounts to \(O(\log \log \sigma \log \log \log \sigma)\). They can perform \texttt{select} in \(O(\log \log \sigma)\) calls to \texttt{access} and \(O(t_a \log \log \sigma)\) total time. The indexes of \cite{BHMR07c} can also be squeezed into entropy bounds, providing a data structure that performs on highly compressed strings.

Our Theorem 4.6 proves that for any fixed \(t = O(\log \sigma)\) one can build an index in \(O((n/t) \log \sigma)\) bits that (i) solves \texttt{select} in \(O(t_a t)\) time and \(O(t)\) probes (ii) solves \texttt{rank} \(O(t(t_a + \log \log \sigma))\) time and \(O(t)\) probes. Hence, we prove that with \(O(n \log \log \sigma)\) extra bits, one can do \texttt{rank} in exactly \(O(\log \log \sigma)\) time. This means that, in term of calls to \texttt{access}, as long as \(t = o(\log \sigma / \log \log \sigma)\), our data struc-
ture is optimal. We also provide, in Corollary 4.2, a way to compress both the original string and our data structure in very little space. Moreover, subsequent studies [BN11a] shows that, within those space bounds, \(O(\log \log \sigma)\) is the optimal time complexity to execute rank and select on strings with large alphabets, namely when \(\log \sigma / \log w = O(\log \sigma)\).

**Approximate substring selectivity estimation**

Chapter 5 contains a novel application of rank and select data structures to approximately count the number of occurrences of a substring.

Given a text \(T\) of length \(n\) over alphabet \([\sigma]\), there exist many data structures for compressed text indexing that can compress \(T\) and allow for substring search in the compressed version of \(T\): given any pattern \(P\) such data structures are able to count the exact number of occurrences of \(P\) inside \(T\). Not all applications require the exact count; for example, the selectivity estimation problem [JNS99] in databases requires only an approximate counting of pattern occurrences to perform efficient planning for query engines. More formally, given a fixed value \(l \leq 1\), if \(y = \text{Count}(P)\) is the actual occurrence count for \(P\), we expect to give an answer in \([y, y + l - 1]\).

The database community provided multiple results [JNS99, KV196, LNK09, CGG04, JL05] aimed mainly at practical speed. We perform the first theoretical and practical study strictly regarding space efficiency. We study the problem in two variants: the uniform error range, where we answer within \([y, y + l - 1]\) for any pattern \(P\), and the lower-sided error range, where for any pattern that occurs more than \(l\) times in \(T\), the count is actually correct.

We initially extend existing results in compressed text indexing to prove in Theorem 5.2 that, for \(l = o(\sigma)\), an index for the uniform error case can be built in optimal \(\Theta(|T| \log \sigma / \sigma)\) bits. The time complexity depends on the underlying rank/select data structure for strings that is chosen at construction time.

We also provide a solution for the lower sided error range based on pruned suffix trees. It is not easy to immediately state the space occupancy of our data structure, which scales with the amount of patterns that appear more than \(\ell\) times in the text. We refer the reader to Theorem 5.4 for details.

Finally, Section 5.6 rejoins our results with the original works of selectivity estimation. For comparison, we give the average error measured on a number of random patterns, provided that previously known data structures and our data structure match in space occupancy. As a result, we have cases in which the average error is always less than 2.5, even when we set \(l = 32\). This gives improvements up to 790 times with respect to previous solutions.
Chapter 2

Basic concepts

This chapter recalls some fundamental techniques and concepts used throughout the rest of this thesis. A first part describes the very basic notation, including the computational model into which we deliver our work and how succinct data structures are categorized. Next, we describe data structures that are connected to the original results presented in forthcoming chapters.

Throughout the whole thesis, we will refer to logarithms as in base 2, that is, \( \log x = \log_2 x \). To ease readability, we define \( \log^{(1)} x = \log(x) \) and \( \log^{(i)} x = \log \log^{(i-1)} x \) for \( i > 1 \) and we use the notation \( \log^k f(x) \) as a shortcut for \( \log(f(x))^k \) for any \( k \) and \( f(\cdot) \). Finally, we will often use forms similar to \( \log(x)/y \). To ease readability, we will omit parentheses, writing them as \( \log x/y \), so that \( \log x/\log \log(1/x) \) is actually \( \log(x)/\log \log(1/x) \) for example.

We also recall the definition of \( \tilde{O} \): \( f(x) \) is said to be \( \tilde{O}(g(x)) \) if \( f(x) \) is \( O(g(x) \text{polylog}(x)) \).

2.1 Time complexities

Data structures and algorithms can exploit the specific computational model they operate on. As such, they are also subject to specific time lower bounds. Some models are more abstract (like the bit-probe model), some are more realistic (e.g., Word-RAM) and some are tightly related to lower bounds (cell-probe model). We now describe the ones that will be used throughout the thesis.

Word RAM model

The Word Random Access Machine is a very realistic model, i.e., it tries to model a physical computer, where a CPU executes the algorithm step by step and accesses

\footnote{Here we refer to the basic AC0 operations, plus multiplication}
locations of memory in a random fashion in \(O(1)\) time (this is in contrast to the classic Turing machine with sequential access). The model relies on the concept of \textit{word}: a contiguous area of memory (or working memory inside the CPU) of \(w\) bits, for some parameter \(w\). The memory is organized in contiguous words of \(w\) bits. When computing time complexities, single accesses to a word costs \(O(1)\) time units. Every elementary mathematical and logical operation takes effect on a constant number of words in \(O(1)\) time. It is usually assumed that the Word RAM is \textit{trans-dichotomous}: supposing the problem has a memory size driven by some value \(n\), it holds that \(w \geq \log n\). Note that a non-trans-dichotomous RAM would not be able to address a single cell of the data structure using \(w < \log n\) bits.

**Cell probe model**

The classic \textit{cell probe model}, introduced by Yao \cite{Yao81}, is the usual environment into which lower bounds for the Word RAM model are proved. The model is exactly the same, however, the computational cost of an execution is given by the sole number of probes executed over the memory. In other words, the cell probe model is a word RAM model where the CPU is arbitrarily powerful, in contrast to being limited to basic mathematical and logical operations.

**Bit probe model**

The \textit{bit probe model} is a model based on the assumption that a random access memory of finite size exists and the algorithm may perform single bit probes. More formally, a CPU exists and an algorithm may perform elementary mathematical and logical operations plus accessing one single bit of memory in \(O(1)\) time. Computational time is computed by summing the time of elementary operations and accesses to memory. In other words, it is the cell probe model, with \(w = 1\).

The bit probe model can also be used to prove lower bounds, where only the amount of memory accesses is considered.

**Character RAM model**

A variation over the Word RAM model is the \textit{character RAM model}, which we formally introduce in this thesis. It is defined by two parameters: \(\sigma\) (alphabet size, bits) and \(w\) (word size). The CPU works as in the Word RAM model, with operations over entire words, and it is supposed to be trans-dichotomous. The memory is instead divided into two different areas: a \textit{data} and an \textit{index} area. Accesses to the data area are limited to contiguous area of \(\log \sigma\) bits each, performed through a specific \texttt{access}\((\cdot)\) call, instead of \(w\) bits as in the Word RAM model. Accesses to the
index are of normal word size. The time complexity here is provided by the number of probes in the data area, plus the probes in the index area, plus the computation time. The character RAM model is a good approximation of communication-based models, where access to the underlying data is abstract and is designed to retrieve the logical unit of information (i.e., the character).

The character probe model is a hybrid between the cell probe model and the character RAM model: the machine has an index memory (which may be initialized with some data structure) and every computation done with such index memory is completely free. The only parameter of cost is the number of access- es to the underlying data, which are character-wide. In other words, it is the character RAM model where the cost for index computations is waived. This model was already used in [Gol07b].

2.2 Space occupancy

2.2.1 Empirical entropy

Shannon’s definition of entropy [CT06] can be given as follows:

**Definition 2.1.** Given an alphabet $\Sigma$, a source $X$ is a discrete random variable over $\Sigma$, whose probability density function is denoted by $p(\cdot)$. Its entropy is defined as

$$H(X) = \sum_{c \in \Sigma} p(c) \log \frac{1}{p(c)}.$$ 

Shannon’s entropy [CT06] is tightly connected to compression, as it defines how easy to compress is a probabilistic source. As all generic definitions, it is acceptable when no further knowledge is available. It can be used to deduce the compressibility of distributions as a whole. On the other hand, from one factual output of the source, it is impossible to use Shannon’s entropy to bound the compressibility the output text.

In recent years [Man01] empirical entropy has shown to be a powerful tool in the analysis of compressors performance, overcoming the limitation of Shannon’s entropy. At a high level the definition of entropy starts from the text output of an unknown source:

**Definition 2.2.** Given a text $T$ over an alphabet $\Sigma$ of size $\sigma$, let $n_c$ be the number of occurrences of $c$ in $T$ and let $n = |T|$. Then, the zero-th order empirical entropy of $T$ is defined as

$$H_0(T) = \frac{1}{n} \sum_{c \in \Sigma} n_c \log \frac{n}{n_c}.$$
CHAPTER 2. BASIC CONCEPTS

Viewing the \( n_c/n \) factor as an estimator of the probability of character \( c \) appearing in the text, we can assume \( T \) to be generated by a probabilistic source in Shannon’s setting where \( p(c) = n_c/n \) (note that we created the source onto the specific text). Assume to have the text \( T \) and to build \( p(\cdot) \) as described, then, the Shannon’s entropy of the text is

\[
H(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c},
\]

hence proving the same definition. Therefore, the value \( nH_0(T) \) provides an information-theoretic lower bound on the compressibility of \( T \) itself for any compressor that encodes \( T \) by means of encoders that do not depend on previously seen symbols \[Man01\]. An example of such encoder is Huffman’s coding (we assume the reader familiar with the concept \[CLRS00\].

More powerful compressors can achieve better bounds by exploiting the fact that when encoding/decoding a real string, some context is also known. Empirical entropy can be extended to higher orders, exploiting the predictive power of contexts in the input string. To help define \( k \)th order empirical entropy of \( T \), for any string \( U \) of length \( k \), let \( U_T \) be the string composed by juxtaposing single symbols following occurrences of \( U \) in \( T \). For example if \( T = \text{abracadabra} \$ \) and \( U = \text{ab} \), then \( U_{ab} = \text{rr} \), as the two occurrences of \( \text{ab} \) in \( T \) are both followed by \( r \). Then, the \( k \)-order entropy of \( T \) is defined as:

\[
H_k(T) = \frac{1}{n} \sum_{U \in \Sigma^k} |U_T|H_0(U_T),
\]

where \( \Sigma^k \) is the set of all strings of length \( k \) built with characters in \( \Sigma \).

Empirical entropy has been intensively used \[Man01\] to build a theory explaining the high performance of Burrows-Wheeler Transform based compressors (e.g., bzip2), since higher order entropies \( (k \geq 1) \) always obey the relationship

\[
H_k(T) \leq H_{k-1}(T).
\]

Burrows-Wheeler transform and its relation with empirical entropy are thoroughly discussed in Section 2.6.3. More details about empirical entropy and the relation to compressors can be found in \[Man01\].

2.3 Information-theoretic lower bounds

Given a class of combinatorial objects \( C \), the minimum amount of data necessary for expressing an object \( c \in C \) is \( \log |C| \) bits, equivalent to the expression of entropy
in an equiprobable setting. This quantity is often called the *information-theoretical bound*. Inasmuch \( \log |C| \) poses a lower bound, it is not always a barrier. The first strategy to avoid the limitation is to further refine class \( |C| \), by introducing more constraints on the data. If \( C \) is the class of all bitvectors of length \( u \), then \( \log |C| = u \), but if we can restrict to a subclass of \( C \), say \( C_n \) of bitvectors of length \( u \) with \( n \) bit set to \( 1 \), then \( \log |C_n| = \log \left( \binom{u}{n} \right) \). This for bitvectors is widely used in literature since it is, similarly to empirical entropy, a data aware measure.

**Definition 2.3.** Given a bitvector \( B \) of size \( u \) with cardinality \( n \), the information theoretical bound is

\[
B(n, u) = \left\lceil \log \left( \binom{u}{n} \right) \right\rceil.
\]

By means of Stirling inequality it is easy to see that

\[
B(n, u) = n \log \left( \frac{u}{n} \right) + (u - n) \log \left( \frac{u}{u - n} \right) - O(\log u)
\]

The function is symmetric and has an absolute maximum for \( n = u/2 \), as \( B(u/2, u) = u - O(\log u) \), so that one can restrict to the case \( n \leq u/2 \) and avoid considering the remaining values, obtaining that

\[
B(n, u) = n \log(u/n) + O(n).
\]

As one can see, for bitvectors, the information theoretical bound is equivalent, up to lower order terms, to the empirical entropy bound: let \( B \) be a bitvector of \( u \) bits with \( n \) 1s; then, for \( c = 1 \) we have \( n \) occurrences and for \( c = 0 \) we have \( u - n \) occurrences. Substituting into Definition 2.2 gives that

\[
H_0(B) = \frac{n}{u} \log \left( \frac{u}{n} \right) + \frac{u - n}{u} \log \left( \frac{u - n}{n} \right),
\]

so that \( uH_0(B) = B(n, u) + O(\log u) \). In general, the information theoretical bound is mainly used where more sophisticated methods of measuring compressibility are not viable: higher order empirical entropy, for example, is suitable mainly for strings, whereas trees and graphs have no affirmed compressibility measure.

Another example is given by the information theoretic limit used to represent ordinal trees. An *ordinal* tree is a tree where children of a node are ordered, whereas in a cardinal tree the \( i \)th child of a node is specified by explicitly specifying the index \( i \). For example, in a cardinal tree a node can have child 3 and lack child 1. Many basic trees in computer science books are usually ordinal; cardinal trees are frequently associated with labelled trees such as tries, or binary search trees. We remark that, since there exists a bijection between binary cardinal trees and arbitrary
degree cardinal trees, it is sufficient to have a lower bound over the representation of binary trees to induce one over cardinal trees. The number of cardinal binary trees of \( n \) nodes is given by the Catalan number \( C_n \simeq 4^n/(\pi n)^{3/2} \) (see [GKP94] for a proof), hence

**Lemma 2.1.** An ordinal tree of \( n \) nodes requires \( \log C_n = 2n - \Theta(\log n) \) bits to be described.

## 2.4 Succinct bounds

The concepts of entropy and information theoretical lower bound are vital for succinct data structures too. Consider a set \( \Pi \) of operations to be performed on any object \( c \), with some defined time complexity. Beyond the sole space occupancy for \( c \)'s data, many data structures involve encoding additional data used to perform operations on the representation of \( c \) itself. The space complexity of the data structure can be split into two parts: the information theoretical lower bound (\( \log |C| \) bits) and the additional bits required to implement \( \Pi \) over \( c \), usually referred to as *redundancy* of the data structure. We assume the setting to be static: the data is stored once and the operations do not modify the object or the index representation. For example, consider the class \( C_B \) of balanced bitvectors, having \( u \) bits of which \( u/2 \) are set to 1. Then, \( \log |C| = \log \left( \binom{u}{u/2} \right) = u - O(\log u) \) bits. Suppose a data structure must store \( C \) and providing operation \( \text{count}(a, b) \), returning the number of bits set to 1 in the interval \( [a, b] \). A data structure employing \( u + O(u/\log u) \) bits for that has a redundancy of \( O(u/\log u) \) bits. Data structures can be classified with respect to their size: data structures employing \( \omega(\log |C|) \) bits are standard data structures. An example is the data structure that solves \( \text{rank/select} \) on a balanced bitvector \( B \), with \( O(uw) \) bits: the data structures stores, in a sorted array, the value of \( \text{rank}(i) \) for each \( i \in [u] \) and, in another sorted array, the value of \( \text{select}(j) \) for each \( j \) such that \( B_j = 1 \). Data structures using \( O(\log |C|) \) bits are called compact; proper succinct data structures use \( \log |C| + o(\log |C|) \); finally, we have implicit data structures that use \( \log |C| + O(1) \) space. For succinct data structures, the quantity \( \log |C| + o(\log |C|) \) is often referred to as the *succinct bound*. Among succinct data structures, the smaller the redundancy for a fixed set of time settings for the operations, the better the data structure.

In the dynamic setting, succinct data structures must maintain the succinct bound among updates: after any update operation, at instant \( r \), supposing the description of the object requires \( \log |C_r| \) bits, the data structure must maintain a \( \log(|C_r|)(1 + o(1)) \) occupancy rate.

For binary vectors, succinct \( \text{rank/select} \) data structures use \( B(n, u) + o(B(n, u)) \)
2.5. On binary rank/select

bits and, possibly, $B(n, u) + o(n)$ when available. In the string setting, where $n$ is the string length and $\sigma$ is the alphabet size, a succinct data structure will use $n \log \sigma + o(n \log \sigma)$ for any string $S$. It is easy to note that for the arbitrary string the succinct bound is a more permissive definition. This is dictated by the presence of certain lower bounds, which will be presented later.

Succinct bounds must also be seen in the logic of the data structure actual needs. The class $C$ of elements should be as much restricted as possible, constrained to the minimum set of information that are actually needed to perform the operation. The more those operations are data-independent, the smaller the bound becomes. An important example is the data structure for Range Minimum Queries (RMQ) over integer arrays developed by Fischer [Fis10].

The data structure allows one to create an index that can answer RMQ over an array $A$ of $n$ integers as $\text{rmq}(i, j) = z$, where $i, j$ and, more importantly, $z$ are indexes of positions in $A$. Considering the data structure does not require to access $A$ at all, it does not need to represent it. Hence, the succinct bound to be compared with is not $n \log n$ (space to represent $A$) but $2n - \Theta(\log n)$. The latter is given by the equivalence of RMQ and the need to represent a Cartesian Tree shape over the elements of $A$, which requires, as per Lemma 2.1 $2n - \Theta(\log n)$ bits at least.

2.5 On binary rank/select

This section explores some results and applications of basic rank and select data structures in the binary case, excluding the results of the thesis.

Let us consider the Word RAM model with $w$-bit words and a universe set $U$ from which we extract a subset $X$. Here, $|U| = u$ and $|X| = n$. We often refer to $n$ as the cardinality of $X$ and to $u$ as the universe (size). Note that we will often change our point of view from the one of a bitvector to the one of a subset. This is possible by a trivial, although important, duality where a bitvector $B$ can be interpreted as a subset:

**Definition 2.4.** Let $B$ be a bitvector of length $u$ of cardinality $n$. Let $U = [u]$ be called the universe. Then $B$ can be interpreted as the characteristic function for a (sorted) set $X \subseteq U$, so that $X$ contains $a \in U$ iff $B[a] = 1$. $X$ is called the corresponding set of $B$ over $U$ and $B$ is the corresponding bitvector of $X$.

As an example, consider the bitvector $B = 00101010$, where $n = 3$ and $u = 8$. The corresponding set of $B$ is $X = \{2, 4, 6\}$.

Data structures should use at least $B(n, u) = n \log (u/n) + O(n)$ bits to store $X$. In the Word-RAM model, storing $X$ implies using $\Theta(nw)$ bits, where $w = \Omega(\log u)$,
thus employing $\Omega(n \log u)$ bits, way more than necessary. Succinct data structures are appealing, especially w.r.t. plain representations, when $n$ approaches $u$, since they can use can use as little as $B(n, u) = n \log(u/n) + O(n)$ bits to represent the data; when $n$ approaches $u$, $\log(u/n)$ draws towards 1, which is better than the plain $\log u$. Systematic data structures become appealing when the data is shared with other structures or, in terms of space, again when the density of the set is high, i.e., when $n = \Theta(u)$. Different time/space trade-offs, illustrated in Table 2.1, are presented for binary rank/select data structures. The table describes only some of the latest developments and complexities at the time of writing. More results can be found, for example, in [RRR07, Gol07a, GGG+07, GRR08]. Some are also described later.

Table 2.1: Latest time-space tradeoffs for static binary rank/select data structures. NS/S stands for non-systematic/systematic.

<table>
<thead>
<tr>
<th>#</th>
<th>rank</th>
<th>select$_1$</th>
<th>select$_0$</th>
<th>Type</th>
<th>Space</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(\log(u/n))$</td>
<td>$O(1)$</td>
<td>-</td>
<td>NS</td>
<td>$B(n, u) + O(n)$</td>
<td>[Eli74b, Fan71b, OS07]</td>
</tr>
<tr>
<td>2</td>
<td>$O(t)$</td>
<td>$O(t)$</td>
<td>$O(t)$</td>
<td>NS</td>
<td>$B(n, u) + O(\frac{u^t}{\log^t u}) + O(u^{1/3})$</td>
<td>[FG08]</td>
</tr>
<tr>
<td>3</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>S</td>
<td>$u + O(u \log u / \log \log u)$</td>
<td>[Cla96, Gol07a]</td>
</tr>
</tbody>
</table>

Note that, as described in [RRR07], non-systematic binary rank/select data structures are sometimes referred to as *fully indexable dictionaries*.

### 2.5.1 Elias-Fano scheme

In this section we describe the Elias-Fano rank/select data structure. It is a non-systematic data structure that is comprised by a simple encoding scheme originally developed by Elias and Fano independently (see [Eli74a, Fan71a]) and later revised by [GV05, OS07, Vig08] to support rank and select operations. The scheme will be later used in Chapter 3 as the grounds of an improved data structure. It is also an elegant and simple data structure that serves as a gentle introduction to rank and select internals. We will prove the following:

**Theorem 2.1.** Given a set $X$ from universe $U$ so that $n = |X|$ and $u = |U|$, there exists a data structure that uses $B(n, u) + O(n)$ bits that answers select$_1$ in $O(1)$ time and rank in $O(\log(u/n))$ time.

After reviewing the plain encoding as originally described by Elias and Fano (Theorem 2.2) we briefly review how to support rank and select$_1$ (but no select$_0$)

---

3The original definition of fully indexable dictionaries constrains to $O(1)$ time but later papers presented such data structures using $\omega(1)$ time, hence the two terms can be considered interchangeable.
on it (proving Theorem 2.1). For the sake of explanation, we will assume that $n \leq u/2$, otherwise we store the complement of $X$.

Recall that $X = \{x_1 < x_2 < x_3 < \cdots < x_n\}$ is equivalent to its characteristic function mapped to a bitvector $S$ of length $u$, so that $S[x_i] = 1$ for $1 \leq i \leq n$ while the remaining $u - n$ bits of $S$ are 0s.

**Theorem 2.2.** Given a set $X$ from universe $U$ so that $n = |X|$ and $u = |U|$, there exists an encoding of $X$ that occupies $n \log(u/n) + O(n)$ bits.

*Proof.* Let us arrange the integers of $X$ as a sorted sequence of consecutive words of $\log u$ bits each. Consider the leftmost $[\log n]$ bits of each integer $x_i$, called $h_i$, where $1 \leq i \leq n$. We say that any two integers $x_i$ and $x_j$ belong to the same superblock if $h_i = h_j$. For example, assuming $\log n = 3$ and $\log u = w = 5$, then values 2, 10, 22 all are in different superblocks, while 4 and 6 are in the same one (their leftmost 3 bits out of 5 coincide).

The sequence $h_1 \leq h_2 \leq \cdots \leq h_n$ can be stored as a bitvector $H$ in $3n$ bits, instead of using the standard $n[\log n]$ bits. The representation is unary, in which an integer $x \geq 0$ is represented with $x$ 0s followed by a 1. Namely, the values $h_1, h_2 - h_1, \ldots, h_n - h_{n-1}$ are stored in unary as a bitvector. In other words, we can start from an all-zero bitvector of length $n + 2^{\lceil \log n \rceil}$ and for each $h_i$ we set the bit $h_i + i, i \geq 0$). For example, the sequence $h_1, h_2, h_3, h_4, h_5 = 1, 1, 2, 3, 3$ is stored as $H = 01101011$. Note that the position of the $i$th 1 in $H$ corresponds to $h_i$, and the number of 0s from the beginning of $H$ up to the $i$th 1 gives $h_i$ itself. The remaining portion of the original sequence, that is, the last $\log u - \lceil \log n \rceil$ bits in $x_i$ that are not in $h_i$, are stored as the $i$th entry of a simple array $L$. Hence, we can reconstruct $x_i$ as the concatenation of $h_i$ and $L[i]$, for $1 \leq i \leq n$. The total space used by $H$ is at most $2^{\lceil \log n \rceil} + n \leq 3n$ bits and that used by $L$ is $n(\log u - \lceil \log n \rceil) \leq n \log(u/n)$ bits.

The plain storage of the bits in $L$ is related to the information-theoretic minimum, namely, $n \log(u/n) \leq B(n, u)$. To see that, recall that

$$B(n, u) = n \log \left( \frac{u}{n} \right) + (u - n) \log \left( \frac{u}{u - n} \right) - O(\log u).$$

For $n \leq u/2$ the term $(u - n) \log \left( \frac{u}{u - n} \right) - O(\log u)$ is always asymptotically positive (the minimum is located at $n = u/2$). Since the space to represent $H$ is upper bounded by $3n$, the total footprint of the data structure is $B(n, u) + O(n)$ bits.

\[ \text{Here we use Elias’ original choice of ceiling and floors, thus our bounds slightly differ from the sdarray structure of [OS07], where they obtain } n[\log(u/n)] + 2n. \text{ We also assume that the most significant bit of a word is the leftmost one.} \]
Proof of Theorem 2.1

To support rank and select, we store \( H \) using the technique described in [BM99]:

**Theorem 2.3.** Given a bitvector \( B \) of length \( v \) there exists a systematic rank and select data structure using \( O(v \log \log / \log v) \) bits of redundancy supporting rank and select on \( B \) in \( O(1) \) time.

Hence, we are now able to perform all \texttt{fid} operations on \( H \) using \( o(n) \) additional bits of redundancy. To compute \( \texttt{select}_1(q) \) in the main Elias-Fano data structure, we operate as follows: we first compute \( x = (\texttt{select}_1(q) - q) \times 2^{|\log n|} \) on \( H \) and then we return \( x + L[q] \). The rationale behind the computation of \( x \) is as follows: \( \texttt{select}_1(q) \) over \( H \) returns the value of \( h_q + q \), since there is one 1 for each of the \( q \) elements in \( X_0, \ldots, X_{q-1} \), plus a 0 each time the superblock changes. Hence, \( \texttt{select}_1(q) - q \) is the superblock value \( h_q \), i.e. the \( \lfloor \log n \rfloor \) bits of \( X_q \). The remaining bits of \( X_q \) are stored verbatim in \( L[q] \), so it suffices to shift \( h_q \) to the left and add the two. The time to compute \( \texttt{select}_1 \) is the time of computing the same operation over \( H \) plus a read in \( L \), namely \( O(1) \) time.

To compute \( \texttt{rank}_1(p) \) on \( X \), let \( \mu = \lfloor u/2^{\lfloor \log(n) \rfloor} \rfloor \). We know that position \( p \) belongs to superblock \( h_p = \lfloor p/\mu \rfloor \). We first find the value of \( y = \texttt{rank}_1(p \mod \mu) \) on \( X \) and then add the remaining value. The former quantity can be obtained by performing \( y = \texttt{select}_0(h_p) - h_p \) on \( H \): there are \( h_p \) 0s in \( H \) that lead to superblock \( h_p \) and each 1 before the position of the \( h_p \)th 0 represent an element of \( X \). The value \( y \) also indicates who is the first position in superblock \( h_p \) in \( L \). Computing \( y' = \texttt{select}_0(h_p + 1) - h_p - 1 \), still over \( H \), finds the ending part, so that \( X[y..y'-1] \) all share the leftmost \( \lfloor \log n \rfloor \) bits. Finding the difference of \( y \) and \( \texttt{rank}_1(p) \) on \( X \) now boils down to finding the predecessor of \( p \mod \mu \) in \( L[y..y'-1] \), which we do via simple binary search. The first operations on \( H \) still account for \( O(1) \) time, but each superblock can potentially fit \( u/n \) over items, which the binary search requires \( \log(u/n) + O(1) \) steps to explore. Hence, the total time for \( \texttt{rank} \) is \( O(\log(u/n)) \).

### 2.5.2 Relations with predecessor problem

The predecessor problem is an ubiquitous problem in computer science, and is tightly connected to \( \texttt{rank} \) and \( \texttt{select} \) data structures. As such, it makes sense to introduce it here:

**Definition 2.5.** The predecessor problem is defined as follows:

**Input:** \( X \), a sorted subset of a known finite universe set \( U \subseteq \mathbb{N} \).

**Query:** \( \texttt{pred}(q) \), where \( q \in U \), returns \( \max\{x \in X | x \leq q\} \).
For example, let $X = \{1, 4, 7, 91\}$ where $U = [100]$. Then, $\text{pred}(4) = 4$ and $\text{pred}(10) = 7$. It is usually assumed that if $q$ is less than any element in $X$, then $\text{pred}(q) = -1$. Also, predecessor search is usually intended in a systematic fashion, i.e., a plain copy of the input data is always accessible. Solutions to predecessor search have a large impact on the algorithmic and related communities, being a widely known problem. As an example, consider network routers: each router possesses a set of destination networks in the form IP address / netmask, each associated to a specific network interface. As a bitwise and of IP address and netmask delivers the lowest IP address appearing in the destination network. If those lowest IP addresses are stored in a predecessor network, routing an incoming packet is reduced to finding the lowest IP address for a given destination IP address, i.e. a predecessor query. Given the size of nowadays routing tables, providing space efficient and fast predecessor search is important: data structures for rank and select with fast operations can supply that.

