Universität Bielefeld

Suffix Arrays in Theory and Practice

Klaus-Bernd Schürmann

Thesis submitted to the
Faculty of Technology of Bielefeld University, Germany
for the degree of Dr. rer. nat.

Supervised by
Prof. Dr. Jens Stoye

Referees
Prof. Dr. Jens Stoye, Prof. Dr. Enno Ohlebusch

Defense on
September 24, 2007

Gedruckt auf alterungsbeständigem Papier – ISO 9706

for Anja

Abstract

The suffix array of a string is a permutation of all starting positions of the string's suffixes in lexicographical order. In this thesis, we investigate mathematical and algorithmical aspects of suffix arrays.

The first part mainly deals with combinatorial properties of suffix arrays and their enumeration. For a fixed alphabet size and string length, we divide the set of all strings into equivalence classes of strings that share the same suffix array. For each such equivalence class, we count the number of strings contained in it and enumerate those strings. We also give exact formulas for computing the number of equivalence classes and efficient algorithms for enumerating them. Alternatively, we count the number of suffix arrays and enumerate them. Our methods yield lower bounds for the compressibility of suffix arrays and build the foundation for the efficient generation of appropriate test data sets for suffix-array-based algorithms. We also show that summing up the elements of all equivalence classes forms a particular instance for some summation identities of Eulerian numbers.

The second part of the thesis deals with suffix array construction. We first present a new classification of suffix array construction algorithms and provide an in-depth review of the classified algorithms. We classify the algorithms regarding two different categories: the progress in the suffix sorting process and the usage of dependencies among suffixes. After the survey of the previous algorithms, we present our new practical algorithm for suffix array construction that consists of two easy-to-implement components. It first sorts the suffixes with respect to a fixed length prefix; then it refines each bucket of suffixes sharing the same prefix using the order of already sorted suffixes. Other suffix array construction algorithms follow more complex strategies. We achieve a very fast construction for common strings as well as for worst-case strings by enhancing our algorithm with further techniques; this is shown by an in-depth experimental study that compares our algorithm to other fast suffix array construction algorithms.

Acknowledgements

Foremost, I would like to thank my supervisor Jens Stoye for encouraging me to develop and to follow my own ideas. He has been a great boss over these past years; I could always rely on his support.

Thanks to the working group Genome Informatics in Bielefeld for the nice working atmosphere. In particular, thanks to Hans-Michael Kaltenbach (Mitch) and Constantin Bannert (Conni) for various fruitful discussions, breakfast sessions, and coffee breaks. Karla and Sergio Carvalho showed me the "Brasilian way of life". Sven Rahmann was a nice room mate at some conference trips. Veli Mäkinen and Katharina Jahn were nice office mates; from Veli I learned a lot about text compression and compressed indices. Together with Zsuzsanna Lipták and Ferdinando Cicalese, we had many nice experiences on life with children. Special thanks to Heike Samuel for her kind help on dealing with administrative subtleties. Basically, thanks to all former and current members of the group for many relaxing lunch and coffee breaks.

Many thanks to Hans-Michael Kaltenbach, Sergio Carvalho, Katharina Jahn, Wolfgang Gerlach, Marcel Martin, and Manuela Schürmann for proofreading parts of this thesis and to Peter Husemann for being a helping hand.

Finally, I would like to thank my parents and my family. My parents Maria and Bernhard Schürmann have been supporting me through all my life. I particularly thank my wife, Anja Schürmann, for her love and patience over these past years and my children, Alexander and Niklas, for showing me the most important things in life.

Contents

1 Introduction

The most common type of information is a written text as we find it in books, newspapers, and in other printed media. We treat such a text as a sequence of symbols and call it string, sequence, word, or text. Such strings play a fundamental role in many software applications: Word processing systems provide advanced facilities for the modification of texts, e-mail tools are used to send text messages and other data, and Internet browsers allow to retrieve and to read texts from the Internet, among many other applications. There are other sequences that are used in the background of software applications. The data that are interchanged via the Internet, for example, are first translated into a sequence of binary digits (bits). Then the real transmission is carried out by a sequence of digital signals that corresponds to the binary sequence. In molecular biology, we encounter DNA, RNA, or amino acid sequences (peptides), and there are many other types of sequences.

In sequence analysis, we are interested in the development of efficient data structures and algorithms to process all types of sequences. A fundamental problem in sequence analysis is pattern matching, which deals with the following question: Does a query pattern occur exactly or approximately in a given sequence, and if so, where in the sequence does it occur?

Full-text indices are data structures used to process different kinds of sequences for such applications. In contrast to other text indices, such as inverted files [27], full-text indices allow the efficient access to every substring, or subword, of a given input string. The *suffix tree* is arguably the best known full-text index, which can be computed and stored in $\mathcal{O}(n)$ time and space for an input string t of length n. It was introduced by Weiner [143] in 1973, who presented a linear-time construction algorithm. Further linear-time algorithms were given by McCreight [104] in 1976, Ukkonen [141, 142] in 1993, and Farach [45] in 1997. McCreight's algorithm is considered to be simpler and more space efficient than Weiner's algorithm, Ukkonen's algorithm constructs suffix trees online, and Farach's algorithm runs in linear time even for alphabets of arbitrary size. For an in-depth study of the connections between the former three algorithms, we refer to a study of Giegerich and Kurtz [53].

There are many applications of suffix trees. The classical one is the exact pattern matching: For a query string of length m, we use a suffix tree of another database string to decide in time $\mathcal{O}(m)$ if the query appears as a substring in the indexed string. But the real virtue of suffix trees comes from their use in solutions of more complex string problems [8] (for example, repeat finding); Gusfield presents more than twenty in his book about string processing algorithms [58]. Unfortunately, those construction and query algorithms do not explicitly consider the locality of memory reference, which is very important on current computer architectures with a memory hierarchy of multi-level cache and main memory. Hence, the practical run time of those algorithms, which is often asymptotically optimal,

suffers from many cache misses. These problems have been approached by representing the suffix tree data structure in different ways [88, 54, 129] for particular applications. In general, it remains an open problem.

Further drawbacks of suffix trees are their large space requirements, which exceed the space requirements of the input string by an order of magnitude. Until the early 1990s, the most space-efficient implementation of McCreight's algorithm required $28n$ bytes for a string of length n in the worst case (for 4-byte integer words). Manber and Myers [96, fourth column in Table 1 on page 946] state that their own implementation requires between $14.2n$ and $27.8n$ bytes in practice. Even today, the most space-efficient implementation of McCreight's algorithm by Kurtz [88] still uses between $8n$ and $14n$ bytes in total. These large space requirements of suffix trees are incompatible with the increasing amount of accessible sequence data that needs to be indexed. Typical data mainly come from the Internet and from several genome sequencing projects, which produce long DNA sequences. In the 1990s, two technology projects stressed the requirement of string indices for huge amounts of sequence data: *Google* and the *Human Genome Project*. Google attempts to index the human readable information available through the Internet, and the Human Genome Project provides the genomic sequence data for the human species.

As a result, space-efficient alternatives to suffix trees have been developed: In the early 1990s, Manber and Myers [96] and Gonnet *et al.* [55] introduced the *suffix array* (Gonnet *et al.* under the name PAT array), which is the most popular alternative to suffix trees. Other space-efficient full-text indices are the suffix cactus of Kärkkäinen [70], the factor oracle of Allauzen *et al.* [4], and the suffix vector of Monostori *et al.* [108] (ordered historically). Unlike suffix arrays, however, these developments have not found their way into the mainstream of research on full-text indices. This is presumably so because the suffix array with its space requirements of $5n$ bytes (including the input string) is more space efficient than those indices. Furthermore, its simple one-dimensional structure is easy to handle in software implementations.

1.1 Suffix arrays

In their seminal article [96], Manber and Myers gave the first algorithm to directly construct suffix arrays in $\mathcal{O}(n \log n)$ time. In addition, they enhanced the suffix array with an auxiliary array, the LCP array, that stores the length of the longest common prefix of adjacent suffixes in the suffix array. Based on the suffix array and the corresponding LCP array, they present an algorithm for the exact pattern matching problem, which decides in $\mathcal{O}(m + \log n)$ time whether a query string of length m is a substring of the indexed string.

1.1.1 Suffix array construction

The further interest in suffix arrays was then initially attracted by the close relation to the *Burrows–Wheeler transform* [32] (presented in 1994), which is often used as the basis for text compression algorithms. This interest can be explained by the fact that computing the Burrows–Wheeler transform by block-sorting the input string is equivalent

to constructing a suffix array. Therefore, most of the research on suffix arrays regard their construction. But although Farach *et al.* [47] correlated suffix sorting and linear-time suffix tree construction in 2000, up until 2003 all known algorithms reaching this bound took a detour over suffix tree construction and afterwards derived the suffix array from the suffix tree (see [58, Section 7.14.1]), instead of directly constructing suffix arrays. In 2003, the problem of direct linear-time construction of suffix arrays was solved independently by Kärkkäinen and Sanders [71, 73], Kim *et al.* [79, 80], and Ko and Aluru [84, 85]. Shortly after, Hon *et al.* [63] gave a linear-time algorithm that needs $\mathcal{O}(n)$ bits of working space.

Apart from these more theoretical results, there has also been much progress in practical suffix array construction. Larsson and Sadakane [90] presented a fast algorithm, called *qsufsort*, running in $\mathcal{O}(n \log n)$ worst-case time using $8n$ bytes. Kim *et al.* [78] introduced a divide-and-conquer algorithm based on [80] with $\mathcal{O}(n \log \log n)$ worst-case time complexity, but with faster practical running times than the previously mentioned linear-time algorithms.

Other viable algorithms mainly consider space requirements. They are called *lightweight algorithms* due to their small space requirements. Itoh and Tanaka [67], Seward [135], and Manzini and Ferragina [102] proposed algorithms using only $5n$ bytes and little additional auxiliary space. In theory, their worst-case time complexity is $\Omega(n^2)$. However, they are very fast in practice if the average LCP is small. The most recent lightweight algorithm, developed by Burkhardt and Kärkkäinen [31] (see also [73]), is called *difference-cover* algorithm. Its worst-case running time is $\mathcal{O}(n \log n)$, and it uses sublinear extra space. For common real-life data, though, the algorithm is on average slower than Manzini and Ferragina's [102] algorithm. These are the major developments in the field of in-memory suffix array construction algorithm. Other approaches are presented by Lee and Park [91], Baron and Bresler [15], Maniscalco and Puglisi [98, 99], and Ahlswede *et al.* [3].

Besides the in-memory suffix array construction algorithms, there are several others that address specific sub-branches of practical suffix array construction, namely distributed algorithms and external memory algorithms: Distributed or parallel suffix array construction algorithms were studied by Navarro *et al.* [112] and Kulla and Sanders [86], among others. External memory suffix array construction algorithms were proposed, for example, by Crauser and Ferragina [39] and Dementiev *et al.* [42].

We observe that the previous in-memory suffix array construction algorithms either perform well for common strings with short LCPs or for degenerated strings with large LCPs. Based on our experience with biological sequence data, we believe that further properties are required. There are many applications where very long sequences with mainly small LCPs, interrupted occasionally by very large LCPs, are investigated. In genome comparison, for example, concatenations of similar sequences are indexed to find common subsequences, repeats, and unique regions. Thus, to compare genomes of closely related species, one has to build suffix arrays for strings with highly variable LCPs. We believe that the characteristics as observed in this context can also be found in other application areas. These facts stress the importance of efficient ubiquitous suffix array construction algorithms.

1.1.2 Using suffix arrays

Beyond the development of suffix array construction algorithms, there has been progress on algorithmical applications of suffix arrays. In 2001, Kasai *et al.* [76] presented an algorithm that constructs the LCP array from the suffix array in linear time, and they show how every bottom-up traversal of a suffix tree can be simulated on those two arrays. Manzini [101] later presented more space-efficient algorithms for the construction of the LCP array from the suffix array. The LCP information, however, only allows the simulated traversal of suffix trees from child nodes to parent nodes. Abouelhoda *et al.* [1, 2] enhanced the suffix array with additional auxiliary arrays that further allow the traversal from parent nodes to child nodes. Based on their *enhanced suffix array*, they established the concept of *lcp-interval trees*. These conceptual trees, which do not need to be constructed in practice, are equivalent to suffix trees. Furthermore, the enhanced suffix array contains information allowing suffix link traversal. Chang and Lawler [33], for example, use suffix links for computing *matching statistics*. Hence, basically every algorithm working on suffix trees can be ported to an equivalent algorithm on enhanced suffix arrays with identical asymptotic time bound. Abouelhoda *et al.* showed how to do that for algorithms performing different types of suffix tree traversals.

The enhanced suffix array has many practical advantages compared to suffix trees. Firstly, it is possible to store it on secondary memory without serialising the data structure, which would be necessary for suffix trees. Secondly, the different auxiliary arrays are independent such that for particular applications only a subset of arrays has to be accessed, which decreases main memory load. Finally, additional annotations are easily added (see [121] for example annotations). We believe that virtually all algorithms that were originally designed for suffix trees can be implemented more efficiently on enhanced suffix arrays. Hence, (enhanced) suffix arrays have the potential to fully replace suffix trees for practical applications.

Suffix arrays are already used in many bioinformatics applications. We give some examples: Burkhardt *et al.* [30] applied suffix arrays for searching similar DNA sequences and Malde *et al.* [95] for EST clustering. Kurtz's [87] implementation of enhanced suffix arrays is used in several other bioinformatics tools and projects. Höhl *et al.* [60] used it for multiple sequence alignment and Beckstette *et al.* [16] for the matching of position specific scoring matrices, see [87] for a longer list. Apart from suffix array applications in bioinformatics, there are other application areas: Suffix sorting algorithms have been applied for the computation of the Burrows–Wheeler transform, for example, in the *bzip2* compressor [134]. Moreover, in linguistics Yamamoto and Church [144] used them to count term frequencies.

In brief, the various time-efficient algorithms on suffix trees can be ported to enhanced suffix arrays, and these algorithms have proved their practical efficiency on suffix arrays. At the moment, we see no room for significant improvements regarding algorithmical applications of suffix arrays.

1.1.3 Suffix array compression and suffix arrays in theory

The task of full-text index compression emerged after Grossi and Vitter introduced the *compressed suffix array* [57] that reduces the space requirements to a linear number of bits. Other compressed indices of that type are: Ferragina and Manzini's *FM-index* [49] based on the Burrows–Wheeler transform, a *compressed-suffix-array-based index* by Sadakane [123] that does not use the text itself, and Mäkinen's *compact suffix array* [94]. There are various subsequent developments; most of them improve upon the compressed indices of Grossi and Vitter [57], Ferragina and Manzini [49], or Sadakane [123]. For an in-depth study of compressed full-text indices and their space requirements, we refer to the survey of Navarro and Mäkinen [113]. Moreover, Sadakane [125] recently presented a compressed full-text index providing the full functionality of suffix trees, although not with the same asymptotic time bounds.

All these developments on compressed indices trade space occupancy for querying time. Experimental results of Ferragina and Manzini [50] show that suffix arrays use 8 to 13 times as much space as their FM-index. For the exact pattern matching with the reporting of occurrences, however, the running times on their FM-index are by a factor between 3 and 33 higher than the running times on their suffix array implementation. The reason for the greater running times on compressed indices is that redundant information, which would have been necessary for more efficient querying, is lost when compressing an original base index, like the suffix array. We believe that a profound knowledge of the algebraic and combinatorial properties of suffix arrays is essential to develop suffix-array-based, succinct indices that allow efficient querying.

Besides those practical aspects, suffix arrays are also interesting from the purely theoretical perspective. They are represented as permutations, which are widely studied in group theory and combinatorics. Nevertheless, in that regard, they have been less studied than we expected. Duval and Lefebvre [44] characterised the set of strings that share the same suffix array. A combinatorial approach that partly includes suffix arrays was presented by Hohlweg and Reutenauer [61]. Hence, further research on the theoretical aspects of suffix arrays was required.

1.2 Organisation of the thesis

Throughout the thesis, we investigate the function sa that maps each string to its suffix array. The thesis consists of two major parts: In the first part (Chapters 3–6), we investigate the function sa from a more theoretical point of view. In particular, we study combinatorial aspects of strings and their suffix arrays. In the second part (Chapters 7–10), we deal with the efficient implementation of the function sa, namely, the construction of suffix arrays.

We first give the basic definitions and notations regarding suffix arrays in Chapter 2. Chapter 3 contains the preliminaries for the subsequent investigations: We define different equivalences of strings regarding their structure. In particular, for a fixed alphabet size and string length, we divide the set of all strings into equivalence classes of strings that share the

same suffix array. We also define the data structures for the subsequent reasoning on such equivalence classes and characterise the strings in each class. In Chapter 4, we count the number of particular strings in any equivalence class and present enumeration algorithms for those strings. Chapter 5 contains exact formulas for the number of equivalence classes or, alternatively, for the number of respective suffix arrays; we also present an efficient enumeration algorithm for those equivalence classes, or rather, for their representatives. We then apply the counting results to more practical problems in Chapter 6: From the exact number of suffix arrays, we derive lower bounds on the compressibility of suffix-array-based compressed indices. Apart from that (also in Chapter 6), we show that summing up the elements of all equivalence classes forms a particular instance for some summation identities of Eulerian numbers.

In the second part of the thesis, we study the problem of efficient suffix array construction. Chapter 7 contains the suffix-array-construction-specific definitions and notations. In Chapter 8, we provide new comprehensive classifications of previous suffix array construction algorithms and survey those algorithms. In Chapter 9, we present our new bucket-pointer refinement algorithm, show a runtime analysis and provide implementation details. Experimental results on the practical performance of our algorithm and the previously fastest suffix array construction algorithms are given in Chapter 10.

We conclude and give an outlook to future research in Chapter 11.