Predecessor search is a very well studied algorithmic problem. Different data structures exist since the seventies: beyond classical searching algorithms and k-ary trees, we find skiplists [Pug90], van Emde Boas trees [vEBKZ77], x- and y-fast tries [Wil83] up to the latest developments of [PT06b]. Searching a predecessor is immediately connected to a pair of rank and select queries, as

$$\text{pred}(i) = \text{select}_1(\text{rank}_1(i))$$

holds. The connection with rank/select data structures also extends to lower bounds. The first non-trivial lower bound in the cell probe model is due to Ajtai [Ajt88], later strengthened by Miltersen et al. [MNSW95] and further by Beame and Fich [BF02]. Matching lower and upper bounds were finally presented by Pătraşcu and Thorup [PT06a, PT06b], providing a multiple-case space-time trade-off. On the lower bound side, they prove:

**Theorem 2.4.** For a data structure using $b \geq nw$ bits of storage, the time to perform rank is no less than

$$\min \left\{ \begin{array}{l}
\log_n n \\
\log \left( \frac{\log(u/n)}{\log(b/nw)} \right) \\
\log \left( \frac{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)}{\log \left( \frac{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)}{\log \left( \frac{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)}{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)} \right)} \right)} \\
\log \left( \frac{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)}{\log \left( \frac{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)}{\log \left( \frac{\log(u/n)}{\log(b/nw)} \right)} \right)} \right)
\end{array} \right\}$$

A direct consequence of Theorem 2.4 has been stressed out multiple times in the literature (see [GRR08]):
Theorem 2.5. Given $|U| = u$ and $|X| = n$, rank and select operations for non-

systematic data structures in $O(1)$ time and $o(n)$ redundancy are possible only when

$n = O(\log^{O(1)} u)$ or $u = O(n \log^c n)$, for some constant $c > 1$.

A much stronger lower bound exists for systematic data structures (see [Gol07a]):

Theorem 2.6. Given $|U| = u$, rank and select operations for systematic data structures in $O(t)$ time require a redundancy of $\Omega((u \log u)/t \log u))$.

A matching upper bound has been proposed in [Gol07a] itself, practically closing

the problem when $n$ is not considered as a part of the equation. Further work for

so-called density-sensitive lower bounds was presented in [GRR08], yielding to:

Theorem 2.7. Given $|U| = u$, $|X| = n$, rank and select operations for systematic data structures in $O(t)$ time require redundancy $\Omega(r)$ where

\[
  r = \begin{cases} 
  \Omega \left( \frac{n}{t} \log \left( \frac{nt}{u} \right) \right) & \text{if } \frac{nt}{u} = \omega(1) \\
  \Omega(n) & \text{if } \frac{nt}{u} = \Theta(1) \\
  \Omega \left( n \log \left( \frac{u}{nt} \right) \right) & \text{if } \frac{nt}{u} = o(1)
  \end{cases}
\]

2.5.3 Weaker versions

The original problem [RRR07] for rank/select data structures was also stated in a

weaker version:

Definition 2.6. The weak rank/select problem requires to build a data structure

as follows:

Input: Set $U$ of size $u$ and set $X \subseteq U$ of size $n$.

Queries: select$_1(x)$ and rank$_1^-(q)$ where

$$
  \text{rank}_1^-(q) = \begin{cases} 
  i \text{ s.t. } X_i = q & \text{if } q \in X \\
  \bot & \text{otherwise}
  \end{cases}
$$

Data structures solving the weak rank/select problem are usually called index-

able dictionaries.

Considering select$_0$ is not supported and rank$_1^-$ can answer, essentially, arbitrary values when the query is not in $X$, this is enough to invalidate predecessor lower bounds. It turns out [RRR07, Pag01] that a clever combination of perfect hashing schemes and bucketing techniques proves the following:
Theorem 2.8. There exists an indexable dictionary that uses $B(n,u) + o(n) + O(\log \log u)$ bits of space and has query time of $O(1)$ in the RAM model.

Indexable dictionaries have a narrower application range w.r.t. to fully indexable ones, but they are still useful as basic data structures: when select$_0$ is not required, an indexable dictionary query solves the same problem of order-preserving minimal perfect hash functions [BPZ07, MWHC96] with membership queries and, in the way, adding compression. An example application is well known in information retrieval: consider having a static, precomputed dictionary of strings $S$. Given a text $T$ made of arbitrary words (some of which are in $S$) one can translate words as dictionary indexes as follows: a reasonably wide output hash function (such as CRC64) is employed, $U$ is set to $2^{64}$ and $X$ is set to be the set of fingerprints built over elements of $S$ (considering it collision-free for simplicity). Then, an indexable dictionary can be used to iterate over the words in $T$ and transform the words into $S$ in their indexes in the dictionary.

Further improvements in this scenario were made by Belazzougui et al. [BBPV09]. The authors provide data structures implementing the $\text{rank}^-$ operation of indexable dictionaries with the typical twist of hash functions: they always answer $\text{rank}^-(x)$ for any input $x$, except the output value is correct iff $x \in X$. The authors introduce a data structure called monotone minimal perfect hash, of which multiple flavours exist:

Theorem 2.9. Given a universe $U$ of size $u$ and a subset $X \subseteq U$ of size $n$, there exist monotone minimal perfect hash functions on a word RAM of size $w = \Omega(\log u)$, answering $\text{rank}^-(x)$ correctly for $x \in X$ in

- $O(\log u/w)$ time, occupying $O(n \log \log u)$ bits,
- $O(\log u/w + \log \log u)$ time, occupying $O(n \log^{(3)} u)$ bits.

The data structures of Theorem 2.9 data structures are designed as pure indexes, i.e. they do not require access to the real data. As a consequence, any data structure performing select$_1$ would be sufficient to build an indexable dictionary out of the original results by Belazzougui et al. Using simple solutions, one can reach the following:

Theorem 2.10. Let $s_{mmphf}$ and $t_{mmphf}$ be, respectively, space and time complexities of a data structure as described in Theorem 2.9. Given an universe $U$ of size $u$, where $u \leq 2^w$ and a subset $X \subseteq U$ of size $n$, there exist indexable dictionaries that use

$$n \log(u/n) + O(n + s_{mmphf})$$

bits of space, answering select$_1$ and $\text{rank}^-$ over $X$ in $O(1)$ time and $O(t_{mmphf})$ respectively.
Proof. Use the data structure of Table 2.1, line 1, to store $X$ and implement $\text{select}_1$ over it in $O(1)$ time. $\text{rank}^-$ is implemented using the given data structure. □

2.5.4 Basic employment and usage

Predecessor problems and rank/select data structures are at the heart of identifier remapping: consider the universe $U$ of elements and the set $X$ of extracted elements. For concreteness, consider a database where the rowset of a table is indexed by $U$ and a subset of the rows is indexed by $X$. Here, $\text{select}_1(x)$ remaps the rank of a column in the sub-table as the rank of the column in the original table, and $\text{rank}_1$ vice-versa. This kind of operation in the static setting is ubiquitous in massive data sets analysis, in which projecting data dimensions and selecting sub-sets of sorted data is a frequent operation.

Inverted indexes

Inverted indexes [WMB99] [BYRN11] are both a theoretical and practical example. In information retrieval, a set of documents $D$ is given, where each document $D_i$ may be described, at the very least, as a “bag of words”, i.e. a set (in the proper mathematical sense) of unique words taken from some vocabulary $W$. In an inverted index the relationship is transposed, i.e. a set of so-called posting lists $L = \{L_0, L_1, L_2, \ldots, L_{|W|-1}\}$ are stored on disk, where, for each $j \in [W]$, $L_j = \{i | j \in D_i\}$. $L_j$ contains then the set of indexes of documents containing the word $j$ at least once. Posting lists are stored as a sorted sequence so that answering boolean queries of the form “Which documents in $D$ contain word $a$ and (resp. or) $b$?” boils down to intersection (resp. union) of lists $L_a, L_b$.

Storing $L_j$ for some $j$ requires to store a subset of the indexes of $D$, and can be solved through binary $\text{rank}$ and $\text{select}$ data structures to achieve zero-th order entropy compression. Moreover, the intersection algorithm benefits from the $\text{rank}$ operation to be sped up. The “standard” compression algorithm for posting lists is to use a specific, statically modelled compressor [WMB99], such as Golomb codes or Elias codes, which are developed to encode a natural integer $x \geq 0$ in roughly $\log(x + 1)$ bits of space. The encoder is then fed with the gap sequence of each posting list, i.e. the sequence $G(L) = L_{j0}, L_{j1} - L_{j0}, L_{j2} - L_{j1}, \ldots$. The rationale is that the denser the posting list is (even just in some region), the smaller the gaps will be, leading to a compact encoding.

Some authors [Sad03] [GGV04] [MN07] [GHSV07] [BB04] also tried to perform $\text{rank/select}$ over gap encoding, to obtain a more data-aware compression level, similar to traditional approaches, while retaining enhanced capabilities. In fact,
2.5. ON BINARY RANK/SELECT

for a set $L$ of length $n$ from the universe $[u]$, defining $\text{gap}(L) = \log(L_0 + 1) + \sum_{k=1}^{n-1} \log(L_k - L_{k-1} + 1)$, it is easy to see that $\text{gap}(L) \leq n \log(u/n) + O(n)$, since the maximum of $\text{gap}(L)$ occurs when all the gaps are equal to $u/n$. All in all, the following theorems where proved in [GHSV06] and [MN07] respectively, the first being a space effective and the second one being a time effective result:

**Theorem 2.11.** Given a bitvector $B$ of length $u$ with cardinality $n$, let $L$ be the set represented by $B$ as in Definition 2.4. Then, there exists a non-systematic data structure occupying $(1 + o(1))\text{gap}(L)$ bits of space and performing rank and select in $o(\log^2 \log n)$ time.

**Theorem 2.12.** Given a bitvector $B$ of length $u$ with cardinality $n$, let $L$ be the set represented by $B$ as in Definition 2.4. Then, there exists a non-systematic data structure occupying $\text{gap}(L) + O(n) + o(u)$ bits performing rank and select in $O(1)$ time.

It is worth noting that, in Theorem 2.12, the $O(n)$ term comes from the specific encoding that has been chosen for the gap sequence and invalidates the possibility to have an $o(n)$ redundancy when the bitvector is dense enough. Whether a more compact code is available for these schemes seems to be an open problem.

**Succinct trees**

Fully indexable dictionaries are bread and butter of a variety of succinct data structures. Overall, the most studied seem to be succinct trees, both cardinal and ordinal. As discussed before, Lemma 2.1 states that binary trees require $2n - \Theta(\log n)$ bits to be represented. In fact, there exist a number of bijections between binary strings of $2n$ bits and ordinal trees, which can be exploited to represent shapes of binary trees. The first one [Jac89] is BP (Balanced Parentheses) which introduces a bijection between certain sequences of parentheses and shapes of binary trees. A straightforward bijection between parentheses and binary strings is the final step to obtain a representation of a tree. One of the most studied and applied [BDM+05, JSS12, Fis10] is, instead, DFUDS: Depth First Unary Degree Sequence, defined as follows [BDM+05]:

**Definition 2.7.** Let $T$ be an ordinal tree. The function $\text{Enc}(v)$ for node $v \in T$ outputs the 0-based unary representation of the degree count of $v$ expressing such value with a sequence of $[\text{]}$ terminated with a $\text{]}$. Let $v_0, v_1, \ldots$ be the sequence of nodes in $T$ as a result of a depth first visit of the tree (preorder for binary trees). The DFUDS encoding of $T$ is the sequence $[\text{]}\text{Enc}(v_0)\text{Enc}(v_1)\text{Enc}(v_2)\ldots$. 
For example the tree of Figure 2.1 is represented as follows:\footnote{Historically, trees are mapped to strings of parentheses, which in turn are represented as binary strings. For consistency, we show parenthesis sequences.}

$$(((())())())()()()$$

DFUDS representations can be stored with additional redundancy and provide multiple operations in constant (or almost constant) time: fully indexable dictionaries are involved when performing basic navigational and informational operations over trees.

Although data structures using DFUDS were believed to be the most functionality-rich, the work by Navarro and Sadakane [NS10] disproved it, reaching the following result using a BP-based representation:

\begin{theorem}
On a Word-RAM model, there exists a succinct data structure that supports the following operations in $O(1)$ time and $2n + O(n/\log^{\Theta(1)} n)$ space:
\end{theorem}
The natural question then arises: what is the best way to encode a tree in terms of functionality? A related answer, quite surprising, is given in [FRR09], where the authors prove that one can emulate many known representations (BP, DFUDS, micro-tree decompositions [FM08a]) while storing just one.

Another result that helped better understand the compressibility of succinct trees is due to Jansson et al. [JSS12] and their ultra-succinct representation of ordinal trees. Instead of considering to represent the whole classes of trees, the authors devise an encoding mechanism that represents a tree matching a degree-aware form of entropy:

**Theorem 2.14.** For an ordered tree $T$ with $n$ nodes, having $n_i$ nodes with $i$ children,
let 
\[ H^*(T) = \frac{1}{n} \sum_i n_i \log \left( \frac{n}{n_i} \right). \]

Then, there exists a data structure that uses \( nH^*(T) + o(n) \) bits and supports navigational, LCA and level-ancestor queries on a tree in \( O(1) \) time.

Note that the definition of \( H^*(T) \) closely resembles the one of Definition 2.2, and can be much better than the standard bound of \( \sim 2n \) bits. As an example, a binary tree with no node of degree 1 (except the root) requires no more than \( n + o(n) \) bits (see [JSS12] for a proof).

**Succinct relations**

`rank/select` data structures can be employed also to represent succinct relations: a binary relation is defined as an \( a \times b \) binary matrix. The basic problem set here, is to support the following operations:

**Definition 2.8.** Input A binary matrix \( R \) of size \( a \times b \) of weight \( n \).

Queries
- **row_rank** \( (i, j) \): find the number of 1-bits in the \( i \)th row of \( R \) up to column \( j \).
- **row_sel** \( (i, x) \): find the column of \( R \) at the \( i \)th row with the \( x \)-th occurrence of \( R \) set to 1, or \(-1\).
- **row_nb** \( (i, j) \): return the cardinality of the \( i \)th row, i.e. **row_rank** \( (i, b) \).
- **col_rank** \( (i, j) \): orthogonal to **row_rank**.
- **col_sel** \( (i, j) \): orthogonal to **row_sel**.
- **col_nb** \( (i, j) \): orthogonal to **row_nb**.
- **access** \( (i, j) \): access \( R_{ij} \) (this is non trivial in non-systematic data structures).

Building data structures for this problem is an intermediate tool for higher complexity data structures that work on two-dimensional data structures (strings [BHMR07a], graphs [PM08d], or in general binary relations [BCN10]).

Given the total size \( u = ab \) of the matrix and its cardinality (cells set to 1) \( n \), the succinct bound of a generic matrix can be set to \( B(n, u) = \log \binom{n}{u} = n \log(u/n) + O(n) \), giving a density-sensitive bound. The redundancy instead can be bound using the work in [Gol09], which formally states the intuition that faster row operations correspond to slower column ones, and vice-versa:
2.5. ON BINARY RANK/SELECT

Theorem 2.15. Let \( t \) be the time to answer \texttt{col\_sel} and \( t' \) be the time to answer \texttt{row\_sel}, and assume that \( a < b \). Also assume that \( t \) and \( t' \) are both \( O((\log(u/n)/\log w)) \), and

\[
\min{tt'} = O \left( \frac{n \log^2(u/n)}{b \log(n/b)w} \right).
\]

Then, on a standard Word-RAM model with word size \( \Theta(\log u) \) the redundancy for a non-systematic data structure must be

\[
\Omega \left( \frac{n \log^2(u/n)}{w^2 t t'} \right).
\]

Rewording Theorem 2.15 yields to the discovery that for polynomially sized matrices (w.r.t. to the cardinality), when \( u = a^{1+\alpha} \) and \( n = a^\beta \) for some \( \alpha < \beta < 1 + \alpha \), if both \( t \) and \( t' \) are \( O(1) \), then linear sized redundancy is needed: \( r = \Omega(n) \) bits.

Multiple designs have been presented to solve the succinct relations problem, leading to different upper bounds.

Dynamization

Succinct data structures are mainly thought for the static case: large data sets are the ones gaining the most significant benefits from data compression and operations on large data sets are still mainly batch \cite{DG08, MAB+09, CKL+06}; operating in batch mode with static data structures can help reducing design issues and produce more compact data structures. Nevertheless recent years have seen an effort in producing succinct data structures whose underlying combinatorial objects can be incrementally updated. The rule here is as follows: let \( C \) the class of combinatorial objects to represent, let \( n_1, n_2, \ldots \) be the size of the combinatorial object after update \( 1, 2, \ldots \), let \( f(n) \) be the function that defines the space occupancy of the data structures for any \( n \). We say the data structure has redundancy \( r(n) \) iff for any \( e \geq 1, f(n_e) = \log |C_{n_e}| + r(n_e), \) where \( C_{n_e} \subseteq C \) is the class of objects of size \( n_e \). Concretely, assume a data structure is initially built for an ordered tree of size \( n \) and uses \( 2n_1 + O(n_1 \log \log n_1 / \log n_1) \) bits and supports insertion of \( m \) nodes adding \( 2m + O(m \log m / \log m) \) bits of space, then the redundancy of the data structure is \( r(n) = O(n \log \log n / \log n) \) for any \( n \).

Dynamization of succinct data structures has been studied for binary rank/select data structures \cite{MN08}, also under the form of partial sums \cite{HSS11}, strings \cite{HM10} and ordered trees \cite{NS10}. The main objective is to keep the data structure succinct (so that the redundancy is at any step a lower order term w.r.t. to the object entropy), and the results are as follows (refer to \cite{MN08} and \cite{NS10} respectively):
Theorem 2.16. Let $B$ be a bitvector before the $i$th update, having length $u_i$, and cardinality $n_i$. Then, there exists a dynamic succinct rank/select data structure over binary sequences that supports rank, select, insertion and deletion of bits at arbitrary positions in $O(\log u_i / \log \log u_i)$ time. The space of the data structures is $\log (\binom{u_i+1}{n_i+1}) + o(u_i)$.

Theorem 2.17. Let $T$ be an ordered tree of $n$ nodes, then there exists a data structure that supports all operations of Theorem 2.13 in $O(\log n_i / \log \log n_i)$ time, except for levelancestor, levellmost and levelrmost that require $O(\log n_i)$ time. Updating is performed by adding or removing arbitrary nodes in $O(\log n_i / \log \log n_i)$ time and $n_i + O(n_i \log \log n_i / \log n_i)$ bits of space.

2.6 Operating on texts

2.6.1 Generic rank/select

Fully indexable dictionaries cover only the binary case. Using Definition 2.2 for empirical entropy, we can extend the concept of succinct data structures for binary alphabets supporting rank/select (recall Definition ??). Operating on a string $S$ of length $n$ over the alphabet $\Sigma = [\sigma]$, a data structure is considered succinct if its space bound is $n \log \sigma + o(n \log \sigma)$ bits. A considerable portion of literature about non-systematic data structures also consider stronger space bounds, providing bounds of the form $nH_0(S) + o(n \log \sigma)$. To understand the different upper bounds, one shall first consider that the space of solutions is partitioned in two classes:

- Polylogarithmic alphabet: when $\sigma = O(\log^{O(1)} n)$, on a Word RAM with size $\Theta(\log(n))$, it is possible to achieve both succinct bounds with constant time rank and select.

- Large alphabet: in the remaining case, the intrinsic time for rank is non-constant for both succinct bounds.

The low execution times that can be achieved on polylogarithmic alphabets are justified by the fact that for $w = \Theta(\log n)$ one word can contain $\omega(1)$ alphabet symbols, so that ad-hoc algorithms can perform parallel computations. The first technique to achieve such running time within succinct bounds appears in [FMMN07]. For larger alphabets on non-systematic data structures the current state of the art is the work of Belazzougui and Navarro [BN11b]. In total, the resulting theorem is:

Theorem 2.18. The arbitrary alphabet rank/select on a string $S$ of length $n$ from alphabet $[\sigma]$ can be solved using
2.6. OPERATING ON TEXTS

1. non-systematic succinct data structures such that for any \( f(n, \sigma) \), where \( f = \omega(1) \) and \( f = o(\log(\log \sigma / \log w)) \), supporting operations access and select in \( O(1) \) and \( O(f(n, \sigma)) \) (or vice-versa), and rank can be performed in \( O(\log(\log w)) \). The data structure uses \( nH_0(S) + o(n(1 + H_0(S))) \) bits of space.

2. systematic succinct data structures such that for a given \( t = \Omega(1) \) and \( t = O(\log \sigma / \log \log \sigma) \) the string \( S \) can be represented with a redundancy of \( O((n/t) \log \sigma) \) bits supporting rank in \( O(\log \log \sigma) \) time and select in \( O(t) \) time.

Furthermore, non-systematic schemes received further attention in terms of compression factors: a series of techniques \([GN06, FV07, SG06, BGNN10]\) can be used to reach \( nH_k(S) \) for any \( k = o(\log n / \log \log \sigma) \). Hence, we can also state:

**Theorem 2.19.** The arbitrary alphabet rank/select on a string \( S \) of length \( n \) from alphabet \([\sigma]\) can be solved in \( nH_k(S) + o(n \log \sigma) \) bits and keep the same running times of Theorem 2.18 (1).

2.6.2 Compressed text indexing

Arbitrary alphabet rank/select data structures have been initially inspired by a related problem:

**Definition 2.9.** The compressed text indexing problem requires building a data structure as follows:

Input An alphabet size \( \sigma \), a text \( T \) of length \( n \) over \([\sigma]\).

Queries count\((P)\), where \( P \) is a string over \( \sigma \) of length \( m \), returns the number of times the string \( P \) appears in \( T \) (if \( P \) overlaps with itself, it is counted multiple times) and locate\((P)\), which outputs all the occurrences in \( T \) where \( P \) is located.

Binary rank/select data structures were used to bring down the space complexity of data structures for text indexing \([GV05]\). A component heavily involved in various solutions for compressed text indexing is the Burrows-Wheeler Transform (henceforth, BWT), which we now introduce, and the backward search algorithm performed over it, described just afterwards. The backward search algorithm exploits the connection between BWT and rank operations to perform time and space efficient compressed text indexing. For the sake of simplicity and given the scope of this thesis, we will restrict to Count\((\cdot)\) only operations. For further discussion on compressed text indexing one can also refer to the survey of \([NM07]\) (although limited to 2006) and newer results.
2.6.3 Burrows-Wheeler Transform

Burrows and Wheeler [BW94] introduced a new compression algorithm based on a reversible transformation, now called the Burrows-Wheeler Transform (BWT from now on). The BWT transforms the input string $T$ into a new string that is easier to compress. The BWT of $T$, hereafter denoted by $\text{bwt}(T)$, consists of three basic steps (see Figure 2.2): (1) append at the end of $T$ a special symbol $\$\$ smaller than any other symbol of $\Sigma$; (2) form a conceptual matrix $M(T)$ whose rows are the cyclic rotations of string $T\$\$ in lexicographic order; (3) construct string $L$ by taking the last column of the sorted matrix $M(T)$. Then it holds $\text{bwt}(T) = L$.

Every column of $M(T)$, hence also the transformed string $L$, is a permutation of $T\$\$. In particular the first column of $M(T)$, call it $F$, is obtained by lexicographically sorting the symbols of $T\$\$ (or, equally, the symbols of $L$). Note that when we sort the rows of $M(T)$ we are essentially sorting the suffixes of $T$ because of the presence of the special symbol $\$\$. This shows that: (1) there is a strong relation between $M(T)$ and the suffix array data structure built on $T$; (2) symbols following the same substring (context) in $T$ are grouped together in $L$, thus giving raise to clusters of nearly identical symbols when the string has low $k$th order entropy. Property 1 is crucial for designing compressed indexes (see e.g. [NM07, FGNV08]), Property 2 is the key for designing modern data compressors (see e.g. [Man01]), since it is related to higher-order empirical entropy. One of the less immediate properties of $\text{bwt}(\cdot)$
is that it is invertible, so that one can compress \( \text{bwt}(T) \) and then rebuild \( T \) out of it. Shortly, when properly partitioned, compressing \( \text{bwt}(T) \) to zero-th order entropy is equivalent to compressing \( T \) to \( k \)-th order entropy for \( k = o(\log_\sigma n) \). To invert \( \text{bwt}(T) \), one can resort to the following properties:

**Lemma 2.2.** For any text \( T \) built over \( \Sigma \),

(a) Since the rows in \( \mathcal{M}(T) \) are cyclically rotated, \( L[i] \) precedes \( F[i] \) in the original string \( T \).

(b) For any \( c \in \Sigma \), the \( \ell \)-th occurrence of \( c \) in \( F \) and the \( \ell \)-th occurrence of \( c \) in \( L \) correspond to the same position in \( T \).

In order to map symbols in \( L \) to their corresponding symbols in \( F \), [FM05] introduced the following function:

\[
\text{LF}(i) = C[L[i]] + \text{rank}_{L[i]}(L, i)
\]

where \( C[c] \) counts the number of symbols smaller than \( c \) in the whole string \( L \). Given Lemma 2.2 and the alphabetic ordering of \( F \), it is not difficult to see that symbol \( L[i] \) corresponds to symbol \( F[\text{LF}(i)] \). For example in Figure 2.2 we have \( \text{LF}(10) = C[a] + \text{rank}_a(L, 10) = 1+5 = 6 \) and, in fact, both \( L[10] \) and \( F[6] \) correspond to the symbol \( T[6] \). Given Lemma 2.2 and the definition of \( \text{LF} \), it is easy to see that \( L[i] \) (which is equal to \( F[\text{LF}(i)] \)) is preceded by \( L[\text{LF}(i)] \), and thus the iterated application of \( \text{LF} \) allows to move backward over the text \( T \). Of course, we can compute \( T \) from \( L \) by moving backward from symbol \( L[1] = T[n] \).

All in all, compression to zero-th order empirical entropy for a \textbf{rank} and \textbf{select} data structure translates immediately in a \( k \)-th order empirical entropy compressor for \( k = o(\log_\sigma n) \), extending Theorem 5.1 to another result of [BN11b]:

**Theorem 2.20.** Given a text \( T \) of length \( n \) over alphabet \( [\sigma] \), for any \( \omega(1) = f(n, \sigma) = o(\log(\log \sigma / \log w)) \), where \( w = \Theta(\log n) \) is the Word RAM size, there exists a non-systematic data structure storing \( T \) in \( nH_k(S) + o(n \log \sigma) \) bits solving

- for \( \sigma = \omega(\log^{O(1)} n) \), \textbf{access} in \( O(1) \) time, \textbf{select} in \( O(f(n, \sigma)) \) time and \textbf{rank} in \( O(\log(\log \sigma / \log w)) \) bits.

- for \( \sigma = O(\log^{O(1)} n) \), \textbf{access}, \textbf{rank} and \textbf{select} in \( O(1) \) time.
Algorithm Count($P[1,p]$)

1. $i = p$, $c = P[p]$, First$_p = C[c] + 1$, Last$_p = C[c + 1]$;
2. while ((First$_i$ $\leq$ Last$_i$) and ($i \geq 2$)) do 
3. $c = P[i - 1]$;
4. First$_{i-1} = C[c] + \text{rank}_c(L, \text{First}_i - 1) + 1$;
5. Last$_{i-1} = C[c] + \text{rank}_c(L, \text{Last}_i)$;
6. $i = i - 1$;
7. if (Last$_i$ $<$ First$_i$) then return “no rows prefixed by $P$” else return [First$_i$, Last$_i$].

Figure 2.3: The algorithm to find the range [First$_1$, Last$_1$] of rows of $\mathcal{M}(T)$ prefixed by $P[1,p]$ (if any).

2.6.4 Backward search

Our purposes are, however, to support the search of any pattern $P[1,p]$ as a substring of the indexed string $T[1,n]$, by requiring a space that is close to the one of the best compressors.

Indeed [FM05] proposed a surprisingly simple algorithm to identify with $O(p)$ steps the range of rows of $\mathcal{M}(T)$ that are prefixed by an arbitrary pattern $P$. In detail, they proved that data structures for supporting rank queries on the string $L$ are enough to search for an arbitrary pattern $P[1,p]$ as a substring of the indexed text $T$. The resulting search procedure, called backward search, is illustrated in Figure 2.3. It works in $p$ phases. In each phase it is guaranteed that the following invariant is kept: At the end of the $i$-th phase, [First$_i$, Last$_i$] is the range of contiguous rows in $\mathcal{M}(T)$ that are prefixed by $P[i,p]$. Count starts with $i = p$ so that First$_p$ and Last$_p$ are determined via the array $C$ (step 1). At any other phase, the algorithm (see pseudo-code in Figure 2.3) has inductively computed First$_{i+1}$ and Last$_{i+1}$, and thus it can derive the next interval of suffixes prefixed by $P[i,p]$ by setting First$_i = C[P[i]] + \text{rank}_{P[i]}(L, \text{First}_{i+1} - 1)$ and Last$_i = C[P[i]] + \text{rank}_{P[i]}(L, \text{Last}_{i+1})$. These two computations are actually mapping (via LF) the first and last occurrences (if any) of symbol $P[i]$ in the substring $L[\text{First}_{i+1}, \text{Last}_{i+1}]$ to their corresponding occurrences in $F$. As a result, the backward-search algorithm requires to solve at most $2p - 2$ rank queries on $L = \text{bwt}(T)$ in order to find out the (possibly empty) range [First$_1$, Last$_1$] of text suffixes prefixed by $P$. The number of occurrences of $P$ in $T$ is, thus, $\text{occ}(P) = \text{Last}_1 - \text{First}_1 + 1$.

In general, we have the following:
Theorem 2.21. Given a text $T[1, n]$ drawn from an alphabet $\Sigma$ of size $\sigma$, there exists a compressed index that takes $p \times t_r$ time to support $\text{Count}(P[1, p])$ where $t_r$ is the time required to perform a rank query.

2.6.5 A more generic framework

The relationship of rank/select data structures with compressed text indexing is very tight: if the main interest is only to implement $\text{count}(\cdot)$, one can rely on the following theorem:

Theorem 2.22. Let us assume there exists a data structure for rank/select on arbitrary alphabet encoding any text $S$ of length $n'$ over alphabet $\sigma'$ in $n'\mathcal{H}_0(S) + n'r(n', \sigma')$ bits, for some concave non-decreasing function $r$, and executes rank in $t(n', \sigma', w)$ time. Then, the Burrows-Wheeler Transform $T^{\text{bwt}}$ of a text $T$ of length $n$ over alphabet $\sigma$ can always be partitioned as $T^{\text{bwt}}_1 \circ T^{\text{bwt}}_2 \circ \ldots \circ T^{\text{bwt}}_m$ for some $m \geq 1$. Using such partition, one can build a data structure that, simultaneously over all $k \leq \alpha \log \sigma n$ for some constant $0 < \alpha < 1$, occupies $n\mathcal{H}_k(T) + nr(n/\sigma^k, \sigma) + o(n)$ bits and solves $\text{count}(P)$ in $O(|P|(1 + t(n, \sigma, w)))$ time.