Parts of Chapters 3–6 have been published in a technical report [130], in a refereed conference proceeding [131], and are to appear in a refereed journal article [128]. Parts of Chapters 7–10 have been published in a refereed conference proceeding [132] and in a refereed journal article [133].

2 Basic Definitions and Terminology

The interval $[l,r] = \{z \in \mathbb{Z} : l \leq z \leq r \text{ with } l,r \in \mathbb{Z}\}$ denotes the set of all integers greater than or equal to l and less than or equal to r. The set of natural numbers starting with 1 is denoted by $\mathbb{N}$, and $\mathbb{N}_0$ further contains the additional 0, that is, $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$.

Alphabet and strings. Let Σ be a finite set of size $|\Sigma|$, the *alphabet*, and $t \in \Sigma^n$ a string over Σ of length n, the *text*. For $i \in [1,n]$, $t[i]$ denotes the i^{th} character of t, and for all pairs of indices (l,r), $1 \leq l \leq r \leq n$, $t[l,r] = t[l], t[l+1], \ldots, t[r]$ denotes the substring of t starting at position l and ending at position r. Substrings $t[i,n]$ ending at position n are *suffixes* of t; $t[i,n]$ is called the suffix i. The starting position i of a suffix $t[i,n]$ is called its *suffix number*. For $1 \leq i < n$, $t[i+1,n]$ is called the *successor suffix* of $t[i,n]$, and conversely, $t[i,n]$ the *predecessor suffix* of $t[i+1,n]$. For more distant suffixes $t[i,n]$ and $t[i+\ell,n]$ with $\ell \in \mathbb{N}$ and $i+\ell \leq n$, $t[i+\ell,n]$ is called the *ℓ-successor* of $t[i,n]$ and $t[i,n]$ the *ℓ-predecessor* of $t[i+\ell,n]$. $\Sigma(t) := \{t[i] : 1 \leq i \leq n\} \subseteq \Sigma$ is the subset of characters actually occurring in t and is called the *character set* of t. We usually use σ for the alphabet size $|\Sigma|$, but if the strings are required to use all characters such that their character set equals the alphabet, we use κ.

Permutations and suffix arrays. Let $\mathcal{P}^n$ denote the set of all permutations of $[1,n]$, and let $P \in \mathcal{P}^n$. Then $i \in [1,n-1]$ is a *permutation descent* of P if $P[i] > P[i+1]$. Conversely, a non-extendable ascending segment $P[l,r]$ of P with $P[l] < P[l+1] < \ldots < P[r]$ of P is called a *permutation run*. Each permutation run of P begins right after a permutation descent or at the leftmost position 1 of P, and ends at the next permutation descent or at the last position n of P. Hence, the permutation runs define the permutation descents and vice versa. Figure 2.1 shows the permutation descents and permutation runs for the permutation $P = (5,6,3,2,4,8,9,1,7)$.

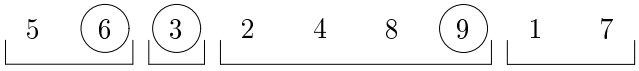

Figure 2.1: Permutation descents and permutation runs for $P = (5,6,3,2,4,8,9,1,7)$. The encircled entries mark the positions of the permutation descents, and the underlined segments mark the permutation runs.

The function

$$sa : \begin{cases} \Sigma^n & \longrightarrow & \mathcal{P}^n \\ t & \longmapsto & P, \end{cases} \tag{2.1}$$

maps each string t of length $n \in \mathbb{N}$ to its suffix array, where the *suffix array* $sa(t)$ of t is a permutation of the suffix numbers $[1, n]$ according to the lexicographic ordering of the n suffixes of t. More precisely, a permutation P of $[1, n]$ is the suffix array for a string t of length n, $P = sa(t)$, if for all pairs of indices (i, j), $1 \leq i < j \leq n$, the suffix with suffix number $P[i]$ is lexicographically smaller than the suffix with suffix number $P[j]$. Moreover, the sequence $t[P[1]], t[P[2]], \ldots, t[P[n]]$, which is formed of the first characters of the ordered suffixes, is called the *First sequence* for t (similar to the *first column* used for the Burrows–Wheeler transform [32]).

The *rank array* R_P for the permutation P (further on simply denoted by R), sometimes called the inverse permutation or the inverse suffix array, is defined as follows: For all indices $i \in [1, n]$ the rank of i is j, $R[i] = j$, if i occurs at position j in the permutation, $P[j] = i$. We extend the rank array by $R[n + 1] = 0$, indicating that the empty suffix, not contained in the suffix array, is always the lexicographically smallest. $R[i] = j$ implies that the suffix $t[i, n]$ is the lexicographically j^{th} among all suffixes of t. The rank array and also other rank functions are an important tool throughout the rest of this thesis. The rank array allows to directly determine the location of a suffix number in the suffix array and defines the relative lexicographical order of the suffixes:

$$t[i, n] < t[j, n] \Longleftrightarrow R[i] < R[j] \text{ for all } (i, j) \in [1, n]^2,$$

where $t[i, n] < t[j, n]$ accords to the lexicographical order of the suffixes and $R[i] < R[j]$ to the order of the natural numbers.

Further definitions. Besides the binomial coefficient $\binom{x}{y} = \frac{x!}{y!(x-y)!}$, the Stirling numbers and the Eulerian numbers are important for this work. Although these numbers have a venerable history, their notation is less standard. We follow the notation of Graham *et al.* [56, Chapter 6] where the Stirling number of the second kind $\left\{ {n \atop k} \right\}$ is the number of ways to partition a set of n elements into k non-empty subsets, and the Eulerian number $\left\langle {n \atop d} \right\rangle$ gives the number of permutations of $[1, n]$ having exactly d permutation descents, also defined through the recursion (i) $\left\langle {n \atop 0} \right\rangle = 1$, (ii) $\left\langle {n \atop d} \right\rangle = 0$ for $d \geq n$, and (iii) $\left\langle {n \atop d} \right\rangle = (d + 1) \left\langle {n-1 \atop d} \right\rangle + (n - d) \left\langle {n-1 \atop d-1} \right\rangle$ for $0 < d < n$.

Part I

COMBINATORICS OF SUFFIX ARRAYS

3 Background, Definitions, and Basic Observations

For certain applications, we are not always interested in the total number of strings. Instead, we are interested in equivalence classes of strings sharing the same structural properties. A suffix array construction algorithm, for example, produces the same suffix array for `ABBAA` and `ACCAA`, but a different one for `CBBCC`. Therefore, we would count two classes of strings: the first class containing `ABBAA` and `ACCAA`, and the second one containing `CBBCC`.

A different notion of equivalence on strings arises from the preprocessing phase of the substring search algorithm of Knuth *et al.* [83] (Knuth-Morris-Pratt algorithm). It returns a prefix function (also called failure function or border array) for the query string that encapsulates information about how the suffixes of the query match against the prefixes (see also [38, Section 32.4]). Our example strings `ABBAA`, `ACCAA`, and `CBBCC` share the same prefix function. Hence, we consider them equivalent and only count one equivalence class. Moore *et al.* [109] counted the number of such distinct prefix functions.

To the best of our knowledge, there are no previous studies counting the number of permutations that are the suffix arrays for a particular set of strings. Although the combinatorics of permutations is a research field that has been widely studied (see, for example, [28]), there are only a few combinatorial results for suffix arrays. In 2002, Duval and Lefebvre [44] characterised the set of strings that share the same suffix array. Recently, Crochemore *et al.* [40] presented combinatorial properties of the related Burrows–Wheeler transform, but these properties are unassignable to suffix arrays. They rely on the fact that the Burrows–Wheeler transform is based on the order of cyclic shifts of the input sequence, whereas the suffix array is based on the order of suffixes cut at the end of the string, which destroys that nice group structure. A combinatorial approach that partly includes suffix arrays was presented by Hohlweg and Reutenauer [61]. They study connections between binary planary trees, Lyndon words, and suffix arrays.

This chapter provides the basic definitions and tools for counting the strings and suffix arrays in the subsequent chapters. In Section 3.1 we define different equivalences of strings regarding their various structural properties and further combinatorial structures related to suffix arrays in Section 3.2. Although the given general definition of suffix arrays in Chapter 2 is quite concise, we need a more specific, "handy" proposition for the subsequent reasoning, which is given in Section 3.3, Theorem 3.2.

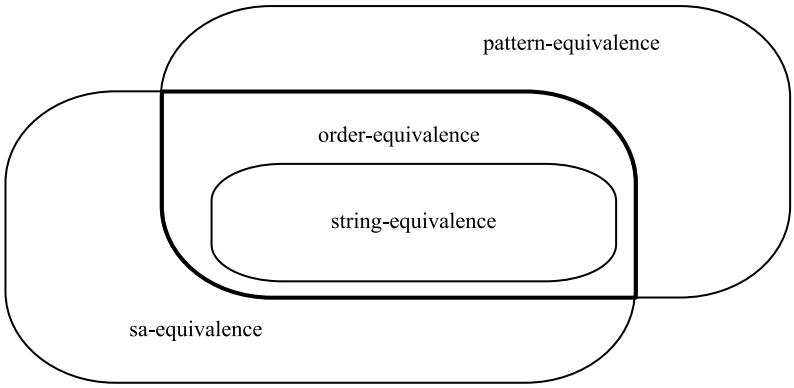

Figure 3.1: The relationships among the different equivalences on strings.

3.1 Equivalences on strings

We use three different kinds of equivalences on strings. The natural definition is that strings are *(string-)equivalent* if they are equal, and *(string-)distinct* otherwise. In order to define the other two equivalences, we first introduce a bijective mapping rk of the characters of a string t onto the first $|\Sigma(t)|$ integers, $\mathrm{rk} : \Sigma(t) \longrightarrow [1, |\Sigma(t)|]$. We call rk *order-preserving* if $c_1 < c_2 \Leftrightarrow \mathrm{rk}(c_1) < \mathrm{rk}(c_2)$ for all pairs of characters $(c_1, c_2) \in \Sigma(t) \times \Sigma(t)$. The mapped string $\mathrm{rk}(t)$ is then defined by $\mathrm{rk}(t) := \mathrm{rk}(t[1]), \mathrm{rk}(t[2]), \ldots, \mathrm{rk}(t[n])$. We call two strings t and t' *order-equivalent* if there exists an order-preserving bijection rk for t and another such bijection rk' for t' such that $\mathrm{rk}(t) = \mathrm{rk}'(t')$; otherwise the strings are *order-distinct*. If bijective mappings rk and rk' exist such that $\mathrm{rk}(t) = \mathrm{rk}'(t')$ (not necessarily order-preserving), we call t and t' *pattern-equivalent*; otherwise the strings are *pattern-distinct*. String-equivalent strings are also order-equivalent and order-equivalence implies pattern-equivalence. The strings `ABBAA` and `ACCAA`, for example, are string-distinct but order-equivalent, and the strings `ABBAA` and `CBBCC` are order-distinct but pattern-equivalent.

Additionally, we define the equivalence of strings sharing the same suffix arrays. Two strings t and t' are *suffix-array-equivalent* or, alternatively, *sa*-equivalent if they share the same suffix array, $sa(t) = sa(t')$; otherwise the strings are *sa*-distinct. Order-equivalence implies *sa*-equivalence since the order of suffixes is not affected through an order-preserving mapping of the characters.

If two strings are order-distinct, they can either be *sa*-equivalent or pattern-equivalent, but not both. Let t and t' be two order-distinct strings. Then either there are no bijective character mappings rk and rk' such that $\mathrm{rk}(t) = \mathrm{rk}'(t')$ or the bijective mappings are not order-preserving. If there are no such bijective character mappings, then t and t' are pattern-distinct, but can still be *sa*-equivalent. Otherwise, if such bijective mappings exist but are not order-preserving, then t and t' are yet pattern-equivalent; a rearrangement of the alphabet that changes the relative alphabetical order, however, induces a different relative order of the suffixes, which implies suffix array distinctness. The order-

distinct strings ABBAA and BDCAA, for example, are *sa*-equivalent but pattern-distinct, and the order-distinct strings ABBAA and CBBCC are pattern-equivalent but *sa*-distinct. The relationships among the mentioned four equivalences on strings are shown in Figure 3.1.

The regarded equivalences of strings are obviously reflexive, symmetric, and transitive. Hence, they are equivalence relations and thus induce a partitioning of the set of strings into equivalence classes. Our main focus is on the *sa*-equivalence classes. We recall the function *sa* that maps each string of length $n \in \mathbb{N}$ to its suffix array P,

$$sa : \begin{cases} \Sigma^n & \longrightarrow & \mathcal{P}^n (\supset \mathcal{P}^n_\Sigma) \\ t & \longmapsto & P, \end{cases}$$

where $\mathcal{P}^n_\Sigma$ is the image of Σ^n under *sa*. Alternatively, $\mathcal{P}^n_\Sigma$ is called the set of suffix arrays of Σ^n. For large n and fixed small alphabet Σ of size σ, *sa* is not surjective; hence $\mathcal{P}^n_\Sigma \subsetneqq \mathcal{P}^n$. Moreover, it is not injective for $\sigma > 1$. We define the function sa^{-1} that maps each permutation P to its *preimage* under *sa*

$$sa^{-1} : \begin{cases} \mathcal{P}^n & \longrightarrow & 2^{(\Sigma^n)} \\ P & \longmapsto & \mathcal{T}_{P,\Sigma} = \{t \in \Sigma^n \,:\, sa(t) = P\}. \end{cases}$$

The function sa^{-1} maps each permutation to the *sa*-equivalence class of suffixes sharing the same suffix array P, $sa^{-1}(P) = \mathcal{T}_{P,\Sigma}$. If $P \in \mathcal{P}^n_\Sigma$, then the preimage of P under *sa* is not empty; otherwise $sa^{-1}(P) = \emptyset$. Hence, the function sa^{-1} partitions the set of strings Σ^n into $|\mathcal{P}^n_\Sigma|$ non-empty equivalence classes. In Chapter 4, we count the number of specific elements in an equivalence class $sa^{-1}(P)$ for any $P \in \mathcal{P}^n$. The number $|\mathcal{P}^n_\Sigma|$ of non-empty equivalence classes is counted in Chapter 5.

3.2 The ^+R-array

We define the ^+R-array, the basic data structure for the subsequent analysis of the suffix array equivalences.

Definition 3.1 (^+R-array). Let $P \in \mathcal{P}^n$ be a permutation of $[1, n]$. The ^+R-array of P is defined as

$$^+R[i] := R[P[i] + 1] \quad \text{for all } i \in [1, n].$$

In the compressed indexing literature the ^+R-array is usually called Ψ-array or Ψ-function. We define the ^+R-descents and the ^+R-runs of P similar to the permutation descents and the permutation runs respectively: A position $i \in [1, n-1]$ is called a ^+R-*descent* if $^+R[i] > {}^+R[i+1]$. For $l \le r$, a non-extendable ascending segment $^+R[l] < {}^+R[l+1] < \ldots < {}^+R[r]$ is called a ^+R-*run*; it will be denoted $^+R[l, r]$. The set of ^+R-descents $\{i \in [1, n-1] \,:\, {}^+R[i] > {}^+R[i+1]\}$ is denoted by ^+R-*desc*(P). If the ordered set of ^+R-descents of P equals $\{i_1, i_2, \ldots, i_d\}$ with $i_j < i_{j+1}$ for all $j \in [1, d-1]$, then i_j is called the j^{th} ^+R-descent. The list of ^+R-runs is $^+R[1, i_1], {}^+R[i_1+1, i_2], \ldots, {}^+R[i_{d-1}+1, i_d], {}^+R[i_d+1, n]$, where $^+R[i_{j-1}+1, i_j]$ is called the j^{th} ^+R-run. Note that ^+R-runs can be of length 1.

Table 3.1: The permutation P, which is the suffix array for the string ABBAA, the sorted suffixes of the string $t[P[i], n]$, the rank array R, the ${}^{+}R$-array, and the ${}^{+}R$-descent at position 3.

i	$P[i]$	$t[P[i], n]$	$R[i]$	${}^{+}R[i]$	${}^{+}R$-$desc(P)$
0	6	ε			
1	5	A	3	0	
2	4	AA	5	1	
3	1	ABBAA	4	5	$\longleftarrow$
4	3	BAA	2	2	
5	2	BBAA	1	4	
6			0		

Moreover, let d_i be the number of ${}^{+}R$-descents in the prefix $P[1, i]$ of the permutation P, $d_i := |\{j \in {}^{+}R\text{-}desc(P) : j < i\}|$.

If $P = sa(t)$ is the suffix array of a string t, then the ${}^{+}R$-array reflects the connection between consecutive suffixes of t. ${}^{+}R[i] = j$ has the following interpretation: The successor suffix $t[P[i] + 1, n]$ of the lexicographically i^{th} suffix $t[P[i], n]$ is the lexicographically j^{th} among all suffixes of t. Since there does not exist a predecessor for the suffix number 1, the position j in the suffix array P with $P[j] = 1$ never appears in the ${}^{+}R$-array. If a position i is a ${}^{+}R$-descent, then the successor suffixes of $t[P[i], n]$ and $t[P[i + 1], n]$ are in descending lexicographical order: $t[P[i] + 1, n] > t[P[i + 1] + 1, n]$. A ${}^{+}R$-run ${}^{+}R[l, r]$ corresponds to a continuous suffix array segment, in which also the respective successor suffixes are in ascending lexicographical order.

For the permutation $P = (5, 4, 1, 3, 2)$, which is the suffix array of the string ABBAA, Table 3.1 shows the ${}^{+}R$-annotations. The columns show the array indices i, the permutation P, the sorted suffixes of the string $t[P[i], n]$, the rank array R, the ${}^{+}R$-array, and the only ${}^{+}R$-descent at position 3. The suffix array P is extended with the number 6 at position 0 and the R-array with the number 0 at position 6, indicating that the empty suffix, which does not appear in P, is always the smallest. Note that P contains a ${}^{+}R$-descent at position 3. Hence, ${}^{+}R[1, 3]$ and ${}^{+}R[4, 5]$ are the ${}^{+}R$-runs.