Proof. The technique to build this data structure is a trivial generalization of the work of [FMMN04]. Given the function $r(\cdot, \cdot)$, we define

$$f(x) = xr(x, \sigma) + (\sigma + 1)\log x,$$

and we go through the same proof of Theorem 5 from [FMMN04], using the given data structure instead of the one contained in the paper. Briefly, the final data structure encodes $T^{\text{bwt}}$ piecewise, using the given rank/select data structure as a black box. Some additional boilerplate is needed to be able to answer rank consistently at any position, yielding the limitation over $k$. Given rank queries, the backward search algorithm of [FM05] performs $O(1)$ rank queries over $T^{\text{bwt}}$ per pattern character, hence the given time performance.

---

6The theorem requires the redundancy to be based on length and alphabet only. Nonetheless, many existing data structures in the RAM model also require access to precomputed tables which are usually based on the word size or other non-input specific parameters. The theorem still applies as long as such extra tables may be shared across all instances of the same data structure.

7Here $\circ$ denotes juxtaposition of two strings.
However the above approach is limited by how fast one can perform \texttt{rank}. Just recently, [BN11a] proved that in general,

\textbf{Theorem 2.23.} For a Word RAM of size $w$ any data structure using $O(n \log^{O(1)} n)$ space to represent a text of length $n$ from alphabet $[\sigma]$, where \(\sigma = \omega(w^{O(1)})\), needs time $\Omega(\log(\log \sigma / \log w))$ to solve \texttt{rank}. For $\sigma = O(w^{O(1)})$ the lower bound is trivial.

Combining Theorems 2.23 and 2.22, one could argue that the minimum time for a data structure solving the \texttt{count}-ing only version of the compressed text indexing can be obtained by employing the solution of Theorem 2.18. Such conjecture has been disproved again in [BN11a].
Chapter 3

Improving binary rank/select

This chapter deals with a new binary rank and select data structure, described in Section 3.4, building over the Elias-Fano scheme of Section 2.5.1. During the chapter we will consider a bitvector $B$ of length $u$ and cardinality $n$, which is equivalent (see Definition 2.4) to representing a set $X = \{x_1 < x_2 < \cdots < x_n\}$ of $n$ elements coming from a universe $U$ of size $u$, assuming $n \leq u/2$. We will consider non-systematic data structures only and assume, where not stated, to work on a Word RAM of word size $w = \Theta(\log u)$. The contents of this chapter mainly follow those of [GORR09]; Section 3.5 is inspired by the works of [GRR08] later revised in [GORR12].

3.1 Our results

In this chapter, our main result is given by the following two theorems, the first being targeted at a broader range of parameters and the latter at lower redundancies and higher times.

**Theorem 3.1.** For any bitvector $S$, $|S| = u$, having cardinality $n$, let $s$ be an integer such that $s = O(\log \log u)$ and let $0 < \varepsilon, \delta \leq 1/2$ be parameters. There exists a fully indexable dictionary solving rank and select operations in time $O(\log 1/(\delta^s \varepsilon))$ and

$$B(n, u) + O(n^{1+\delta} + n \log \log u + n^{1-s\varepsilon} u^\varepsilon)$$

bits of space.

**Theorem 3.2.** For any bitvector $S$, $|S| = u$, having cardinality $n$, let $s > 0$ be $\Theta(1)$ and let $0 < \varepsilon, \delta \leq 1/2$ be two parameters such that either

- $\log \frac{1}{\varepsilon \delta} = o(\log \log n)$ and $u = O(n^{O(1)})$, or
- $u = O(n^{2\log n/\log \log n})$. 
Then, there exists a fully indexable dictionary supporting rank and select in $O(\log(\frac{1}{\varepsilon}))$ and

$$B(n, u) + O(n^{1+\delta} + n^{1-s}\varepsilon)$$

bits of space.

A simplified bound from Theorem 3.1 also exists, where we just set $s = 1$ and $\delta = \varepsilon$:

**Theorem 3.3.** Let $0 < \varepsilon \leq 1/2$ be a real (also $o(1)$). For any bitvector $S$, $|S| = u$, of cardinality $n$, there exists a FID solving operations in time $O(\log(\varepsilon^{-1}))$ and using $B(n, u) + O(n^{1+\varepsilon}) + O(n^{1-\varepsilon}\varepsilon)$ bits.

To better understand the role of our data structure among the variety of results presented in Section 2.5 we can reason as follows. The dense case, where $u = O(n \log^{O(1)} n)$ is already covered tightly by Table 2.1 (row 2). The case $u = \Omega(n^{1+\alpha})$ for some $\alpha > 0$, is partially covered by the upper bounds presented in [PT06b], since the data structure is space/time optimal but does not support select. Our improvements are then twofold: for the case of polynomial universes, we can build data structures with the same time bounds of [PT06b] while supporting select. Then, we can actually perform better than [PT06b] and Table 2.1 (row 2) in the gap where $u = o(n^{1+\varepsilon})$ but $u = \omega(n \log^{O(1)} n)$. Such operation is possible since the data structures of [PT06b] are optimal w.r.t. the lower bound of Theorem 2.4, which requires at least $n$ words of storage. Indeed, let

$$u = n \log^{\sqrt{\log n}} n$$

and assume that in our case $\delta = \varepsilon = 1/\log n$ and $s = 1$. Then, we can perform rank and select in $O(\log n \log \log n)$ time and $B(n, u) + O(n)$ bits. Note that $B(n, u) = O(n \sqrt{\log n} \log \log n) = o(n \log u)$, so that less than $n$ words of $\Theta(\log u)$ bits are used for the whole data structure. The upper bound of Table 2.1 (row 2), when set to the same space complexity, will produce a data structure that will perform rank in $O(\sqrt{\log n})$ time, i.e. slower than us. Also, we are able to support select in the same time as the one of rank, which is a non-trivial extension.

While $\delta$ and $\varepsilon$ can frequently be $o(1)$ in the need of smaller footprints, the parameter $s$ has a more subtle relation, and is almost always kept as $\Theta(1)$ or even just 1: $s$ is driven by the number of applications of our main Theorem (described later), and $s = 1$ is usually sufficient.

### 3.2 Relationship with predecessor search

This section further explores the relationship among rank and select and predecessor search, introducing a simple result linking predecessor search also to select.
3.2. RELATIONSHIP WITH PREDECESSOR SEARCH

Answering \( \text{rank}_1(k) \) in \( X \) is equivalent to finding the predecessor \( x_i \) of \( k \) in \( X \), since \( \text{rank}_1(k) = i \) when \( x_i \) is the predecessor of \( k \). Note that \( \text{rank}_0(k) = k - \text{rank}_1(k) \), so performing this operation also amounts to finding the predecessor. As for \( \text{select}_0(i) \) in \( X \), let \( \overline{X} = [u] \setminus X = \{ v_1, v_2, \ldots, v_{u-n} \} \) be the complement of \( X \), where \( v_i < v_{i+1} \), for \( 1 \leq i < u-n \). Given any \( 1 \leq i \leq u-n \), our goal is to find \( \text{select}_0(i) = v_i \) in constant time, thus we can freely assume that \( n \leq u/2 \) is w.l.o.g.: whenever \( n \leq u/2 \), we store the complement set of \( X \) and swap the zero- and one-related operations.

The key observation comes from the fact that we can associate each \( x_l \) with a new value \( y_l = |\{ v_j \in \overline{X} \text{ such that } v_j < x_l \}| \), which is the number of elements in \( \overline{X} \) that precede \( x_l \), where \( 1 \leq l \leq n \). The relation among the two quantities is simple, namely, \( y_l = x_l - l \), as we know that exactly \( l-1 \) elements of \( X \) precede \( x_l \) and so the remaining elements that precede \( x_l \) must originate from \( \overline{X} \). Since we will often refer to it, we call the multiset \( Y = \{ y_1, y_2, \ldots, y_n \} \) the dual representation of the set \( X \).

Returning to the main problem of answering \( \text{select}_0(i) \) in \( X \), our first step is to find the predecessor \( y_j \) of \( i \) in \( Y \), namely, the largest index \( j \) such that \( y_j < i \). As a result, we infer that \( x_j \) is the predecessor of the unknown \( v_l \) (which will be our answer) in the set \( X \). We now have all the ingredients to deduce the value of \( v_l \). Specifically, the \( y_{i/2} \)th element of \( \overline{X} \) occurs before \( x_j \) in the universe, and there is a nonempty run of elements of \( X \) up to and including position \( x_j \), followed by \( i - y_j \) elements of \( \overline{X} \) up to and including (the unknown) \( v_l \). Hence, \( v_l = x_j + i - y_j \) and, since \( y_j = x_j - j \), we return \( v_i = x_j + i - x_j + j = i + j \). (An alternative way to see \( v_i = i + j \) is that \( x_1, x_2, \ldots, x_j \) are the only elements of \( X \) to the left of the unknown \( v_i \).) We have thus proved the following:

**Lemma 3.1.** Using the Elias-Fano encoding, the \( \text{select}_1 \) operation takes constant time, while the \( \text{rank} \) and \( \text{select}_0 \) operations can be reduced in constant time to predecessor search in the sets \( X \) and \( Y \), respectively.

The following theorem implies that we can use both lower and upper bounds of the predecessor problem to obtain a FID, and vice versa. Below, we call a data structure storing \( X \) set-preserving if it stores \( x_1, \ldots, x_n \) verbatim in a contiguous set of memory cells.

**Theorem 3.4.** For a given set \( X \) of \( n \) integers over the universe \([u]\), let FID \((t,s)\) be a FID that takes \( t \) time and \( s \) bits of space to support \( \text{rank} \) and \( \text{select} \). Also, let \( \text{pred}(t,s) \) be a static data structure that takes \( t \) time and \( s \) bits of space to support predecessor queries on \( X \), where the integers in \( X \) are stored in sorted order using \( n \log u \leq s \) bits. Then,
1. given a FID \((t, s)\), we can obtain a \(\text{PRE}(O(t), s)\);

2. given a set-preserving \(\text{PRE}(t, s)\), we can obtain a FID \((O(t), s-n \log n + O(n))\)

3. if there exists a non set-preserving \(\text{PRE}(t, s)\), we can obtain a FID \((O(t), 2s + O(n))\) with constant-time \(\text{SELECT}_1\).

**Proof.** The first statement easily follows by observing that the predecessor of \(k\) in \(X\) is returned in \(O(1)\) time by \(\text{SELECT}_1(S, \text{RANK}_1(S, k-1))\), where \(S\) is the characteristic bitvector of \(X\). Focusing on the second statement, it suffices to encode \(X\) using the Elias-Fano encoding, achieving space \(s-n \log n + O(n)\).

We review the second statement more thoroughly. By Lemma 3.1, it suffices to encode \(X\) by the Elias-Fano encoding of Section 2.5.1, so the space becomes \(s-n \log n + O(n)\) bits. This allows us to search for the predecessor on \(X\) in \(O(t)\) time since each \(x_i\) can be retrieved in \(O(1)\) time using \(\text{SELECT}_1\) on Elias-Fano (see Section 2.5.1). However, we do not want to replicate data for storing also \(Y\), but we exploit the fact that \(y_i = x_i - i\) for \(1 \leq i \leq n\).

However, the integers in \(Y\) are in non-decreasing order while the predecessor data structure might have been designed to work on a set of distinct integers. Hence, we split \(X\) into two sorted sub-sequences, namely \(X'\) and \(X''\), dividing the elements of \(X\) according to their homologous entries in \(Y\). Namely, \(X' = \{x_i \mid y_i \neq y_{i+1} \text{ for } 1 \leq i \leq n-1\} \cup \{x_n\}\) and \(X'' = X \setminus X'\). In this way, the corresponding set \(Y' = \{y_i \in Y \mid x_i \in X'\}\) contains only the distinct values of \(Y\). We now store \(X'\) and \(X''\) with two separate Elias-Fano encoding, as well as a bitvector \(\pi\) of length \(n\) such that \(\pi[i] = 1\) if \(x_i \in X'\) and \(\pi[i] = 0\) if \(x_i \in X''\), for \(1 \leq i \leq n\). Note that a constant-time FID on \(\pi\) requires \(O(n)\) bits. In other words, storing \(X'\) and \(X''\) with Elias-Fano, \(\pi\) and its FID do not change the space complexity of \(\mathbf{B}(n, u) + O(n)\) of storing \(X\) with Elias-Fano.

First, we observe that a predecessor search for \(k\) in \(X\) can be implemented by finding the predecessors of \(k\) in both \(X'\) and \(X''\), returning the largest of the two.

Second, we observe that \(Y\) does not need to be stored. Indeed, the predecessor \(y_i\) of \(k\) in \(Y\) is also the predecessor of \(k\) in \(Y'\), and the corresponding element \(x_i \in X\) should belong to \(X'\) by construction.

Third, \(Y'\) does not need to be stored either. In fact, we know that \(y_i = x_i - i\), and so we can perform a search on \(X'\) as if we had \(Y'\) available, since they are in one-to-one correspondence. However, \(x_i\) does not necessarily have rank \(i\) in \(X'\) (it has rank \(i\) in \(X\)) but it has a certain rank, say \(i'\) in \(X'\). We then use the FID on \(\pi\) to get \(i = \text{SELECT}_1(i')\). Hence, given \(x_i \in X'\), we can reconstruct the corresponding \(y_i \in Y'\) in \(O(1)\) time.
In summary, we are able to perform the predecessor queries on $X$ and $Y$ in $O(t)$ time, just using the Elias-Fano encoding of $X'$ and $X''$ and the FID on $\pi$, totaling $s - n \log n + O(n)$ bits of space.

If the predecessor structure is not set-preserving, then we have to store both $X$ (split) and $Y'$, so that space bound is doubled. \hfill \Box

### 3.3 Searching the integers

We now describe some techniques used to search integers, i.e. to perform predecessor search on a set of integers. The problem is very well studied and differs from the succinct rank and select data structure problem because it does not necessarily involve compression and separation between rank and select operations.

Predecessor search in the static setting was mainly associated with standard data structures in the beginning: in the comparison model the best option is a balanced search tree, that obtains $O(\log n)$ time to search among $n$ elements. With $O(1)$ Word RAM time per comparison, as in the case of integers, the result also holds in the Word RAM model.

#### 3.3.1 Previous work: searching in $O(\log \log u)$ time

Better data structures follow the same limited universe hypothesis of rank and select data structures: setting the universe size $u$, the van Emde Boas tree (also called vEB tree) of \cite{vEBKZ77} implements the following:

**Theorem 3.5.** A sequence $X$ of $n$ integers from a universe $U$ of size $u$ can be searched for predecessor in $O(\log \log u)$ time and $O(uw)$ bits on a Word RAM of size $w \geq \log u$.

In the static setting this is not interesting because using $O(uw)$ bits one could explicitly state the predecessor of any $x \in U$, but the space of the vEB tree can be lowered to $O(n \log u)$ bits and keep $O(\log \log u)$ search time. The data structure is then called $y$-fast trie (see \cite{Wil83}), and can be seen as a structural improvement upon van Emde Boas, although it implements the same search principle. We now briefly summarize the main ideas behind the $y$-fast trie construction, since we will need it later. The gist of the idea to reach $O(\log \log u)$ time is the following: given a query $q$ of $\log u$ bits, we perform a binary search over the bit representation of $q$ to compute the maximum length of the longest common prefix (LCP) between $q$ and the whole set $X$, namely finding the element $x \in X$ that shares the longest common prefix with $q$ (overloading notation we indicate this by $\text{LCP}(q,X)$). The original solution of \cite{Wil83} lowers the space to $O(n \log^2 u)$ bits. We now show one
of the possible, known, ways on how to bring it down to $O(n \log u)$, restricting to the static setting. Instead of storing $X$ in the $y$-fast trie, we partition $X$ into $\Theta(n/\log^2 u)$ chunks of $\Theta(\log^2 u)$ elements each. We then choose both the smallest and largest element of each subset to be part of a new set $Y$. We store $Y$ in the $y$-fast trie, adding the necessary pointers to the beginning of a chunk in the hash table records. The algorithm then finds the correct chunk from the $y$-fast trie and performs a binary search in the array. The first step has (in the static case) a worst-case time of $O(\log \log u)$ due to the time to search the $y$-fast trie and then another $O(\log \log u)$ to finish the search in $X$. The space is now down to $O(n \log u)$ bits.

Hence:

**Theorem 3.6.** A sequence $X$ of $n$ integers from a universe $U$ of size $u$ can be searched for predecessor in $O(\log \log u)$ time and $O(n \log u)$ bits on a Word RAM of size $w \geq \log u$.

In a static setting $y$-fast tries/van Emde Boas trees can be pushed [PT06b] to match the second branch of the lower bound of Theorem 2.4, hence giving:

**Lemma 3.2.** There exists a data structure that can store a set $X$ of $n$ integers from an universe $U$ of size $u$ on a Word RAM of $w = \Omega(\log u)$ bits in $z > n \log u$ bits and perform predecessor search in $t = O(\log(\log(u/n)/\log(z/(nw))))$ time, for any $z$.

Reversing the formula

$$t = O(\log(\log(u/n)/\log(z/(nw))))$$

for $t = O(1)$, and letting $\varepsilon = 2^{-t}$, we obtain a data structure that uses $z = \Theta(n(u/n)^c)$ bits for constant time queries. The authors of [PT06b] target the use of their data structure for polynomial universes, i.e. $u = \Theta(n^c)$, $c > 1$, since for different cases they build different data structures, but the data structure can be effortlessly extended to the remaining cases. Joining Lemma 3.2 with Theorem 3.4 yields the following result.

**Corollary 3.1.** Using a modified vEB tree, we can implement a FID that uses $B(n,u) + O(n(u/n)^c)$ of space supporting all the operations in $O(\log(1/\varepsilon))$ time, for any fixed $\varepsilon > 0$.

### 3.3.2 Previous work: searching in $o(\log n)$ time

A completely different class of tools to search over a static set of integers is given by fusion trees [FW94]. A balanced binary search tree can search in $O(\log n)$ time, and this has been a hard limit for long time. Fusion trees take advantage of Word RAMs and bounded universes:
Theorem 3.7. A sequence $X$ of $n$ integers from a universe $U$ of size $u$ can be searched for predecessor in $O(\log n / \log w)$ time and $O(n \log u)$ bits on a Word RAM of size $w \geq \log u$.

Fusion trees are B-trees with fan-out $k = \Theta(w^{1/5})$. The main trickery here is to perform branching of a node in $O(1)$ time. Briefly, at each node, one can carefully construct a single word of data that can be used to perform predecessor search for all $k$ keys by means of bit-twiddling operations: since the LCP of $k$ keys can change at most $k$ times, information on where it changes can be embedded in a single word. Redundancy is then added so that the computation can be performed by means of elementary arithmetic and logical operations ($AC^0$ class).

3.3.3 New result: searching systematically

One main drawback of using vEB trees, y-fast tries and fusion trees as part of rank/select data structures is their need to organize the data to speed up search i.e., they are non-systematic. The other drawback is that we plan to use predecessor search data structures also on polylogarithmically sized blocks of elements. Namely, we want to be able to have a systematic representation of $m = O(\log^c u)$ elements, where $c = \Theta(1)$, and search them. In fully indexable dictionaries where $u = n \log^{O(1)} n$ this is not problematic, since $\log u / \log \log u$ elements can fit in a word and one can use precomputed tables to traverse a polylogarithmic amount of elements in $O(c)$ time. In our case, we will have to perform the same operation on objects of arbitrary universe size. One can resort to Theorem 3.7 or Theorem 3.5 to perform this search fast. We now introduce a novel data structure, of independent interest, that uses $O(m \log \log u)$ bits of redundancy can traverse a polylogarithmically sized block in $O(c)$ time, systematically. We name it succinct SB-tree.

Theorem 3.8. Let $w = \Omega(\log u)$ for some $u$, and let $A$ be a set of $m = O(\log^c u)$ elements drawn from $[u]$, where $c = \Theta(1)$. Then, there exists a data structure performing predecessor search on $A$ that uses $O(1 + c)$ accesses to elements of $A$ plus $O(c)$ internal computations. The data structure has redundancy $O(m \log \log u)$, requiring access to tables of size $O(u^\gamma)$ bits for some $\gamma < 1/2$, not depending on the actual content of the data structure.

Proof. The data structure is based on the building piece of String B-trees [FG99a], which in turn are based on the Patricia tries of [Mor68]. We start by reviewing the overall structure and then provide a succinct index out of it. The String B-tree [FG99b] of $A$ is basically a B-tree with fan out $\log_w n$, where each node contains a trie. The trie is used to select which exit node to continue searching into. To select
which keys to put in a node, the tree is built bottom up: for some $p$ to be set later, we partition $A$ into chunks of $p$ consecutive elements and assign each leaf one element of the partition. For each element of such partition, we take the minimum and maximum from the set of keys associated to each trie, group them $p$ at the time, and continue building the tree all the way up recursively.

Given the set of keys associated to a node $N$, say $Y = A_i, A_{i+1}, \ldots, A_{i+p}$, we build a compacted binary trie out of the representation of $Y$. It is easy to see that the trie has at most $2|Y|$ nodes. We annotate each node with its \textit{skip value}, namely the depth of such node in the original trie or, said otherwise, the position of the discriminating bit that forks the tree. By convention, we expect the left subtree to contain the 0 branch and the right subtree 1. We also add a fake root with skip value 0 at the beginning. An example is depicted in Figure 3.1. We can search for a predecessor inside a node if we can find $\ell = |\text{LCP}(q, Y)|$ for some query $q$: given $\ell$ and $q$ we can find a node in the compacted trie that we can use to find the predecessor of $q$. The main difficulty is how to compute $\ell$: the used technique is the known \textit{blind search}. In a first traversal of the trie, the query $q$ drives the descent by comparing the bits dictated in the skip values, irrespectively of whether there may have been a discrepancy in the erased nodes. For example, assume the skip values during the descent are 0, 1, 3, 5 and $q = 11001$. Then only bits 1 and 5 will be used, even though it may be that $\ell = 0$, i.e., there is no element in $Y$ prefixed by 1. This behavior is expected: it may be easily proved that the value $A_F$ associated to leaf $F$ found during the process shares the longest common prefix of length $\ell$ with $q$. Hence, it suffices to retrieve $A_F$ through \texttt{access} and the compute $\ell = \text{LCP}(A_F, q)$.\footnote{This is equivalent to computing the position of leftmost bit in $A_F \oplus q$, which may be done in $O(1)$ time through precomputed tables.}

We now describe how to store a single node: we must encode the trie shape and the skip values. As described in Lemma 2.1 the former element requires $2p$ bits for a node with fan-out $p$, while the latter requires $O(p \log \log u)$ bits. The challenge is now to branch from a node in $O(1)$ time. To do so, we have to set $p \leq \log u / 3 \log \log u$ so that the entire representation of a node fits in $\frac{1}{3} \log u$ bits. We perform the initial descent using precomputed tables: we navigate the trie in 3 steps, using 3 tables, so that at each time the table contains (i) the trie and skip values, (ii) the node at which the search starts and (iii) $1/3$ of the representation of $q$, outputting the exit node of the navigation procedure. Hence, this step can be performed by adding $O(u^\gamma)$ bits, for some $\gamma < 1/2$, to the whole data structure (shared by all instances). The other navigation in the trie can be performed by means of tables of negligible sizes as well.

In total, the space for the tree is $O(m \log \log u) + O(u^\gamma)$ bits, the branching factor is $\Theta(\log u / \log \log u)$ and the time to search is $O(\log_p n)$. When $m = O(\log^c u)$ and
Figure 3.1: Compacted trie for $Y = 00100, 00110, 010010, 010011$. Skip values are written in the nodes.

$$w = \Theta(\log u) = \Omega(p),$$ the time complexity becomes $O(c)$. \qed

### 3.4 An improved data structure

The current section is devoted to improve upon existing data structures for rank and select. In particular, we will exploit Lemma 3.1 and Theorem 3.4 to prove Theorem 3.1.

#### 3.4.1 Overview of our recursive dictionary

We consider the $\text{rank}_1$ operation only, leaving the effective development of the details to the next sections. A widely used approach to the FID problem lies in splitting the universe $[u]$ into different chunks and operating independently in each chunk, storing the rank at the beginning of the block. Queries are redirected into a chunk via a preliminary distributing data structure and the local data structure is used to solve it. Thus, the space occupancy is the distributing structure (once) plus all chunks. An orthogonal approach is given by Elias-Fano (see Section 2.5.1); the upper bit “cutting” operation performed by the construction $H$ builds up to $n$ different rank$_1$ problems, one per superblock. As much as one may be tempted to solve each of them recursively and independently, the real drawback of such an approach lies in the lack of global conditions that one can import in each subproblem: either one splits the sequence or one divides the universe, with no implication between the two dimensions. Our approach has then three-phases: we bucket the integers into superblocks by building the $H$ bitvector and then re-join all the sub-problems
together creating a new sequence of integers, repeating the scheme a number of times before storing the final sequence in another predecessor data structure.

More formally, let \( X \) be the integer sequence of \( n \) values drawn from \([u]\) and let \( q \in [u] \) be the argument of a generic rank query. Our goal is to encode a simple function \( f : [u] \to [u/n] \) and a machinery that generates a sequence \( \tilde{X} \) (to be described later) from \( X \) of length at most \( n \) coming from the universe \([u/n]\), so that given the predecessor of \( \tilde{x} = f(q) \in \tilde{X} \), we can recover the predecessor of \( q \) in \( X \).

Easily enough, \( f \) is the “cutting” operation of the upper \( \log n \) bits operated by the Elias Fano construction, which generates \( g \) different superblocks. Let \( X'_1, \ldots, X'_g \) be the sets of lower \([\log(u/n)] \) bits of values in \( X \), one per superblock. We define \( \tilde{X} = \text{sort}(\bigcup_{1 \leq i \leq g} X'_i) \), that is, the set of unique values we can extract from the \( X'_i \)'s, removing duplicates, and sorted in ascending order.

Suppose we have an oracle function \( \psi \), so that given a value \( \hat{x} \in \tilde{X} \) and an index \( j \in [g] \), \( \psi(j, \hat{x}) \) is the predecessor of \( \hat{x} \) in \( X'_j \). We also recall from Section 2 that the upper bit vector \( H \) of the Elias-Fano construction over \( X \) can answer the query \( \text{rank}_1([x/2^{\lceil \log u \rceil}]_2^{\lceil \log u \rceil}) \) in constant time (by performing \( \text{select}_v(H, x/2^{\lceil \log u \rceil}) - x/2^{\lceil \log u \rceil} \)). That is, it can give the rank value at the beginning of each superblock.

Given a query \( q \) we can perform \( \text{rank}_1(q) \) in the following way: we use \( H \) to reduce the problem within one superblock and know the rank at the beginning of the superblock \( j \). We then have the lower bits of our query \( (f(q)) \) and the sequence \( \tilde{X} \): we rank \( f(q) \) there, obtaining a certain result, say \( v \); we finally refer to our oracle \( \psi \) to find the predecessor of \( v \) into \( X'_j \), and thus find the real answer for \( \text{rank}_1(q) \).

As an example, consider, with \( n = 8 \) and \( u = 2^7 \), so that the leftmost 3 bits are the ones stored in \( H \):

\[
X = \{ 0, 3, 4, 8, 9, 11, 15, 33 \} \\
= \{ 000000, 000011, 001000, 001001, 001011, 001111, 100001 \} \\
\text{(superblock 1 prefix 000)} \\
\text{(superblock 2 prefix 001)} \\
\text{(superblock 3 prefix 100)}
\]

Then the sequences of low bits values are \( X'_1 = \{0,3,4\} \), \( X'_2 = \{0,1,3,7\} \) and \( X'_3 = \{1\} \). The sequence \( \tilde{X} \) is then \( \{0,1,3,4,7\} \). To perform \( \text{rank}_1(q) \) where \( q = 13 = 001101 \), we first use \( H \) to obtain \( \text{rank}_1(001000) = \text{rank}_1(8) = 3 \). Then, we solve the predecessor into superblock 2 using the value \( 101 \): we find that the predecessor of 5 in \( \tilde{X} \) is \( \tilde{X}_4 = 4 \). We then query the oracle \( \psi \) (not shown in the example) and obtain \( v = \psi(4,2) = 3 \). Hence we sum up \( \text{rank}_1(8) = 3 \) and \( v = 3 \) to have \( \text{rank}_1(q) = 6 \).

The main justification for the architecture is the following: in any superblock, the predecessor of some value can exhibit only certain values in its lower bits (those in \( \tilde{X} \)), thus once given the predecessor of \( f(q) \) our necessary step is only to reduce
the problem within \(|X|\) as the lower bits for any superblock are a subset of \(\hat{X}\). The impact of such choice is, as explained later, to let us implement the above oracle in just \(O(n^{1+\delta})\) bits, for any \(0 < \delta \leq 1/2\). That is, by using a superlinear number of bits in \(n\), we will be able to let \(u\) drop polynomially both in \(n\) and \(u\) (since \(\hat{X}\) has a smaller universe).

The above construction, thus, requires us to write \(X\) in an Elias-Fano dictionary, plus the oracle space and the space to solve the predecessor problem on \(\hat{X}\). The first part accounts for \(B(n,u) + O(n)\) bits, to which we add \(O(n^{1+\delta})\) bits for the oracle. By carefully employing the succinct String B-tree of Section 3.3.3 we can shrink the number of elements of \(\hat{X}\) to \(O(n/\log^2 u)\), leaving us with the problem of ranking on a sequence of such length and universe \([u/n]\). We continue replicating the entire scheme from the beginning up to \(s\) times: at each step \(i\), we cut out the leftmost \(\lfloor \log n \rfloor\) bits from \(X_i\), obtaining the description of the superblocks \(X_{i,1}^l, X_{i,2}^l\) and so on. We build our oracle data structure, \(\psi_i\) which takes as input the \(X_{i,j}^l\)s, and we create \(\hat{X}_i\) as described above. The next step has \(X_{i+1} = \hat{X}_i\) as input. Up to the final stage of the recursion, the total space occupancy is

\[
B(n, u) + O\left( \frac{n \log(u/n)}{\log^2 u} + \left( \frac{n}{\log^2 u} \right)^{1+\delta} \right)
\]

bits at the \(i\)-th step, descending geometrically. Interestingly, at each step we reduce the universe size of the outcome sequence to \(un^{-i}\). Thus, at the final step \(s\), we employ the previous result of Corollary 3.1 and obtain a final redundancy of \(O(u^n n^{1-s\epsilon})\).