3.3 Characterising strings sharing the same suffix array

The following theorem was first given, without proof, by Burkhardt and Kärkkäinen [31] and equivalent results were proved by Duval and Lefebvre [44].

Theorem 3.2. *Let $P \in \mathcal{P}^n$ be any permutation of $[1, n]$ and t a string of length n. Then $t \in sa^{-1}(P)$ if and only if the following two conditions hold for all $i \in [1, n - 1]$:*

(a) $t[P[i]] \leq t[P[i + 1]]$ *and*

(b) ${}^{+}R[i] > {}^{+}R[i + 1] \Rightarrow t[P[i]] < t[P[i + 1]]$.

Table 3.2: The permutation P, the ${}^{+}R$-array, the ${}^{+}R$-descent at position 3, and the First sequences for the strings $t_1 =$ ABBAA, $t_2 =$ BDCAA, $t_3 =$ BDDBB, and $t_4 =$ CDDCA that share the same suffix array P.

				Strings with suffix array P			
				$t_1 =$ ABBAA	$t_2 =$ BDCAA	$t_3 =$ BDDBB	$t_4 =$ CDDCA
i	$P[i]$	${}^{+}R[i]$	${}^{+}R\text{-}desc(P)$	$t_1[P[i]]$	$t_2[P[i]]$	$t_3[P[i]]$	$t_4[P[i]]$
1	5	0		A	A	B	A
2	4	1		A	A	B	C
3	1	5	⟵	A	B	B	C
4	3	2		B	C	D	D
5	2	4		B	D	D	D

Theorem 3.2 has the following interpretation. Condition (a) states that the First sequence for t is non-decreasing, and condition (b) states: if the successor suffixes of $t[P[i], n]$ and $t[P[i+1], n]$ are in descending lexicographical order, that is, if $t[P[i]+1, n] > t[P[i+1]+1, n]$, then the relative order of $t[P[i], n]$ and $t[P[i+1], n]$ is determined by their first character, $t[P[i]] < t[P[i+1]]$.

Table 3.2 shows the permutation $P = (5, 4, 1, 3, 2)$ and the strings $t_1 =$ ABBAA, $t_2 =$ BDCAA, $t_3 =$ BDDBB, and $t_4 =$ CDDCA in the respective sa-equivalence class $sa^{-1}(5, 4, 1, 3, 2)$. The leftmost four columns show the array indices i, the permutation P, the ${}^{+}R$-array, and the ${}^{+}R$-descent; the remaining columns show the First sequences for t_1, t_2, t_3, and t_4. From reading each of the First sequences top down, it becomes evident that Theorem 3.2(a) holds for each of the four strings. Moreover, for the ${}^{+}R$-descent 3, the character $t_k[P[3]]$ is smaller than $t_k[P[3+1]]$ for each $k \in [1, 4]$, satisfying Theorem 3.2(b).

3.3.1 Proving the characterisation – Proof of Theorem 3.2

We first prove two auxiliary lemmas (Lemma 3.3 and Lemma 3.4), which are eventually used in the main proof of Theorem 3.2. First of all, Lemma 3.3 generalises a proposition about consecutive elements in a permutation to arbitrary pairs of elements.

Lemma 3.3. *Let $P \in \mathcal{P}^n$ be any permutation of $[1, n]$ and t a string of length n. If for all $i \in [1, n-1]$ we have that*

(a) $t[P[i]] \leq t[P[i+1]]$ *and*

(b) $t[P[i]] = t[P[i+1]] \;\Rightarrow\; R[P[i]+1] < R[P[i+1]+1]$,

then we also have that for all pairs (i, j), $1 \leq i < j \leq n$,

$$t[P[i]] = t[P[j]] \Rightarrow R[P[i]+1] < R[P[j]+1].$$

Proof. Due to (a), the sequence of characters $t[P[i]], t[P[i+1]], \ldots, t[P[j]]$ is non-decreasing. Combining this property with $t[P[i]] = t[P[j]]$ implies that $t[P[i']] = t[P[i'+1]]$ for all $i' \in [i, j-1]$. Then applying (b) on $t[P[i']] = t[P[i'+1]]$ leads us to $R[P[i']+1] < R[P[i'+1]+1]$ for all $i' \in [i, j-1]$. By transitivity, we finally obtain $R[P[i]+1] < R[P[j]+1]$. □

Before we can prove the main result of this section, we continue with a further generalisation. We extend our proposition from elements of the permutation referring to equal characters in the string to elements referring to starting positions of equal substrings.

Lemma 3.4. *Let $P \in \mathcal{P}^n$ be any permutation of $[1, n]$ and t a string of length n. If for all pairs (i, j) with $1 \leq i < j \leq n$ we have that*

$$t[P[i]] = t[P[j]] \quad \Rightarrow \quad R[P[i]+1] < R[P[j]+1], \tag{3.1}$$

then we also have that for all pairs (i, j) with $1 \leq i < j \leq n$ and for all $k > 0$ with $P[i]+k-1 \leq n$ and $P[j]+k-1 \leq n$

$$t[P[i], P[i]+k-1] = t[P[j], P[j]+k-1] \Rightarrow R[P[i]+k] < R[P[j]+k]. \tag{3.2}$$

Proof (Induction over k). For $k = 1$, the equation $t[P[i], P[i]+1-1] = t[P[j], P[j]+1-1]$ accords to $t[P[i]] = t[P[j]]$; and hence, implication (3.2) accords to implication (3.1).

We now perform the induction step starting with

$$t[P[i], P[i]+k] = t[P[j], P[j]+k],$$

which is obviously equivalent to

$$t[P[i], P[i]+k-1] = t[P[j], P[j]+k-1] \tag{3.3}$$

$$\text{and} \quad t[P[i]+k] = t[P[j]+k]. \tag{3.4}$$

Applying the induction hypothesis (3.2) to (3.3) gives $R[P[i]+k] < R[P[j]+k]$. Then we choose i' and j' such that $P[i'] = P[i]+k$ and $P[j'] = P[j]+k$. Since R is the inverse of P, we obtain

$$i' = R[P[i']] = R[P[i]+k] < R[P[j]+k] = R[P[j']] = j'. \tag{3.5}$$

Combining equation (3.4) with $P[i'] = P[i]+k$ and $P[j'] = P[j]+k$ implies

$$t[P[i']] = t[P[i]+k] = t[P[j]+k] = t[P[j']].$$

By (3.5) i' is smaller than j', so implication (3.1) is applicable and leads to

$$R[P[i']+1] < R[P[j']+1].$$

Substituting $P[i']$ by $P[i]+k$ and $P[j']$ by $P[j]+k$ results in $R[P[i]+k+1] < R[P[j]+k+1]$, completing the proof. □

We are now ready for proving Theorem 3.2.

Proof of Theorem 3.2. If $t \in sa^{-1}(P)$ or, alternatively, if the permutation P is the suffix array for the string t, then the conditions (a) and (b) of the theorem clearly hold.

The opposite direction is more intricate. Assume that both conditions (a) and (b) hold. If P is not the suffix array of t, then there must be two incorrectly ordered suffixes in P. Let i and j be the positions of these suffixes in P such that $i < j$ and $t[P[i], n] > t[P[j], n]$.

Negating condition (b) and using the definition of ${}^{+}R$ gives for all $i \in [1, n-1]$

$$t[P[i]] \geq t[P[i+1]] \Rightarrow R[P[i]+1] \leq R[P[i+1]+1],$$

and by (a) and by the fact that both R and P are different at unequal positions, we obtain for all $i \in [1, n-1]$ that

$$t[P[i]] = t[P[i+1]] \Rightarrow R[P[i]+1] < R[P[i+1]+1].$$

We apply Lemma 3.3 and Lemma 3.4 to obtain for all $i, j \in [1, n]$, $i < j$,

$$t[P[i], P[i]+k-1] = t[P[j], P[j]+k-1] \Rightarrow R[P[i]+k] < R[P[j]+k]. \tag{3.6}$$

Now let ℓ be the length of the longest common prefix of $t[P[i], n]$ and $t[P[j], n]$, then we distinguish between two cases.

(i) If $\ell = 0$, the suffixes differ in their first position. Since $t[P[i], n] > t[P[j], n]$, the first character $t[P[i]]$ of $t[P[i], n]$ must be greater than the first character $t[P[j]]$ of $t[P[j], n]$, which contradicts (a).

(ii) If $\ell > 0$, the suffixes $t[P[i], n]$ and $t[P[j], n]$ share a longest common prefix of length ℓ, that is, $t[P[i], P[i]+\ell-1] = t[P[j], P[j]+\ell-1]$. Then implication (3.6) leads to $R[P[i]+\ell] < R[P[j]+\ell]$. We choose i' and j' such that $P[i'] = P[i]+\ell$ and $P[j'] = P[j]+\ell$. Since R is the inverse of P, we have $i' = R[P[i']] = R[P[i]+\ell] < R[P[j]+\ell] = R[P[j']] = j'$. Therefore, using (a) we obtain

$$t[P[i]+\ell] = t[P[i']] \leq t[P[j']] = t[P[j]+\ell]. \tag{3.7}$$

This contradicts the assumption that $t[P[i], n] > t[P[j], n]$ with longest common prefix of length ℓ such that $t[P[i]+\ell] > t[P[j]+\ell]$.

Since both cases lead to contradictions, all suffixes represented in P must be in the correct order; hence $t \in sa^{-1}(P)$. □

4 Counting and Enumerating the Strings per Suffix Array

Enumerative combinatorics is a major subfield of combinatorics (see, for example, [138, 103, 34, 29]). For any particular combinatorial structure, it poses the following questions: How many combinatorial objects of a particular type are there (Counting), and how can we list all these objects (Enumeration). To the best of our knowledge, such questions relating to suffix arrays have not been studied before. In this and the next chapter, we are the first providing answers on that.

In this chapter, we count and enumerate, for any permutation $P \in \mathcal{P}^n$ and a fixed-sized alphabet Σ, the strings in the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ of all strings in Σ^n with P as their suffix array (see page 13), considering particular subsets of strings: string-distinct strings composed of *up to* $|\Sigma|$ distinct characters (not all characters of the alphabet must appear) and string-distinct strings composed of *exactly* $|\Sigma|$ distinct characters (all characters must appear). We proceed as follows: We first present the number of the different sets of counted strings, especially Theorem 4.1 and Theorem 4.2. Then, after introducing the foundations for the subsequent string counting in Section 4.1, we prove Theorem 4.1 in Section 4.2 and Theorem 4.2 in Section 4.3. Finally, we give enumeration algorithms for both sets of counted strings in Section 4.4.

The main results of this chapter are the following two theorems.

Theorem 4.1. *Let $P \in \mathcal{P}^n$ be any permutation of length n with d ${}^+R$-descents and Σ an alphabet of $\sigma = |\Sigma|$ ordered symbols. Then the number of string-distinct strings in the sa-equivalence class $\mathcal{T}_{P,\Sigma}$ is given by $\binom{n+\sigma-d-1}{\sigma-d-1}$.*

Theorem 4.2. *Let $P \in \mathcal{P}^n$ be any permutation of length n with d ${}^+R$-descents and Σ an alphabet of $\kappa = |\Sigma|$ ordered symbols. Then the number of string-distinct strings composed of exactly κ distinct characters in the sa-equivalence class $\mathcal{T}_{P,\Sigma}$ is given by $\binom{n-d-1}{\kappa-d-1}$.*

For the various settings, Table 4.1 summarises the number of string-distinct, order-distinct, and pattern-distinct strings of length n. Some of the results were previously proven by other authors or are given by Theorems 4.1 and 4.2. We discuss the results presented in Table 4.1 row-wise, beginning with the first row. Moore *et al.* [109] showed that the number of pattern-distinct strings composed of exactly κ distinct characters is $\left\{{n \atop \kappa}\right\}$. For each pattern-distinct string, we permute the alphabet in $\kappa!$ different ways to get a total of $\left\{{n \atop \kappa}\right\}\kappa!$ order-distinct strings. These are already all the string-distinct strings since we have no flexibility to choose different characters to produce string-distinct strings that are yet order-equivalent.

Table 4.1: Summary of the previous and new results on the number of string-distinct, order-distinct and pattern-distinct strings of length n. In the analyses d is always the number of ${}^{+}R$-descents for the respective suffix array P. Moreover, Σ is the underlying alphabet of $\kappa = \sigma = |\Sigma|$ ordered symbols.

Number of	string-distinct	order-distinct	pattern-distinct
strings composed of exactly κ distinct letters	$\left\{ {n \atop \kappa} \right\} \cdot \kappa!$	$\left\{ {n \atop \kappa} \right\} \cdot \kappa!$	$\left\{ {n \atop \kappa} \right\}$ [109]
strings composed of up to σ distinct letters	σ^n	$\sum_{\kappa=1}^{\sigma} \left\{ {n \atop \kappa} \right\} \cdot \kappa!$	$\sum_{\kappa=1}^{\sigma} \left\{ {n \atop \kappa} \right\}$
strings in $\mathcal{T}_{P,\Sigma}$ composed of exactly κ distinct letters	$\binom{n-d-1}{\kappa-d-1}$ [Thm. 4.2]	$\binom{n-d-1}{\kappa-d-1}$	—
strings in $\mathcal{T}_{P,\Sigma}$ composed of up to σ distinct letters	$\binom{n+\sigma-d-1}{\sigma-d-1}$ [Thm. 4.1]	$\sum_{\kappa=d+1}^{\sigma} \binom{n-d-1}{\kappa-d-1}$	—

The numbers of all strings over a given alphabet of size σ are shown in the second row. There are σ^n string-distinct strings. For the order- and pattern-distinct strings, we sum up the number of strings for all possible κ.

The number of string-distinct strings composed of exactly κ distinct characters in the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ for any permutation P with d ${}^{+}R$-descents and an alphabet Σ of fixed size κ is given in Theorem 4.2. These strings are again order-distinct. For pattern-distinct strings, we cannot necessarily determine a unique suffix array. This fact has already been investigated in Chapter 3.1 and a graphical representation is shown in Figure 3.1. It is indicated by a dash in the table.

The number of string-distinct and order-distinct strings in the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ for any permutation P and an alphabet Σ of size σ are given in the fourth row. Theorem 4.1 gives the number of string-distinct strings; to count the order-distinct strings, we sum up over all possible κ. Again, the dash denotes that we cannot necessarily determine a unique suffix array for pattern-distinct strings.

4.1 Foundations

Before we prove Theorem 4.1 in Section 4.2 and Theorem 4.2 in Section 4.3, we first repeat an observation of Bannai *et al.* [14] that links the minimal alphabet size of the strings in the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ to the number of ${}^{+}R$-descents of P: For a permutation P with d ${}^{+}R$-descents, the number of different characters in a string $t \in \mathcal{T}_{P,\Sigma}$ is at least the number of ${}^{+}R$-descents plus one, $|\Sigma(t)| \geq d + 1$. Furthermore, Bannai *et al.* presented an algorithm to construct a unique string $b_P \in \mathcal{T}_{P,\Sigma}$ consisting of exactly $d + 1$ different

Algorithm 4.1.
BaseString(P, n)
  $c \leftarrow 1$
  for $i \leftarrow 1$ to n **do**
    $b_P[P[i]] \leftarrow c$
    if $i \in {}^+R\text{-desc}(P)$ **then**
      $c \leftarrow c + 1$
    end if
  end for
  return b_P

Table 4.2: Construction of the base string b_P of the permutation P having the ${}^+R$-descent 3.

i	$P[i]$	${}^+R[i]$	$b_P[P[i]]$	b_P
1	5	0	A	____A
2	4	1	A	___AA
3	1	5	A	A__AA
4	3	2	B	A_BAA
5	2	4	B	ABBAA

characters, $|\Sigma(b_P)| = d + 1$. Note that b_P is only defined for non-empty *sa*-equivalence classes $\mathcal{T}_{P,\Sigma}$ with $P \in \mathcal{P}_\Sigma^n$.

Without loss of generality, we assume that the character set of b_P contains the first natural numbers, $\Sigma(b_P) = [1, d+1]$, and call b_P the *base string* of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$, its lexicographically smallest representative. Nevertheless, we synonymously use the characters $\{\mathtt{A}, \mathtt{B}, \ldots\}$ for illustrations. The algorithm suggested in [14] works as follows. It starts with the initial character $c = 1$. For each index position $i \in [1, n]$ in ascending order, the algorithm proceeds through all suffix numbers from $P[1]$ to $P[n]$ by assigning c to $b_P[P[i]]$. If i is a ${}^+R$-descent, c is incremented by one to satisfy condition (2) of Theorem 3.2, such that $b_P[P[i]] = d_i + 1$; we recall that d_i is the number of ${}^+R$-descents in the prefix $P[1, i]$ of the suffix array P (see page 14). The pseudo-code is given in Algorithm 4.1. Note that the algorithm can only construct a correct base string if the size of the underlying alphabet exceeds the number of ${}^+R$-descents of the input permutation, and fails otherwise. For the permutation $P = (5, 4, 1, 3, 2)$ with ${}^+R$-descent 3, Table 4.2 shows the successive assignment of characters to the base string b_P. The columns show the array indices i, the permutation P, the ${}^+R$-array, the First sequence for the base string $b_P[P[i]]$, and the assignment of characters to the base string b_P.