### 3.4.2 Multiranking: Oracle \(\psi\)

We now give further details on our construction, describing how to build the oracle. Mainly, we show that using our choice on how to build \(\hat{X}\) and the function \(f\), being able to rank over \(\hat{X}\) we can build the oracle in \(O(n^{1+\delta})\) bits. We do it by illustrating, in a broader framework, the multiranking problem.

We are given a universe \([m]\), and a set of nonempty sequences \(A_1, \ldots, A_c\) each containing a sorted subset of \([m]\). We also define \(d = \sum_{1 \leq j \leq n} |A_j|\) as the global number of elements. The goal is, given two values \(1 \leq i \leq c\) and \(1 \leq q \leq u\), perform \(\text{rank}_i(q)\) in the set \(A_i\) in \(O(1)\) time and small space.

A trivial solution to this problem would essentially build a \(\text{FID}\) for each of the sequences. Assume that \(g(n, m)\) is the best redundancy for a \(\text{FID}\) and that it increases with \(m\) (as it usually is). Using one \(\text{FID}\) per sequence, we would require space occupancy

\[
d \log \left( \frac{mc}{d} \right) + g(d, mc),
\]
while we will be able to obtain
\[
d \log \left( \frac{m}{d} \right) + g(d, m) + O(d^{1+\delta}),
\]
for any \( \delta \leq 1/2 \). By a very simple case analysis one can easily see that this is always better when \( \delta = \Theta(1) \). (Note that when \( \delta = o(1) \) our data structure does not perform in \( O(1) \) time.)

The core of our technique is the universe scaling procedure. We perform the union of all the sequences \( A_1, \ldots, A_c \) and extract a new, single sequence \( \Lambda = \text{sort} \left( \bigcup_{1 \leq i \leq c} A_i \right) \), that is, containing only the distinct values that appear in the union (that is, we kill duplicates). \( \Lambda \) is named the alphabet of our problem and we denote its length with \( a \leq d \). Next, we substitute each element of a sequence by using the rank of the element itself in the alphabet. Hence, each element of a sequence is now in \( [a] \).

The multiranking problem is solved in two phases. We first perform ranking of the query \( q \) on \( \Lambda \) and then we exploit the information to recover the predecessor in the given set. Here we achieve our goal to (i) decouple a phase that depends on the universe size from one that depends on the alphabet size and (ii) have only one version of the problem standing on the initial universe. The following lemma solves the multiranking problem completely.

**Lemma 3.3.** There exists a data structure solving the multirank problem over non-empty increasing sequences \( A_1, \ldots, A_c \) with elements drawn from the universe \( [m] \), having \( d = \sum_{i=1}^c |A_i| \) elements in total, using \( B(d, m) + O(d^{1+\delta}) + o(m) \) bits for any given \( 0 < \delta \leq 1/2 \). Each multirank query takes \( O(\log(1/\delta)) \) time.

**Proof.** Let \( \Lambda \) be the alphabet defined over \( m \) by the sequences and let \( a = |\Lambda| \). For each sequence \( A_i \) we create a bitvector \( \beta_i \) of length \( a \) where \( \beta_{ij} = 1 \) iff \( \Lambda_j \in A_i \). We first view \( \beta_i \)'s as rows of a matrix of size \( a \times c \); since \( a \leq d \) and each of the sequences is non-empty (hence, \( c \leq d \)), the matrix has size \( O(d^2) \). We linearize the matrix by concatenating its rows and obtain a new bitvector \( \beta' \) on which we want to perform predecessor search. We note that the universe size of this bitvector is \( O(d^2) \), that is, the universe is polynomial w.r.t. to the elements stored. We store \( \beta' \) using the data structure of Corollary 3.1 setting the time to \( \log(1/\delta) \), so that that space is \( O(d^{1+\delta}) \). Finally, we store a fid \( G \) occupying \( B(d, m) + o(m) \) bits that represents the subset \( \Lambda \) of the universe \( [m] \).

Solving the multirank is now easy: given a query for \( q \) and a sequence index \( i \), we use \( G \) and find \( \lambda = \text{rank}_1(q) \), which leads to the predecessor into the alphabet \( \Lambda \) of our query \( q \). Since \( \lambda \in [a] \) we can now use the fid over \( \beta' \) to find \( x = \text{rank}_1(ai + \lambda) \). The final answer is clearly \( x - \text{rank}_1(ai) \). \( \square \)
3.4. AN IMPROVED DATA STRUCTURE

3.4.3 Completing the puzzle

With respect to Section 3.4 we are left with just one major detail. Each time we produce the output sequence \( \tilde{X} \), containing the lower bits for all elements, our only clue for the number of elements is the worst case upper bound \( n \), which is unacceptable. We now review the whole construction and employ the succinct SB-tree to have a polylogarithmic reduction on the number of elements, paying \( O(n \log \log u) \) bits per recursion step, as follows. Recall that \( X_i \) is the input to one step of our recursive scheme.

**Theorem 3.9.** Let \( n \) and \( u \) be two parameters, and let \( 0 < \delta < 1/2 \) be a parameter. Given a sorted sequence \( X_i \) of length \( n_i \) > \( \log^2 u \) taken from the universe \([u_i]\) where \( n_i \leq u_i \leq n \), there exists a procedure that receives \( X_i \) and outputs \( X_{i+1} \) of length \( n_{i+1} \) taken from \( u_{i+1} \) such that \( n_{i+1} \leq n_i / \log^2 u \) and \( u_{i+1} = u_i / n \). Also, the procedure creates a data structure that occupies

\[
B(n_i, u_i) + O(n + n_i \log \log u_i + n_i^{1+\delta}),
\]

so that a predecessor query on \( X_i \) can be reduced to a predecessor query on \( X_{i+1} \) in \( O(\log 1/\delta) \) time.

**Proof.** \( X_i \) is stored in an Elias-Fano dictionary, and the sets of superblocks and lower bits sequences are built as explained before. We then apply a further reduction step on the problem cardinality. Each superblock can be either **slim** or **fat** depending on whether it contains less than \( \log^2 u \) elements or not. Each fat superblock is split into **blocks** of size \( \log^2 u \), apart from the last block, and for each block we store a succinct SB-tree of Theorem 3.8. Since the block is polylogarithmic in size, we can perform predecessor search in constant time.

Slim superblocks are handled directly by the SB-tree and they do not participate further in the construction. For each block in a fat superblock, we logically extract its **head**, that is, the smallest element in it. We now use heads in the multiranking problems and build the output sequence \( X_{i+1} \) using only lower bits from the heads. As there can only be at most \( O(n/\log^2 u) \) blocks in fat superblocks, the size of the output sequence is at most \( O(n/\log^2 u) \). The oracle is built as usual, on the heads, using \( O(n^{1+\delta}) \) bits.

Ranking now performs the following steps: for each recursive step, it uses the Elias-Fano \( H \) vector to move into a superblock and at the same time check if it is slim or fat. In the latter case, it first delegates the query for the lower bits to the next dictionary, then feeds the answer to the multiranking instance and returns the actual answer. By Theorem 3.8 the added time complexity is \( O(1) \) per step. \( \square \)
3.4.4 Proof of main theorems

Proof of Theorem 3.1

Let $X \subseteq [u]$ be the set whose characteristic vector is $S$. The data structure involves recursive instances of Theorem 3.9 by starting with $X_0 = X$ and using each step’s output as input for the next step. As previously mentioned, we must only cover the base case and the last recursive step. We partition $X$ into $X'$ and $X''$ as described in the proof of Theorem 3.4, so that the construction is operated on both $X'$ and $X''$. We give a representation of $X'$, and $X''$ is stored in a similar way. We recursively build smaller sequences by invoking Theorem 3.9 exactly $s$ times. By Corollary 3.1 the space bound easily follows. To support select_0 on the original sequence, we operate on the $X'$ sequence alone, since when transformed to its dual $Y'$, we obtain a strictly monotone sequence. Interpreting $X'$ as an implicit representation of $Y'$ we build a multiset representation for the high bits ($H'$), a new set of succinct string B-trees using the superblocks of the dual sequence $Y'$ and think of as operating on $Y'\square$ and a new set of $s$ recursive applications of Theorem 3.9.

select_1 is trivial, thanks to the machinery of Theorem 3.4. The rank_1 algorithm for a query $q$ is performed on both $X'$ and $X''$ FIDs: we start by querying $H_0$, the upper bits of $F'_0$ ($F''_0$ respectively) for $q/2^{\lceil \log n \rceil}$, thus identifying a certain superblock in which the predecessor for $q$ can appear. Unless the superblock is slim (refer to proof of Theorem 3.9) we must continue to search through the next lower-order bits. This is done via multiranking, which recurses in a cascading manner with the same technique on the $s$ steps up to the last FID, that returns the answer. The chain is then walked backwards to find the root FID representative. We finally proceed through the succinct SB-tree to find the head and the next succinct string B-tree until we find the predecessor of $q$. The last step for recursion takes $O(\log 1/\varepsilon)$ time. All the middle steps for multiranking and succinct string B-tree traversals take $O(s \log 1/\delta)$ time. To support select_0, we act on $X'$, using exactly the same algorithm as before, but with the collection of data structures built for the dual representation $Y'$, and following the steps of Theorem 3.4.

During the buildup of the recursive process, say being at step $i$, the size $n'_i$ for sequence $X'_i$ ($i > 1$) is upper bounded by $n_i / \log^2 u$, while the universe has size $u/n'_i$. If at any step $2 \leq j \leq s$ the condition $u_j < n$ does not apply, we cannot apply Theorem 3.9 so we truncate recursion and use a $O(u_j) = O(n)$ bits FID to store the sequence $X_j$. This contributes a negligible amount to the redundancy.

Suppose we can recurse for $s$ steps with Theorem 3.9, we end up with a sequence
over a universe \( u_s = u/n^s \). By using Corollary 3.1 the space bound to store the final sequence is no more than \( O(n(u/n^s)\varepsilon) \).

The \( B(n_i, u_i) + O(n_i^{1+\delta}) \) factors decrease geometrically, so the first factor dominates and we can show that, apart from lower order terms, the space bound is as claimed. In other words, the total space \( S(n_i, u_i) \) of the recursive data structure satisfies:

\[
S(n_i, u_i) = S(n_{i+1}, u_{i+1}) + \text{space(FID for high bits)} + \text{space(SB-trees)} + O(n_i^{1+\delta})
\]

where \( n_{i+1} = n_i / \log^2 m \) and \( u_{i+1} = u_i / n \). The claimed redundancy follows easily.

**Proof of Theorem 3.2**

A slightly different definition of the data structure of Theorem 3.1 leads to Theorem 3.2, where we replace the \( O(n \log \log u) \) term with an \( O(n^\varepsilon) \) term. The substitution is possible when

(i) \( \log(1/\delta^s \varepsilon) = O(\log \log n) \) and \( \log u = \Theta(\log n) \) or

(ii) \( u = O\left(n^2 \frac{\log n}{\log \log n}\right) \).

When condition (i) applies, we know that either \( O(n^{1+\delta}) \) or \( O(n^{1-s\varepsilon}u^\varepsilon) \) are \( o(n \log \log u) \) bits, so that the \( O(n \log \log u) \) term of the SB-trees would dominate the redundancy. However, in such a case, we act as follows: instead of building a succinct SB-tree on each element of \( X' \) and \( X'' \), we sample \( X' \) and \( X'' \) one every \( \log \log u \) elements. When we have to search for the predecessor of \( q \), we find the predecessor of \( q \) among the sampled elements and then access the chunk of \( O(\log \log u) \) elements and scan them linearly.

We claim that the additional time complexity of \( O(\log \log u) \) is negligible. For the condition (i) to be satisfied, recalling that \( \log n = \alpha \log u \), for some \( \alpha < 1 \), we must have:

\[
n^\delta < \log \log u \Rightarrow \delta < \frac{\log(n)}{\log u} \Rightarrow \log(1/\delta) = \Omega(\log \log u),
\]

or

\[
(u^{n-s})^\varepsilon < \log \log u \Rightarrow \varepsilon < \frac{\log(n)}{\log(u/n)} = \frac{\log(n)}{(1-\alpha) \log u} \Rightarrow \log(1/\varepsilon) = \Omega(\log \log u).
\]

Hence, \( O(\log \log u) \) is subsumed.

Under condition (ii), we sample \( X \) every \( \Theta(\log \log u) \) elements again. Differently from case (i), we claim that we can navigate among these elements in \( O(1) \) time, so
that no extra data structures are needed. We give a lower bound for $\log u / \log(u/n)$, the amount of lower bits for elements in $X$ that can fit into a word of $\Theta(\log u)$ bits. The more we can fit, the more we can search in $O(1)$ time by means of precomputed tables of negligible size.

To give a lower bound, we note that $\log u / \log(u/n) = 1 + (\log n / \log(u/n))$ is minimized for large $u$ (as a function of $n$). Therefore, we set

$$u = n2^{\log n},$$

since constants do not matter, and obtain

$$\frac{\log u}{\log(u/n)} \geq \frac{\log n + (\log n / \log n)}{\log n / \log n} \geq \log \log n$$

Also, we know that, under condition (ii), $\log \log u = O(\log \log n)$, so that $\Omega(\log \log u)$ elements fit in a word. Hence, we obtain $O(1)$ navigation.

### 3.5 Density sensitive indexes

Although the lower bound of \cite{Gol07a} (Theorem 2.6) states that for a systematic data structure with constant time rank/select, the optimal (tight) redundancy is $\Theta(u \log \log u / \log u)$, the lower bound does not involve density-sensitive arguments, i.e. the ratio between $u$ and $n$ is not considered as part of the parameters. For example, when $n = 1$, it is easy to see that the redundancy of $O(\log u)$ bits suffices to support all operations in $O(1)$ time. Subsequent work \cite{GRR08} bridged the gap and proved Theorem 2.7. The work of this chapter is discussed in \cite{GORR11} and \cite{GORR12}. We also recall that we will use systematic data structure and succinct index as interchangeable terms.

#### 3.5.1 Preliminary discussion

The aim of this section is to provide a matching upper bound. However, we state a different redundancy, since the lower bound of Theorem 2.7 is in the bit-probe model, but we operate in the Word RAM model with $w = \Theta(\log u)$. Hence, we begin by proving:

**Lemma 3.4.** Let $u > 0$ be an integer, let $n = f(u)$ for some $f$, where $1 \leq n < u/2$. Let $t > 0$ be an integer and let $\mu = \lceil t \log u \rceil$. Then defining

$$R(n, u, t) = \begin{cases} n \frac{\log \left(\frac{nu}{u}\right)}{\mu}, & \text{if } n = \omega(u/\mu) \\ n \left(1 + \log \left(\frac{u}{\mu n}\right)\right), & \text{if } n = O(u/\mu), \end{cases}$$

implies $B(n, n + \left\lceil u/\mu \right\rceil) = O(R(n, u, t))$. 

Proof. Follows from the standard approximation of binomial coefficients, namely
\[ \log \binom{a}{b} = O(b \log(a/b)) \] if \( b \leq a/2 \).

Lemma 3.4 allows us to build an index for a bitvector \( S \) forgetting about the
triple definition of Theorem 2.7. We will also need the following intermediate result,
stated as Theorem 1 in [GRR08]:

**Theorem 3.10.** When \( u \leq n \log^{O(1)} n \), we can store a bitvector \( S \)
of length \( u \) and cardinality \( n \) using \( O(\mathcal{B}(n,u)) \) bits such that \( \text{rank} \) and \( \text{select} \) operations can be
supported in \( O(1) \) time.

We remark that, under the assumption of \( u \leq n \log^{O(1)} n \), there exist results
such as [GGG+07, P08], that can obtain \((1 + o(1))\mathcal{B}(n,u)\) space if employed as
succinct indexes (i.e., one can duplicate the original content and compress it in
the index). However, these approaches are usually complex and difficult to employ
in practice. The experimental work of [OS07] suggests that our approach can be
practical. In what follows, indeed, we will build upon Elias-Fano dictinaries and the
work of [OS07] and prove that our simple extension is powerful enough to match
the density-sensitive lower bounds of [GRR08].

We now prove the following:

**Theorem 3.11.** Given a bitvector \( S \) of length \( u \) with cardinality \( n \), where
\( \min\{n, u - n\} \geq u/\log^{O(1)} u \) and given an integer \( t = O(\log^{O(1)} u) \) there is a systematic data
structure that stores \( S \) and supports \( \text{rank} \) and \( \text{select} \) in \( O(t) \) time and \( O(t) \) calls to
\( \text{access} \). The data structure has redundancy \( O(R(n,u,t)) \) as defined in Lemma 3.4.

We remark that the condition \( \min\{n, u - n\} \geq u/\log^{O(1)} u \) is essential to get
\( O(1) \) time operations, as the predecessor lower bound of [PT06b] also applies in this
setting, through the following reduction. Given a set \( X \subseteq [u], |X| = n \), let \( S \) be the
characteristic function of \( X \). We can then represent \( S \) in \( O(n) \) words of space using
the naive array representation of \( X \). Since \( R(n,u,t) \) is also at most \( O(n) \) words of
memory, if we could achieve an index of size \( R(n,u,t) \) and support \( O(1) \)-time \( \text{rank} \) operation on the characteristic vector of \( S \) for \( n = u/\log^{O(1)} u \), we would be able to
solve predecessor queries in linear space and constant time, which is impossible.

### 3.5.2 A succinct index for \( \text{rank}/\text{select}_1 \)

**Lemma 3.5.** Given a bitvector \( S \) of length \( u \) with cardinality \( n \), where \( \min\{n, u - n\} \geq u/\log^{O(1)} n \), there is a succinct index that supports \( \text{rank} \) and \( \text{select}_1 \) on \( S \) in
\( O(t) \) time, \( O(t) \) access calls, and \( O(R(n,u,t)) \) bits of space, for any \( t = (\log u)^{O(1)} \).
Proof. We build on a variation of the Elias-Fano scheme. Partition $S$ into contiguous blocks of size $\mu = \lceil t \log u \rceil$ each and let $n_i \geq 0$ denote the number of 1s in the $i$-th block. Using standard approaches we can assume that the “local” computation needed to perform \texttt{rank} and \texttt{select}$_1$ operations on a block can done in $O(t)$ time using pre-computed lookup tables of size $O(u^\gamma)$ for some $\gamma < 1$.

We represent the sequence $OD = 1^{n_0}01^{n_1}0 \ldots$, which has $n$ 1s and $\lceil u/\mu \rceil$ 0s using Theorem \ref{thm:elias_fano}. The index size is $B(n, n + \lceil u/\mu \rceil) + O(\min\{u/\mu, n\}) = O(R(n, u, t))$ bits, by Lemma \ref{lem:rank_select1_bits}.

To compute \texttt{rank}$_1(i)$, let $j = \lceil i/\mu \rceil$. If $j = 0$, then the answer is obtained by reading the first block of $S$ with $O(t)$ \texttt{access} operations. Otherwise, select$_0(OD, j) - j$ gives the number of 1s in blocks $0, \ldots, j - 1$ and reading the next block with $O(t)$ \texttt{access} operations gives the answer to the query.

To compute select$_1(i)$ we first compute $j = \text{select}_1(OD, i) - i + 1$, giving us the block in which the $i$-th 1 lies. A call to select$_0(OD, j - 1)$ gives the number of 1s in blocks $0, \ldots, j - 1$, after which $O(t)$ calls to \texttt{access} suffice to compute the answer. \hfill $\square$

3.5.3 A succinct index for select$_0$

\textbf{Lemma 3.6.} Given a bitvector $S$ of length $u$ with cardinality $n$, where $\min\{n, u - n\} \geq u/\log^O(1) u$, there is a succinct index that supports select$_0$ on $S$ in $O(t)$ time, $O(t)$ \texttt{access} calls, and $O(R(n, u, t))$ bits of space, for any $t = (\log u)^{O(1)}$.

\textbf{Proof.} We divide $S$ into blocks of size $\mu$ as before. Let $x_1 < x_2 < \cdots < x_z$ be the positions of 0s in $S$ such that rank$_0(x_i) = i\mu$ for $i = 1, 2, \ldots, z = \lceil (u - n)/\mu \rceil$. Taking $x_0 = 0$, if there are $b_i$ 1s between $x_{i-1}$ and $x_i$, then the bitvector $SP = 1^{b_0}01^{b_1}0 \ldots 1^{b_z}0$ has at most $n$ 1s and $u/\mu$ 0, and is represented using at most $B(n, n + \lceil u/\mu \rceil) + O(\min\{n, u/\mu\})$ bits using Theorem \ref{thm:elias_fano} so that select$_0$ and select$_1$ on SP are supported in $O(1)$ time. Observe that select$_0(i\mu)$ on $S$ is equivalent to $i\mu + \text{select}_0(i)$ on SP, so that we are now able to answer select$_0$ queries for positions $i\mu$ ($i \in [z]$).

To answer general select$_0$ queries, we proceed as follows. With each position $x_i$ we associate the gap $G_i = [x_i, x_{i+1})$. We say that position $x_i$ is the starting point of a long gap if $|G_i| \geq 2\mu$ and define a set $LG$ of those positions that are the starting point of a long gap. A key property is that there are at most $n/(2\mu)$ long gaps and that

$$\sum_{i \in LG} |G_i| = O(n).$$

This is because any long gap contains exactly $\mu$ 0s, hence at least $\mu$ 1s in $S$ and so there are at most $n/\mu$ long gaps. Moreover, since there are at most $n$ 1s that lie
3.5. DENSITY SENSITIVE INDEXES

within long gaps, hence their total length is $O(n)$. Consider LG as a bitvector whose $i$-th bit is 1 iff $x_i$ is the starting point of a long gap. Note that LG has $z \leq m/\mu$ 0s and at most $n/\mu$ 1s, and can be represented using Elias-Fano (Theorem 2.1) dictionary\footnote{If LG has fewer than $n/\mu$ 1s we append trailing 1s, so that LG is not too sparse}. The representation has takes $B\left(\lceil u/\mu \rceil, \lceil n/\mu \rceil \right) + O(n/\mu) = O(n)$ bits in total, which is negligible.

Observe that $\text{select}_0(i)$ can be computed directly in $O(t)$ time if $x_{\mu[i/\mu]}$ is not the starting point of a long gap (which can be checked using LG), as we can read all bits in the gap starting at $x_{\mu[i/\mu]}$ with $O(t)$ accesses and use $O(t)$ table-lookups to process them. If $x_i$ lies in a long gap, instead, the maximum number of blocks $b$ a long gap can straddle in $S$ is $O(n/\mu)$, since there are at most $O(n/\mu)$ long gaps in $O(n)$. Furthermore, the maximum number $v$ of 0s in long gaps is $O(n)$. Hence, if the $i$th block which is (partially or fully) contained in a long gap contains $z_i$ 0s, the bitvector $ZD = 0^{z_1}10^{z_2}1\ldots0^{z_v}1$ can be represented using Lemma 3.10. The representation uses $O(B(b,v+b)) = O((n/\mu) \log \mu)$ bits. This is always $O(R(n,u,t))$, since if $n = \omega(u/\mu)$, then $R(n,u,t) = O((u/\mu) \log \mu)$, and if $n = O(u/\mu)$, then $R(n,u,t) = O(n \log \mu)$.

The steps to answer $\text{select}_0$ when the answer lies in a long gap are then as follows:

1. Let $j = \mu[i/\mu]$, and obtain $x_j = \text{select}_0(j)$ using SP.

2. If $x_j$ is the starting point of a long gap, then $q = \text{rank}_1(LG, j/\mu)$ gives the number of long gaps preceding $x_j$.

3. The number of block boundaries in $S$ crossed by the interval from $x_j$ to $\text{select}_0(i)$ can be obtained by taking the difference in position between 0s corresponding to these in $ZD$ (that is, $\text{select}_0(i \mod \mu + q\mu) - \text{select}_0(q\mu)$ on ZD) and subtracting the number of 0s in ZD between these two positions ($i \mod \mu - 1$). Since we now know the block in which $x_j$ lies, we also know the block, say $A$, in which $\text{select}_0(i)$ lies.

4. To compute the number of 0s before the beginning of $A$ in $S$, we use a $\text{rank}_0$ operation, answered through the OD bitvector of Lemma 3.5.

5. Load the block $A$ using $O(t)$ access operation and process it in $O(t)$ time. \qed
Chapter 4

Rank and select on sequences

We are given a read-only sequence $S$ of $n$ symbols over an integer alphabet $\Sigma = [\sigma] = \{0, 1, \ldots, \sigma - 1\}$, where $2 \leq \sigma \leq n$. The symbols in $S$ can be read using $\text{access}(i)$, for $0 \leq i \leq n - 1$: this primitive probes $S$ and returns the symbol $S[i]$. Hence, we will be working in the character RAM model of Section 2.1. Given the sequence $S$, its length $n$, and the alphabet size $\sigma$, we want to support $\text{rank}$ and $\text{select}$ operations systematically. An auxiliary data structure, a succinct index as in Section 1.1.1, is constructed in a preprocessing step to help answer these queries rapidly.

In this chapter we study the natural and fundamental time-space trade-off between the time complexity (in terms of access calls) and the redundancy $r$ in bits occupied by the index. Looking at extremal points, these queries can be answered in negligible space but $O(n)$ probes by scanning $S$, or in zero probes by making a copy of $S$ in auxiliary memory at preprocessing time, but with redundancy of $\Theta(n \log \sigma)$ bits. We are interested in indices that use few probes, and have redundancy $o(n \log \sigma)$, i.e., succinct w.r.t. the plain representation of $S$. Specifically, we obtain upper and lower bounds on the redundancy $r$ viewed as a function of the maximum number $t$ of probes, the length $n$ of $S$, and the alphabet size $\sigma$. We assume that $t > 0$ from now on.

4.1 Our results

Our first contribution is to show that the redundancy $r$ in bits

$$r = \Theta\left(\frac{n \log \sigma}{t}\right)$$

(4.1)

is tight for any succinct index solving our problem, for $t = O(\log \sigma / \log \log \sigma)$. We provide matching upper and lower bounds for this range of values on $t$, under the
assumption that $O(t)$ probes are allowed for rank and select, i.e. we ignore multiplicative constant factors and we work in the character probe model. In the Word RAM model with word size $w = \Theta(\log n)$, one can access $\log_\sigma n$ characters at the same time, but such case is not covered by our results, unless $\log_\sigma n = \Theta(1)$. We discuss recent results for a lower bound in the Word RAM model in Section 4.7.

Our result is composed by a lower bound of $r = \Omega\left(\frac{n \log \sigma}{t}\right)$ bits that holds for $t = O(\log \sigma)$ and by an upper bound of $r = O\left(\frac{n \log \sigma}{t} + n \log \log \sigma\right)$, thus leaving open what is the optimal redundancy when $t = \omega(\log \sigma)$. We also provide a lower bound of $r = \Omega\left(\frac{n \log t}{t}\right)$ for any $t = O(n)$.

Running times in the Word RAM model for the upper bound are $O(t + \log \log \sigma)$ for rank and $O(t)$ for select.

An interpretation of Eq. (4.1) is that, given a data collection $D$, if we want to build an additional succinct index on $D$ that saves space by a factor $t$ over that taken by $D$, we have to pay $\Omega(t)$ access cost for the supported queries. Note that the plain storage of the collection $D$ itself requires $n \log \sigma$ bits.

Lower bounds are our main findings, while the matching upper bounds are derived from known algorithmic techniques. Thus, our second contribution is a general lower bound technique that extends the algorithmic encoding/decoding approach in \cite{DLO03} in the sense that it abstracts from the specific query operation at hand, and focuses on its access pattern solely. For this, we can single out a sufficiently large, conflict free subset of the queries that are classified as stumbling or z-unique. In the former case, we extract direct knowledge from the probed locations; in the latter, the novelty of our approach is that we can extract (implicit) knowledge also from the unprobed locations. We are careful not to exploit the specific semantics of the query operations at this stage.

We also provide further time bounds for the rank/select problem. For example, if $\sigma = (\log n)^{O(1)}$, the rank operation requires only $O(t)$ time; also, we can get $O(t \log \log \sigma \log^{(3)} \sigma)$ time for rank and $O(t \log \log \sigma)$ time for select (Theorem 4.6). We also have a lower bound of $r = \Omega\left(\frac{n \log t}{t}\right)$ bits for the redundancy when $1 \leq t \leq n/2$, which leaves open what is the optimal redundancy when $t = \Omega(\log \sigma)$. As a corollary, we can obtain an entropy-compressed data structure that represents $S$ using $n H_k(S) + O\left(\frac{n \log \sigma}{\log \log \sigma}\right)$ bits, for any $k = O\left(\frac{\log \sigma}{\log \log \sigma}\right)$, supporting access in $O(1)$ time and rank and select in $O(\log \log \sigma)$ time (here, $H_k(S)$ is the $k$th-order empirical entropy).
4.2 Related work

In terms of time-space trade-off, our results extend the complexity gap between systematic and non-systematic succinct data structures for sequences (which was known for $\sigma = 2$) to any integer alphabet of size $\sigma \leq n$. This is easily seen by considering the case of $O(1)$ time/probes for select. Our systematic data structure requires $\Theta(n \log \sigma)$ bits of redundancy whereas the non-systematic data structure of [GMR06] uses just $O(n)$ bits of redundancy. However, if the latter should also provide $O(1)$-time access to the encoded string, then its redundancy becomes $\Theta(n \log \sigma)$. Note that Eq. (4.1) is targeted for non-constant alphabet size $\sigma$ whereas, for constant size, the lower and upper bounds for the $\sigma = 2$ case of [Gol07a] can be extended to obtain a matching bound of $\Omega(n \log t)$ bits (see Section 4.3).

The conceptual separation of the index from the input data was introduced to prove lower bounds in [GM07]. It was then explicitly employed for upper bounds in [FV07, GN06, SG06] to provide access, and was fully formalized in [BHMR07b]. The latter contains the best known upper bounds for our problem\(^1\) i.e. $O(s)$ probes for select and $O(s \log k)$ probes for rank, for any two parameters $s \leq \log \sigma / \log \log \sigma$ and $k \leq \sigma$, with redundancy $O(n \log k + n(1/s + 1/k) \log \sigma)$. For example, fixing $s = k = \log \log \sigma$, they obtain $O(\log \log \sigma)$ probes for select and $O(\log \log \sigma \log \log n)$ probes for rank, with redundancy $O(n \log \sigma / \log \log \sigma)$. By Eq. (4.1), we get the same redundancy with $t = O(\log \log \sigma)$ probes for both rank and select. Hence, our probe complexity for rank is usually better than [BHMR07b] while that of select is the same. Our $O(\log \log \sigma)$ time complexities are all better when compared to $O((\log \log \sigma)^2 \log \log \sigma)$ for rank and $O((\log \log \sigma)^2)$ for select in [BHMR07b].

4.3 Extending previous lower bound work

In this section, we prove a first lower bound for rank and select operations. We extend the existing techniques of [Gol07a], originally targeted at $\sigma = 2$. The bound has the advantage to hold for any $1 \leq t \leq n/2$, but it is weaker than Eq. (4.1) when $\log t = o(\log \sigma)$.

Theorem 4.1. Let $S$ be an arbitrary string of length $n$ over the alphabet $\Sigma = [\sigma]$, where $\sigma \leq n$. Any algorithm solving rank or select queries on $S$ using at most $t$ character probes (i.e. access queries), where $1 \leq t \leq n/2$, requires a succinct index with $r = \Omega(n \log t / t)$ bits of redundancy.

Intuitively speaking, the technique is as follows: it first creates a set of queries the data structure must answer and then partitions the string into classes, driven

\(^1\)We compare ourselves with the improved bounds given in the full version of [BHMR07b].
by the algorithm behavior. A bound on the entropy of each class gives the bound. However, our technique proves that finding a set of queries adaptively for each string can give a higher bound for \( t = o(\log \sigma) \).

Before getting into the full details we prove a technical lemma that is based on the concept of distribution of characters in a string: Given a string \( T \) of length \( n \) over alphabet \( \varphi \), the distribution (vector) \( d \) for \( n \) over \( \varphi \) is a vector in \( \mathbb{N}^\varphi \) containing the frequency of each character in \( T \). We can state:

**Lemma 4.1.** For any \( \varphi \geq 2 \), \( n \geq \varphi \) and distribution \( d \) for \( n \) on \( \varphi \), it holds

\[
\frac{n!}{(\frac{n}{\varphi})^n} \leq \varphi^n \left( \frac{\varphi}{n} \right)^{\varphi/2} \frac{\sqrt{\varphi}}{\sqrt{n}}.
\]

**Proof.** The maximization follows from the concavity of the multinomial function and the uniqueness of its maximum, which is obtained for the uniform distribution \( d = (n/\varphi, n/\varphi, \ldots, n/\varphi) \). The upper bound arises from double Stirling inequality, as we have:

\[
\frac{n!}{(\frac{n}{\varphi})^n} \leq \frac{\sqrt{2\pi}n^{n+1/2}e^{-n+1/12}}{(\sqrt{2\pi})^n (n/\varphi)^{n+\varphi/2}e^{-n+1/12(n/\varphi)^{1/2}}}
\]

\[
\leq (2\pi)^{(1-\varphi)/2} n^{n+1/2} \left( \frac{\varphi}{n} \right)^{n+1/2+(\varphi-1)/2}
\]

\[
\leq \varphi^{n} \varphi^{1/2} \left( \frac{\varphi}{n} \right)^{(\varphi-1)/2}
\]

as for \( \varphi \geq 2 \), \( (2\pi)^{(1-\varphi)/2} < 1 \). The lemma follows. \( \square \)

Let \( L = 3\sigma t \) and assume for the sake of simplicity that \( L \) divides \( n \). Denoting \( q(a,p) \) the arguments of a query for character \( a \) in position \( p \), we define the query set for \texttt{select} as

\[
Q = \{ q(c,3i) | c \in [\sigma] \land i \in [n/L] \}
\]

having size \( \gamma = \frac{n\sigma}{L} = \frac{n}{3t} \). The set of strings on which we operate, \( S \), is the set of all strings, so that \( \log |S| = n \log \sigma \).

A \textit{choices} tree for \( Q \) is a composition of smaller decision trees. At the top, we build the full binary tree of height \( r \), each leaf representing a possible choice for the \( r \) bits of redundancy. At each leaf of the tree, we append the decision tree of our algorithm for the first query \( q' \in Q \) on every possible string conditioned on the choice of the index. Since \( t \) is the time (in probes) to solve one \texttt{select} query, the decision tree has height at most \( t \) and each node has fan-out \( \sigma \), being all possible results of probing a location of the string. Each node is labeled with the location the algorithm chooses to analyze, although we are not interested in this information.
The choice tree has now at most \(2^r \sigma^t\) leaves. At each leaf we append the decision tree for the second query \(q''\), increasing the number of leaves again, and so on up to \(q''\). Without loss of generality we will assume that all decision trees have height exactly \(t\) and that each location is probed only once (otherwise we simply remove double probes and add some padding ones in the end). In the end, the entire tree will have \(2^r \sigma^t \gamma\) leaves. Leaves at the end of the whole choices tree are assigned strings from \(S\) that are compatible with the root-to-leaf path: each path defines a set of answers \(A\) for all the \(\gamma\) queries and a string is said to be compatible with a leaf if the answers to \(Q\) on that string is exactly \(A\) and all probes during the path match the path. For any leaf \(x\), we will denote the number of compatible strings by \(C(x)\).

Note that the tree partitions the entire set of strings, i.e. \(\sum_{x\text{ is a leaf}} C(x) = |S|\). Our objective is to prove that \(C(x)\) cannot be too big, and so prove that to distinguish all the answer sets the topmost tree (whose size is determined by \(r\)) must have at least some minimum height. More in detail, we will first compute \(C^*\), an upper bound on \(C(x)\) for any \(x\), so that

\[
|S| = \sum_{x\text{ is a leaf}} C(x) \leq (\# \text{ of leaves}) \times C^*,
\]  

which yields to (passing to logarithms):

\[
\log |S| \leq r + t\gamma \log \sigma + \log C^*.
\]  

Before continuing, we define some notation. For any path, the number of probed locations is \(t\gamma = n/3\), while the number of unprobed locations is denoted by \(U\). We divide a generic string in some leaf \(x\) into consecutive blocks of length \(L\), \(B_1, \ldots, B_{n/L}\). It is important to note that the division does not depend on the value of unprobed characters in the specific string.

We now associate a conceptual value \(u_i\) to each block, which represents the number of unprobed characters in that block, so that \(\sum_{i=1}^{n/L} u_i = U\). As in a leaf of the choices tree all probed locations have the same values, the only degree of freedom distinguishing between compatible strings lies in the unprobed locations.

We will compute \(C^*\) by analyzing single blocks, and we will focus on the right side of the following:

\[
\frac{C^*}{\sigma U} \triangleq c_1^* c_2^* \cdots c_{n/L}^* \triangleq \frac{g_1}{\sigma^{u_1}} \frac{g_2}{\sigma^{u_2}} \frac{g_3}{\sigma^{u_3}} \cdots \frac{g_{n/L}}{\sigma^{u_{n/L}}},
\]  

where \(g_i \leq \sigma^{u_i}\) represents the amount of possible assignments of unprobed characters for block \(i\) and \(c_i^*\) represents the ratio \(g_i/\sigma^{u_i}\).

We categorize blocks into two classes: determined blocks, having \(u_i < \sigma t\) and the remaining undetermined ones. For determined ones, we will assume \(g_i = \sigma^{u_i}\), as this approximation can only weaken our lower bound, hence having \(c_i^*\) values equal to
Figure 4.1: A choice tree for $|Q|$ queries, formed by a binary decision tree over $r$ bits, and $|Q|$ repetitions of a decision tree (see box) over the alphabet $\sigma$. Here $\sigma = 3$. At the bottom, colored nodes depict $C_j$s: the sets of strings associated with each root-to-leaf path.
1. For the remaining ones we bound $g_i$ from above by its maximum value. Namely, we employ Lemma 4.1 to bound the entropy of choices in each undetermined block: probed characters are fixed by the leaf in the decision tree we have chosen, so we can alter only the unprobed ones. We use Lemma 4.1 to obtain the worst assignment (i.e., maximum entropy) for any unprobed block:

$$g_i \leq \sigma^{1/2} \sigma_{u_i} \left( \frac{\sigma}{u_i} \right)^{\sigma/2}$$

Using that $u_i > \sigma t$, we can say

$$c_i^* = \frac{g_i}{\sigma_{u_i}} \leq \frac{\sigma^{1/2} \sigma_{u_i} \left( \frac{\sigma}{\sigma t} \right)^{\sigma/2}}{\sigma_{u_i}} \leq \sigma^{1/2} \left( \frac{1}{t} \right)^{\sigma/2} \quad (4.5)$$

The last step involves finding the number of such determined and undetermined blocks. As the number of global probes is at most $t \gamma$, the maximum number of determined blocks (where the number of probed locations is $L - u_i > 2\sigma t$) is $(t \gamma)/(2\sigma t) = n/(2L)$. Since $t \gamma = n/3$, the number of undetermined blocks is then at least $n/L - n/(2L) = n/(2L)$. Recalling that our upper bound in Eq. 4.4 increases with the number of determined blocks, we keep it to the minimum. Therefore, we can rework it by using only $c_i^*$ values for the $n/(2L)$ undetermined blocks, and upper bound each of them by the RHS of Eq. (4.5):

$$\frac{C^*}{\sigma U} \leq \left( \sigma^{1/2} \left( \frac{1}{t} \right)^{\sigma/2} \right)^{n/(2L)},$$

which in turn, recalling $L = 3\sigma t$, yields:

$$\log C^* \leq U \log \sigma + \frac{n}{2L} \log \left( \frac{1}{t} \right) + \frac{n}{2L} \log \sigma \quad (4.6)$$

$$= U \log \sigma + \Theta \left( \frac{n \log t}{t} \right). \quad (4.7)$$

Combining Eq (4.7), Eq. (4.3) and the fact that $t \gamma + U = n$, we obtain that

$$n \log \sigma = \log |S| \leq r + t \gamma \log \sigma + U \log \sigma - \Theta \left( \frac{n \log t}{t} \right)$$

and the bound follows since $t \gamma + U = n$.

We can prove an identical result for operation rank. The set $S$ of hard strings is the set of all strings of length $n$ over $\sigma$. We conceptually divide the strings in blocks of $L = 3\sigma t$ consecutive positions, starting at 0. With this in mind, we define the set of queries for rank,

$$Q = \{ q(c, L) | c \in [\sigma] \land i \in [n/L] \},$$

i.e. we ask for the distribution of the whole alphabet every $L$ characters, resulting in a batch of $\gamma = \frac{n}{3\sigma}$ queries. The calculations are then parallel to the previous case.
4.4 A general lower bound technique

This section aims at stating a general lower bound technique, of independent interest, which applies not only to both rank and select but to other query operations as well. Suppose we have a set \( S \) of strings of length \( n \), and a set \( Q \) of queries that must be supported on \( S \) using at most \( t \) probes each and an unknown amount \( r \) of redundancy bits. Under certain assumptions on \( S \) and \( Q \), we can show a lower bound on \( r \). Clearly, any choice of \( S \) and \( Q \) is allowed for the upper bound.

**Terminology.** The framework for our discussion extends the algorithmic encoding/decoding approach \[DLO03\]. Consider an arbitrary algorithm \( A \) that can answer to any query in \( Q \) performing at most \( t \) probes on any \( S \in S \), using a succinct index with \( r \) bits. We describe how to encode \( S \) using \( A \) and the succinct index as a black box, thus obtaining \( E(S) \) bits of encoding. Then, we describe a decoder that knowing \( A \), the index of \( r \) bits, and the encoding of \( E(S) \) bits, is able to reconstruct \( S \) in its original form. The encoding and decoding procedure are allowed unlimited (but finite) computing time, recalling that \( A \) can make at most \( t \) probes per query.

The lower bound on \( r \) arises from the necessary condition \( \max_{S \in S} E(S) + r \geq \log |S| \), since otherwise the decoder cannot be correct. Recall that, for this work, \( r \) must not be a function of a single \( S \), namely, we will bound the worst case redundancy. Indeed, \( r \geq \log |S| - \max_{S} E(S) \): the lower \( E(S) \), the tighter the lower bound for \( r \). Our contribution is to give conditions on \( S \) and \( Q \) so that the above approach can hold for a variety of query operations, and is mostly oblivious of the specific operation at hand since the query access pattern to \( S \) is relevant. This appears to be novel.

First, we require \( S \) to be sufficiently dense, that is, \( \log |S| \geq n \log \sigma - \Theta(n) \). Second, \( Q \) must be a subset of \([\sigma] \times [n] \), so that the first parameter specifies a character \( c \) and the second one an integer \( p \). Elements of \( Q \) are written as \( q_{c,p} \). Third, answers to queries must be within \([n] \).

Fourth, the set \( Q \) must contain a sufficiently large number of “stumbling” or “z-unique” queries, as we define now. Consider an execution of \( A \) on a query \( q_{c,p} \in Q \) for a string \( S \). The set of accessed position in \( S \), expressed as a subset of \([n] \), is called an access pattern, and is denoted by \( \text{Pat}_S(q_{c,p}) \).

- **Stumbling** queries imply the occurrence of a certain symbol \( c \) inside their own access pattern: the position of \( c \) can be decoded by using just the answer and the parameters of the query. Formally, \( q_{c,p} \in Q \) is stumbling if there exists a computable function \( f \) that takes in input \( c, p \) and the answer of \( q_{c,p} \) over \( S \), and outputs a position \( x \in \text{Pat}_S(q_{c,p}) \) such that \( S[x] = c \). The position \( x \) is called the target of \( q_{c,p} \). The rationale is that the encoder does not need to store
any information regarding \( S[x] = c \), since \( x \) can be extracted by the decoder from \( f \) and the at most \( t \) probed positions by \( A \). We denote by \( Q'_S \subseteq Q \) the set of stumbling queries over \( S \).

- **\( z \)-unique** queries are at the heart of our technique, where \( z \) is a positive integer. Informally, they have specific answers implying unique occurrences of a certain symbol \( c \) in a segment of \( S \) of length \( z + 1 \). Formally, a set \( U \subseteq [n] \) of answers is \( z \)-unique if for every query \( q_{c,p} \) having answer in \( U \), there exists a unique \( i \in [p, p + z] \) such that \( S[i] = c \) (i.e. \( S[j] \neq c \) for all \( j \in [p, p + z], j \neq i \)). A query \( q_{c,p} \) having answer in \( U \) is called \( z \)-unique and the corresponding position \( i \) is called the target of \( q_{c,p} \). Note that, for our purposes, we will restrict to the cases where \( |U| = 2^{O(n)} \). The rationale is the following: when the decoder wants to rebuild the string it must generate queries, execute them, and test whether they are \( z \)-unique by checking if their answers are in \( U \). Once that happens, it can infer a position \( i \) such that \( S[i] = c \), even though such a position is not probed by the query.

We denote by \( Q''_S(z) \subseteq Q \setminus Q'_S \) the set of \( z \)-unique queries over \( S \) that are not stumbling. Examples of \( z \)-unique queries and stumbling ones are illustrated in Figure 4.2 and Figure 4.3 respectively.

\[
\text{rank}(b, 7) - \text{rank}(b, 0) = 1
\]

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
c & a & b & f & e & d & g & h & e \\
\end{array}
\]

\( z \)  

\( \text{Tgt}_S(b, 0) = 2 \)  
\( \text{Pat}_S(b, 0) = \{5, 8\} \)  
\( \text{Trace}_S(b, 0) = dh \)  
Reduced trace: \( dh \)

Figure 4.2: Analysis of \( z \)-unique query \( q_{b,0} \) for \( \text{rank}(c, p + z) - \text{rank}(c, p) \), where \( U = \{1\}, \sigma = 8, t = 2, z = \lceil \sigma^{3/4} \sqrt{t} \rceil = 7 \). For explanation purposes, we are violating the rule that \( z \) should divide \( \sigma \).

**Main statement.** We now state our main theorem. Let \( S \) be a set of strings such that \( \log |S| \geq n \log \sigma - \Theta(n) \). Consider a set of queries \( Q \) that can be answered by performing at most \( t \) probes per query and using \( r \) bits of redundancy. We let \( \text{Tgt}_S(q_{c,p}) \) denote the target of query \( q_{c,p} \) over \( S \), if it exists, and let \( \text{Tgt}_S(Q) = \bigcup_{q \in Q} \text{Tgt}_S(q) \) for any set of queries \( Q \).
CHAPTER 4. RANK AND SELECT ON SEQUENCES

select(a, 2) = 3

\[
S = \begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
a & a & b & a & d & c & b & b & a
\end{array} \cdots
\]

Tgt_S(a, 2) = 3

Pats_S(a, 2) = \{2, 3, 5, 7, 8\}

Traces_S(a, 2) = bacba

Reduced trace: bcba

Figure 4.3: Analysis of one stumbling select query \(q_{a,2}\), where \(\sigma = 4\) and \(t = 5\).

**Theorem 4.2.** For any \(z \in [\sigma]\), let \(\lambda(z) = \min_{S \in S} |\text{Tgt}_S(Q'_s) \cup \text{Tgt}_S(Q''_s(z))|\). Then, there exist integers \(\gamma\) and \(\delta\) with \(\lambda(z)/(15t) \leq \gamma + \delta \leq \lambda(z)\), such that any succinct index has redundancy

\[
r \geq \gamma \log \left(\frac{\sigma}{z}\right) + \delta \log \left(\frac{\sigma \delta}{t |Q|}\right) - \Theta(n).
\]

The proof goes through a number of steps, each dealing with a different issue and is deferred to Section 4.5.

**Applications.** We now apply Theorem 4.2 to our two main problems, for an alphabet size \(\sigma \leq n\).

**Theorem 4.3.** Any algorithm solving rank queries on a string \(S \in [\sigma]^n\) using at most \(t = o(\log \sigma)\) character probes (i.e. access queries), requires a succinct index with \(r = \Omega \left(\frac{n \log \sigma}{t}\right)\) bits of redundancy.

**Proof.** We start by defining the set \(S\) of strings. For the sake of presentation, suppose \(\sigma\) divides \(n\). Also define \(z = \sigma^{3/4} \sqrt{t}\), assuming \(z\) divides \(\sigma\). \(S\) is the set of all the possible concatenations of \(n/\sigma\) permutations of \([\sigma]\). Therefore, \(|S| = (\sigma!)^{n/\sigma}\) and so we have \(\log |S| \geq n \log \sigma - \Theta(n)\) bits (by Stirling’s approximation).

Without loss of generality, we prove the bound on a derivation of the rank problem. We define the set \(Q\) so that the queries are \(q_{c,p} = \text{rank}(c, p+z) - \text{rank}(c, p)\), where \(c \in [\sigma]\) and \(p \in [n]\) with \(p \mod z \equiv 0\). In this setting, the \(z\)-unique answers are in \(U = \{1\}\). Indeed, whenever \(q_{c,p} = 1\), there exists just one instance of \(c\) in \(S[p-1, p+z-1]\). Note that \(|Q| = n\sigma/z > n\), for \(\sigma\) larger than some constant.

Observe that \(\lambda(z) = n\), as each position \(i\) in \(S\) such that \(S[i] = c\), is the target of exactly one query \(q_{c,p}\), supposing the query is not stumbling, such a query is surely \(z\)-unique. This is because we do not consider queries whose answer is different from 1, since they have no clear definition of target. The situation is illustrated in Figure 4.2. By Theorem 4.2, \(\gamma + \delta \geq n/(30t)\) since a single query is allowed to make
up to $2t$ probes now (this causes just a constant multiplicative factor in the lower bound.)

Still by the same application of Theorem 4.2, recalling $|Q| = n\sigma/z$, we have

$$r \geq \gamma \log \left( \frac{\sigma}{z} \right) - \delta \log \left( \frac{nt}{z\delta} \right) - \Theta(n). \quad (4.8)$$

We distinguish between two cases. At first, we assume $\delta \leq n/\sigma^{1/4}$. Considering $t = o(\log \sigma)$ and $\gamma + \delta \geq n/(30t)$, we know that for $\sigma$ and $t$ larger than suitable constants, $\sigma^{1/4} > \log \sigma > 60t$. We can then state that $\delta < n/(60t)$ and $\gamma \geq n/(60t)$ hold, yielding:

$$\gamma \log \left( \frac{\sigma}{z} \right) \geq \frac{n}{60t} \log \left( \frac{\sigma^{1/4}}{\sqrt{t}} \right).$$

Hence, Eq. (4.8) translates into

$$r \geq \frac{n}{240t} \log \sigma - \frac{n}{120t} \log t - \Theta(n) - \delta \log \left( \frac{nt}{z\delta} \right).$$

The term $\frac{n}{120t} \log t$ is subsumed by $\Theta(n)$. For the next step, we recall that, by convexity, functions of the form $\delta \log(k/\delta)$ for some $k \geq 2$, have their maximum in $\delta = k/e$. Hence, we know that

$$\delta \log \left( \frac{nt}{z\delta} \right) \leq \frac{nt}{ez} \log e = \frac{n\sqrt{t}}{e\sigma^{3/4}} \log e = o(n),$$

since $\sigma^{3/4}$ overcomes $\sqrt{t}$ for $\sigma$ larger than a constant. Hence, Equation 4.8 reduces to

$$r \geq \frac{n}{240t} \log \sigma - \Theta(n).$$

In the other case, we have $\delta > n/\sigma^{1/4}$, and

$$\delta \log \left( \frac{nt}{z\delta} \right) \leq \delta \log \left( \frac{\sigma^{1/4}}{\sigma^{3/4}\sqrt{t}} \right) = \frac{\delta}{2} \log \left( \frac{t}{\sigma} \right) = -\frac{\delta}{2} \log \left( \frac{\sigma}{t} \right).$$

Therefore, we know in Eq. (4.8) that, since $z = \Omega(\log \sigma) = \omega(t)$,

$$\gamma \log \left( \frac{\sigma}{z} \right) + \frac{\delta}{2} \log \left( \frac{\sigma}{t} \right) \geq \frac{1}{2}(\gamma + \delta) \log \left( \frac{\sigma}{z} \right) = \frac{1}{2}(\gamma + \delta) \log \left( \frac{\sigma^{1/4}}{\sqrt{t}} \right).$$

Again, we obtain

$$r \geq \frac{n}{240t} \log \sigma - \Theta(n).$$

In both cases, the $\Theta(n)$ term is negligible as $t = o(\log \sigma)$, hence the bound.
Theorem 4.4. Any algorithm solving \texttt{select} queries on a string \( S \in [\sigma]^n \) using at most \( t = o(\log \sigma) \) character probes (i.e. \texttt{access} queries), requires a succinct index with \( r = \Omega\left(\frac{n \log \sigma}{t}\right) \) bits of redundancy.

\textit{Proof.} The set \( S \) of strings is composed by full strings, assuming that \( \sigma \) divides \( n \). A full string contains each character exactly \( n/\sigma \) times and, differently from Theorem 4.3, has no restrictions on where those characters can be found. Again, we have \( \log |S| \geq n \log \sigma - \Theta(n) \). The set \( Q \) of queries is \( q_{c,p} = \texttt{select}(c,p) \), where \( p \in [n/\sigma] \), and all queries in \( Q \) are clearly stumbling ones, as \( \texttt{select}(c,p) = x \) immediately implies that \( S[x] = c \) (so \( f \) is the identity function). There are no \( z \)-unique queries here, so we can fix any value of \( z \): we choose \( z = 1 \). The situation is illustrated in Figure 4.3. It is immediate to see that \( \lambda(z) = n \), and \( |Q| = n \), as there are \( n/\sigma \) queries for each symbols in \([\sigma]\). By applying Theorem 4.2, we obtain that
\[
\gamma + \delta \geq \frac{n}{15t} \quad \text{and} \quad r \geq \gamma \log \left(\frac{\sigma}{t^2}\right) - \Theta(n).
\]
To obtain the final bound we substitute the \( \gamma \log \sigma \) term in Eq. (4.9) with a weaker \( \gamma \log(\sigma/(t^2)) \) and proceed by means of a case analysis for the term \(-\delta \log \left(\frac{\alpha t^2}{\sigma^2}\right)\).

1. When \( \delta = o(n/t) \) and \( \delta < nt/\sigma \), we know that \(-\delta \log (nt/(\sigma \delta)) \) is negative. We try to maximize its absolute value to obtain the worst case lower bound. By concavity again, this occurs for \( \delta^* = nt/(e \sigma) \). Since \( \delta^* = o(n/t) \), we have \(-\delta \log (nt/(\sigma \delta)) \geq -\delta^* \log (nt/(\sigma \delta^*)) = -\delta^* \log e \geq -\Theta(n/t) \) and Eq. (4.9) becomes
\[
\gamma + \delta \geq \frac{n}{15t} - \Theta(n) \geq \Theta\left(\frac{n}{t} \log \left(\frac{\sigma}{t^2}\right)\right) - \Theta(n),
\]
where we use the fact that \( \gamma + \delta = \Omega(n/t) \) and \( \delta = o(n/t) \) implies \( \gamma = \Omega(n/t) \).

2. When \( \delta = o(n/t) \) but \( \delta \geq nt/\sigma \), we know \(-\delta \log (nt/(\sigma \delta)) \geq 0 \). Since \( \gamma + \delta = \Omega(n/t) \) and \( \delta = o(n/t) \) implies \( \gamma = \Omega(n/t) \), Eq. (4.9) becomes like in the previous case, namely,
\[
r \geq \gamma \log \left(\frac{\sigma}{t^2}\right) - \Theta(n).
\]

3. When \( \delta = \Omega(n/t) \), in general, the term \(-\delta \log (nt/(\sigma \delta)) \) is always positive and we try to minimize it again. Setting \( \delta = n/(\alpha t) \) for a sufficiently large constant \( \alpha > 1 \), we obtain \(-\delta \log (nt/(\sigma \delta)) \geq -\delta \log (\alpha t^2/\sigma) \geq -\delta \log (t^2/\sigma) \) - \(\Theta(n)\).
4.5 Proof of Theorem 4.2

Hence, Eq. (4.9) becomes

\[ r \geq \gamma \log \left( \frac{\sigma}{t^2} \right) - \delta \log \left( \frac{t^2}{\sigma} \right) - \Theta(n) \]

\[ = (\gamma + \delta) \log \left( \frac{\sigma}{t^2} \right) - \Theta(n) \]

\[ \geq \frac{n}{15t} \log \left( \frac{\sigma}{t^2} \right) - \Theta(n). \]

Since in all cases the \( \Theta(n) \) term is subsumed when \( t = o(\log \sigma) \), we have the bound. \( \Box \)

4.5 Proof of Theorem 4.2

We give an upper bound on \( E(S) \) for any \( S \in \mathbb{S} \) by describing an encoder and a decoder for \( S \). In this way we can use the relation \( \max_{S \in \mathbb{S}} E(S) + r \geq \log |S| \) to induce the claimed lower bound on \( r \) (see Section 4.4). We start by discussing how we can use \( z \)-unique and stumbling queries to encode a single position and its content compactly. Next, we will deal with conflicts between queries: not all queries in \( Q \) are useful for encoding. We describe a mechanical way to select a sufficiently large subset of \( Q \) so that conflicts are avoided. Bounds on \( \gamma \) and \( \lambda \) arise from such a process. To complete the encoding, we present how to store the parameters of the queries that the decoder must run.

Information of a single position and its content. We first evaluate the entropy of positions and their contents by exploiting the knowledge of \( z \)-unique and stumbling queries. We use the notation \( \log |S|_{\Omega} \) for some event \( \Omega \) as a shortcut for \( \log |S'| \) where \( S' = \{ S \in \mathbb{S} | S \text{ satisfies } \Omega \} \).

Lemma 4.2. For any \( z \in [\sigma] \), let \( \Omega_{c,p} \) be the condition “\( q_{c,p} \) is \( z \)-unique”. Then it holds

\[ \log |S| - \log |S|_{\Omega_{c,p}}| \geq \log(\sigma/z) - O(1). \]

Proof. It holds \( |(S|_{\Omega_{c,p}})| \leq (z + 1)|S|/\sigma \) since there at most \( z + 1 \) candidate target cells compatible with \( \Omega_{c,p} \) and at most \( |S|/\sigma \) possible strings containing \( c \) at a fixed position. So, \( \log |S|_{\Omega_{c,p}}| \leq \log(z + 1) + \log |S| - \log \sigma \), hence the bound. \( \Box \)

Lemma 4.3. Let \( \Omega'_{c,p} \) be the condition “\( q_{c,p} \) is a stumbling query”. Then, it holds that

\[ \log |S| - \log |S|_{\Omega'_{c,p}}| \geq \log(\sigma/t) - O(1). \]

Proof. The proof for this situation is already known from [Go07b]. In our notation, the proof goes along the same lines as that of Lemma 4.2 except that we have \( t \) choices instead of \( z + 1 \). To see that, let \( m_1, m_2, \ldots, m_t \) be the positions, in temporal
order, probed by an algorithm \( \mathcal{A} \) on \( S \) while trying to answer \( q_{c,p} \). Since the query is stumbling, the target will be one of \( m_1, \ldots, m_t \). It suffices to remember which one of the \( t \) steps probes that target, since their values \( m_1, \ldots, m_t \) are deterministically characterized given \( \mathcal{A}, S \) and \( q_{c,p} \).

\( \square \)

**Conflict handling.** In general, multiple instances of Lemma \[1.2\] and/or Lemma \[4.3\] cannot be applied independently. We introduce the notion of conflict on the targets and show how to circumvent this difficulty. Two queries \( q_{b,o} \) and \( q_{c,p} \) conflict on \( S \) if at least one of the following three condition holds:

1. \( \text{Tgt}_S(q_{c,p}) \in \text{Pat}_S(q_{b,o}) \).
2. \( \text{Tgt}_S(q_{b,o}) \in \text{Pat}_S(q_{c,p}) \).
3. \( \text{Tgt}_S(q_{c,p}) = \text{Tgt}_S(q_{b,o}) \).

A set of queries where no one conflicts with another is called conflict free.