Proposition 4.3. *Let P be a permutation with d ${}^+R$-descents, then the base string b_P has the properties*

(a) $b_P[P[1]] = 1$ *and* $b_P[P[n]] = d + 1$,

(b) $b_P[P[i+1]] = b_P[P[i]]$ *if* $i \in [1, n-1]$ *is not a* ${}^+R$*-descent of* P,

(c) $b_P[P[i+1]] = b_P[P[i]] + 1$ *if* $i \in [1, n-1]$ *is a* ${}^+R$*-descent of* P.

Note that each ${}^+R$-run ${}^+R[l, r]$ of the base string corresponds to an interval of equal characters of the First sequence for the base string, $b_P[P[l]] = b_P[P[l+1]] = \ldots = b_P[P[r]]$.

4.2 Counting strings composed of up to σ distinct characters

For a permutation $P \in \mathcal{P}_\Sigma^n$, the strings contained in the respective *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ can be derived from the base string b_P of $\mathcal{T}_{P,\Sigma}$ by applying a certain sequence of rewrite

Table 4.3: The permutation P, the ${}^{+}R$-array, the first characters of the ordered suffixes of the base string $b_P = \texttt{ABBAA}$, and the m-incremented strings $t_{P,m} = \texttt{BDCAA}$, $t_{P,m'} = \texttt{BDDBB}$ and $t_{P,m''} = \texttt{CDDCA}$ over the alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$.

				m-incremented string					
				$t_{P,m} = \texttt{BDCAA}$		$t_{P,m'} = \texttt{BDDBB}$		$t_{P,m''} = \texttt{CDDCA}$	
i	$P[i]$	${}^{+}R[i]$	$b_P[P[i]]$	m	$t_{P,m}[P[i]]$	m'	$t_{P,m'}[P[i]]$	m''	$t_{P,m''}[P[i]]$
1	5	0	A	+0	A	+1	B	+0	A
2	4	1	A	+0	A	+1	B	+2	C
3	1	5	A	+1	B	+1	B	+2	C
4	3	2	B	+1	C	+2	D	+2	D
5	2	4	B	+2	D	+2	D	+2	D

operations to the base string after which the order of suffixes remains untouched. The sequence of rewrite operations starts with the largest suffix. Increasing the first character of the largest suffix by any number $a \in \mathbb{N}$ does not change the order of suffixes. Then the first character of the second largest suffix can be increased by at most a without changing the order of suffixes, and so on.

We proceed as follows: We first define the sequence of rewrite operations (Definition 4.4), establish a bijection between a particular set of rewrite operations and the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ for any permutation $P \in \mathcal{P}_\Sigma^n$ (Lemma 4.5), count the number of these rewrite operations (Lemma 4.6), and finally derive the size of $\mathcal{T}_{P,\Sigma}$, which gives the proof of Theorem 4.1.

Definition 4.4. Let Σ be the underlying alphabet, $P \in \mathcal{P}_\Sigma^n$ a permutation of $[1, n]$ and b_P the base string of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$. Moreover, let m be an integer sequence of length n, $m \in \mathbb{Z}^n$ (usually m is a sequence of non-negative integers). The *m-incremented string* $t_{P,m}$ of b_P is defined as

$$t_{P,m}[P[i]] := b_P[P[i]] + m[i] \quad \text{for all } i \in [1, n].$$

That is, the i^{th} smallest character of b_P is increased by $m[i]$. Note that we assume $\Sigma = [1, |\Sigma|]$ and allow m-incremented strings $t_{P,m}$ with $\Sigma(t_{P,m}) \nsubseteq \Sigma$. In particular, the m-incremented strings span the set of integer strings of length n: $\mathbb{Z}^n = \{t_{P,m} \in \mathbb{Z}^n : m \in \mathbb{Z}^n\}$ for any permutation $P \in \mathcal{P}_\Sigma^n$. We use this property in Lemma 4.5.

For the permutation $P = (5, 4, 1, 3, 2)$, Table 4.3 shows the connection between the base string ABBAA and three m-increment sequences over the alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$. The leftmost four columns show again the array indices i, the permutation P, the ${}^{+}R$-array, and the First sequence for the base string. Each of the following three pairs of columns show the modification of the base string b_P, or rather, the modification of the corresponding first array by non-decreasing sequences to produce m-incremented strings: m-incrementing

the base string by $m = 0,0,1,1,2$ produces $t_{P,m} =$ BDCAA, $m' = 1,1,1,2,2$ produces $t_{P,m'} =$ BDDBB, and $m'' = 0,2,2,2,2$ produces $t_{P,m''} =$ CDDCA. Like the base string ABBAA, the m-incremented strings BDCAA, BDDBB, CDDCA are contained in $\mathcal{T}_{(5,4,1,3,2),\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}}$.

Lemma 4.5. *Let Σ be an ordered alphabet of size $\sigma := |\Sigma|$, $P \in \mathcal{P}^n_\Sigma$ a permutation of $[1,n]$ with d ${}^+R$-descents. Moreover, let $\mathcal{M}_{P,\sigma}$ be the set of non-decreasing sequences of length n over the ordered alphabet $[0, \sigma - d - 1]$.*

Then there exists an isomorphism between $\mathcal{T}_{P,\Sigma}$ and $\mathcal{M}_{P,\sigma}$, $\mathcal{T}_{P,\Sigma} \simeq \mathcal{M}_{P,\sigma}$.

Proof. Let b_P be the base string of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$. Without loss of generality, we assume $\Sigma = [1,\sigma]$. We show: (i) each non-decreasing sequence $m \in \mathcal{M}_{P,\sigma}$, generates an m-incremented string $t_{P,m} \in \mathcal{T}_{P,\Sigma}$ and (ii) each other sequence $o \in \mathbb{Z}^n$ of length n, $o \notin \mathcal{M}_{P,\sigma}$, generates a string $t_{P,o} \notin \mathcal{T}_{P,\Sigma}$.

(i) Let $m \in \mathcal{M}_{P,\sigma}$, such that $m[i] \leq m[i+1]$ for all $i \in [1, n-1]$. We verify the conditions of Theorem 3.2 for $t_{P,m}$:

(i.1) For all $i \in [1, n-1]$, we obtain $b_P[P[i]] \leq b_P[P[i+1]]$ from Proposition 4.3 (b) and (c). That implies

$$t_{P,m}[P[i]] = b_P[P[i]] + m[i] \leq b_P[P[i+1]] + m[i+1] = t_{P,m}[P[i+1]],$$

verifying Theorem 3.2(a).

(i.2) If ${}^+R[i] > {}^+R[i+1]$, then $i \in {}^+R\text{-desc}(P)$. Proposition 4.3(c) gives $b_P[P[i]]+1 = b_P[P[i+1]]$, which leads to

$$\begin{aligned} t_{P,m}[P[i]] &= b_P[P[i]] + m[i] \\ &< (b_P[P[i]] + 1) + m[i] \\ &\leq b_P[P[i+1]] + m[i+1] = t_{P,m}[P[i+1]], \end{aligned}$$

verifying Theorem 3.2(b).

Therefore, $sa(t_{P,m}) = P$.

Moreover, for each position j of $t_{P,m}$ with $j = P[i]$ for some $i \in [1,n]$,

$$t_{P,m}[j] = t_{P,m}[P[i]] = b_P[P[i]] + m[i] \leq (d+1) + (\sigma - d - 1) = \sigma$$

and analogously $1 \leq t_{P,m}[j]$. Hence, each $m \in \mathcal{M}_{P,\sigma}$ generates a sequence $t_{P,m} \in \mathcal{T}_{P,\Sigma}$ $(\subset \Sigma^n)$.

(ii) For $o \notin \mathcal{M}_{P,\sigma}$ containing a descending adjacent index pair such that $o[i] > o[i+1]$ for some $i \in [1, n-1]$, we concern ourselves with two cases:

(ii.1) If i is not a ${}^+R$-descent of P, then Proposition 4.3(b) states $b_P[P[i]] = b_P[P[i+1]]$. Hence,

$$t_{P,o}[P[i]] = b_P[P[i]] + o[i] > b_P[P[i+1]] + o[i+1] = t_{P,o}[P[i+1]],$$

which contradicts Theorem 3.2(a).

(ii.2) If i is a ${}^{+}R$-descent of P, then Proposition 4.3(c) states $b_P[P[i]] = b_P[P[i{+}1]]{-}1$ and, because of $o[i] > o[i+1]$, also $o[i] \geq o[i+1]+1$ is true. This results in

$$\begin{aligned} t_{P,o}[P[i]] &= b_P[P[i]] + o[i] \\ &\geq (b_P[P[i+1]] - 1) + (o[i+1] + 1) \\ &= b_P[P[i+1]] + o[i+1] \\ &= t_{P,o}[P[i+1]], \end{aligned}$$

which contradicts Theorem 3.2(b).

Therefore, only the non-decreasing sequences m produce a string $t_{P,m}$ such that $sa(t_{P,m}) = P$.

The non-decreasing sequences $o \notin \mathcal{M}_{P,\sigma}$, for which $\Sigma(o) \not\subseteq [0, \sigma - d - 1]$, remain. For all these strings, we show that $t_{P,o} \notin \Sigma^n$. If o is non-decreasing, but not in $\mathcal{M}_{P,\sigma}$, it must contain a character greater than $\sigma - d - 1$ or smaller than 0 at some position i. Since o is non-decreasing, such a character appears at position n or 1. That is, $o[n] > \sigma - d - 1$ or $o[1] < 0$. Combining $o[n] > \sigma - d - 1$ with the fact from Proposition 4.3(a) that $b_P[P[n]] = d + 1$ implies

$$t_{P,o}[P[n]] = b_P[P[n]] + o[n] > (d+1) + (\sigma - d - 1) = \sigma.$$

Using $b_P[P[1]] = 0$ for $o[1] < 0$ analogously implies $t_{P,o}[P[1]] < 0$. Thus, $t_{P,o} \notin \Sigma^n$, completing the proof. □

Finally, we prove that the number of sequences in the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ for any permutation P is the same as the number of non-decreasing sequences over $\sigma - d$ characters. To count the number of non-decreasing sequences of length n composed of μ elements, we observe the following:

Lemma 4.6. *Let $M(n, \mu)$ be the number of non-decreasing sequences of length n of elements in $[0, \mu - 1]$. For any positive integers n and μ*

$$M(n, \mu) = \binom{n + \mu - 1}{\mu - 1}.$$

Proof. The non-decreasing sequences of length n composed of μ symbols can be modelled as a sequence of two different operations. Initially, the current symbol is set to 0. Then we apply a sequence of operations to generate non-decreasing sequences of length n. One possible operation is to write the current symbol behind the so far written symbols and the other one is to increment the symbol by 1. To generate a non-decreasing sequence, we apply $n + \mu - 1$ operations, n to write down the non-decreasing sequence and $\mu - 1$ to increment the current symbol until $\mu - 1$ is reached. For this sequence of length $n + \mu - 1$, we have $\binom{n+\mu-1}{\mu-1}$ possibilities to choose the $\mu - 1$ positions of the increment operations. □

The respective representation of the sequence $2, 2, 2, 2, 4, 5, 5$ is shown in Figure 4.1.

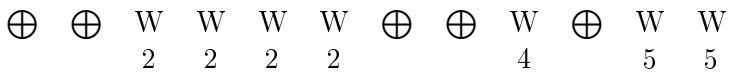

Figure 4.1: Representation of the non-decreasing sequence 2,2,2,2,4,5,5 for $\mu = 6$, where $\bigoplus$ denotes an increment operation and W denotes a write operation.

When applying this observation to Lemma 4.5, we get the number of strings in an *sa*-equivalence class.

Proof of Theorem 4.1. For each permutation $P \in \mathcal{P}^n_\Sigma$, the claim follows directly from the bijection shown in Lemma 4.5 and the equality $|\mathcal{M}_{P,\sigma}| = M(n, \sigma - d) = \binom{n+\sigma-d-1}{\sigma-d-1}$ from Lemma 4.6. For each other permutation $P \in \mathcal{P}^n$ with $P \notin \mathcal{P}^n_\Sigma$, we have $d \geq \sigma$ and thus $\binom{n+\sigma-d-1}{\sigma-d-1} = 0$. □

Remark. There are further instances for the number $\binom{n+\sigma-d-1}{\sigma-d-1}$. We have, for example, $\binom{n+\sigma-d-1}{\sigma-d-1} = \left(\!\binom{n+1}{\sigma-d-1}\!\right) = (n, \sigma - d - 1)!$, where $\left(\!\binom{x}{y}\!\right)$ denotes the number of distinct multisets of size y on x symbols and $(a, b)!$ is a multinomial coefficient that denotes the number of ways of depositing $a + b$ distinct objects into two sets, the first set of size a and the second of size b. Hence, for the strings counted in Theorem 4.1, there exist further bijections to other combinatorial objects: a bijection to the family of multisets of size $\sigma - d - 1$ on $n + 1$ symbols and a bijection to the ways of depositing $n + \sigma - d - 1$ distinct objects into two sets, the first set of size n and the second of size $\sigma - d - 1$.

For $n = 2$, $\sigma = 4$, and $d = 1$, Table 4.4 shows a specific instance for each of the bijective combinatorial objects: The set of strings $\mathcal{T}_{(2,1),\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}}$ over the alphabet $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}\}$ sharing the suffix array $(2, 1)$, the family of multisets $\left(\!\binom{\{a,b,c\}}{2}\!\right)$ of size 2 on the symbols $\{a, b, c\}$, and the ways $\binom{\{\mathrm{a,b,c,d}\}}{2,2}$ of depositing the symbols {a,b,c,d} into two sets both of size 2.

Table 4.4: The three bijective sets $\mathcal{T}_{(1,2),\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}}$, $\left(\!\binom{\{a,b,c\}}{2}\!\right)$, and $\binom{\{\mathrm{a,b,c,d}\}}{2,2}$.

$\mathcal{T}_{(1,2),\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}}$	AB	AC	AD	BC	BD	CD
$\left(\!\binom{\{a,b,c\}}{2}\!\right)$	$\{a, a\}$	$\{a, b\}$	$\{a, c\}$	$\{b, b\}$	$\{b, c\}$	$\{c, c\}$
$\binom{\{\mathrm{a,b,c,d}\}}{2,2}$	{a,b}, {c,d}	{a,c}, {b,d}	{a,d}, {b,c}	{b,c}, {a,d}	{b,d}, {a,c}	{c,d}, {b,c}

4.3 Counting strings composed of exactly κ distinct characters

So far, we have counted all strings of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ for a permutation P. Now, we count the subset $\mathcal{T}^{\kappa}_{P,\Sigma}$ of strings composed of exactly κ ($= \sigma = |\Sigma|$) distinct symbols or, alternatively, the isomorphic set of non-decreasing sequences $\mathcal{M}^{\kappa}_{P,\sigma} := \{m \in \mathcal{M}_{P,\sigma} : t_{P,m} \in \mathcal{T}^{\kappa}_{P,\Sigma}\}$; obviously $\mathcal{T}^{\kappa}_{P,\Sigma} \simeq \mathcal{M}^{\kappa}_{P,\sigma}$.

We have to determine the non-decreasing sequences $m \in \mathcal{M}_{P,\sigma}$ for which $t_{P,m}$ consists of exactly κ letters. To assure that none of the κ characters $[1,\kappa]$ is left out, it is sufficient that $t_{P,m}[P[1]] = 0$, $t_{P,m}[P[n]] = \kappa$, and consecutive characters in the resulting sequence $t_{P,m}$ are not differing by more than one.

Proposition 4.7. *Let Σ be an ordered alphabet of size $\kappa := \sigma = |\Sigma|$ and $P \in \mathcal{P}^n$ a permutation of $[1,n]$ with d ${}^+R$-descents. Moreover, let $m \in \mathcal{M}_{P,\sigma}$.*

Then $m \in \mathcal{M}^{\kappa}_{P,\sigma}$, if and only if, for all $i \in [1, n-1]$

(a) $m[1] = 0$ and $m[n] = \kappa - d - 1$,

(b) $m[i+1] = m[i]$ or $m[i+1] = m[i] + 1$ if $i \notin {}^+R\text{-}desc(P)$, and

(c) $m[i+1] = m[i]$ if $i \in {}^+R\text{-}desc(P)$.

We are now prepared to prove Theorem 4.2.

Proof of Theorem 4.2. The proof works similar as for Theorem 4.1. We again represent the non-decreasing sequences of $m \in \mathcal{M}^{\kappa}_{P,\sigma}$ as n write operations and $\mu - 1$ increment operations, as it has been modelled above. Here, for the placement of the $\kappa - d - 1$ increment operations, we are restricted by the mentioned conditions of Proposition 4.7. In order not to break these conditions, (a) an increment operation must not appear before the first or after the last write operation, (b) at most one increment operation must appear between two write operations, and (c) the d ${}^+R$-descent positions are forbidden for the increments. We are thus left with $n - d - 1$ mutually exclusive positions from which we choose the $\kappa - d - 1$ increment operations. □

Table 4.3 shows that among the three non-decreasing sequences m, m', and m'' only m generates an m-incremented string $t_{P,m}$ that fulfills the three conditions of Proposition 4.7; m' violates conditions (a) and (c), and m'' violates condition (b).

4.4 Enumerating the strings

In combinatorics, we are mainly interested in counting combinatorial objects of a particular type. As computer scientists, we are further interested in the *efficient* enumeration of those objects. This section presents two new algorithms enumerating the strings that we have previously counted. For a fixed alphabet Σ of size σ and a permutation $P \in \mathcal{P}^n_{\Sigma}$, the first algorithm enumerates all strings of $\mathcal{T}_{P,\Sigma}$, and the second enumerates the subset $\mathcal{T}^{\kappa}_{P,\Sigma}$ of such strings composed of exactly $\kappa = \sigma$ distinct characters.

Algorithm 4.2.
ENUM$_{P,\sigma}(m, t, i, \mu, enum)$

```
1:  m_enum ← m
2:  t_enum ← t
3:  enum ← enum + 1
4:  if i>0 then
5:    for h ← 1 to μ − 1 do
6:      m[i] ← m[i] + 1
7:      t[P[i]] ← t[P[i]] + 1
8:      ENUM_{P,σ}(m, t, i − 1, h + 1, enum)
9:    end for
10:   m[i] ← m[i] − (μ − 1)
11:   t[P[i]] ← t[P[i]] − (μ − 1)
12: end if
```

Table 4.5: Enumeration of the strings t_{enum} that share the suffix array $P = (5, 4, 1, 3, 2)$ with base string $b_P =$ ABBAA.

$enum$	m_{enum}	t_{enum}
1	00000	ABBAA
2	00001	ACBAA
3	00011	ACCAA
4	00111	BCCAA
5	01111	BCCBA
6	11111	BCCBB
7	00002	ADBAA
8	00012	BDBAA
⋮	⋮	⋮

4.4.1 Strings composed of up to σ distinct characters

The non-decreasing sequences of length n over $[0, \sigma - d - 1]$ can be enumerated in-place by applying one change operation at a time, beginning with the sequence 0^n. The bijection described by Definition 4.4 suggests to apply these enumeration steps directly to the base string b_P of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$.

Algorithm 4.2 shows the simultaneous enumeration of the non-decreasing sequences $m \in \mathcal{M}_{P,\sigma}$ and the strings $t \in \mathcal{T}_{P,\Sigma}$ for a permutation $P \in \mathcal{P}_\Sigma^n$; m_{enum} denotes the $enum^{th}$ enumerated non-decreasing sequence and t_{enum} the respective m-incremented string, $t_{enum} = t_{P,m_{enum}}$. The parameters of the algorithm are the current non-decreasing sequence m, the corresponding m-incremented string t, the position i according to which the modifications are performed, the current upper bound μ for the value $m[i]$ such that $m[i] < \mu$, and the current enumeration number $enum$. The enumeration is invoked with ENUM$_{P,\sigma}(0^n, b_P, n, \sigma - d, 1)$. Starting with the sequence $m = 0^n$, the algorithm increments $m[n]$ and recursively enumerates all $(n-1)$-length non-decreasing prefixes of $m = 0^{n-1}1$ over the numbers $\{0, 1\}$. Then it increments $m[n]$ again and enumerates the $(n-1)$-length non-decreasing prefixes of $0^{n-1}2$ over $\{0, 1, 2\}$. The recursive call is repeated for each sequence $0^{n-1}h$ with $1 \leq h < \mu$. Moreover, each modification operation of $m[i]$ is simultaneously applied to $t[P[i]]$ such that the strings in $\mathcal{T}_{P,\Sigma}$ are enumerated in parallel. In this way, the algorithm enumerates all $|\mathcal{T}_{P,\Sigma}|$ strings of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ over an alphabet Σ for the suffix array P in optimal $\mathcal{O}(n + |\mathcal{T}_{P,\Sigma}|)$ time, where n steps are used to construct the initial non-decreasing sequence 0^n and the base string. Moreover, it has further features: It works in-place. After each single step of the algorithm, the current sequence $m \in \mathcal{M}_{P,\sigma}$ is non-decreasing and $t \in \mathcal{T}_{P,\Sigma}$. Moreover, the enumeration works correctly for countable ordered alphabets.

Table 4.5 shows the first eight enumerated non-decreasing sequences and the respec-

tive enumerated strings of $\mathcal{T}_{P,\Sigma}$ for the permutation $P = (5,4,1,3,2)$ and the alphabet $\Sigma = \{\mathtt{A}, \mathtt{B}, \ldots\}$. The columns show the enumeration number *enum*, the enumerated non-decreasing sequences m_{enum}, and the enumerated strings t_{enum}, where t_1 is the base string of $\mathcal{T}_{(5,4,1,3,2),\{\mathtt{A},\mathtt{B},\ldots\}}$ with $t_1 = b_P = \mathtt{ABBAA}$.

4.4.2 Strings composed of exactly κ distinct characters

We modify the previous algorithm to enumerate only the subset $\mathcal{T}^{\kappa}_{P,\Sigma} (\subset \mathcal{T}_{P,\Sigma})$ of strings composed of exactly κ distinct characters for any permutation $P \in \mathcal{P}^n_\Sigma$ or, alternatively, the elements of the bijective set of non-decreasing sequences $\mathcal{M}^{\kappa}_{P,\sigma}$.

For each non-decreasing sequence $m \in \mathcal{M}^{\kappa}_{P,\sigma}$, Proposition 4.7(c) states that $m[i] = m[i+1]$ if i is a ${}^{+}R$-descent of the input permutation P. That is, some positions of m, or rather some non-increments, are pre-determined by the ${}^{+}R$-descents of P. We skip the redundant entries at the ${}^{+}R$-descent positions and confine ourselves to the isomorphic set $\mathcal{M}^{\kappa,*}_{P,\sigma}$ of non-decreasing sequences of length $n-d$ over $\mu = (\kappa - d)$ distinct symbols that fulfill Proposition 4.7(a) and (b), but ignore the ${}^{+}R$-descents.

Recall that d_i is the number of ${}^{+}R$-descents in the prefix $P[1,i]$ of the suffix array P (see page 14). We obtain the sparse permutation P^* of length $n-d$ by erasing the ${}^{+}R$-descent positions from the permutation P:

$$P^*[i - d_i] := P[i] \qquad \text{for all } i \in [1,n] \text{ with } i \notin {}^{+}R\text{-desc}(P).$$

The set of values in P^* and the set of values at the ${}^{+}R$-descent positions of P form a partitioning of the set of suffix numbers: $[1,n] = \{P^*[i] : 1 \le i \le n-d\} \uplus \{P[j] : j \in {}^{+}R\text{-desc}(P)\}$, where $\uplus$ denotes the disjoint union of two sets.

For $m^* \in \mathcal{M}^{\kappa,*}_{P,\sigma}$ (of length $n-d$), the sparse m^*-incremented string t^*_{P,m^*} of b_P (both t^*_{P,m^*} and b_P have length n) is defined by:

$$\begin{array}{lll} t^*_{P,m^*}[P^*[i]] & := b_P[P^*[i]] + m^*[i] & \text{for all } i \in [1, n-d], \\ t^*_{P,m^*}[P[j]] & := \texttt{'-'} & \text{for all } j \in {}^{+}R\text{-desc}(P). \end{array}$$

Let $\mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*}$ denote the set of m^*-incremented strings for P, $m^* \in \mathcal{M}^{\kappa,*}_{P,\sigma}$.

Algorithm 4.3 recursively enumerates the strings $m^* \in \mathcal{M}^{\kappa,*}_{P,\sigma}$ and the m^*-incremented strings $t^*_{P,m^*} \in \mathcal{T}^{\kappa,*}_{P,\Sigma}$ in parallel, in the same order as in Algorithm 4.2, while skipping the *invalid* sequences. Besides the sparse permutation P^*, the parameters of the algorithm are the current non-decreasing sequence m^*, the respective m^*-incremented sparse string t^*, the position i according to which the modifications are performed, the current upper bound μ for the number of distinct symbols in the prefix of the current non-decreasing sequence, and the current enumeration number *enum*. The enumeration is invoked with $\textsc{Enum}^{\kappa}_{P^*,\sigma}(minit^*, t^*_{P,minit^*}, n-(\kappa-d-1), \kappa-d, 1)$, where $minit^* = 0^{n-d-\mu}, 0, 1, 2, \ldots, \mu-1$, and $t^*_{P,minit^*}$ is the $minit^*$-incremented base string b_P. Starting with the sequence $m^* = minit^* = 0^{n-d-\mu}, 0, 1, 2, \ldots, \mu-1$, the algorithm increases $m^*[n-d-\mu+1]$ such that $m^* = 0^{n-d-\mu}, 1, 1, 2, \ldots, \mu-1$ and recursively enumerates the $(n-d-\mu)$-length

Algorithm 4.3.
ENUM$^{\kappa}_{P^*,\sigma}(m^*, t^*, i, \mu, enum)$
1: $t^*_{enum} \leftarrow t^*$
2: $m^*_{enum} \leftarrow m^*$
3: $enum \leftarrow enum + 1$
4: **if** $i > 1$ **then**
5: **for** $h \leftarrow 1$ to $\mu - 1$ **do**
6: $m^*[i+h-1] \leftarrow m^*[i+h-1] + 1$
7: $t^*[P^*[i+h-1]] \leftarrow t^*[P^*[i+h-1]] + 1$
8: ENUM$^{\kappa}_{P^*,\sigma}(m^*, t^*, i-1, h+1, enum)$
9: **end for**
10: **for** $h \leftarrow \mu - 1$ down to 1 **do**
11: $m^*[i+h-1] \leftarrow m^*[i+h-1] - 1$
12: $t^*[P^*[i+h-1]] \leftarrow t^*[P^*[i+h-1]] - 1$
13: **end for**
14: **end if**

Table 4.6: Enumeration of the sparse strings representing the strings composed of exactly the four distinct symbols A, B, C, and D sharing the suffix array $P = (6, 5, 1, 2, \textcircled{4}, 3)$ with base string $b_P =$ AABBAA.

$enum$	m^*_{enum}	t^*_{enum}
1	00012	A$^{\text{B}}$-DCAA
2	00112	B$^{\text{B}}$-DCAA
3	01112	B$^{\text{B}}$-DCBA
4	00122	B$^{\text{C}}$-DDAA
5	01122	B$^{\text{C}}$-DDBA
6	01222	C$^{\text{C}}$-DDBA

proper non-decreasing prefixes composed of the numbers $\{0, 1\}$. Then $m^*[n-d-\mu+2]$ at the position to the right is incremented such that $m^* = 0^{n-d-\mu}, 1, 2, 2, \ldots, \mu - 1$, and the proper prefixes composed of $\{0, 1, 2\}$ are recursively enumerated. The recursive enumeration is repeated for each sequence $m^* = 0^{n-d-\mu}, 1, 2, \ldots, h, h, h+1, \ldots, \mu-1$ with $1 \leq h < \mu$. Moreover, each modification operation of $m^*[i]$ is simultaneously applied to $t^*[P^*[i]]$ such that the strings in $\mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*}$ are enumerated in parallel.

We now show how $\mathcal{T}^{\kappa}_{P,\Sigma}$ derives from $\mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*}$. The characters at the blank positions of the enumerated sparse strings are implicitly defined. We construct $t_{enum} \in \mathcal{T}^{\kappa}_{P,\Sigma}$ from $t^*_{enum} \in \mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*}$ by assigning

$$t_{enum}[P[i]] = \begin{cases} t^*_{enum}[P[i]] & \text{if } i \notin {}^{+}R\text{-desc}(P) \\ t_{enum}[P[i+1]] - 1 & \text{if } i \in {}^{+}R\text{-desc}(P), \end{cases} \tag{4.1}$$

for each $i \in [1, n]$, where $t_{enum}[P[i]]$ depends on the previous assignment of $t_{enum}[P[i+1]]$ for each ${}^{+}R$-descent i. Equation (4.1) obviously defines an isomorphism between $\mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*}$ and $\mathcal{T}^{\kappa}_{P,\Sigma}$, $\mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*} \simeq \mathcal{T}^{\kappa}_{P,\Sigma}$. Hence, the enumeration of the sparse strings in $\mathcal{T}^{\kappa,*}_{P,\mathcal{M}^*}$ induces the enumeration of the strings in $\mathcal{T}^{\kappa}_{P,\Sigma}$. In this way, we implicitly enumerate all $|\mathcal{T}^{\kappa}_{P,\Sigma}|$ strings composed of exactly κ distinct characters contained in the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$ for a permutation $P \in \mathcal{P}^n_\Sigma$ in optimal $\mathcal{O}(n + |\mathcal{T}^{\kappa}_{P,\Sigma}|)$ time, where $\mathcal{O}(n)$ steps are used to construct b_P, P^*, $minit^*$, and $t^*_{P,minit^*}$.

Table 4.6 shows the enumerated non-decreasing sequences and the enumerated sparse strings over the alphabet $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ for the permutation $P = (6, 5, 1, 2, 4, 3)$ with base string $b_P =$ AABBAA. The only ${}^{+}R$-descent of P is 4, which is marked by a circle in the table caption ($P[4] = 2$). Deleting the encircled value 2 from P results in the sparse permutation $P^* = (6, 5, 1, 4, 3)$. The columns show the enumeration numbers *enum*, the

enumerated non-decreasing sequences m^*_{enum}, and the enumerated sparse strings t^*_{enum}. Moreover, the blank character $t^*_{enum}[2]$ ($P[4] = 2$) of each enumerated sparse string is annotated with the implicitly defined character $t^*_{enum}[4] - 1$ ($P[4 + 1] = 4$) forming the complete string t_{enum}, as it has been defined by equation (4.1).

5 Counting and Enumerating the Suffix Arrays for Strings with a Fixed Alphabet

In this chapter, we encounter two other classical counting problems: the counting of equivalence classes and the counting of permutations of a particular type. We count and enumerate the non-empty *sa*-equivalence classes $\mathcal{T}_{P,\Sigma}$ for a fixed-sized alphabet Σ with $P \in \mathcal{P}_\Sigma^n$ or, alternatively, the bijective set of suffix arrays for strings over that fixed-sized alphabet. We first concentrate on the equivalent problem of counting the number of suffix arrays with a fixed number of ${}^+R$-descents and then use the result to count the distinct suffix arrays for strings over a given alphabet.

Bannai *et al.* [14] stated that the number of suffix arrays of length n with exactly d ${}^+R$-descents is equal to the Eulerian number $\left\langle {n \atop d} \right\rangle$. In their explanation, they interpret Eulerian numbers as the number of permutations of length n with d permutation descents and explain how their algorithm checks for these permutation descents. In fact, their algorithm counts the number of ${}^+R$-descents, but the ${}^+R$-array is not a permutation. Nevertheless, as we show in this chapter, their proposition is true.

Theorem 5.1. *Let $A(n,d)$ be the number of permutations of length n with d ${}^+R$-descents, then*

$$A(n,d) = \left\langle {n \atop d} \right\rangle.$$

Bannai *et al.* [14] also showed that each suffix array with d ${}^+R$-descents can be associated with a string of at least $d+1$ different characters. Therefore, for strings over an alphabet of size σ, we sum up the suffix arrays with up to $\sigma - 1$ ${}^+R$-descents to obtain the number of non-empty *sa*-equivalence classes.

Corollary 5.2. *For a fixed alphabet Σ of size σ, the number $|\mathcal{P}_\Sigma^n|$ of non-empty sa-equivalence classes for permutations of length n is given by $\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle$.*

5.1 Counting suffix arrays – Proof of Theorem 5.1

Our counting or, alternatively, our enumeration scheme for suffix arrays of length n starts with the permutation (1), which is the suffix array of every string of length 1. Then it gradually extends the suffix arrays in a particular way until the maximum length n is reached.

We first have a look at the recursive definition of the Eulerian number $\left\langle {n \atop d} \right\rangle$ that denotes the number of permutations of $[1, n]$ with exactly d permutation descents. For such permutations, Graham *et al.* [56, Section 6.2] presented a counting scheme that in fact works for permutation ascents, but can be adapted for permutation descents by reading the permutations from right to left. There are n ways to insert the element n into a permutation of $[1, n-1]$ with d permutation descents, leading to n permutations of length n: $d+1$ with d permutation descents and $(n-1)-d$ with $d+1$ permutation descents. The desired recursion for the Eulerian numbers becomes evident from the reverse perspective: The $\left\langle {n \atop d} \right\rangle$ permutations of length n with d permutation descents are constructed from $(d+1)\left\langle {n-1 \atop d} \right\rangle$ permutations of length $n-1$ with d permutation descents and from $((n-1)-(d-1))\left\langle {n-1 \atop d-1} \right\rangle$ permutations of length $n-1$ with $d-1$ permutation descents, which implies $\left\langle {n \atop d} \right\rangle = (d+1)\left\langle {n-1 \atop d} \right\rangle + (n-d)\left\langle {n-1 \atop d-1} \right\rangle$ for $0 < d < n$.

Although the counting scheme of Graham *et al.* works for the permutations with a certain number of permutation descents, it does not work for the permutations with a certain number of ${}^{+}R$-descents. In general, there is a significant difference between the number of permutation descents and the number of ${}^{+}R$-descents of a permutation. An extreme case is the permutation $P = (n, n-1, \ldots, 1)$, which is the suffix array for the string $\mathtt{A}^n$. It has the maximum number of $n-1$ permutation descents, but not a single ${}^{+}R$-descent. Nevertheless, the counting scheme of Graham *et al.* and also the recursion formula for Eulerian numbers suggest a recursive counting scheme: A permutation should be extended by one element, thereby the number of ${}^{+}R$-descents should either be retained or increased by one.

Theorem 3.2 revealed a close connection between the ${}^{+}R$-array of a permutation P and the strings in the *sa*-equivalence class for P. Therefore, we do not confine ourselves to the investigation of permutations only, but rather study the modification of strings and the induced effect on the ${}^{+}R$-arrays of the affected suffix arrays instead, yielding the desired counting scheme.

The first promising modification is to append a character at the end of the string. Ukkonen [142] follows this approach for the online construction of suffix trees. This extension of the string, however, affects the relative order of the suffixes and thus inappropriately rearranges the ${}^{+}R$-array. If we start, for example, with `BCCAA` having the suffix array $(5, 4, 1, 3, 2)$ with the only ${}^{+}R$-descent at position 3 (see Tables 4.6 and 3.2) and append `D`, the resulting string `BCCAAD` has the suffix array $(4, 5, 1, 3, 2, 6)$ with ${}^{+}R$-array $(2, 6, 5, 1, 4, 0)$, which has three ${}^{+}R$-descents. The recursive formula for the Eulerian numbers, however, suggests that the number of ${}^{+}R$-descents d should not increase by more than one during a single extension step. Hence, this is apparently not the appropriate extension scheme.