We now prove a lemma similar to the one found in [Gol09], in a different context. Namely, Lemma \[1.4\] defines a lower bound on the maximum size of a conflict free subset \( \mathcal{Q}^* \) of \( \mathcal{Q} \). We use an iterative procedure that maintains at each step \( i \) a set \( \mathcal{Q}^i \) of conflict free queries and a set \( C_i \) of available targets, such that no query \( q \) whose target is in \( C_i \) will conflict with any query \( q' \in \mathcal{Q}^i_{i-1} \). Initially, \( C_0 \) contains all potential targets for stumbling and \( z \)-unique queries for the string \( S \), so that by definition \( |C_0| \geq \lambda(z) \) (since \( \lambda(z) \) is a lower bound over all strings in \( \mathbb{S} \)). Also, \( \mathcal{Q}^0 \) is the empty set. In the proof, we work on subsets of \( \mathcal{Q} \), denoted by \( \mathcal{Q}^E \), containing the set of eligible queries to build \( \mathcal{Q}^* \) from. At each step \( i \), given \( C_i \), we consider \( \mathcal{Q}^E_i \), created as follows: for each \( x \in C_i \), we pick one arbitrary element from the set \( \text{Tgt}_S^{-1}(x) \). Note that \( |\mathcal{Q}^E_i| = |C_i| \) for each \( i \).

**Lemma 4.4.** Let \( i \geq 1 \) be an arbitrary step and assume \( |C_{i-1}| > 2|C_0|/3 \). Then, there exists \( \mathcal{Q}^*_i \) and \( C_i \) such that (a) \( |\mathcal{Q}^*_i| = 1 + |\mathcal{Q}^*_{i-1}| \), (b) \( \mathcal{Q}^*_i \) is conflict free, (c) \( |C_i| \geq |C_0| - 5it \geq \lambda(z) - 5it \).

**Proof.** We now prove that there exists a target position \( u \in C_{i-1} \) such that less than \( 3t \) queries in \( \mathcal{Q}^E_{i-1} \) probe \( u \). Assume by contradiction that for any \( u \), at least \( 3t \) queries in \( \mathcal{Q}^E_{i-1} \) probe \( u \). Then, we would collect \( 3t|\mathcal{Q}^E_{i-1}| = 3t|C_{i-1}| > 2|C_0|t \) probes in total. However, any query can probe at most \( t \) cells; summing up over the whole \( \mathcal{Q}^E_0 \), we obtain \( |C_0|t \), giving a contradiction. At step \( i \), we choose \( u \) as a target and the unique query in \( \mathcal{Q}^E_{i-1} \) that has \( u \) as target, say query \( q_{c,p} \) for some \( c,p \). This maintains invariant (a) as \( \mathcal{Q}^*_i = \mathcal{Q}^*_{i-1} \cup \{q_{c,p}\} \). As for invariant (b), we remove the potentially conflicting targets from \( C_{i-1} \), and produce \( C_i \). Specifically, let \( I_u \subseteq C_{i-1} \) be the set of targets for queries probing \( u \) over \( S \), where by the
above properties \(|I_u| \leq 3t\). We remove \(u\) and the elements in \(I_u\) and \(\text{Pat}_S(q_{c,p})\). So, 
\(|C_i| = |C_{i-1}| - |\{u\}| - |I_u| - |\text{Pat}_S(q_{c,p})| \geq |C_{i-1}| - 1 - 3t - t \geq |C_0| - 5it,\) proving also (c).

By applying Lemma 4.4 until \(|C_i| \leq 2|C_0|/3\), we obtain a final set \(Q^*\):

**Corollary 4.1.** For any \(S \in \mathcal{S}\), \(z \in [\sigma]\), there exists a set \(Q^*\) containing \(z\)-unique and stumbling queries of size \(\gamma + \delta \geq \lambda(z)/(15t)\), where \(\gamma = |\{q \in Q^* | q \text{ is stumbling on } S\}|\) and \(\delta = |\{q \in Q^* | q \text{ is } z\text{-unique on } S\}|\).

**Proof.** We want to find a value \(i^* = \gamma + \delta\) which determines the first iteration in Lemma 4.4 such that \(|C_{i^*}| \leq 2|C_0|/3\). By the lemma, we also know that \(|C_{i^*}| > |C_0| - 5i^*t\). Hence, \(|C_0| - 5i^*t \leq 2|C_0|/3\), which gives \(i^* \geq |C_0|/(15t) \geq \lambda(z)/(15t)\).

### 4.5.1 Encoding

We are left with the main task of describing the encoder. Ideally, we would like to encode the targets, each with a cost as stated in Lemma 4.2 and Lemma 4.3, for the conflict free set \(Q^*\) mentioned in Corollary 4.1. Characters in the remaining positions can be encoded naively as a string. This approach has a drawback. While encoding which queries in \(Q\) are stumbling has a payoff when compared to Lemma 4.3, we do not have such a guarantee for \(z\)-unique queries when compared to Lemma 4.2.

Without getting into details, according to the choice of the parameters \(|Q|, z\) and \(t\), such an ‘encoding sometimes saves space and sometimes does not: it may use even more space than \(\log |\mathcal{S}|\). For example, when \(|Q| = O(n)\), even the naive approach works and yields an effective lower bound. Instead, if \(Q\) is much larger, savings are not guaranteed. The main point here is that we want to lower the effect that the parameters have on the lower bound and always guarantee a saving, which we obtain by means of an implicit encoding of \(z\)-unique queries. Some machinery is necessary to achieve this goal.

**Archetype and trace**

Instead of trying to directly encode the information of \(Q^*\) as discussed above, we find a query set \(Q^A\) called the archetype of \(Q^*\), that is indistinguishable from \(Q^*\) in terms of \(\gamma\) and \(\delta\) as given by Corollary 4.1. The extra property of \(Q^A\) is to be decodable using just \(O(n)\) additional bits, hence \(E(S)\) is smaller when \(Q^A\) is employed. The other side of the coin is that our solution requires a two-step encoding. We need to introduce the concept of trace of a query \(q_{c,p}\) over \(S\), denoted by \(\text{Trace}_S(q_{c,p})\). Given the access pattern \(\text{Pat}_S(q_{c,p}) = \{m_1 < m_2 < \cdots < m_t\}\) (see Section 4.4), the trace
is defined as the string $\text{Trace}_S(q, p) = S[m_1] \cdot S[m_2] \cdots S[m_t]$. We also extend the concept to sets of queries, so that for $\hat{Q} \subseteq Q$, we have $\text{Pat}_S(\hat{Q}) = \bigcup_{q \in \hat{Q}} \text{Pat}_S(q)$, and $\text{Trace}_S(\hat{Q})$ is defined using the distinct, sorted positions in $\text{Pat}_S(\hat{Q})$.

Then, we define a canonical ordering between query sets. We define the predicate $q, c, p \prec q, d, g$ iff $p < g$ or $p = g$ and $c < d$ over queries, so that we can sort queries inside a single query set. Let $Q_1 = \{q_1 < q_2 < \cdots < q_x\}$ and let $Q_2 = \{q'_1 < q'_2 < \cdots < q'_y\}$ be two distinct query sets. We say that $Q_1 \prec Q_2$ if either $q_1 < q'_1$ or recursively $q_1 = q'_1$ and $(Q_1 \setminus \{q_1\}) \prec (Q_2 \setminus \{q'_1\})$.

Given $Q^*$, its archetype $Q^A$ must obey to the following conditions for the given $S$:

- It is conflict free and has the same number of queries of $Q^*$.
- It contains exactly the same stumbling queries of $Q^*$, and all remaining queries are $z$-unique (note that they may differ from those in $Q^*$, but they are still the same amount).
- If $p_1, p_2, \ldots, p_x$ are the positional arguments of queries in $Q^*$, then the same positions are found in $Q^A$ (while character $c_1, c_2, \ldots, c_x$ may change).
- $\text{Pat}_S(Q^*) = \text{Pat}_S(Q^A)$.
- Among those query sets complying with the above properties, $Q^A$ is the minimal w.r.t. to the canonical ordering $\prec$.

Note that $Q^*$ complies with all the conditions above for sure, but the last. Therefore, the archetype of $Q^*$ always exists, being either a smaller query set (w.r.t. to $\prec$) or $Q^*$ itself. The encoder can compute $Q^A$ by exhaustive search, since its computational time complexity is not relevant to the lower bound, being in the character probe model.

**First step: encoding for trace and stumbling queries**

As noted above the stumbling queries for $Q^*$ and $Q^A$ are the same, and there are $\delta$ of them. Here, we encode the trace together with the set of stumbling queries. The rationale is that the decoder must be able to rebuild the original trace only, whilst encoding of the positions which are not probed is left to the next step, together with $z$-unique queries. Here is the list of objects to be encoded:

- (a) The set of stumbling queries expressed as a subset of $Q$.
- (b) The access pattern $\text{Pat}_S(Q^A)$ encoded as a subset of $[n]$, the positions of $S$. 
4.5. PROOF OF THEOREM 4.2

(c) The reduced trace, obtained from Trace$_S(Q^A)$ by removing all the characters in positions that are targets of stumbling queries. Encoding is performed naively by storing each character using $\log \sigma$ bits. The positions thus removed, relatively to the trace, are stored as a subset of $||\text{Trace}_S(Q^A)||$.

(d) For each stumbling query $q_{c,p}$, in the canonical order, an encoded integer $i$ of $\log t$ bits indicating that the $i$th probe accesses the target of the query.

The decoder starts with an empty string, it reads the access pattern in (b), the set of removed positions in (c), and distributes the contents of the reduced trace (c) into the remaining positions. In order the fill the gaps in (c), it recovers the stumbling queries in (a) and runs each of them, in canonical ordering. Using the information in (d), as proved by Lemma 4.3, it can discover the target in which to place its symbol $c$. Since $Q^A$ is conflict free, we are guaranteed that each query will always find a symbol in the probed positions during execution.

**Lemma 4.5.** Let $\ell$ be the length of Trace$_S(Q^A)$. The first step encodes information (a)–(d) using at most $\ell \log \sigma + O(n) + \delta \log(|Q|/\delta) - \delta \log(\sigma/t)$ bits.

**Proof.** Recall that by construction, $|Q^A| = |Q^*| = \Theta(n/t)$. For all objects: (a) uses $\log \left(\frac{|Q|}{\delta}\right) = \delta \log(|Q|/\delta) + O(\delta)$; (b) uses $\log \binom{n}{\ell} \leq n$ bits; (c) uses $(\ell - \delta) \log \sigma$ bits for the reduced trace plus at most $\ell$ bits for the removed positions; (d) uses $\delta \log t$ bits.

**Second step: encoding of $z$-unique queries and unprobed positions**

We now proceed to the second step, where targets for $z$-unique queries are encoded along with the unprobed positions. Contents of the target cells can be rebuilt using queries in $Q^A$. To this end, we assume that encoding of Lemma 4.5 has already been performed and, during decoding, we assume that the trace has been already rebuilt. Recall that $\gamma$ is the number of $z$-unique queries. Here is the list of objects to be encoded:

(e) The set of queries in $Q^A$ that are $z$-unique, expressed as a subset of $Q^A$ according to the canonical ordering $\prec$. Also the set of $z$-unique answers $U$ is encoded as a subset of $[n]$.

(f) For each $z$-unique query $q_{c,p}$, in canonical order, the encoded integer $p$. This gives a multiset of $\gamma$ integers in $[n]$.

(g) The reduced unprobed region of the string, obtained by removing all the characters in positions that are targets of $z$-unique queries. Encoding is performed
naively by storing each character using \( \log \sigma \) bits. The positions thus removed, relatively to the unprobed region, are stored as a subset of \([n - \ell]\).

\((h)\) For each \(z\)-unique query \(q_{c,p}\), in the canonical order, an encoded integer \(i\) of \(\log z + O(1)\) bits indicating which position in \([p - 1, p + z - 1]\) contains \(c\).

The decoder first obtains \(Q^A\) by exhaustive search. It initializes a set of \(|Q^A|\) empty couples \((c, p)\) representing the arguments of each query in canonical order. It reads \((e)\) and reuses \((a)\) to obtain the parameters of the stumbling queries inside \(Q^A\). It then reads \((f)\) and fills all the positional arguments of the queries. Then, it starts enumerating all query sets in canonical order that are compatible with the arguments known so far. That is, it generates characters for the arguments of \(z\)-unique queries, since the rest is known. Each query set is then tested in the following way. The decoder executes each query by means of the trace. If the execution tries a probe outside the access pattern, the decoder skips to the next query set. If the query conflicts with any other query inside the same query set, the decoder skips. If the query answer denotes that the query is not \(z\)-unique (see Section 4.4 and \((e)\)), it skips. In this way, all the requirements for the archetype are met, hence the first query set that is not skipped is \(Q^A\).

Using \(Q^A\) the decoder rebuilds the characters in the missing positions of the reduced unprobed region: it starts by reading positions in \((g)\) and using them to distribute the characters in the reduced region encoded by \((g)\) again. For each \(z\)-unique query \(q_{c,p} \in Q^A\), in canonical order, the decoder reads the corresponding integer \(i\) inside \((h)\) and infers that \(S[i + p] = c\). Again, conflict freedom ensures that all queries can be executed and the process can terminate successfully. Now, the string \(S\) is rebuilt.

**Lemma 4.6.** The second step encodes information \((e)-(h)\) using at most \((n - \ell) \log \sigma + O(n) - \gamma \log(\sigma/z)\) bits.

**Proof.** Space occupancy: \((e)\) uses \(\log \left(\frac{|Q^A|}{\gamma}\right) \leq |Q^A|\) bits for the subset plus, recalling Section 4.4, \(O(n)\) bits for \(U\); \((f)\) uses \(\log \left(\frac{n + \gamma}{\gamma}\right) \leq 2n\) bits; \((g)\) requires \((n - \ell - \gamma) \log \sigma\) bits for the reduced unprobed region plus \(\log \left(\frac{n - \ell}{\gamma}\right)\) bits for the positions removed; \((h)\) uses \(\gamma \log z + O(\gamma)\) bits.

**Putting it all together**

**Proof of Theorem 4.2.** By combining Lemma 4.5 and Lemma 4.6 we obtain that for each \(S \in \mathbb{S}\),

\[
E(S) \leq n \log \sigma + O(n) + \delta \log \left(\frac{t|Q|}{\delta \sigma}\right) - \gamma \log \left(\frac{\sigma}{z}\right). 
\]
4.6. UPPER BOUND

On one side we know that, for the encoder to be correct, it must be able to distinguish among all the strings in $S$. Hence, we know that $r + \max_{S \in S} E(S) \geq \log |S|$. On the other side, we know by assumption that $\log |S| \geq n \log \sigma - \Theta(n)$. Combining everything together we obtain the bound.

4.6 Upper bound

Our approach follows substantially the one in [BHMR07b], but uses two new ingredients, that of monotone hashing [BBPV09] and succinct SB-trees of Theorem 3.8, to achieve an improved (and in many cases optimal) result. We first consider these problems in a slightly different framework and give some preliminaries.

4.6.1 Bootstrapping

We are given a subset $T \subseteq [\sigma]$, where $|T| = m$. Let $R(i) = |\{j \in T | j < i\}|$ for any $i \in [\sigma]$, and $S(i)$ be the $i + 1$th element of $T$, for any $i \in [m]$.

For any subset $T \subseteq [\sigma]$, given access to $S(\cdot)$, a succinct SB-tree of Theorem 3.8 is used as the systematic data structure that supports predecessor queries on $T$, using $O(|T| \log \log \sigma)$ extra bits. For any $c > 0$ such that $|T| = O(\log^c \sigma)$, the succinct SB-tree supports predecessor queries in $O(c)$ time plus $O(c)$ calls to $S(\cdot)$. The data structure relies on a precomputed table of $O(m^\alpha)$ bits for some $0 < \alpha < 1$, depending only on $\sigma$ and not on $T$.

A monotone minimal perfect hash function for $T$ is a function $h_T$ such that $h_T(x) = R(x)$ for all $x \in T$, but $h_T(x)$ can be arbitrary if $x \notin T$. We use the following result, which is a rewording of Theorem 2.9:

**Theorem 4.5 (BBPV09).** There is a monotone minimal perfect hash function for a string $T$ of length $m$ over alphabet $[\sigma]$ that:

(a) occupies $O(m \log \log \sigma)$ bits and can be evaluated in $O(1)$ time;

(b) occupies $O(m \log^3 \sigma)$ bits and can be evaluated in $O(\log \log \sigma)$ time.

Although function $R(\cdot)$ has been studied extensively in the case that $T$ is given explicitly, we consider the situation where $T$ can only be accessed through (possibly expensive) calls to $S(\cdot)$. We give the following extension of known results:

**Lemma 4.7.** Let $T \subseteq [\sigma]$ and $|T| = m$. Then, for any $1 \leq k \leq \log \log \sigma$, there is a data structure that supports $R(\cdot)$ in $O(\log \log \sigma)$ time plus $O(1 + \log k)$ calls to $S(\cdot)$, and uses $O((m/k) \log \log \sigma)$ bits of space.
Proof. We construct the data structure as follows. We store every \((\log \sigma)\)th element of \(T\) in a y-fast trie. This divides \(T\) into buckets of \(\log \sigma\) consecutive elements. For any bucket \(B\), we store every \(k\)th element of \(T\) in a succinct SB-tree. The space usage of the y-fast trie is \(O(m)\) bits, and that of the succinct SB-tree is \(O((m/k) \log \log \sigma)\) bits.

To support \(R(\cdot)\), we first perform a query on the y-fast trie, which takes \(O(\log \log \sigma)\) time. We then perform a query in the appropriate bucket, which takes \(O(1)\) time by looking up a pre-computed table (which is independent of \(T\)). The query in the bucket also requires \(O(1)\) calls to \(S(\cdot)\). We have so far computed the answer within \(k\) keys in \(T\): to complete the query for \(R(\cdot)\) we perform binary search on these \(k\) keys using \(O(\log k)\) calls to \(S(\cdot)\).

4.6.2 Supporting \(\text{rank}\) and \(\text{select}\)

In what follows, we use instances of Lemma 4.7 choosing \(k = 1\) and \(k = \log \log \sigma\) respectively. We now give the following result, contributing to Eq. (4.1). Note that the first option in Theorem 4.6 has optimal index size for \(t\) probes, for \(t \leq \log \sigma/\log \log \sigma\). The second option has optimal index size for \(t\) probes, for \(t \leq \log \sigma/\log (3\sigma)\), but only for \(\text{select}\).

Theorem 4.6. Given a string of length \(n\) over alphabet \([\sigma]\), for any \(1 \leq t \leq \log \sigma\), there exist systematic data structures with the following complexities:

(a) \(\text{select}\) in \(O(t)\) probes and \(O(t)\) time, and \(\text{rank}\) in \(O(t + \log \log \sigma)\) time using a succinct index with \(r = O(n(\log \log \sigma + (\log \sigma)/t))\) bits of redundancy. If \(\sigma = (\log n)^{O(1)}\), the \(\text{rank}\) operation requires only \(O(t)\) time and \(O(t)\) probes.

(b) \(\text{select}\) in \(O(t)\) probes and \(O(t \log \log \sigma)\) time, and \(\text{rank}\) in \(O(t \log (3\sigma))\) probes and \(O(t \log \log \sigma \log (3\sigma))\) time, using \(r = O(n(\log (3\sigma) + (\log \sigma)/t))\) bits of redundancy for the succinct index.

Proof. We divide the given string \(S\) into contiguous blocks of size \(\sigma\) (assume for simplicity that \(\sigma\) divides \(n = |S|\)). We start by showing how to reduce \(\text{rank}\) and \(\text{select}\) operations from the whole string to inside a block. Along the lines of \([\text{BHMR07d}]\) \([\text{GMR06}]\) we build an initial bitvector for each character \(c \in [\sigma]\) as \(V_c = 0^v_0 10^v_1 1 \cdots 0^{v_{(n/\sigma) - 1}} 1\), where \(v_i\) is \(|\{j \in [i\sigma, (i + 1)\sigma - 1] | S[j] = c\}|\). Concatenating all the bitvectors in a single one \(V\) and building a \(FID\) with \(O(1)\)-time \(\text{rank}\) and \(\text{select}\) requires in total \(O(n)\) bits and lets us reduce \(\text{select}\) and \(\text{rank}\) operations inside a block. To answer \(\text{select}(c, i)\) we have to perform the operation \(x = \text{select}_0(i) - i\) on the bitstring \(V_c\), as \(x\) then translates into the index of the...
4.6. UPPER BOUND

block where the $i$th instance of $c$ resides. The query can be emulated on $V$ by finding in $O(1)$ time where the bitvector for $V_c$ starts and how many 0s to offset the query with (notice that each $V_c$ ends after its share of $n/\sigma$ 1s). For $\text{rank}(c, p)$, we have to compute $p' = \sigma \lfloor p/\sigma \rfloor$ and $y = \text{select}_1(p') - p'$, so that $y$ contains the value of $\text{rank}(c, \sigma \lfloor p/\sigma \rfloor)$.

We now explain how to support operations in a block $B$. We denote the individual characters of $B$ by $B[0], \ldots, B[\sigma - 1]$.

To implement $\text{select}$, we continue one step further along the lines of [BHMR07b]: letting $n_c$ denote the multiplicity of character $c$ in $B$, we store the bitvector $Z = 1^{n_0}01^{n_1}0 \ldots 1^{n_{\sigma - 1}}0$, which is of length $2\sigma$, and augment it with the binary $\text{rank}$ and $\text{select}$ operations, using $O(\sigma)$ bits in all. Let $c = B[i]$ for some $0 \leq i \leq \sigma - 1$, and let $\pi[i]$ be the position of $c$ in a stably sorted ordering of the characters of $B$ ($\pi$ is a permutation). As in [BHMR07b], $\text{select}(c, \cdot)$ is reduced, via $Z$, to determining $\pi^{-1}(j)$ for some $j$. As shown in [MRRR03], for any $1 \leq t \leq \log \sigma$, permutation $\pi$ can be augmented with $O(\sigma + (\sigma \log \sigma)/t)$ bits so that $\pi^{-1}(j)$ can be computed in $O(t)$ time plus $t$ evaluations of $\pi(\cdot)$ for various arguments.

We now describe how to compute $\pi(i)$. If $T_c$ denotes the set of indexes in $B$ containing the character $c$, we store a minimal monotone hash function $h_{T_c}$ on $T_c$, for all $c \in [\sigma]$. We probe $B$ to find $c = B[i]$, and observe that $\pi(i) = \text{rank}_{B'}(c, i) + \sum_{i=0}^{c-1} n_i$. The term $\text{rank}_{B'}(c, i)$ can be solved by evaluating $h_{T_c}(i)$. The term $\sum_{i=0}^{c-1} n_i$ can be obtained in $O(1)$ time by $\text{rank}$ and $\text{select}$ operations on $Z$. By Theorem 4.5, the complexity of $\text{select}(c, i)$ is as claimed: we use Theorem 4.5(a) for our Theorem 4.6(a) solution and Theorem 4.5(b) for our Theorem 4.6(b) solution.

We now describe how to compute $\text{rank}$ inside a block. If $T_c$ is as above, we apply Lemma 4.7 to each $T_c$, once with $k = 1$ and once with $k = \log \log \sigma$ (to distinguish between Theorem 4.6(a) and Theorem 4.6(b)). Lemma 4.7 requires some calls to $S(\cdot)$, but this is just $\text{select}(c, \cdot)$ restricted to $B$, and is solved as described above. If $\sigma = (\log n)^{O(1)}$, then $|T_c| = (\log n)^{O(1)}$, and we store $T_c$ itself in the succinct SB-tree, which allows us to compute $\text{rank}_{B}(c, i)$ in $O(1)$ time using a (global, shared) lookup table.

The enhancements described here also lead to more efficient non-systematic data structures. Namely, for $\sigma = \Theta(n^\varepsilon)$, $0 < \varepsilon < 1$, we match the lower bound of [Gol09] Theorem 4.3. Moreover, we improve asymptotically both in terms of space and time upon the results of [BHMR07b], giving one of the major contributions of this section:

**Corollary 4.2.** There exists a data structure that represents any string $S$ of length $n$ using $nH_k(S) + O(n \log \log \sigma)$ bits, for any $k = O(\log n)$, supporting $\text{access}$ in $O(1)$ time, and $\text{rank}$ and $\text{select}$ in $O(\log \log \sigma)$ time.
Proof. We take the data structure of Theorem 4.6(a), where \( r = O\left(\frac{n \log \sigma}{\log \log \sigma}\right) \). We compress \( S \) using the high-order entropy encoder of [FV07, GN06, SG06] resulting in an occupancy of \( nH_k(S) + a \) bits, where

\[
a = O\left(\frac{n}{\log_\sigma n} (k \log \sigma + \log \log n)\right)
\]

is the extra space introduced by encoding. We observe that \( a = O\left(\frac{n \log \sigma}{\log \log \sigma}\right) \) for our choice of \( k \), hence it does not dominate on the data structure redundancy. Operation \texttt{access} is immediately provided in \( O(1) \) time by the encoded structure, thus the time complexity of Theorem 4.6 applies.

\[ \square \]

4.7 Subsequent results

The discussion of this chapter pivoted over the character probe/RAM model, which is a good model for realistic applications (e.g. 16-bit alphabets and 64-bit words) or for strings where alphabets are actually words of some underlying text, as it is required in certain applications. The upper bound also falls in the character probe model. The question “Is it possible to obtain a better systematic data structure in the Word RAM model – as opposite to the character RAM model?” as well as “Is it possible to have non-systematic data structures in the Word RAM model that perform better?” are perfectly natural. We cannot answer the first one positively, and we conjecture that unless \( w = \Omega(\sigma) \) it is not possible to change the space/time trade-off of Theorem 4.6 unless a deep rethinking of the data structure is involved.

The rationale behind that stands in the \texttt{select} data structure: proving a better bound requires to tackle the permutation inversion problem differently, since larger words are not directly usable (the pattern followed by the permutation can easily disrupt the gain of caching multiple characters of the string together).

Very recently, the authors of [BN11b], pointed out that the second question can be answered positively:

\textbf{Theorem 4.7.} Given a string \( S \) of length \( n \) over alphabet \([\sigma]\), for any \( \omega(1) = f(n, \sigma) = o(\log(\log \sigma / \log w)) \), where \( w \geq \log(\sigma + n) \) is the Word RAM size, there exists a non-systematic data structure storing \( S \) in

\[
n \log \sigma + O\left(\frac{n \log \sigma}{\log \sigma / \log w}\right) = n \log \sigma + o(n \log \sigma)
\]

bits. The data structure performs \texttt{access} in \( O(1) \) time, \texttt{select} in \( O(f(n, \sigma)) \) time and \texttt{rank} in \( O(\log(\log \sigma / \log w)) \) time.
The same paper also points out that the execution time is tight for \textbf{rank}:

\textbf{Theorem 4.8.} Any data structure on a Word RAM of size $w$ storing a string $S$ of length $n$ over alphabet $[\sigma]$ in $O(nw^{O(1)})$ bits must answer \textbf{rank} queries in $\Omega(\log(\log \sigma / \log w))$ time.

Theorem 4.8 proves that the data structure of Theorem 4.6 is optimal as long as $\log \sigma = (\log w)n^{\Theta(1)}$. Since we always need $w \geq \log(\sigma + n)$, to maintain such condition we need

$$\max\{w, \log n\} = O\left(2^{\log^{1-\varepsilon} \sigma}\right),$$

for some $0 \leq \varepsilon < 1$. As such, in standard conditions where $w = \Theta(\log(\sigma + n))$, our data structure is optimal for situations such as $n = \sigma^{O(1)}$ or $n = 2^{\sqrt{\log n}}$. Or, again in standard conditions, when $\log \sigma = \Omega(\log^c w)$, for some $c > 1$, which is for example when $\log \sigma = \Omega(\log^2 \log n)$. 

Chapter 5

Approximate substring selectivity estimation

The current chapter deals with the application of rank and select data structures for sequences and Burrows-Wheeler transform based compressed indexing algorithms. Given a text $T$ of length $n$ over an alphabet $\sigma$, the substring selectivity estimation problem requires to answer queries $\text{Count}(P)$, for some $P \in [\sigma]^n$, namely, to approximately count the number of occurrences of $P$ inside $T$. We seek indexes that can be created in sublinear space and answer queries with predictable error bounds and fast times. We propose both theoretical and practical results. The work of this chapter is based on the contents of [OV11].

5.1 Scenario

When massive data sets are involved, the cost for indexing the data may be non-negligible, and thus compressing the data is mandatory. Compressed text indexing discussed in Section 2.6.2 meets this pattern. However, there exists a lower bound on the compression ratio it can achieve, and such a limit can be surpassed by allowing pattern matching operations to have approximated results. This is a realistic scenario, as with massive amounts of data and answers that provide millions of strings, a small absolute error is clearly tolerable in many situations.

In this chapter we follow such idea by studying the above problem of Substring Occurrence Estimation.

Given a text $T[1,n]$ drawn from an alphabet $\Sigma$ of size $\sigma$ and fixed any error parameter $l$, we design an index that, without the need of accessing/storing the original text, is able to count the number of occurrences of any pattern $P[1,p]$ in $T$. The index is allowed to err by at most $l$: precisely, the reported number of
occurrences of \( P \) is in the range \([\text{Count}(P), \text{Count}(P) + l - 1]\) where \( \text{Count}(P) \) is the actual number of occurrences of \( P \) in \( T \). In the following we will refer to it as \( \text{Count} \approx_l(P) \), which has uniform error range. We also consider a stronger version of the problem denoted \( \text{Count} \geq_l(P) \), namely having lower-sided error range, where \( \text{Count} \geq_l(P) = \text{Count}(P) \) whenever \( \text{Count}(P) \geq l \), and \( \text{Count} \geq_l(P) \in [0, l - 1] \) otherwise.

We also consider multiplicative error, that is, when the estimation ranges in \([\text{Count}(P), (1 + \varepsilon)\text{Count}(P)]\) for some fixed \( \varepsilon > 0 \). In theory, this error could provide better estimates for low frequency patterns. Obtaining multiplicative error would imply an index able to discover for sure whether a pattern \( P \) appears in \( T \) or not (set \( \text{Count}(P) = 0 \) in the above formulas). This turns out to be the hard part of estimation. In fact, we are able to prove (Theorem 5.8) that an index with multiplicative error would require as much space as \( T \) to be represented; hence, the forthcoming discussion will focus solely on additive error.