A second possibility is to attach a character to the front of a string t. Let $t^{\triangleleft}$ denote such a front-extended string, $t^{\triangleleft} = c\,t$ for some character $c \in \Sigma$. We transfer the concept of the upper triangle ${}^{\triangleleft}$ to the other data structures that are affected by the front extension: If x is an instance of a data structure related to the string t, then $x^{\triangleleft}$ is an instance of the same data structure related to $t^{\triangleleft}$.

Table 5.1: The extension of the string $t =$ `ABBAA` by adding the character `A` to the front, and the effect on the suffix array and the ${}^{+}R$-array.

$t =$ `ABBAA`				$t^{\triangleleft} =$ `AABBAA`			
j	$P[j]$	${}^{+}R[j]$	$t[P[j], n]$	$j^{\triangleleft}$	$P_3^{\triangleleft}[j^{\triangleleft}]$	${}^{+}R^{\triangleleft}[j^{\triangleleft}]$	$t^{\triangleleft}[P_3^{\triangleleft}[j^{\triangleleft}], n]$
1	5	0	`A`	1	6	0	`A`
2	4	1	`AA`	2	5	1	`AA`
				3	1	4	`AABBAA`
3	1	5	`ABBAA`	4	2	6	`ABBAA`
4	3	2	`BAA`	5	4	2	`BAA`
5	2	4	`BBAA`	6	3	5	`BBAA`

Table 5.1 shows such an extension of `ABBAA` by `A`. For the string $t =$ `ABBAA`, the first four columns show the array indices i, the permutation P, the ${}^{+}R$-array, and the sorted suffixes. The remaining four columns show the respective data for the front-extended string $t^{\triangleleft} =$ `AABBAA`. The front extension of `ABBAA` by `A` shifts the existing suffixes by one position to the right, while keeping the relative order of the suffixes and the interdependencies among suffixes and their successors. Only the suffix number 1 of the new suffix `AABBAA` is inserted at the position 3 (or rather between positions 2 and 3) of the suffix array P, but the number of ${}^{+}R$-descents remains one. This is an appropriate extension scheme.

Based on our observations, we define an extension of a permutation P of length $n-1$ to a set $\mathbf{P}^{\triangleleft}$ of extended permutations, each of length n. This definition is the key for the further reasoning throughout Lemmas 5.4–5.8, ultimately leading to Theorem 5.1.

Definition 5.3. Let $P \in \mathcal{P}^{n-1}$ be a permutation of length $n-1$. A *set of extended permutations* $\mathbf{P}^{\triangleleft}$ of P is defined as $\mathbf{P}^{\triangleleft} = \{P_i^{\triangleleft} : i \in [1, n]\} \subset \mathcal{P}^n$ where the *extended permutation* $P_i^{\triangleleft}$ evolves from P by incrementing each element of P by one and inserting the missing 1 at position i, such that each index position j of P corresponds to an index position $j^{\triangleleft}$ of $P_i^{\triangleleft}$:

$$\begin{aligned} & j^{\triangleleft} := j && \text{if } j < i \\ \text{and} \quad & j^{\triangleleft} := j + 1 && \text{if } j \geq i, \end{aligned}$$

and

$$\begin{aligned} & P_i^{\triangleleft}[j^{\triangleleft}] := P[j] + 1 && \text{if } j^{\triangleleft} \neq i \\ \text{and} \quad & P_i^{\triangleleft}[j^{\triangleleft}] := 1 && \text{if } j^{\triangleleft} = i. \end{aligned}$$

$R^{\triangleleft}$ analogously denotes the rank array and ${}^{+}R^{\triangleleft}$ the ${}^{+}R$-array of an extended permutation $P^{\triangleleft}$, alternatively with an additional subscript i for an extended permutation with insertion position i.

The insertion at position i shifts the elements at positions j with $j \geq i$ to the right, resulting in an increased rank for the respective elements of $P_i^{\lhd}$. In this way, the insertion position i determines the rank array of the extended permutation.

Lemma 5.4. *Let $P \in \mathcal{P}^{n-1}$ be a permutation of length $n-1$ and $P^{\lhd} \in \mathbf{P}^{\lhd}$ an extended permutation with insertion position i. Then we have for all $e \in [1, n-1]$ that*

(a) $R^{\lhd}[e+1] = R[e]$ *if* $R[e] < i$,

(b) $R^{\lhd}[e+1] = R[e]+1$ *if* $R[e] \geq i$, *and*

(c) $R^{\lhd}[1] = i$.

Proof. Let e be an arbitrary element of the permutation P occurring at position j, $e = P[j]$ and $R[e] = j$.

(a) If $R[e] < i$, then $j = R[e] < i$. Therefore, according to Definition 5.3, $j^{\lhd}$ equals j and hence $P^{\lhd}[j^{\lhd}] = P[j]+1 = e+1$. Altogether, this implies $R^{\lhd}[e+1] = R^{\lhd}[P^{\lhd}[j^{\lhd}]] = j^{\lhd} = j = R[e]$.

(b) If $R[e] \geq i$, then $j = R[e] \geq i$. Therefore, $j^{\lhd} = j+1$ and $P^{\lhd}[j^{\lhd}] = P[j]+1 = e+1$. This implies $R^{\lhd}[e+1] = R^{\lhd}[P^{\lhd}[j^{\lhd}]] = j^{\lhd} = j+1 = R[e]+1$.

(c) $R^{\lhd}[1] = i$ holds because 1 is inserted at position i, $P^{\lhd}[i] = 1$. □

Furthermore, mapping P to $P^{\lhd}$ basically preserves the ${}^{+}R$-order, except for the insertion position i:

Lemma 5.5. *Let $P \in \mathcal{P}^{n-1}$ be a permutation of length $n-1$ and $P^{\lhd} \in \mathbf{P}^{\lhd}$ an extended permutation. Then, for all indices $g, h \in [1, n-1]$,*

$$ {}^{+}R[g] < {}^{+}R[h] \Longrightarrow {}^{+}R^{\lhd}[g^{\lhd}] < {}^{+}R^{\lhd}[h^{\lhd}]. $$

Proof. Let g and h be two positions of P such that ${}^{+}R[g] < {}^{+}R[h]$. Then, according to the definition of ${}^{+}R$, $R[P[g]+1] < R[P[h]+1]$. Moreover, let i be the insertion position of $P^{\lhd}$. We distinguish two cases.

(i) If $R[P[g]+1] < i$, then Lemma 5.4 (a and b) gives

$$ R^{\lhd}[P[g]+1+1] = R[P[g]+1] < R[P[h]+1] \leq R^{\lhd}[P[h]+1+1]. $$

Combining this with Definition 5.3 and the definition of ${}^{+}R^{\lhd}$ yields

$$ {}^{+}R^{\lhd}[g^{\lhd}] = R^{\lhd}[P^{\lhd}[g^{\lhd}]+1] < R^{\lhd}[P^{\lhd}[h^{\lhd}]+1] = {}^{+}R^{\lhd}[h^{\lhd}]. $$

(ii) If $R[P[g]+1] \geq i$ the proof works analogously using the fact that $R[P[h]+1] > R[P[g]+1] \geq i$. Hence, Lemma 5.4(b) has to be used for $R[P[g]+1]$ as well as for $R[P[h]+1]$, and then the rest of the proof proceeds as before. □

Lemma 5.5 considers the ${}^{+}R$-order of $P^{\lhd}$, but leaves out the insertion position i. The next lemma states that the ${}^{+}R$-order at position i just depends on the position $R[1]$ of element 1 in the permutation P.

Lemma 5.6. *Let $P^\lhd \in \mathbf{P}^\lhd$ be an extended permutation of $P \in \mathcal{P}^{n-1}$ with insertion position $i \in [1,n]$, and let g be an index of P, then*

$$ {}^+R[g] < R[1] \iff {}^+R^\lhd[g^\lhd] < {}^+R^\lhd[i] \qquad \text{for all } g \in [1, n-1]. $$

Proof. We first show that ${}^+R[g] < R[1] \implies {}^+R^\lhd[g^\lhd] < {}^+R^\lhd[i]$.

If ${}^+R[g] < R[1]$, then using the definition of ${}^+R$ leads to $R[P[g]+1] < R[1]$. We consider two cases.

(i) If $R[P[g]+1] < i$, then $R^\lhd[P[g]+1+1] = R[P[g]+1]$ by Lemma 5.4(a). Moreover, Lemma 5.4 (a and b) implies $R[1] \leq R^\lhd[1+1]$. This together leads to

$$ R^\lhd[(P[g]+1)+1] < R^\lhd[1+1]. \tag{5.1} $$

According to Definition 5.3, $P^\lhd[g^\lhd] = P[g]+1$ and $P^\lhd[i] = 1$. Combining this with inequality (5.1) leads to

$$ {}^+R^\lhd[g^\lhd] = R^\lhd[P^\lhd[g^\lhd]+1] = R^\lhd[(P[g]+1)+1] < R^\lhd[1+1] = R^\lhd[P^\lhd[i]+1] = {}^+R^\lhd[i]. $$

(ii) If $R[P[g]+1] \geq i$, then the proof proceeds analogously by considering $R[1] > R[P[g]+1] \geq i$.

In order to show the opposite direction ${}^+R[g] < R[1] \impliedby {}^+R^\lhd[g^\lhd] < {}^+R^\lhd[i]$, we observe that ${}^+R[g] > R[1] \implies {}^+R^\lhd[g^\lhd] > {}^+R^\lhd[i]$. Since, for all $g \in [1, n-1]$, ${}^+R[g] \neq R[1]$ and ${}^+R^\lhd[g^\lhd] \neq {}^+R^\lhd[i]$, we obtain the stated equivalence. □

After characterising the ${}^+R$-order of extended permutations, we now prove that the number of ${}^+R$-descents is either preserved or increased by exactly one through the mapping from P to an arbitrary extended permutation $P^\lhd$.

Lemma 5.7. *Let $P \in \mathcal{P}^{n-1}$ be a permutation of length $n-1$ with d ${}^+R$-descents and $\mathbf{P}^\lhd$ the set of extended permutations of P, then we have, for all extended permutations $P_i^\lhd \in \mathbf{P}^\lhd$,*

$$ |desc(P)| \leq |desc(P_i^\lhd)| \leq |desc(P)| + 1. $$

Proof. According to Lemma 5.5, the mapping with respect to the insertion position i does not touch the ${}^+R$-order of consecutive positions not adjacent to i. More precisely, for all $j \in [2, n-1]$ with $j \neq i$,

$$ {}^+R[j-1] > {}^+R[j] \iff {}^+R_i^\lhd[(j-1)^\lhd] > {}^+R_i^\lhd[j^\lhd]. $$

This means that each ${}^+R$-descent at position $j-1$ with $j \neq i$ corresponds to a ${}^+R$-descent at position $(j-1)^\lhd$ in $P_i^\lhd$ and vice versa. Therefore, we only have to examine the ${}^+R$-order of the remaining pair of positions $(i-1, i)$ in P and the respective interval $[(i-1)^\lhd, i^\lhd]$ in $P_i^\lhd$. Note that $[(i-1)^\lhd, i^\lhd] = \{i-1, i, i+1\}$. We distinguish whether position $i-1$ of P is a ${}^+R$-descent or not.

(i) If $i-1$ is a ${}^+R$-descent of P such that ${}^+R[i-1] > {}^+R[i]$, then applying Lemma 5.5 leads to

$$ {}^+R_i^{\triangleleft}[(i-1)^{\triangleleft}] > {}^+R_i^{\triangleleft}[i^{\triangleleft}]. \tag{5.2} $$

Since $R[1] \neq {}^+R[g]$ for all $g \in [1, n-1]$, we consider three subcases:

(i.1) If $R[1] > {}^+R[i-1]$, then Lemma 5.6 implies ${}^+R_i^{\triangleleft}[i] > {}^+R_i^{\triangleleft}[(i-1)^{\triangleleft}]$ and together with inequality (5.2) ${}^+R_i^{\triangleleft}[i] > {}^+R_i^{\triangleleft}[(i-1)^{\triangleleft}] > {}^+R_i^{\triangleleft}[i^{\triangleleft}]$ follows. That is, ${}^+R_i^{\triangleleft}[i-1] = {}^+R_i^{\triangleleft}[(i-1)^{\triangleleft}] < {}^+R_i^{\triangleleft}[i]$ and ${}^+R_i^{\triangleleft}[i] > {}^+R_i^{\triangleleft}[i^{\triangleleft}] = {}^+R_i^{\triangleleft}[i+1]$. Hence, i is a ${}^+R$-descent of $P_i^{\triangleleft}$ and the number of ${}^+R$-descents of $P_i^{\triangleleft}$ equals the number of ${}^+R$-descents of P.

(i.2) If ${}^+R[i-1] > R[1] > {}^+R[i]$, then Lemma 5.6 implies ${}^+R_i^{\triangleleft}[(i-1)^{\triangleleft}] > {}^+R_i^{\triangleleft}[i] > {}^+R_i^{\triangleleft}[i^{\triangleleft}]$. Hence, $(i-1)^{\triangleleft}$ and i are ${}^+R$-descents of $P_i^{\triangleleft}$. The number of ${}^+R$-descents in $P_i^{\triangleleft}$ is thus one more than in P.

(i.3) If ${}^+R[i] > R[1]$, then ${}^+R_i^{\triangleleft}[(i-1)^{\triangleleft}] > {}^+R_i^{\triangleleft}[i] < {}^+R_i^{\triangleleft}[i^{\triangleleft}]$. Hence, the number of ${}^+R$-descents in $P_i^{\triangleleft}$ equals the number of ${}^+R$-descents in P.

(ii) If $i-1$ is not a ${}^+R$-descent of P, then an argument similar to (i) can be used to show that the number of ${}^+R$-descents is retained or increases by one.

Combining all these cases shows, for each i, that the number of ${}^+R$-descents is preserved by the mapping from P to $P_i^{\triangleleft}$ or is increased by one. □

Lemma 5.8. *Let P be a permutation of length $n-1$ with d ${}^+R$-descents and $\mathbf{P}^{\triangleleft}$ the set of extended permutations of P; then the number of extended permutations with d ${}^+R$-descents is $d+1$,*

$$ |\{P^{\triangleleft} \in \mathbf{P}^{\triangleleft} \,:\, |desc(P^{\triangleleft})| = d\}| = d+1. $$

Proof. We assign to each ${}^+R$-run ${}^+R[l, r]$ of P a *proper insertion position* $i \in [l, r+1]$ that preserves the number of ${}^+R$-descents through the mapping from P to $P_i^{\triangleleft}$ and show that the number of ${}^+R$-descents increases for the other, non-proper insertion positions.

Let ${}^+R[l, r]$ be a ${}^+R$-run defined by a pair of consecutive ${}^+R$-descents, $(l-1, r)$, such that ${}^+R[l-1] > {}^+R[l] < {}^+R[l+1] < \ldots < {}^+R[r] > {}^+R[r+1]$. Remember that, according to Lemma 5.5, the ${}^+R$-descents not adjacent to the insertion position are preserved through the mapping to $P_i^{\triangleleft}$. Therefore, it suffices to investigate the ${}^+R$-order of positions touched by the insertion. Since $R[1] \neq {}^+R[g]$ for all $g \in [1, n-1]$, we consider three mutually exclusive cases.

(i) For $R[1] < {}^+R[l]$, the proper insertion position is i, $i = l$, such that

$$ {}^+R[l-1] > R[1] < {}^+R[l] < \ldots < {}^+R[r] > {}^+R[r+1]. $$

According to Lemmas 5.5 and 5.6, we obtain the series of inequalities

$$ {}^+R_i^{\lhd}[(l-1)^{\lhd}] > {}^+R_i^{\lhd}[i] < {}^+R_i^{\lhd}[l^{\lhd}] < \ldots < {}^+R_i^{\lhd}[r^{\lhd}] > {}^+R_i^{\lhd}[(r+1)^{\lhd}]. $$

Hence, for the insertion position l, there exist exactly as many ${}^+R$-descents in the respective interval $[l-1, r+1]$ of P as in the interval $[(l-1)^{\lhd}, (r+1)^{\lhd}]$ of $P_i^{\lhd}$, and, according to Lemma 5.5, the other ${}^+R$-descents are not affected through the mapping. Thus, $|\,{}^+R\text{-desc}(P)| = |\,{}^+R\text{-desc}(P_i^{\lhd})|$.

For the insertion positions $i \in [l+1, r]$,

$$ {}^+R[l] < {}^+R[l+1] < \ldots < {}^+R[i-1] > R[1] < {}^+R[i] < \ldots < {}^+R[r] \tag{5.3} $$

holds. Then applying Lemmas 5.5 and 5.6 leads to

$$ {}^+R_i^{\lhd}[l^{\lhd}] < {}^+R_i^{\lhd}[(l+1)^{\lhd}] < \ldots < {}^+R_i^{\lhd}[(i-1)^{\lhd}] > {}^+R_i^{\lhd}[i] < {}^+R_i^{\lhd}[i^{\lhd}] < \ldots < {}^+R_i^{\lhd}[r^{\lhd}]. \tag{5.4} $$

Therefore, the number of ${}^+R$-descents increases through the mapping.

The bordering insertion position $r+1$ remains to be investigated, for which we consider two special cases.

(i.1) If $R[1] < {}^+R[r+1]$, then $r+1$ would be the proper insertion position for the next ${}^+R$-run ${}^+R[r+1, h]$ for some h, like in case (i).

(i.2) If $R[1] > {}^+R[r+1]$, then the insertion position $r+1$ increases the number of ${}^+R$-descents through the mapping from P to $P_i^{\lhd}$.

(ii) For ${}^+R[l] < R[1] < {}^+R[r]$, the proper insertion position is $i \in [l+1, r]$ with ${}^+R[i-1] < R[1] < {}^+R[i]$. The other insertion positions j, $j \in [l+1, r]$ with $j \neq i$, increase the number of ${}^+R$-descents. The bordering insertion positions l and $r+1$ either increase the number of ${}^+R$-descents analogously to (i.2), or they are proper insertion positions for the adjacent ${}^+R$-runs.

(iii) For ${}^+R[r] < R[1]$, the proof works analogously to (i) by handling the bordering insertion position l like (i.2).

So far, we concentrated on the inner ${}^+R$-runs ${}^+R[l, r]$ with $l \neq 1$ and $r \neq n-1$. For the bordering ${}^+R$-runs ${}^+R[l, r]$ with $l = 1$ or $r = n-1$, the proper insertion positions are defined in the same way, but the proof is a bit simpler because the insertion positions at the borders 1 and n are both not affected by adjacent ${}^+R$-runs.

Finally, for each of the $d+1$ ${}^+R$-runs in P, there exists a unique insertion position i that preserves the number of ${}^+R$-descents through the mapping from P to $P_i^{\lhd}$. All other insertion positions increase the number of ${}^+R$-descents. □

Proof of Theorem 5.1. For the number of permutations of length n having d ${}^{+}R$-descents, $A(n,d)$, we achieve the following recursive definition with the two base cases (i) and (ii) and the recursion step (iii).

(i) Since the permutation $(n, n-1, \ldots, 1)$ is the only one without any ${}^{+}R$-descent, $A(n,0) = 1$.

(ii) Obviously, the number of ${}^{+}R$-descents is bounded by $n-1$. Hence, there is no permutation of length n with more than $n-1$ ${}^{+}R$-descents, and thus $A(n,d) = 0$ for $d \geq n$.

(iii) As mentioned before, mapping each permutation P of length $n-1$ to $P_i^{\triangleleft}$ leads to n extended permutations, each of length n (one for each possible insertion position i). If P contains d ${}^{+}R$-descents, then Lemma 5.8 implies: There exist exactly $d+1$ extended permutations with d ${}^{+}R$-descents, and, according to Lemma 5.7, the other $n-d$ extended permutations contain $d+1$ ${}^{+}R$-descents. Combining these observations leads to the recursion $A(n,d) = (d+1)A(n-1,d)+(n-d)A(n-1,d-1)$ for $0 < d < n$.

The propositions (i), (ii), and (iii) yield the same recursion as for the Eulerian numbers. Hence, $A(n,d) = \left\langle {n \atop d} \right\rangle$. □

5.2 Enumerating the suffix arrays

We present the first enumeration algorithm for the suffix arrays of the strings *up to* length n over an alphabet of size σ or, alternatively, for the corresponding non-empty *sa*-equivalence classes represented by their base strings. Our enumeration scheme exploits the close relationship between suffix arrays and the Burrows–Wheeler transform. We would like to enumerate only the suffix arrays of (exactly) length n, just as Corollary 5.2 counts them, but we are currently not able to do so. Our enumeration scheme generates the suffix arrays from small to long arrays such that the generation of the suffix arrays of length n depends on the previous generation of all shorter suffix arrays.

We first observe that the attachment of a character at the front of a string causes an index shift of the starting positions of the suffixes: Each index number increases by one, and the newly attached character receives the freed index number 1. For our enumeration algorithm of the base strings up to length n, which also uses such a front extension, we avoid the unfavourable index shift by using a different indexing of the strings: For a string t' of length n' with $n' \leq n$, we use the indexing $n-n'+1, n-n'+2, \ldots, n$. If a new character is attached to the front of t', then it is assigned to the new front index $n-n'$ without increasing the previously existing index numbers $n-n'+1, n-n'+2, \ldots, n$. A more elegant solution would be to replace the left-to-right indexing with a right-to-left indexing $n', \ldots, 2, 1$, which is independent of the final string length. Nevertheless, to be consistent with the literature, we keep the traditional left-to-right indexing throughout the thesis, but start with the index $\mathit{front} = n-n'+1$ in the remainder of this chapter.

The modified indexing is only used for strings. Nevertheless, it requires an adjustment of Definition 5.3. First of all, the indexing of the suffix arrays is not changed. Hence, we still have

$$\begin{aligned} & j^{\triangleleft} := j && \text{if } j < i \\ \text{and} \quad & j^{\triangleleft} := j + 1 && \text{if } j \geq i. \end{aligned}$$

The modified indexing of the strings, however, avoids the shift of the suffix numbers. Therefore,

$$\begin{aligned} & P_i^{\triangleleft}[j^{\triangleleft}] := P[j] && \text{if } j^{\triangleleft} \neq i \\ \text{and} \quad & P_i^{\triangleleft}[j^{\triangleleft}] := \mathit{front}^{\triangleleft} && \text{if } j^{\triangleleft} = i, \end{aligned}$$

where $\mathit{front}^{\triangleleft} := \mathit{front} - 1$.

Furthermore, the proposition of Lemma 5.4 changes; we now have for all $e \in [1, n-1]$ that

(a) $R^{\triangleleft}[e] = R[e]$ if $R[e] < i$,

(b) $R^{\triangleleft}[e] = R[e] + 1$ if $R[e] \geq i$, and

(c) $R^{\triangleleft}[\mathit{front}^{\triangleleft}] = i$.

Nevertheless, the ${}^{+}R$-array is essentially not altered by the different indexing since it reflects the connections between consecutive suffixes, which is independent of the current indexing; only the start index of the string changes from 1 to *front*. Therefore, Lemmas 5.5–5.8 are essentially retained, only $R[1]$ in Lemma 5.6 changes to $R[\mathit{front}]$.

Before we can formulate the enumeration algorithm, we first define the Burrows–Wheeler transform (BWT) and further terms that are frequently used in the compressed indexing literature. Let \$ be a character not contained in Σ with $\$ < c$ for all $c \in \Sigma$. For applying the BWT, we append \$ to the end of t, forming the \$-extended string $t\$$. The suffix array P of t is essentially kept through the extension. Only the new suffix number $n+1$, which refers to the smallest suffix \$, is implicitly attached to the front of P, $P[0] = n+1$, but it does not explicitly appear in P. The BWT string *bwt* of t, or rather the BWT string of the \$-extended string $t\$$, is formed of the characters to the "left" of the suffix numbers in their suffix array order, basically giving the left context of the lexicographically sorted suffixes of $t\$$.

Definition 5.9. Let Σ be the underlying alphabet, $P \in \mathcal{P}_{\Sigma}^{n}$ a permutation of $[1, n]$, and $t \in \mathcal{T}_{P,\Sigma}$ a string of the respective *sa*-equivalence class. Moreover, let $P[0] = n+1$. We define the *BWT string bwt* of t as

$$\mathit{bwt}[i] := \begin{cases} t[P[i]-1] & \text{if } P[i] > 1 \\ \text{'\$'} & \text{if } P[i] = 1, \end{cases}$$

for $i \in [0, n]$.

Note that, different from the string t, the BWT string includes the \$, starts at position 0 and has length $n+1$. Moreover, our definition is only equivalent to the original definition of Burrows and Wheeler [32] for \$-extended strings.

We further define some tools that are frequently used in the compressed text indexing literature, starting with the functions rank and select. For the BWT string bwt, $\text{rank}_c(bwt, j)$ is the number of occurrences of the character c in the prefix $bwt[0, j]$ of bwt:

$$\text{rank}_c(bwt, j) := |\{g \in [0, j] \,:\, bwt[g] = c\}| \text{ for all } j \in [0, n]. \tag{5.5}$$

Conversely, $\text{select}_c(bwt, k)$ gives the position of the k^{th} occurrence of the character c in bwt:

$$\text{select}_c(bwt, k) := j \quad \text{if } bwt[j] = c \text{ and } \text{rank}_c(bwt, j) = k, \tag{5.6}$$

for all $c \in \Sigma$, $j \in [0, n]$, and $k \in [1, n]$; $\text{select}_c(bwt, k)$ is undefined if the number of occurrences of the character c in bwt is less than k.

Recall the First sequence $f = t[P[1]], t[P[2]], \ldots, t[P[n]]$ for a string $t \in \mathcal{T}_{P,\Sigma}$, which is simply composed of the alphabetically ordered characters of t. Without loss of generality, we assume that the underlying alphabet consists of the first natural numbers, $\Sigma = [1, |\sigma|]$. Then we define the array C storing in $C[c]$ the frequency of characters in t that are smaller than c, $C[c] := |\{j \in [1, n] \,:\, t[j] < c\}|$ for all $c \in \Sigma$. Moreover, $f[C[c] + 1] = f[C[c] + 2] = \ldots = f[C[c + 1]]$ for all $c \in \Sigma$. Hence, C uniquely determines the First sequence f.

For a string t with BWT string bwt and First sequence f, the LF-mapping links each positions of bwt to a position of f:

$$LF(j) := \begin{cases} C[bwt[j]] + \text{rank}_{bwt[j]}(bwt, j) & \text{if } bwt[j] \neq \text{'\$'} \\ 0 & \text{if } bwt[j] = \text{'\$'}, \end{cases}$$

for all $j \in [0, n]$. If $bwt[j] = c$ is the k^{th} occurrence of the character c in bwt, then $f[LF(i)] = c$ is the k^{th} occurrence of c in f. The inverse mapping LF^{-1} is realised via a select query:

$$LF^{-1}(h) = \text{select}_{f[h]}(bwt, j - C[h]) \text{ for all } h \in [1, n].$$

Additionally, we maintain a reference $p_\$$ to the position of \$ in bwt such that $bwt[p_\$] = \$$.

There exists a one-to-one correspondence between the suffix arrays with d ${}^+R$-descents and the base strings of the respective sa-equivalence classes, which are composed of exactly $d + 1$ distinct characters (see Chapter 4.1). For the proper insertion position 3, Table 5.2 shows the extension of the permutation $P = (6, 5, 2, 4, 3)$, the respective front extension of the \$-extended base string `ABBAA$` by **A** and the adjustment of the BWT string bwt_P. The symbol '`_`' is a sentinel for the index position 1, which does not belong to the string. The real start index is $front = 2$. The leftmost five columns of the table show the array indices j, the suffix array P, the ${}^+R$-array, the BWT string bwt_P, and the First sequence for the \$-extended base string $b_P\$$, $b_P\$ =$ `_ABBAA$`. The right part shows the respective columns for the extended permutation $P_3^{\triangleleft}$ with the \$-extended base string $b_{P_3^{\triangleleft}}\$ =$ **A**`ABBAA$`. The lines between the BWT column and the column for the First sequence represent the LF-mapping for the `A`s. If **A** is attached to the front of `ABBAA$`, then we find the proper

Table 5.2: The extension of the base string $b_P = \texttt{ABBAA}$ by adding the character `A` to the front and the effect on the suffix array and the ${}^{+}R$-array.

$b_P\$ = \texttt{\_ABBAA\$}$					$b_{P_3^{\triangleleft}}\$ = \mathbf{A}\texttt{ABBAA\$}$				
j	$P[j]$	${}^{+}R[j]$	$bwt_P[j]$	$b_P\$[P[j]]$	$j^{\triangleleft}$	$P_3^{\triangleleft}[j^{\triangleleft}]$	${}^{+}R_3^{\triangleleft}[j^{\triangleleft}]$	$bwt_{P_3^{\triangleleft}}[j^{\triangleleft}]$	$b_{P_3^{\triangleleft}}\$[P_3^{\triangleleft}[j^{\triangleleft}]]$
0			A	$	0			A	$
1	6	0	A	A	1	6	0	A	A
2	5	1	B	A	2	5	1	B	A
					3	**1**	**4**	**$**	**A**
3	2	5	$	A	4	2	6	A	A
4	4	2	B	B	5	4	2	B	B
5	3	4	A	B	6	3	5	A	B

Table 5.3: The extension of the permutation P with base string $b_P = \texttt{ABBAA}$ according to the insertion positions 4 and 2.

$b_{P_4^{\triangleleft}}\$ = \mathbf{B}\texttt{ACCAA\$}$				$b_{P_2^{\triangleleft}}\$ = \mathbf{A}\texttt{BCCBA\$}$				
$j^{\triangleleft}$	$P_4^{\triangleleft}[j^{\triangleleft}]$	$bwt_{P_4^{\triangleleft}}[j^{\triangleleft}]$	$b_{P_4^{\triangleleft}}\$[P_4^{\triangleleft}[j^{\triangleleft}]]$	$j^{\triangleleft}$	$P_2^{\triangleleft}[j^{\triangleleft}]$	$bwt_{P_2^{\triangleleft}}[j^{\triangleleft}]$	$b_{P_2^{\triangleleft}}\$[P_2^{\triangleleft}[j^{\triangleleft}]]$
0		A	$	0		A	$
1	6	A	A	1	6	B	A
2	5	C	A	**2**	**1**	**$**	**A**
3	2	B	A	3	5	C	B
4	**1**	**$**	**B**	4	2	A	B
5	4	C	C	5	4	C	C
6	3	A	C	6	3	B	C

insertion position 3 by moving the \$ at position 3 of bwt_P towards the funnel that is formed by the lines representing the LF-mapping for the `A`s. The \$ in bwt_P is then replaced by the attached **A**, **A** is inserted at position 3 of the first sequence, and the \$ to the "left" of the attached **A** is inserted at position 3 of bwt_P. The other positions of the BWT string and the First sequence remain untouched. Moreover, the new suffix with the suffix number $front^{\triangleleft} = 1$ is inserted at position 3 of the suffix array P. The inserted row 3 is printed in bold face.

For the insertion positions 4 and 2, Table 5.3 shows the respective extended permutations of $P = (6, 5, 2, 4, 3)$, the modifications of the \$-extended base string `_ABBAA$` of the respective *sa*-equivalence class and the adjustment of the Burrows–Wheeler transform; 4 and 2 are non-proper insertion positions. For the insertion position 4, the leftmost four

columns show the array index $j^{\lhd}$, the extended suffix array $P_4^{\lhd}$, the respective BWT string $bwt_{P_4^{\lhd}}$, and the First sequence for the modified \$-extended base string $b_{P_4^{\lhd}}\$ = \mathbf{BACCAA}\$$. The rightmost four columns show the respective data for the insertion position 2. The values of the inserted rows are again printed in bold face. The solid lines show the part of the LF-mapping touching the insertion position i of the First sequence, and the dashed lines show the part of the LF-mapping touching the First sequence at the positions $j^{\lhd}$ with $j^{\lhd} > i$. We observe that the characters after the insertion position i of the First sequence are increased by one, $b_{P_i^{\lhd}}[j^{\lhd}] = b_P[j^{\lhd} - 1] + 1$ for each $j^{\lhd} > i$.

Based on our observations, we define the modification of the base string b_P of the *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$.

Definition 5.10. Let Σ be the underlying alphabet, $P \in \mathcal{P}_\Sigma^{n-1}$ a permutation of length $n-1$, b_P the base string of the respective *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$, and $Prop_P$ the set of proper insertion positions for P. Moreover, let i be a non-proper insertion position, $i \in [1,n]$ with $i \notin Prop_P$. Then we define the *modified base string* $b_{i,P}$ of length n by

(a) $b_{i,P}[P[j^{\lhd}]] := b_P[P[j]]$ if $j^{\lhd} < i$,

(b) $b_{i,P}[P[j^{\lhd}]] := b_P[P[j]] + 1$ if $j^{\lhd} > i$, and

(c) $b_{i,P}[front^{\lhd}] := prop_i + 1$,

where *front* is the start index of the base string b_P and $prop_i$ denotes the number of proper insertion positions in the prefix $P[1, i-1]$ of P, $prop_i = |\{j \in Prop_P : j < i\}|$.

Lemma 5.11. *Let Σ be the underlying alphabet, $P \in \mathcal{P}_\Sigma^{n-1}$ a permutation of length $n-1$, b_P the base string of the respective sa-equivalence class $\mathcal{T}_{P,\Sigma}$, and i a non-proper insertion position of P, $i \in [1,n]$ with $i \notin Prop_P$. Then $b_{i,P}$ is the base string of the sa-equivalence class $\mathcal{T}_{P_i^{\lhd},\Sigma}$ according to the extended permutation $P_i^{\lhd}$, $b_{i,P} = b_{P_i^{\lhd}}$.*

Proof. Lemmas 5.5 and 5.6 imply that the extension with respect to the insertion position i only influences the relative order of the ${}^+R$-values touched by the insertion position. Since i is a non-proper insertion position, the extension of P either produces a new ${}^+R$-descent at position $i-1$ with ${}^+R_i^{\lhd}[i-1] > {}^+R_i^{\lhd}[i]$ or a new ${}^+R$-descent at position i with ${}^+R_i^{\lhd}[i] > {}^+R_i^{\lhd}[i+1]$, implying the following. If $d_{j^{\lhd}}^{\lhd}$ is the number of ${}^+R$-descents in $P_i^{\lhd}$ that are smaller than $j^{\lhd}$ and d_j is the number of ${}^+R$-descents in P that are smaller than j, then we have $d_j = d_{j^{\lhd}}^{\lhd}$ for $j^{\lhd} < i$ and $d_j + 1 = d_{j^{\lhd}}^{\lhd}$ for $j^{\lhd} > i$. Hence, according to Definition 5.10 (a and b) and the definition of the base strings, we have $b_{i,P}[P[j^{\lhd}]] = b_P[P[j]] = d_j + 1 = d_{j^{\lhd}}^{\lhd} + 1 = b_{P_i^{\lhd}}[P_i^{\lhd}[j^{\lhd}]]$ for $j^{\lhd} < i$ and $b_{i,P}[P[j^{\lhd}]] = b_P[P[j]] + 1 = (d_j + 1) + 1 = d_{j^{\lhd}}^{\lhd} + 1 = b_{P_i^{\lhd}}[P_i^{\lhd}[j^{\lhd}]]$ for $j^{\lhd} > i$, verifying the equality for the positions $j^{\lhd} \in [1,n]$ with $j^{\lhd} \neq i$.