Our objective is to heavily reduce the space of compressed text indexes as \( l \) increases.

We provide two different solutions: one in the uniform error model and one in the lower-sided error model. Section 5.3 illustrates the former and shows how to build an index (called \( \text{APX}_l \)) that requires \( O(n \log(\sigma l)/l) \) bits of space. This is the first index that has both guaranteed space, sublinear with respect to the size of the indexed text, and provable error bounds. It turns out (Section 5.5) that the index is space-optimal up to constant factors for sufficiently small \( l \), namely, \( \log l = O(\log \sigma) \).

We also provide a data structure (\( \text{CPST}_l \)) for the lower-sided error model (Section 5.4) that presents a space bound of \( O(m \log(\sigma l)) \) bits where \( m \) is the number of nodes in a pruned suffix tree. This solution needs to be compared to storing a pruned suffix tree plainly, with its edge labels, which would account for up to \( m^2 \cdot O(\log \sigma) \) bits. We remark that, in practice, even when the size of edge labels is smaller than \( m^2 \), it still accounts for the majority of occupied space. The \( \text{CPST}_l \) data structure outperforms our previous solution only when \( m = O(n/l) \); surprisingly, many real data sets exhibit the latter property. The \( \text{CPST}_l \) construction will prove how \texttt{rank} and \texttt{select} over sequences can be crucial contributors to build higher-functionality data structures on strings.

In Section 5.6 we support our claims with tests on real data sets. We show the improvement in space occupancy of both \( \text{APX}_l \) and \( \text{CPST}_l \), both ranging from 5 to 60 w.r.t. standard pruned suffix trees and we also show our sharp advantage over compressed text indexing solutions. As an example, for an English text of about 512 MB, it suffices to set \( l = 256 \) to obtain an index of 5.1 MB (roughly, 1%). We also confirm that \( m \) and \( n/l \) are close most of the times. Combining existing
selectivity estimation algorithms and our CPST with reasonably small error rate, we show it is possible to solve the selectivity estimation problem with an average additive error of 1 by occupying around $1/7$ of the original text size.

5.2 Preliminaries

We will make use of non-systematic rank/select data structures that also compress the original string to zero-th entropy. In particular, we employ another result of [BN11b], which is an extension of Theorem 4.7:

**Theorem 5.1.** Given a string $S$ of length $n$ over alphabet $[\sigma]$, where $w \geq \log(\sigma + n)$ is the Word RAM size, there exists a non-systematic data structure storing $S$ in $(1 + o(1))nH_0(S) + o(n)$ bits, solving

- for $\sigma = \omega(\log O(1)n)$, rank in $O(\log(\log(\sigma/\log w)))$ time, access and select in $O(\sqrt{\log(\log(\sigma/\log w))})$ time.
- for $\sigma = O(\log O(1)n)$, all operations in $O(1)$ time.

In general, our developments rely on the Burrows-Wheeler Transform (BWT) and backward search algorithm described in Section 2.6.3 and on the suffix trees described in Section 5.2.1. Throughout the rest of this chapter, we will assume to deal with a text $T$ of length $n$ from the alphabet $\Sigma = [\sigma]$ for some $2 \leq \sigma \leq n$.

5.2.1 Suffix trees

We now review the suffix tree of [Gus97], introducing useful notation and some of its properties. The suffix tree [Gus97] of a text $T$ is the compacted trie, i.e., a trie in which all unary nodes are omitted, denoted as $ST(T)$ or simply $ST$, built on all the $n$ suffixes of $T$. We ensure that no suffix is a proper prefix of another suffix by simply assuming that a special symbol, say $\$$, terminates the text $T$. The symbol $\$$ does not appear anywhere else in $T$ and is assumed to be lexicographically smaller than any other symbol in $\Sigma$. This constraint immediately implies that each suffix of $T$ has its own unique leaf in the suffix tree, since any two suffixes of $T$ will eventually follow separate branches in the tree. The label of an edge is simply the substring in $T$ corresponding to the edge. For edge between nodes $u$ and $v$ in $ST$, the label of such edge (denoted $label(u, v)$) is always a non-empty substring of $T$. For a given node $u$ in the suffix tree, its path label (denoted $pathlabel(u)$) is defined as the concatenation of edge labels on the path from the root to $u$. The string depth of node $u$ is $|pathlabel(u)|$. In order to allow a linear-space representation of
the tree, each edge label is usually represented by a pair of integers denoting the starting position in $T$ of the substring describing the edge label and its length. In this way, the suffix tree can be stored in $\Theta(n \log n)$ bits of space. It is well-known that to search a pattern $P[1,p]$ in $T$ we have to identify, if any, the highest node $u$ in $ST$ such that $P$ prefixes $\text{pathlabel}(u)$. To do this, we start from the root of $ST$ and follow the path matching symbols of $P$, until a mismatch occurs or $P$ is completely matched. In the former case $P$ does not occur in $T$. In the latter case, each leaf in the subtree below the matching position gives an occurrence of $P$. The number of these occurrences can be obtained in constant time by simply storing in any node $u$ the number $C(u)$ of leaves in its subtree. Therefore, this algorithm counts the occurrences of any pattern $P[1,p]$ in time $O(p \log \sigma)$. This time complexity can be reduced up to $O(p)$ by placing a (minimal) perfect hashing function [HT01] in each node to speed up percolation. This will increase the space just by a constant factor.

### 5.2.2 Pruned suffix trees

The main data structure for occurrence estimation, and the one used in [KV196, JNS99], is the pruned suffix tree $\mathcal{PST}_l(T)$. For a fixed error $l \geq 1$, the $\mathcal{PST}_l(T)$ is obtained from the suffix tree of $T$ by pruning away all nodes of suffixes that appear less than $l$ times in $T$. Examples are illustrated in Figure 5.5 and 5.6. It is immediate to see that the resulting data structure has, indeed, lower-sided error. However, the space occupancy of $\mathcal{PST}_l$ is a serious issue, both in theory and practice: it requires a total of $O(m \log n + g \log \sigma)$ bits where $m$ is the number of nodes surviving the pruning phase and $g$ is the amount of symbols that label the edges of such nodes. There are two main space-related drawbacks in this solution:

1. The number of nodes in the pruned tree may be very high.

2. It may waste a lot of space due to the need of explicitly storing edges’ labels $g = \Theta(n^2)$.

For the first point, we observe that in the worst case the ratio between $l$ (i.e., the error) and the space occupancy of the resulting suffix tree is far away from the optimal one. The number of nodes in the pruned tree could raise to $\Theta(n - l)$ and could slowly decrease as the error $l$ increases: observe that we require to increase the error up to $n/2$ just to halve the number of nodes in the tree. Consider the text $T = a^n \$$. The shape of its suffix tree is a long chain of $n - 1$ nodes with two children each. Therefore, for any value of $l$, the space required to store explicitly its pruned suffix tree is at least $\Omega((n - l) \log n)$ bits. This quantity further increases due to the need of storing explicitly edges’ labels. We point out that the number of
these symbols is at least equal to the number of nodes but can significantly increase whenever the suffixes represented in the tree share long common prefixes. It goes without saying that the number of symbols we need to store can exceed the length of the text itself. One could resort to techniques like blind search over compacted tries \cite{FG99a} to remove the need of storing full labels for the edges. However, as explained later, this would incur in an uncontrollable error when the pattern is not in $\mathcal{PST}_l$, since solutions based on compacted tries require the original text to perform membership queries. Thus, the space occupancy of the pruned suffix tree may be not sublinear w.r.t. the text. Moreover, the lower bound of Theorem \ref{thm:space-complexity} formally proves that the space complexity for an index with threshold $l$ is $\Omega(n \log(\sigma)/l)$ bits, hence stating that a pruned suffix tree is highly non-optimal.

To provide solutions with smaller footprint, one can resort to compressed full-text indexes \cite{FM05, GV05, FGNV08, NM07}, which are well known in the field of succinct data structures. They deliver a framework to keep a copy of text $T$ compressed together with auxiliary information for efficient (i.e., without decompressing the whole $T$) substring search. Such solutions however work on the entire text and are not designed to allow errors or pruning of portions of the string, yet they provide a good baseline for our work.

5.2.3 Naive solutions for occurrence estimation

In this section we describe some naive solutions for occurrence estimation and highlight their weak points.

An alternative pruning strategy w.r.t. pruned suffix trees above consists in building a pruned Patricia Trie $\mathcal{PT}_{l/2}(T)$ that stores just a suffix every $l/2$ suffixes of $T$ sorted lexicographically and resort to Blind Search (refer back to Section 3.3.3 for the binary version). A plain Patricia trie coincides with $\mathcal{ST}(T)$ in which we replace each substring labeling an edge by its first symbol only, which we recall it is named branching symbol. More formally, let $T_1, T_2, \ldots, T_n$ denote the $n$ suffixes of $T$ sorted lexicographically, $\mathcal{PT}_{l/2}(T)$ is the Patricia trie of the set of $O(n/l)$ strings $\mathcal{S} = \{T_i \mid i \equiv 1 \pmod{l/2}\}$. The pruned Patricia trie $\mathcal{PT}_{l/2}(T)$ can be stored in $O(n/l \cdot (\log \sigma + \log n)) = O(n \log n/l)$ bits. We use the blind search described in \cite{FG99a} to search a pattern $P[1, p]$ in time $O(p)$. Such algorithm returns a node $u$ that either corresponds to $P$, if $P$ is a prefix of some string in $\mathcal{S}$ or another node otherwise (whereas there is a connection between such node and $P$, without the original text it is not possible to exploit it). Once we identify the node $u$, we return the number of leaves descending from that node multiplied by $l$. If $P$ occurs at least $l/2$ times in $T$, it is easy to see that the number of reported occurrences is a correct approximation of its occurrences in $T$. Instead, if $P$ occurs less than $l/2$ times in
CHAPTER 5. APPROXIMATE SUBSTRING SELECTIVITY ESTIMATION

T, the blind search may return a different node. Thus, in such cases the algorithm may fail by reporting as result a number of occurrences that may be arbitrarily far from the correct one.

A similar solution resorts to a recent data structure presented by Boldi et al. [BBPV10]. Their solution solves via hashing functions a problem somehow related to ours, called weak prefix search. The problem is as follows: We have a set \( \mathcal{V} \) of \( v \) strings and want to build an index on them. Given a pattern \( P \), the index outputs the ranks (in lexicographic order) of the strings that have \( P \) as prefix; if such strings do not exist the output of the index is arbitrary. Their main solution needs just \( O(|P| \log \sigma/w + \log |P| + \log \log \sigma) \) time and \( O(v \log (L \log \sigma)) \) bits of space, where \( L \) is the average length of the strings in the set and \( w \) is the machine word size. We can use their data structure to index the set of suffixes \( S \), so that we can search \( P[1,p] \) and report its number of occurrences multiplied by \( l \). Since in our case \( L = \Theta(n) \), the index requires \( O(n \log (n \log \sigma)/l) = O(n \log n/l) \) bits of space. As in the case of pruned Patricia tries, the answer is arbitrary when \( P \) is not prefix of any suffix in \( S \) (i.e., it occurs less than \( l \) times). Hence, this solution improves the time complexity but has the same drawback of the previous one.

5.2.4 Previous work

In this section we present in detail the three main algorithms for substring selectivity estimation: KVI [KVI96], the class of MO-based estimators [JNS99] and CRT [CGG04], in chronological order. All the algorithms we will describe suffer from either one or the other of the following drawbacks:

- They are not space optimal, since they require \( \Theta(n \log n/\ell) \) bits of space and/or
- They actually solve a relaxed version of our problem in which we do not care about the results whenever the patterns occur less than \( l \) times.

For a given threshold \( l \), the work of KVI starts by assuming to have a data structure answering correctly to queries \( \text{Count}(P) \) when \( \text{Count}(P) \geq l \) and strives to obtain a one-sided error estimate for infrequent (< \( l \)) strings. It also assumes the data structure can detect if \( \text{Count}(P) < l \). Their main observation is as follows: let \( P = \alpha\beta \) where \( \text{Count}(P) < l \) and assume \( \text{Count}(\alpha) \geq l \) and \( \text{Count}(\beta) \geq l \), then one can estimate \( \text{Count}(P) \) from \( \text{Count}(\alpha) \) and \( \text{Count}(\beta) \) in a probabilistic way, using a model in which the probability of \( \beta \) appearing in the text given that \( \alpha \) appears is roughly the same of \( \beta \) appearing by itself. Generalizing this concept, KVI starts from \( P \) and retrieves the longest prefix of \( P \), say \( P' \), such that \( \text{Count}(P') > l \), and then reiterates on the remaining suffix.
5.3. UNIFORM ERROR RANGE

Requiring the same kind of data structure beneath, the MO class starts by observing that instead of splitting the pattern $P$ into known fragments of information, one can rely on the concept of maximum overlap: given two strings $\alpha$ and $\beta$, the maximum overlap $\alpha \oslash \beta$ is the longest prefix of $\beta$ that is also a suffix of $\alpha$. Hence, instead of estimating $\text{Count}(P)$ from $\text{Count}(\alpha)$ and $\text{Count}(\beta)$ alone, it also computes and exploits the quantity $\text{Count}(\alpha \oslash \beta)$. In probabilistic terms, this is equivalent to introducing a light form of conditioning between pieces of the string, hence yielding better estimates. The change is justified by an empirically proved Markovian property that makes maximum overlap estimates very significant. MO is also presented in different variants: MOC, introducing constraints from the strings to avoid overestimation, MOL, performing a more thorough search of substrings of the pattern, and MOLC, combining the two previous strategies.

In particular, MOL relies on the lattice $L_P$ of the pattern $P$. For a string $P = a \cdot \alpha \cdot b$ ($|\alpha| \geq 0$), the $l$-parent of $P$ is the string $a \cdot b$ and the $r$-parent of $P$ is $a \cdot \alpha$. The lattice $L_P$ is described recursively: $P$ is in the lattice and for any string $\zeta$ in the lattice, also its $l$-parent and $r$-parent are in the lattice. Two nodes $\beta$ and $\zeta$ of the lattice are connected if $\beta$ is an $l$-parent or an $r$-parent of $\zeta$ or vice versa. To estimate $\text{Count}(P)$, the algorithm starts by identifying all nodes in the lattice for which $\text{Count}(\alpha)$ can be found in the underlying data structure and retrieve it, so that $Pr(\alpha) = \text{Count}(\alpha)/N$, where $N$ is a normalization factor. For all other nodes, it computes $Pr(a \cdot \alpha \cdot b) = Pr(a \cdot \alpha) \times Pr(\alpha \cdot b)/Pr(a \cdot \alpha \oslash \alpha \cdot b)$ recursively. In the end, it obtains $Pr(P)$, i.e. the normalized ratio of occurrences of $P$ in $T$.

The CRT method was presented to circumvent underestimation, a problem that may afflict estimators with limited probabilistic knowledge as those above. The first step is to build an a-priori knowledge of which substrings are highly distinctive in the database: in that, they rely on the idea that most patterns exhibit a short substring that is usually sufficient to identify the pattern itself. Given a pattern to search for, they retrieve all distinctive substrings of the pattern and use a machine learning approach to combine their value. At construction time, they train a regression tree over the distinctive substrings by using a given query log; the tree is then exploited at query time to obtain a final estimate.

5.3 Uniform error range

In this section we describe our first data structure which is able to report the number of occurrences of any pattern within an additive error at most $l$. Its error/space trade-off is provably optimal whenever the error $l$ is such that $\log l = O(\log \sigma)$. In this section we will prove the following theorem:
Theorem 5.2. Given \( T[1,n] \) drawn from an alphabet \( \Sigma \) of size \( \sigma \) and fixed an error threshold \( l \), there exists an index that answers \( \text{Count} \approx_i(P[1,p]) \) in \( O(p \times f(\sigma)) \) time by using \( O((n \log(\sigma l))/l + \sigma \log n) \) bits of space, where \( f(\sigma) \) depends on the chosen rank and select data structure (see Theorem 5.1).

The idea behind our solution is that of sparsifying the string \( L = \text{bwt}(T) \) by removing most of its symbols (namely, for each symbol we just keep track of one every \( l/2 \) of its occurrences). Then, we provide an algorithm that, even though, can provide sufficiently good results on this sampled BWT. Similarly to the backward search, our algorithm searches a pattern \( P[1,p] \) by performing \( p \) phases. In each of them, it computes two indexes of rows of \( M(T) \): \( \text{First} \approx_i \) and \( \text{Last} \approx_i \); the two are obtained by first performing rank queries on the sampled BWT and then apply a correction mechanism. Corrections are required to guarantee that both indexes are within a distance \( l/2 \) from the actual indexes \( \text{First}_i \) and \( \text{Last}_i \), the indexes that the backward search would compute for \( P \) in phase \( i \). More formally, in each phase it is guaranteed that \( \text{First} \approx_i \in [\text{First}_i - (l/2) - 1, \text{First}_i] \) and \( \text{Last} \approx_i \in [\text{Last}_i, \text{Last}_i + (l/2) - 1] \). Clearly, also the last step obeys to the invariant, hence all rows in \( [\text{First} \approx_1, \text{Last} \approx_1] \) contain suffixes prefixed by \( P \), with the possible exception of the first and last \( l/2 \) ones. Hence, the maximum error such an algorithm can commit is \( l \).

For each symbol \( c \), the sampling of \( L = \text{bwt}(T) \) keeps track of a set \( D_c \) of positions, called discriminant positions (for symbol \( c \)), containing:

- the position of the first occurrence of \( c \) in \( L \);
- the positions \( x \) of the \( i \)th occurrence of \( c \) in \( L \) where \( i \) mod \( l/2 \equiv 0 \);
- the position of the last occurrence of \( c \) in \( L \).

Algorithm 1 searches a pattern \( P[1,p] \) by performing predecessor and successor queries on sets \( D_c \). The crucial steps are lines 4–9 where the algorithm computes the values of \( \text{First} \approx_{i-1} \) and \( \text{Last} \approx_{i-1} \) using the values computed in the previous phase. To understand the intuition behind these steps, let us focus on the computation of \( \text{First} \approx_{i-1} \) and assume that we know the value of \( \text{First}_i \). The original backward search would compute the number of occurrences, say \( v \), of symbol \( c \) in the prefix \( L[1: \text{First}_i - 1] \). Since our algorithm does not have the whole \( L \), the best it can do is to identify the rank, say \( r \), of the position in \( D_c \) closest to (but larger than) \( \text{First}_i \). Clearly, \( r \cdot l/2 - l/2 < v \leq r \cdot l/2 \). Thus, setting \( \text{First} \approx_{i-1} = C[c] + r \cdot l/2 - l/2 - 1 \)

\[1\]

We recall that a predecessor query \( \text{Pred}(x, A) \) returns the predecessor of \( x \) in a set \( A \), namely, \( \max\{y \mid y \leq x \land y \in A\} \). A successor query is similar but finds the minimum of \( y \geq x \).
Algorithm 1 Our algorithm to find the approximate range $[\text{First}_1, \text{Last}_1]$ of rows of $\mathcal{M}(T)$ prefixed by $P[1, p]$ (if any).

Algorithm $\text{Count}_{\approx i}(P[1, p])$

1. $i = p$, $c = P[p]$, $\text{First}_{\approx p} = C[c] + 1$, $\text{Last}_{\approx p} = C[c + 1]$;

2. while ((First$\approx_i \leq$ Last$\approx_i$) and ($i \geq 2$)) do

3. $c = P[i - 1]$;

4. DiscrFirst$_i = \text{Succ}(\text{First}_{\approx i}, D_c)$

5. $RL = \min(\text{DiscrFirst}_i - \text{First}_{\approx i}, l/2 - 1)$

6. $\text{First}_{\approx i-1} = LF(\text{DiscrFirst}_i) - RL$

7. DiscrLast$_i = \text{Pred}(\text{Last}_{\approx i}, D_c)$

8. $RR = \min(\text{Last}_{\approx i} - \text{DiscrLast}_i, l/2 - 1)$

9. $\text{Last}_{\approx i-1} = LF(\text{DiscrLast}_i) + RR$

10. $i = i - 1$;

11. if (Last$\approx_i <$ First$\approx_i$) then return "no occurrences of $P$" else return $[\text{First}_{\approx i}, \text{Last}_{\approx i}]$. 
would suffice to guarantee that \( \text{First}_{\approx l-1} \in [\text{First}_{l-1} - (l/2 - 1), \text{First}_{l-1}] \). Notice that we are using the crucial assumption that the algorithm knows \( \text{First}_i \). If we replace \( \text{First}_i \) with its approximation \( \text{First}_{\approx i} \), this simple argumentation cannot be applied since the error would grow phase by phase. Surprisingly, it is enough to use the simple correction retrieved at line 5 and applied at line 6 to fix this problem. The following Lemma provides a formal proof of our claims.

**Lemma 5.1.** For any fixed \( l \geq 0 \) and any phase \( i \), both \( \text{First}_{\approx i} \in [\text{First}_i - (l/2 - 1), \text{First}_i] \) and \( \text{Last}_{\approx i} \in [\text{Last}_i, \text{Last}_i + l/2 - 1] \) hold.

**Proof.** We prove only that \( \text{First}_{\approx i} \in [\text{First}_i - (l/2 - 1), \text{First}_i] \) (a similar reasoning applies for \( \text{Last}_{\approx i} \)). The proof is by induction. For the first step \( p \), we have that \( \text{First}_{\approx p} = \text{First}_p \), thus the thesis immediately follows. Otherwise, we assume that \( \text{First}_{\approx i} \in [\text{First}_i - (l/2 - 1), \text{First}_i] \) is true and prove that \( \text{First}_{\approx i+1} \in [\text{First}_{i+1} - (l/2 - 1), \text{First}_{i+1}] \). Recall that \( \text{First}_{i+1} \) is computed as \( C[c] + \text{rank}_c(L, \text{First}_i - 1) + 1 \). We distinguish two cases: (1) \( \text{First}_i \leq \text{DiscrFirst}_i \) and (2) \( \text{First}_i > \text{DiscrFirst}_i \), both of which are illustrated also in Figure 5.1.

**Case 1** Let \( z \) be the number of occurrences of symbol \( c \) in \( L[\text{First}_i, \text{DiscrFirst}_i - 1] \), so that \( \text{First}_{i+1} = \text{LF}(\text{DiscrFirst}_i) - z \). Then, the difference \( \Delta = \text{First}_{i+1} - \text{First}_{\approx i+1} \) equals to \( \text{LF}(\text{DiscrFirst}_i) - z - \text{LF}(\text{DiscrFirst}_i) + \min(\text{DiscrFirst}_i - \text{First}_{\approx i}, l/2 - 1) = \min(\text{DiscrFirst}_i - \text{First}_{\approx i}, l/2 - 1) - z \). Since (by inductive hypothesis) \( 0 \leq \text{First}_i - \text{First}_{\approx i} \leq l/2 \) and \( \text{DiscrFirst}_i \) is the closest discriminant position for \( c \) larger than \( \text{First}_{\approx i} \), we have that \( z \leq \min(\text{DiscrFirst}_i - \text{First}_{\approx i}, l/2 - 1) \). Hence \( 0 \leq \Delta \leq l/2 \).

**Case 2** Let \( k = \text{DiscrFirst}_i - \text{First}_{\approx i} \) and \( z \) be the number of occurrences of \( c \) in \( L[\text{DiscrFirst}_i, \text{First}_i - 1] \). Start by noting that \( z < l/2 \) since \( L[\text{DiscrFirst}_i, \text{First}_i - 1] \) contains at most \( l/2 \) symbols. Since (by inductive hypothesis) our error at step \( i \) is less than \( l/2 \), namely \( \text{First}_i - \text{First}_{\approx i} < l/2 \), we have that \( k + z < |\text{First}_i - \text{First}_{\approx i}| < l/2 \); moreover, \( \text{First}_{i+1} \) can be rewritten as \( \text{LF}(\text{DiscrFirst}_i) + z + 1 \). Since \( \text{First}_{i+1} = \text{LF}(\text{DiscrFirst}_i) - k \), we have \( \text{First}_{i+1} - \text{DiscrFirst}_i = \text{LF}(\text{DiscrFirst}_i) + z + 1 - \text{LF}(\text{DiscrFirst}_i) + k = z + k + 1 \leq l/2 \). Finally, since \( k \) and \( z \) are non negative, \( \text{First}_{i+1} - \text{DiscrFirst}_i \) is non negative.

By combining Lemma 5.1 with the proof of correctness of Backward Search (Lemma 3.1 in [FM05]) we easily obtain the following Theorem.

**Theorem 5.3.** For any pattern \( P[1,p] \) that occurs \( \text{Count}(P) \) times in \( T \) Algorithm 1 returns in \( O(p) \) steps as result a value \( \text{Count}_{\approx i}(P) \in [\text{Count}(P), \text{Count}(P) + l - 1] \).

\(^{2}\)Observe that \( L[\text{DiscrFirst}_i] = c \) by definition of discriminant position for \( c \).
5.3. UNIFORM ERROR RANGE

Notice that, if \([\text{First}, \text{Last}]\) is the range of indexes corresponding to the consecutive suffixes that are prefixed by \(P\), then the algorithm identifies a range \([\text{First}\approx, \text{Last}\approx]\) such that \(\text{First} - l/2 < \text{First}\approx \leq \text{First}\) and \(\text{Last} \leq \text{Last}\approx < \text{Last} + l/2\).

It remains to show how to represent the sets of discriminant positions \(D_c\) to support predecessor and successor queries on them. We represent all of these sets by means of two different objects. We conceptually divide the string \(L = \text{bwt}(T)\) into \(\lceil 2n/l \rceil\) blocks of equal length and for each of them we create the characteristic set \(B_i\), such that \(B_i\) contains \(c\) iff there exists a position in \(D_c\) belonging to block \(i\). Note that since each block has length \(\lfloor l/2 \rfloor\), the construction procedure for \(D_c\) guarantees that there can only be one discriminant position per character in any block. Considering sets \(B_i\) as strings (with arbitrary order), we compute the string \(B = B_0\#B_1\#\ldots B_{2n/l}\#\) where \# is a symbol outside \(\Sigma\) and augment it with \text{rank} and \text{select} data structures (see Theorem \[5.1\]). Let \(r\) be the total number of discriminant positions. We also create an array \(V\) of \(r\) cells, designed as follows. Let \(x\) be a discriminant position and assume that it appears as the \(j\)th one in \(B\), then \(V[j] = x \mod l/2\). The following lemma states that a constant number of \text{rank} and \text{select} queries on \(B\) and \(V\) suffices for computing \(\text{Pred}(x, D_c)\) and \(\text{Succ}(x, D_c)\).

**Lemma 5.2.** \(\text{Pred}(x, D_c)\) and \(\text{Succ}(x, D_c)\) can be computed with a constant number of \text{rank} and \text{select} queries on \(B\) and \(V\).

**Proof.** We show only how to support \(\text{Pred}(x, D_c)\) since \(\text{Succ}(x, D_c)\) is similar. Let \(p = \text{rank}_c(B, \text{select}_\#(B, [2x/l]))\), denoting the number of blocks containing a discriminant position of \(c\) before the one addressed by \([2x/l]\). Let \(q = \text{select}_c(B, p) - [2x/l]\)
be the index of the discriminant position preceding \( x \) (the subtraction removes the 
# spurious symbols). Then, \( g = \text{rank}_\#(B, \text{select}_c(B, p)) \) finds the block preceding 
(or including) \( \lfloor 2x/l \rfloor \) that has a discriminant position for \( c \). Also, \( V[q] \) con-
tains the offset, within that block, of the discriminant position. Such position can 
be either in a block preceding \( \lfloor 2x/l \rfloor \) or in the same block. In the former case, 
\( \text{Pred}(x, D_c) = \lfloor 2x/l \rfloor g + V[q] \). In latter case we have an additional step to make, 
as we have so far retrieved a position that just belongs to the same block of \( x \) but 
could be greater than \( x \). If that happens, we decrease \( p \) by 1 and repeat all the 
calculations. Note that since the first occurrence of \( c \) is also a discriminant than 
this procedure can never fail.

Once we have computed the correct discriminant positions, Algorithm 1 requires 
to compute an LF-step from them (lines 7 and 9). The following Lemma states that 
this task is simple.

**Fact 5.1.** For any symbol \( c \), given any discriminant position \( d \) in \( D_c \) but the largest 
one, we have that \( \text{LF}(d) = C[c] + (i - 1) \cdot l/2 + 1 \) where \( i \) is such that \( D_c \)’s \( i \)th element 
in left-to-right position is \( d \). For the largest discriminant position \( d \) in \( D_c \) we have 
\( \text{LF}(d) = C[c + 1] \).

It follows immediately that while performing the calculations of Lemma 5.2 we 
can also compute the LF mapping of the discriminant position retrieved.

**Proof of Theorem 5.2**

Correctness has been proved. The time complexity is easily seen to be \( O(|P|) \) 
applications of Lemma 5.2 hence the claim. The space complexity is given by three 
elements. The array \( C \), containing counters for each symbol, requires \( O(\sigma \log n) \) 
bits. The number of discriminant positions is easily seen to be at most \( 2n/l \) in total, 
however the array \( V \) requires at most \( O(n/l) \) cells of \( O(\log l) \) bits each. Finally, the 
string \( B \) requires one symbol per block plus one symbol per discriminant position, 
accounting for \( O(n \log \sigma/l) \) bits in total. The theorem follows.

### 5.4 Lower-side error range

Let \( \mathcal{PST}_l(T) \) be the pruned suffix tree as discussed in Section 5.1 and let \( m \) be the 
number of its nodes. Recall that \( \mathcal{PST}_l(T) \) is obtained from the suffix tree of \( T \) by 
removing all the nodes with less than \( l \) leaves in their subtrees, and hence constitutes 
a good solution to our lower-sided error problem: when \( \text{Count}(P) \geq l \), the answer 
is correct, otherwise an arbitrary number below \( l \) can be returned. Compared with
the solution of Section 5.3, it has the great advantage of being perfectly correct if
the pattern appears frequently enough, but it is extremely space inefficient. Our
objective in this section is to present a compact version of $\mathcal{PST}_l(T)$, by means of
proving the following theorem.

**Theorem 5.4.** Given $T[1,n]$ drawn from an alphabet $\Sigma$ of size $\sigma$ and given an error
threshold $l$, there exists a representation of $\mathcal{PST}_l(T)$ using $O(m \log(\sigma l) + \sigma \log n)$
bits that can answer to $\text{Count}_{\geq l}(P)$ in $O(|P| \times f(\sigma))$ time where $m$ is the number of
nodes of $\mathcal{PST}_l(T)$ and $f(\sigma)$ is the chosen rank and select time complexity summed up (see Theorem 5.1).

To appreciate Theorem 5.4 consider that the original $\mathcal{PST}_l(T)$ representation
requires, apart from node pointers, to store labels together with their lengths, for a
total of $O(m \log n + g \log \sigma)$. The predominant space complexity is given by the edge
labels, since it can reach $n \log \sigma$ bits even when $m$ is small. Therefore, our objective
is to build an alternative search algorithm that does not require all the labels to be
stored.

**5.4.1 Computing counts**

As a crucial part of our explanation, we will refer to nodes using their preorder traversal
times, with an extra requirement. Recall from Section 5.2.1 that the branching
symbol in a set of children of node $u$ is the first symbol of children edge labels.
During the visit we are careful to descend into children in ascending lexicographical
order over their branching symbols. Therefore, $u < v$ iff $u$ is either an ancestor
of $v$ or their corresponding path labels have the first mismatching symbols, say in
position $k$, such that $\text{pathlabel}(u)[k] < \text{pathlabel}(v)[k]$.

We begin by explaining how to store and access the basic information that our
algorithm must recover: Given a node $u \in \mathcal{PST}_l(T)$ we would like to compute $C(u)$,
the number of occurrences of $\text{pathlabel}(u)$ as a substring in $T$. A straightforward
storage of such data would require $m \log n$ bits for a tree of $m$ nodes. We prove
we can obtain better bounds and still compute $C(u)$ in $O(1)$ time, based on the
following simple observation:

**Observation 5.1.** Let $u$ be a node in $\mathcal{PST}_l(T)$ and let $v_1, v_2, \ldots, v_k$ be the children
of $u$ in $\mathcal{PST}_l(T)$ that have been pruned away. Denote by $g(u)$ the sum $C(v_1) + C(v_2) + \cdots + C(v_k)$. Then $g(u) < \sigma l$.

**Proof.** Each of the $v_i$s represents a suffix that has been pruned away, hence, for any
$i$, it holds $C(v_i) < l$ by definition. Since each node can have at most $\sigma$ children, the
observation follows. \qed

\footnote{Notice that $C(u)$ is the number of leaves in the subtree of $u$ in the original suffix tree.}
Note that Observation 5.1 applies in a stronger form to leaves, where for a leaf \( x \), \( C(x) = g(x) \). We refer to the \( g(\cdot) \) values as *correction factors* (albeit for leaves they are actual counts). For an example refer to Figure 5.6. It is easy to see that to obtain \( C(v) \) it suffices to sum all the correction factors of all the descendants of \( v \) in \( PST(T) \). Precisely, it suffices to build the binary string \( G = 0^{g(0)}10^{g(1)}1 \cdots 0^{g(m-1)}1 \) together with support for binary select queries.

**Lemma 5.3.** Let \( v \in PST(T) \) and let \( z \) be the identifier of its rightmost leaf. Define \( CNT(u,z) = select_1(G,z) - z - select_1(G,u) + u \). Then \( C(u) = CNT(u,z) \).

**Proof.** By our numbering scheme, \([u,z]\) contains all values in \( G \) for nodes in the subtree of \( u \). \( select_1(G,x) - x \) is equivalent to \( \text{rank}_0(G,select_1(G,x)) \), i.e. it sums up all correction factors in nodes before \( x \) in the numbering scheme. Computing the two prefix sums and subtracting is sufficient.

**Lemma 5.4.** Let \( m \) be the number of nodes in \( PST(T) \), then \( G \) can be stored using at most \( m \log(\sigma l) + O(m) \) bits and each call \( CNT(u,z) \) requires \( O(1) \) time.

**Proof.** Each correction factor has size \( \sigma l \) at most, hence the number of 0s in \( G \) is at most \( m\sigma l \). The number of 1s in \( G \) is \( m \). The thesis follows by storing \( G \) with the binary Elias-Fano data structure of Section 2.5.1.

5.4.2 Finding the correct node

When explaining our solution, we will resort to the concepts of suffix links and inverse suffix links in a suffix tree. For each node \( u \) of \( PST(T) \), the suffix link \( SL(u) = v \) iff we obtain \( \text{pathlabel}(v) \) from \( \text{pathlabel}(u) \) by removing the first letter of the string represented by \( u \). The *inverse suffix link* (also referred to as Weiner link) of \( v \) for some symbol \( c \), denoted \( ISL(v,c) \), is \( u \) iff \( u = SL(v) \) and the link symbol is \( c \). We say that \( v \) possesses an inverse suffix link for \( c \) if \( ISL(v,c) \) is defined. We also refer to the lowest common ancestor of two nodes \( u \) and \( v \) as \( \text{LCA}(u,v) \). An inverse suffix link \( ISL(u,c) = v \) exists only if \( \text{pathlabel}(v) = c \cdot \text{pathlabel}(u) \), however many search algorithms require also *virtual* inverse suffix links to be available. We say a node \( w \) has a virtual inverse suffix link for symbol \( c \) (denoted \( VISL(w,c) \)) if and only if at least one of its descendants (including \( w \)) has an inverse suffix link for \( c \). The value of \( VISL(w,c) \) is equal to \( ISL(u,c) \) where \( u \) is the highest descendant of \( w \) having an inverse suffix link for \( c \). As we will see in Lemma 5.7, it is guaranteed that this highest descendant is unique and, thus, this definition is always well formed. The intuitive meaning of virtual suffix links is the following: \( VISL(w,c) \) links node \( w \) to the highest node \( w' \) in the tree whose path label is prefixed by \( c \cdot \text{pathlabel}(w) \).

\[ \text{Notice that } w \text{ and } u \text{ are the same node whenever } w \text{ has an inverse suffix link for } c. \]
Our interest in virtual inverse suffix links is motivated by an alternative interpretation of the classic backward search. When the backward search is performed, the algorithm virtually starts at the root of the suffix tree, and then traverses (virtual) inverse suffix links using the pattern to induce the link symbols, prefixing a symbol at the time to the suffix found so far. The use of virtual inverse suffix links is necessary to accommodate situations in which the pattern $P$ exists but only an extension $P \cdot \alpha$ of it appears as a node in the suffix tree. Note that the algorithm can run directly on the suffix tree if one has access to virtual inverse suffix links, and such property can be directly extended to pruned suffix trees. Storing virtual inverse suffix links explicitly is prohibitive since there can be up to $\sigma$ of them outgoing from a single node, therefore we plan to store real inverse suffix links and provide a fast search procedure to evaluate the $\text{VISL}$ function.

In the remaining part of this section we will show a few properties of (virtual) suffix links that allow us to store/access them efficiently and to derive a proof of correctness of the search algorithm sketched above.

The following two lemmas state that inverse suffix links preserve the relative order between nodes.

**Lemma 5.5.** Let $w, z$ be nodes in $\mathcal{PST}_l(T)$ such that $\text{ISL}(w, c) = w'$ and $\text{ISL}(z, c) = z'$. Let $u = \text{LCA}(w, z)$ and $u' = \text{LCA}(w', z')$. Then, $\text{ISL}(u, c) = u'$.

*Proof.* If $w$ is a descendant of $z$ or vice versa, the lemma is proved. Hence, we assume $u \neq w$ and $u \neq z$. Let $\alpha = \text{pathlabel}(u)$. Since $u$ is a common ancestor of $w$ and $z$, it holds $\text{pathlabel}(w) = \alpha \cdot \beta$ and $\text{pathlabel}(z) = \alpha \cdot \zeta$ for some non-empty strings $\beta$ and $\zeta$. By definition of inverse suffix link, we have that $\text{pathlabel}(w') = c \cdot \alpha \cdot \beta$ and $\text{pathlabel}(z') = c \cdot \alpha \cdot \zeta$. Since $w$ and $z$ do not share the same path below $u$, the first symbols of $\beta$ and $\zeta$ must differ. This implies the existence of a node $v$ whose path label is $\text{pathlabel}(v) = c \cdot \alpha$ which is the lowest common ancestor between $w'$ and $z'$. Again by definition of inverse suffix link, it follows that $\text{ISL}(u, c) = u' = v$. \hfill $\Box$

**Lemma 5.6.** Given any pair of nodes $u$ and $v$ with $u < v$ such that both have an inverse suffix link for symbol $c$, it holds $\text{ISL}(u, c) < \text{ISL}(v, c)$.

*Proof.* Since $u < v$, we have that $\text{pathlabel}(u)$ is lexicographically smaller than $\text{pathlabel}(v)$. Thus, obviously $c \cdot \text{pathlabel}(u)$ is lexicographically smaller than $c \cdot \text{pathlabel}(v)$. Since $c \cdot \text{pathlabel}(u)$ is the path label of $u' = \text{ISL}(u, c)$ and $c \cdot \text{pathlabel}(v)$ is the path label of $v' = \text{ISL}(v, c)$, $u'$ precedes $v'$ in the preorder traversal of $\mathcal{PST}_l(T)$. \hfill $\Box$

---

5The same property of Lemma 5.5 is also observed in [RNOTI] (Lemma 5.1).
Computing the virtual inverse suffix link of node $u$ for symbol $c$ requires to identify the highest descendant of $u$ (including $u$) having an inverse suffix link for $c$. If such a node does not exist we conclude that the virtual inverse suffix link is undefined. The following lemma states that such node, say $v$, must be unique meaning that if there exists an other descendant of $u$ having an inverse suffix link for $c$, then this node must also be a descendant of $v$.

**Lemma 5.7.** For any node $u$ in $\mathcal{PST}_l(T)$, the highest descendant of $u$ (including $u$) having an inverse suffix link for a symbol $c$, if existing, is unique.

**Proof.** Pick any pair of nodes that descend from $u$ having an inverse suffix link for the symbol $c$. By Lemma 5.5 their common ancestor must also have an inverse suffix link for $c$. Thus, there must exist a unique node that is a common ancestor of all of these nodes.

In our solution we conceptually associate each node $u$ in $\mathcal{PST}_l(T)$ with the set of symbols $D_u$ for which $u$ has an inverse suffix link. We represent each set with a string $\text{Enc}(D_u)$ built by concatenating the symbols in $D_u$ in any order and ending with a special symbol $\#$ not in $\Sigma$. We then build a string $S$ as $\text{Enc}(D_0)\text{Enc}(D_1)\cdots\text{Enc}(D_m-1)$ so that the encodings follow the preorder traversal of the tree $\mathcal{T}$. We also define the array $C[1,\sigma]$ whose entry $C[c]$ stores the number of nodes of $\mathcal{PST}_l(T)$ whose path label starts with a symbol lexicographically smaller than $c$. The next theorem proves that string $S$ together with rank and select capabilities is sufficient to compute VISL. This is crucial to prove that our data structure works, proving virtual inverse suffix links can be recreated from real ones.

**Theorem 5.5.** Let $u \in \mathcal{PST}_l(T)$ and let $z$ be the rightmost leaf descending from $u$. For any character $c \in \Sigma$, let $c_u = \text{rank}_c(S, \text{select}_\#(S, u - 1))$ and, similarly, let $c_z = \text{rank}_c(S, \text{select}_\#(S, z))$. Then (a) if $c_u = c_z$, VISL($u, c$) is undefined. Otherwise, (b) VISL($u, c$) = $C[c] + c_u + 1$ and (c) $C[c] + c_z$ is the rightmost leaf descending from VISL($u, c$).

**Proof.** Let $\mathcal{A}$ be the set of nodes of $\mathcal{PST}_l(T)$ whose path label is lexicographically smaller than the path label of $u$ and let $\mathcal{B}$ be the set of nodes in the subtree of $u$. Let $S(\mathcal{A})$ and $S(\mathcal{B})$ be the concatenations of, respectively, $\text{Enc}(D_w)$ for $w \in \mathcal{A}$ and $\text{Enc}(D_w)$ for $w \in \mathcal{B}$. Due to the preorder numbering of nodes, we know that $\mathcal{A} = [0, u-1]$ and $\mathcal{B} = [u, z]$. Thus, $S(\mathcal{A})$ is a prefix of $S$ that ends where $S(\mathcal{B})$ begins. Notice that the operations $\text{select}_\#(S, u - 1)$ and $\text{select}_\#(S, z)$ return respectively

---

6Interestingly, a similar method of traversing a suffix tree by means of inverse suffix links encoded in a string has been proposed in [ANS12]
Algorithm Count$\geq_i(P[1,p])$

1. $i = p$, $c = P[p]$, $u_p = C[c] + 1$, $z_p = C[c + 1]$;

2. while $((u_i \neq z_i)$ and $(i \geq 2))$ do

3. $c = P[i - 1]$;

4. $u_{i-1} = \text{VISL}(u_i, c) = C[c] + \text{rank}_c(S, \text{select}_\#(S, u_i)) + 1$;

5. $z_{i-1} = \text{VISL}(z_i, c) = C[c] + \text{rank}_c(S, \text{select}_\#(S, z_i))$;

6. $i = i - 1$;

7. if $(u_i = z_i)$ then return “no occurrences of $P$” else return $\text{CNT}(u_1, z_1)$

Figure 5.2: Our algorithm to report the number of occurrences of a pattern $P[1,p]$ in our Compact Pruned Suffix Tree.

5.4. LOWER-SIDE ERROR RANGE

the ending positions of $S(A)$ and $S(B)$ in $S$. Thus, $c_u$ counts the number of inverse suffix links of nodes in $A$ while $c_z$ includes also the number of inverse suffix links of nodes in $B$. Hence, if $c_u = c_z$ no node of $B$ has an inverse suffix link and, thus, proposition (a) is proved.

By Lemma 5.6 we know that inverse suffix links map nodes preserving their relative order. Thus, the first node in $B$ that has an inverse suffix link for $c$ is mapped to node $C[c] + c_u + 1$.

By the node numbering, this first node is obviously also the highest one. Thus, proposition (b) is proved.

Proposition (c) is proved by resorting to similar considerations.

Figure 5.7 illustrates the whole situation. Exploiting VISL, Algorithm 5.2 searches for a pattern $P[1,p]$ backwards. The algorithm starts by setting $u_p$ to be $C[P[p]] + 1$. At the $i$th step, we inductively assume that $u_{i+1}$ is known, and that its path label is prefixed by $P[i + 1,p]$. Similarly, we keep $z_{i+1}$, the address of the rightmost leaf in $u$’s subtree. Using $u_{i+1}$ and $z_{i+1}$ we can evaluate if VISL($u_{i+1}, P[i]$) and, in such case, follow it. In the end, we have to access the number of suffixes of $T$ descending from $u_1$. The next theorem formally proves the whole algorithm correctness:

Theorem 5.6. Given any pattern $P[1,p]$, Algorithm 5.2 retrieves $C(u)$, where $u$ is the highest node of $\text{PST}_l(T)$ such that pathlabel($u$) is prefixed by $P$. If such node does not exist, it terminates reporting $-1$.

There is a caveat: in case the first node of the subtree of $c$ has an edge label with length greater than 1, then the $+1$ factor must be eliminated, since that same node becomes a destination.
Proof. We start by proving that such node $u$, if any, is found, by induction. It is easy to observe that $C[P[p]] + 1$ is the highest node whose path label is prefixed by the single symbol $P[p]$.

By hypothesis, we assume that $u_{i+1}$ is the highest node in $\mathcal{PST}_l(T)$ whose path label is prefixed by $P[i + 1, p]$, and we want to prove the same for $u_i = \text{VISL}(u_{i+1}, P[p - i])$. The fact that $\text{pathlabel}(u_i)$ is prefixed by $P[p - i, p]$ easily follows by definition of inverse suffix link. We want to prove that $u_i$ is the highest one with this characteristic: by contradiction assume there exists another node $w'$ higher that $u_i = \text{VISL}(u_{i+1}, P[p])$. This implies that there exists a node $w = \text{SL}(w')$, prefixed by $P[i + 1, p]$. Also, the virtual inverse suffix link of $u_i$ is associated with a proper one whose starting node is $z = \text{SL}(u_{i+1})$, which by definition of $\text{VISL}$ is also the highest one in $u$’s subtree. Thus, by Lemma 5.7 $w$ is a descendant of $z$. Hence, $w > z$ but $\text{ISL}(w', c) < \text{ISL}(z, c)$, contradicting Lemma 5.6.

Finally, if at some point of the procedure a node $u_{i+1}$ does not have a virtual inverse suffix link, then it is straightforward that the claimed node $u$ does not exist (i.e. $P$ occurs in $T$ less than $l$ times). Once $u$ is found, also $z$ is present, hence we resort to Lemma 5.3 to obtain $C(u) = \text{CNT}(u, z)$. 

**Proof of Theorem 5.4**

We need to store: the $C$ array, holding the count of nodes in $\mathcal{PST}_l(T)$ whose path label prefixed by each of the $\sigma$ characters; the string $G$, together with binary select capabilities and the string $S$, together with arbitrary alphabet rank and select capabilities. Let $m$ be the number of nodes in $\mathcal{PST}_l(T)$. We know $C$ occupies at most $\sigma \log n$ bits. By Lemma 5.4 $G$ occupies at most $m \log(\sigma l) + O(m)$ bits. String $S$ can be represented in different ways, related to $\sigma$, picking a choice from Theorem 5.1 but the space is always limited by $m \log \sigma + o(m \log \sigma)$. Hence the total space is $\sigma \log n + m \log(\sigma l) + O(m) + O(m \log(\sigma)) = O(m \log(\sigma l) + \sigma \log n)$, as claimed. For the time complexity, at each of the $p$ steps, we perform four rank and select queries on arbitrary alphabets which we account as $f(\sigma)$. The final step on $G$ takes $O(1)$ time, hence the bound.

### 5.5 Lower bounds

The following lower bound proves the minimum amount of space needed to solve the substring occurrence estimation problem for both error ranges, uniform and lower-sided.
Theorem 5.7. For a fixed additive error \( l \geq 1 \), an index built on a text \( T[1,n] \) drawn from an alphabet \( \Sigma \) of size \( \sigma \) that approximates the number of occurrences of any pattern \( P \) in \( T \) within \( l \) must use \( \Omega(n \log(\sigma)/l) \) bits of space.

Proof. Assume there exists an index requiring \( o(n \log(\sigma)/l) \) bits of space and answering any approximate counting query within an additive error \( l \). Given any text \( T[1,n] \), we derive a new text \( T'[1,(l+1)(n+1)] \) that is formed by repeating the string \( T\# \) for \( l + 1 \) times, where \$ is a symbol that does not belong to \( \Sigma \). Then, we build the index on \( T' \) that requires \( o((l+1)(n+1)\log(\sigma+1)/l) = o(n \log \sigma) \) bits. We observe that we can recover the original text \( T \) by means of this index: we search all possible strings of length \( n \) drawn from \( \Sigma \) followed by a \# , the only one for which the index answers with a value greater than \( l \) is \( T \). A random (in Kolmogorov’s sense) text has entropy \( \log(\sigma^n) - O(1) = n \log \sigma - O(1) \) bits. Hence, the index would represent a random text using too few bits, a contradiction. \( \square \)

Using the same argument we can prove the following Theorem, which justifies the need of focusing on additive errors.

Theorem 5.8. For a fixed multiplicative error \((1 + \varepsilon) > 1\), an index built on a text \( T[1,n] \) drawn from an alphabet \( \Sigma \) of size \( \sigma \) that approximates the number of occurrences of any pattern \( P \) in \( T \) within \((1 + \varepsilon)\) must use \( \Omega(n \log \sigma) \) bits of space.

5.6 Experiments

In this section we show an experimental comparison among the known solutions and our solutions. We use four different data sets downloaded from Pizza&Chili corpus [FGNV08] that correspond to four different types of texts: DNA sequences, structured text (XML), natural language and source codes. Text and alphabet size for the texts in the collection are reported in the first column of Table 5.1.

We compare the following solutions:

- **FM-index (v2)**. This is an implementation of a compressed full-text index available at the Pizza&Chili site [FGNV08]. Since it is the compressed full-text index that achieves the best compression ratio, it is useful to establish which is the minimum space required by known solutions to answer to counting queries without errors.

- **APPROX-l**. This is the implementation of our solution presented in Section 5.3.

- **PST-l**. This is an implementation of the Pruned Suffix Tree as described in [KVI96].
**Table 5.1:** Statistics on the data sets. The second column denotes the original text in MBytes. Each group of three columns in the lower table describe $\mathcal{PST}_l$ information for a choice of $l$: expected amount of nodes, $|T|/l$; real amount of nodes in $\mathcal{PST}_l(T)$; sum of length of labels in $\mathcal{PST}_l(T)$.

- **CPST-$l$.** This is the implementation of our Compact Pruned Suffix Tree described in Section 5.4.

Recall that APPROX-$l$ reports results affected by an error of at most $l$ while PST-$l$ and CPST-$l$ are always correct whenever the pattern occurs at least $l$ times in the indexed text.

The plots in Figure 5.3 show the space occupancies of the four indexes depending on the chosen threshold $l$. We do not plot space occupancies worse than FM-index, since in those cases FM-index is clearly the index to choose. In fact, Figure 5.3(d) does not contain a plot for PST, since its space performance was always worse than FM-index.

It turns out that in all the texts of our collection the number of nodes in the pruned suffix tree is small (even smaller than $n/l$): these statistics are reported in Table 5.1. This is the reason why our CPST is slightly more space-efficient than APPROX. In practice, the former should be indubitably preferred with respect to the latter: it requires less space and it is always correct for patterns that occur at least $l$ times. Even though, the latter remains interesting due to its better theoretical guarantees. In both solutions, by halving the error threshold, we obtain indexes that are between 1.75 (CPST) and 1.95 (APPROX) times smaller. Thus, we can obtain very small indexes by setting relatively small values of $l$. As an example, CPST with $l = 256$ on text english requires 5.1 Mbytes of space which is roughly 100 times smaller than the original text. We observe that both CPST and APPROX are in general significantly smaller than FM-index and remain competitive even for small values of
5.6. EXPERIMENTS

Figure 5.3: Space occupancies of indexes as a function of the error threshold $l$.

As an example, FM-index requires 232.5 Mbytes on english, which is roughly 45 times larger than CPST−256.

As far as PST is concerned, it is always much worse than CPST and APPROX. As expected, its space inefficiencies are due to the need of storing edge labels since their amount grows rapidly as $l$ decreases (see Table 5.1). Moreover, this quantity strictly depends on the indexed text, while the number of nodes is more stable. Thus, the performances of PST are erratic: worse than CPST by a factor 6 on english that becomes 60 on sources. It is remarkable that on sources we have to increase PST’s error threshold up to 11,000 to achieve a space occupancy close to our CPST with $l = 8$.

For what concerns applications, we use our best index, i.e. CPST together with one estimation algorithm: MOL. The reader can find a brief explanation of the algorithm in Section 5.2.4; the algorithm is oblivious to the underlying data structure as long as a lower-sided error one is used. We performed (details omitted) a comparison between MO, MOL and KVI [KVI96, JNS99] and found out that MOL delivered the best estimates. We also considered MOC and MOLC, but for some of our data sets the creation of the constraint network was prohibitive in terms of memory. Finally, we tried to compare with CRT [CGG04]; however, we lacked the original implementation.
### Chapter 5. Approximate Substring Selectivity Estimation

| Data set | Indices | $|P| = 6$ | $|P| = 8$ |
|----------|---------|-----------|-----------|
| dblp     | PST-256 | 10.06 ± 32.372 | 12.43 ± 34.172 |
|          | CPST-16 | 0.68 ± 1.456   | 0.86 ± 1.714   |
| dna      | PST-256 | 0.47 ± 1.048   | 0.49 ± 2.433   |
|          | CPST-32 | 0.47 ± 0.499   | 0.43 ± 0.497   |
| english  | PST-256 | 7.03 ± 27.757  | 12.45 ± 31.712 |
|          | CPST-32 | 0.80 ± 2.391   | 1.40 ± 3.394   |
| sources  | PST-11000 | 816.06 ± 1646.57 | 564.94 ± 1418.53 |
|          | CPST-8  | 0.70 ± 1.028   | 0.93 ± 1.255   |

| Data set | Indices | $|P| = 10$ | $|P| = 12$ | Average Improvement |
|----------|---------|-----------|-----------|---------------------|
| dblp     | PST-256 | 14.20 ± 35.210 | 15.57 ± 36.044 | 19.03× |
|          | CPST-16 | 1.00 ± 1.884   | 1.14 ± 2.009   |                     |
| dna      | PST-256 | 4.26 ± 15.732  | 11.09 ± 19.835 | 5.51× |
|          | CPST-32 | 0.52 ± 0.904   | 1.77 ± 2.976   |                     |
| english  | PST-256 | 13.81 ± 28.897 | 11.43 ± 23.630 | 9.68× |
|          | CPST-32 | 2.07 ± 3.803   | 2.45 ± 3.623   |                     |
| sources  | PST-11000 | 400.62 ± 1229.35 | 313.68 ± 1120.94 | 792.52× |
|          | CPST-8  | 1.13 ± 1.367   | 1.28 ± 1.394   |                     |

Figure 5.4: Comparison of error (difference between number of occurrences and estimate) for MOL estimates over different pattern lengths. PST and CPST parameters are chosen to obtain close index sizes. Tests performed on 1M random patterns appearing in the text. The last column shows the average factor of improvement obtained by using our CPST instead of PST.

and a significative training set for our data sets. Hence, we discarded the algorithm from our comparison.

Figure 5.4 shows the average error of the estimates obtained with MOL on our collection by using either CPST or PST as the base data structure. For each set we identified two pairs of thresholds such that our CPST and PST have roughly the same space occupancy. For each text, we searched for 4 million patterns of different lengths that we randomly extracted from the text. Thus, this figure depicts the significant boost in accuracy that one can achieve by replacing PST with our solution. As an example, consider the case of sources where the threshold of PST is considerably high due to its uncontrollable space occupancy. In this case the factor of improvement that derives by using our solution is more than 790. The improvements for the other texts are less impressive but still considerable.
Figure 5.5: The suffix tree for the string \textit{banabanab}.

Figure 5.6: The pruned suffix tree of \textit{banabanab} with threshold 2. Each node contains its preorder traversal id and, in brackets, its correction factor. Arrow denotes an inverse suffix link for \textit{b}, the dashed arrow a virtual one.
Figure 5.7: The same PST of Figure 5.6 with information associated with Theorem 5.5. Each node is given the set of symbols for which a virtual inverse suffix link is defined. The string $S$ contains the separated encoding, in preorder traversal, of suffix links chosen by our procedure. Note that the first child with edge label $a$ loses its link for $b$, as $ana$ is sufficient.
Chapter 6

Future directions

We conclude the thesis by presenting various missing results, hoping to inspire further research on the matter.

Binary, distribution aware rank and select.

The work on predecessor search and binary rank/select has been very thorough and in the late years brought up amazing results in terms of upper and lower bound. The basic line of research, i.e. just considering $u$, $n$ and $t$ to create data structures, is almost fulfilled. Nevertheless, real data is usually correlated to specific distributions (e.g. gap lists for indexing). In massive data sets analysis skewed mixtures of binomials or power law distributions are very frequent [WMB99]. Sometimes, through permutations over the universe, bitvectors can be made clustered (see e.g. [BB02]) so that 1s concentrate in contiguous regions. rank and select data structures that take into account such parameters are not known yet. Practical work [Vig08] already proved that different density $(u/n)$ factors can help choosing the correct data structure. It would be interesting to understand how much powerful a data structure can become if the generating source is well known. We conjecture that sensible lowering in total space may be achieved.

select and contiguous memory access.

The index built for select over strings in Theorem 4.6 is an application of the code for permutation inversions presented in [MRRR03]. The original definition regards a permutation $\pi$ over $[n]$ that has to be inverted, and produces an index where one follows the cycles of $\pi$ starting from some position $x$ until one finds $y$ such that $\pi(y) = x$, i.e. $\pi^{-1}(x)$. To limit the number of evaluation of $\pi$ they insert shortcuts in the permutation cycles every $t$ jumps, for some $t$. The original index is optimal when $w = \Theta(\log n)$. In the Word RAM model, however, we may have $w = \omega(\log n)$. 
In theory this should bring the redundancy of the data structure down accordingly, also considering situations in which more elements of the permutation can be stored in one word. So far, no progress has been done on the front and no lower bounds have been proposed. Having either a lower or upper bound would have a significant impact: the BWT LF mapping that one has to follow in the classic BWT inversion procedure is exactly equivalent to following a permutation; a better upper bound would probably give an advantage at decompressing when the alphabet is small; a lower bound would finally justify a limit to research in that sense. In our setting, this would prove that a better select for indexes in the Word RAM model is possible.

**Practical constant-alphabet rank and select.**

The case of $\sigma = 2$ has been studied in practice (see e.g. [GGMN05, OS07, Vig08]), as well as the generic case of any $\sigma$ (see e.g. [CN08]). However, the two kinds of solutions are completely apart, e.g. the latter involves quite complex and impractical machinery and cannot easily compete with the speed of binary solutions. It is still the case that constant time alphabets may need to be used, i.e. $\sigma = 3$ or 4 (an example is BWT over DNA). Adapting binary solutions to this scenario is not straightforward if one wants to keep space consumption low. We think it is worth investigating on that front, producing data structures that can perform rank and select in competitive times for relatively small alphabets.

**Small and fast construction of $CPST_i$.**

The $CPST_i$ of Section 5.4 can be constructed with an easy algorithm. Namely, one can build the full suffix tree and then prune it. This is slow and, worse, extremely memory consuming: a suffix tree can have $O(n)$ nodes for a string of length $n$; when $\ell$ is small this is extremely wasteful. We call for a more efficient and fast construction algorithm in theory at first. Given the final space consumption we also consider important to perform research of a practically employable construction algorithm.
Bibliography


6.0. BIBLIOGRAPHY


[LZO] Lempel-ziv-oberhumer compression algorithm.