For Definition 5.10(c), we exploit the relationship between the ${}^+R$-runs and the proper insertion positions. Let ${}^+R[l,r]$ be the ${}^+R$-run with $l \leq i \leq r$, and assume it is the k^{th} ${}^+R$-run, so $d_i = k-1$. Moreover, in the proof of Lemma 5.8, we have assigned the k^{th} proper insertion position i_{prop} to the k^{th} ${}^+R$-run ${}^+R[l,r]$, $l \leq i_{prop} \leq r+1$ and $k = |\{i \in Prop_P : i \leq i_{prop}\}|$. We distinguish two cases:

(i) If $i < i_{prop}$, then

$$\ldots < {}^{+}R[i-1] < R[\mathit{front}] > {}^{+}R[i] < \ldots < {}^{+}R[i_{prop}-1] < R[\mathit{front}] \ldots,$$

where we have ${}^{+}R[i_{prop}-1] < R[\mathit{front}] < {}^{+}R[i_{prop}]$ since i_{prop} is a proper insertion position (see the series of inequalities (5.3) and (5.4) in the proof of Lemma 5.8). Then applying Lemmas 5.5 and 5.6 leads to

$$\ldots < {}^{+}R_i^{\lhd}[(i-1)^{\lhd}] < {}^{+}R_i^{\lhd}[i] > {}^{+}R_i^{\lhd}[i^{\lhd}] < \ldots$$

Hence, the insertion at position i produces a new ${}^{+}R$-descent i. We have $k-1$ proper insertion positions of P smaller than i and as many ${}^{+}R$-descents of $P_i^{\lhd}$ smaller than i, $\mathit{prop}_i = d_i^{\lhd}$. Moreover, the new suffix number $\mathit{front}^{\lhd}$ is inserted at position i such that $P_i^{\lhd}[i] = \mathit{front}^{\lhd}$. Therefore, according to Definition 5.10(c) and the definition of the base strings, $b_{i,P}[P_i^{\lhd}[i]] = b_{i,P}[\mathit{front}^{\lhd}] = \mathit{prop}_i + 1 = d_i^{\lhd} + 1 = b_{P_i^{\lhd}}[P_i^{\lhd}[i]]$, verifying the equality for the insertion position i.

(ii) If $i > i_{prop}$, then we have

$$R[\mathit{front}] < {}^{+}R[i_{prop}] < \ldots\ldots < {}^{+}R[i-1] > R[front] < {}^{+}R[i] < \ldots$$

Applying Lemmas 5.5 and 5.6 again leads to

$$\ldots < {}^{+}R_i^{\lhd}[(i-1)^{\lhd}] > {}^{+}R_i^{\lhd}[i] < {}^{+}R_i^{\lhd}[i^{\lhd}] < \ldots$$

Hence, the insertion at position i produces a new ${}^{+}R$-descent at position $i-1$. We have k proper insertion positions that are smaller than or equal to i: $k-1$ for the preceding ${}^{+}R$-runs and in addition the proper insertion position i_{prop}. Moreover, we have the same number k of ${}^{+}R$-descents of $P_i^{\lhd}$ that are smaller than i: We have the $k-1$ preceding ${}^{+}R$-runs each terminated by a ${}^{+}R$-descent and in addition the ${}^{+}R$-descent $i-1$ that is produced by the insertion. That is, $\mathit{prop}_i = d_i^{\lhd}$. Hence, according to Definition 5.10 (c) and the definition of base strings, $b_{i,P}[P_i^{\lhd}[i]] = b_{i,P}[\mathit{front}^{\lhd}] = \mathit{prop}_i + 1 = d_i^{\lhd} + 1 = b_{P_i^{\lhd}}[P_i^{\lhd}[i]]$, verifying the equality for the insertion position i. □

We are now prepared to formulate the desired enumeration algorithm. The main procedure EnumSa (Algorithm 5.1) interacts with the procedures EnumProp (Algorithm 5.2), EnumNoProp (Algorithm 5.3), and InsRecDel (Algorithm 5.4). Let $\mathcal{P}_\sigma^{[1,n]}$ be the set of suffix arrays of strings composed of up to σ distinct characters with length up to n. EnumSa simultaneously enumerates the base strings up to length n that are composed of up to σ distinct characters and the corresponding suffix arrays $P \in \mathcal{P}_\sigma^{[1,n]}$. It starts with the suffix array (1) of the base string `A` and gradually extends the suffix arrays $P \in \mathcal{P}_\sigma^{[1,n]}$ emanating from (1) until the maximum length n is reached.

Without loss of generality, we assume again that the character set of a base string b_P equals the first natural numbers $[1, |\Sigma(b_P)|]$. In each step, the BWT string is adjusted to the current base string. The parameters of the algorithm are the current permutation P,

Algorithm 5.1.
$\textsc{EnumSa}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum)$

1: $P_{enum} \leftarrow P$
2: $b_{enum} \leftarrow b_P$
3: $enum \leftarrow enum + 1$
4: **if** $\text{length}(b_P) < n$ **then**
5: $\quad Prop_P \leftarrow \textsc{EnumProp}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum)$
6: $\quad$ **if** $|\Sigma(b_P)| < \sigma$ **then**
7: $\quad\quad \textsc{EnumNoProp}_{n,\sigma}(P, b_P, bwt_P, p_\$, Prop_P, enum)$
8: $\quad$ **end if**
9: **end if**

Algorithm 5.2.
$\textsc{EnumProp}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum)$

1: $Prop_P \leftarrow \emptyset$
2: **for** $c \leftarrow |\Sigma(b_P)|$ down to 1 **do**
3: $\quad i \leftarrow C(c) + \text{rank}_c(bwt_P, p_\$ - 1) + 1$
4: $\quad Prop_P \leftarrow Prop_P \cup \{i\}$
5: $\quad \textsc{InsRecDel}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum, i, c)$
6: **end for**
7: **return** $Prop_P$

Algorithm 5.3.
$\textsc{EnumNoProp}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum, Prop_P)$

1: $c \leftarrow |\Sigma(b_P)| + 1$
2: **if** $\text{length}(b_P) + 1 \in Prop_P$ **then**
3: $\quad c \leftarrow c - 1$
4: **else**
5: $\quad \textsc{InsRecDel}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum, \text{length}(b_P) + 1, c)$
6: **end if**
7: **for** $i \leftarrow \text{length}(b_P)$ down to 1 **do**
8: $\quad bwt_P[LF^{-1}(i)] = b_P[P[i]] + 1$
9: $\quad b_P[P[i]] \leftarrow b_P[P[i]] + 1$
10: $\quad$ **if** $i \in Prop_P$ **then**
11: $\quad\quad c \leftarrow c - 1$
12: $\quad$ **else**
13: $\quad\quad \textsc{InsRecDel}_{n,\sigma}(P, b_P, bwt_P, p_\$, enum, i, c)$
14: $\quad$ **end if**
15: **end for**

Algorithm 5.4.
InsRecDel$_{n,\sigma}(P, b_P, bwt_P, p_\$, enum, i, c)$

1: $front \leftarrow n - \text{length}(b_P)$
2: $b_P[front] \leftarrow c$
3: $bwt_P[p_\$] \leftarrow c$
4: $p_\$ \leftarrow i$
5: insert$(bwt_P, i, \text{'\$'})$
6: insert$(P, i, front)$
7: EnumSa$_{n,\sigma}(P, b_P, bwt_P, p_\$, enum)$
8: $p_\$ \leftarrow LF^{-1}(p_\$)$
9: $bwt_P[p_\$] \leftarrow$ '\$'
10: $b_P[front] \leftarrow$ '_'
11: delete(bwt_P, i)
12: delete(P, i)

Table 5.4: Enumeration of base strings b_{enum} up to length 4 over alphabet {A, B} and the respective suffix arrays P_{enum}.

$enum$	b_{enum}	P_{enum}
1	__A	3
2	_AA	3, 2
3	AAA	3, 2, 1
4	ABA	3, 1, 2
5	ABB	1, 3, 2
6	_AB	2, 3
7	AAB	1, 2, 3
8	BAB	2, 3, 1

the base string b_P of the respective *sa*-equivalence class $\mathcal{T}_{P,\Sigma}$, the BWT string bwt_P for the \$-extended base string $b_P\$$, the index $p_\$$ with $bwt_P[p_\$] = \$$, and the current enumeration number *enum*. It is invoked with EnumSa$_{n,\sigma}((1), \texttt{A}, \texttt{A\$}, 1, 1)$, where (1) is the smallest non-empty suffix array, A is the base string of the *sa*-equivalence class $\mathcal{T}_{(1),\{A,B\}}$ and A\$ is the BWT string for the \$-extended base string A\$. The recursion terminates if the maximal string length n is reached (line 4). Otherwise, EnumProp is called, which enumerates the extended permutations for the proper insertion positions. Moreover, if b_P is composed of less than σ distinct characters, EnumNoProp is called, which enumerates the extended permutations for the non-proper insertion positions.

EnumProp and EnumNoProp both use InsRecDel. In lines 2–6, InsRecDel attaches the character c at the front of the base string b_P, updates the BWT string bwt_P, and inserts the new suffix number *front* at position i of the permutation P, producing the extended permutation $P_i^{\triangleleft}$. Then EnumSa is called, which recursively enumerates the base strings emanating from $b_{P_i^{\triangleleft}} = c\, b_P$ and the suffix arrays emanating from $P_i^{\triangleleft}$ (line 7). Lines 8–12 reverse the modifications of lines 2–6, reconstructing the original data.

For each character c contained in the base string b_P, EnumProp determines the proper insertion position i that accords to the front extension of b_P by c (line 3), stores the insertion position in $Prop_P$ (line 4), and calls InsRecDel (line 5), which produces the base string $b_{P_i^{\triangleleft}}$ of the extended suffix array $P_i^{\triangleleft}$ and recursively enumerates the suffix arrays emanating from $P_i^{\triangleleft}$. Finally, EnumProp returns the set of proper insertion positions $Prop_P$. Note that we assume C is implicitly updated during each insert or delete operation.

For all non-proper insertion positions i in descending order, EnumNoProp in combination with InsRecDel successively produces the base strings $b_{P_i^{\triangleleft}}$ of the *sa*-equivalence classes for the extended permutations $P_i^{\triangleleft}$, realising Definition 5.10, and recursively enumerates the base strings emanating from $b_{P_i^{\triangleleft}}$ and the suffix arrays emanating from $P_i^{\triangleleft}$. EnumNoProp first assigns the smallest not yet used character to c, (line 1). It passes through all the insertion positions i, starting with the largest, which is handled separately

(lines 2–6). When it moves over the position i, then the character $b_P[P[i]]$ at position i of the First sequence and the corresponding character in bwt_P are increased according to Definition 5.10(b) (line 8–9). If i moves over a proper insertion position, then c is decreased to conform with Definition 5.10(c) (lines 2+3 and lines 10+11). Otherwise, if i is a non-proper insertion position, InsRecDel is called (lines 4+5 and lines 12+13), which attaches c to the front of b_P, updates bwt_P, produces the permutation $P_i^{\lhd}$, and recursively enumerates the base strings emanating from $b_{P_i^{\lhd}} = c\,b_P$ and the suffix arrays emanating from $P_i^{\lhd}$. Table 5.4 shows the enumerated base strings b_{enum} up to length 4 composed of up to 2 distinct characters and the corresponding suffix arrays P_{enum}.

Rank and select functions for the implementation of the BWT have been widely studied in the compressed indexing literature, but most of these data structures are rather static. For an in-depth study of the rank and select data structures and their connection to the Burrows–Wheeler transform, we refer to the survey of Navarro and Mäkinen [113]. For the time-efficient implementation of our enumeration scheme, dynamic data structures representing the Burrows–Wheeler transform are required. We may use the dynamic rank index of Mäkinen and Navarro [93], which performs rank and select as well as insert and delete queries in $\mathcal{O}(\log n)$ time. In this way, the algorithm enumerates the base strings of the non-empty *sa*-equivalence classes and the corresponding suffix arrays in $\mathcal{O}(\log n\ |\mathcal{P}_\sigma^{[1,n]}|)$ time, where $\mathcal{P}_\sigma^{[1,n]}$ is the set of suffix arrays of strings composed of up to σ distinct characters with length up to n. We have $|\mathcal{P}_\sigma^{[1,n]}| = \sum_{j=1}^{n} \sum_{d=0}^{\sigma-1} \left\langle {j \atop d} \right\rangle$, which follows from summing up the suffix array count of Corollary 5.2 for all strings up to length n. Furthermore, we anticipate Lemma 6.1 of Chapter 6.1. It states $\sum_{d=0}^{\sigma-1} \left\langle {j \atop d} \right\rangle = \sum_{k=0}^{\sigma-1} \binom{j}{k} (-1)^k (\sigma-k)^j$, which implies $|\mathcal{P}_\sigma^{[1,n]}| = \sum_{j=1}^{n} \sum_{d=0}^{\sigma-1} \left\langle {j \atop d} \right\rangle = \sum_{j=1}^{n} \sum_{k=0}^{\sigma-1} \binom{j}{k} (-1)^k (\sigma-k)^j$. We thus achieve the time bound of $\mathcal{O}(\log n\ \sum_{j=1}^{n} \sum_{k=0}^{\sigma-1} \binom{j}{k} (-1)^k (\sigma-k)^j)$ for the enumeration of the non-empty *sa*-equivalence classes, represented by their base strings, and the parallel enumeration of the corresponding suffix arrays, which is exponential for $\sigma > 1$.

The technique used in InsRecDel (Algorithm 5.4) for the extension of the Burrows–Wheeler transform can also be used for the right-to-left online construction of the BWT or the suffix array: Lippert *et al.* [92] used it for the construction of the BWT for genomic sequence data. Moreover, Gerlach [52] presented a space-efficient implementation of Mäkinen and Navarro's [93] dynamic rank index for the construction of a compressed index that incorporates the Burrows–Wheeler transform.

6 Application of the String and Suffix Array Counting

Many compressed full-text indices are based on suffix arrays: the compressed suffix array of Grossi and Vitter [57], the compressed-suffix-array-based index by Sadakane [123], Mäkinen's compact suffix array [94], and several others that improve upon these three (see Navarro and Mäkinen [113]).

We are interested in the compressibility of such indices, in particular of those based on suffix arrays. Lower bounds for the size of full-text indices are known: Demaine and López-Ortiz [41] proved a lower bound for indices providing substring search, and Miltersen [107] showed lower bounds for selection and rank indices (see equations (5.5) and (5.6) on page 40).

In this chapter, we apply the result of Corollary 5.2 to prove new tight lower bounds on the compressibility of suffix arrays in Section 6.1. Section 6.2 leaves the compressed indexing field; it combines the counting schemes of the previous two chapters to prove summation identities of Eulerian numbers.

6.1 Applications to compressed suffix arrays

Before formally stating and proving the results on the compressibility of suffix arrays, we first perform some preliminary work. At first sight, the counting formula for the number of suffix arrays of Corollary 5.2 looks quite compact. The Eulerian numbers, however, are recursively defined, which is unfavourable in consideration of the subsequent reasoning. We rather convert the formula into a closed form.

Lemma 6.1. *Let σ and n be fixed positive integers, then*

$$\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle = \sum_{k=0}^{\sigma-1} \binom{n}{k} (-1)^k (\sigma - k)^n.$$

Proof. An equality rule for the Eulerian numbers [56, Section 6.2, eq. 6.38], equality rules

for binomial coefficients, and some arithmetics lead to

$$\begin{aligned}
\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle &= \sum_{d=0}^{\sigma-1}\sum_{k=0}^{d} \binom{n+1}{k}(-1)^k(d+1-k)^n && (6.1)\\
&= \sum_{d=0}^{\sigma-1}\sum_{k=0}^{d} \left(\binom{n}{k} + \binom{n}{k-1}\right)(-1)^k(d+1-k)^n && (6.2)\\
&= \sum_{d=0}^{\sigma-1}\sum_{k=0}^{d} \binom{n}{k}(-1)^k(d+1-k)^n \\
&\quad + \sum_{d=0}^{\sigma-1}\sum_{k=0}^{d} \binom{n}{k-1}(-1)^k(d+1-k)^n && (6.3)\\
&= \sum_{d=1}^{\sigma}\sum_{k=1}^{d} \binom{n}{k-1}(-1)^{k-1}(d+1-k)^n \\
&\quad - \sum_{d=1}^{\sigma-1}\sum_{k=1}^{d} \binom{n}{k-1}(-1)^{k-1}(d+1-k)^n && (6.4)\\
&= \sum_{k=1}^{\sigma} \binom{n}{k-1}(-1)^{k-1}(\sigma+1-k)^n && (6.5)\\
&= \sum_{k=0}^{\sigma-1} \binom{n}{k}(-1)^k(\sigma-k)^n, && (6.6)
\end{aligned}$$

where equality (6.1) follows from $\left\langle {n \atop d} \right\rangle = \sum_{k=0}^{d} \binom{n+1}{k}(-1)^k(d+1-k)^n$ [56, eq. 6.38], equality (6.2) from $\binom{n+1}{k} = \binom{n}{k}+\binom{n}{k-1}$, equality (6.3) from the distributivity, equality (6.4) from shifting d and k with respect to the first sum and from $\binom{n}{k-1} = 0$ for $k \leq 0$, equality (6.5) from subtracting both sums, and finally equality (6.6) from shifting k again. □

Many application areas for suffix arrays handle small alphabets like the DNA, amino acid, or ASCII alphabet. Corollary 5.2 thus limits the number of distinct suffix arrays for such applications. For example, for a DNA alphabet of size 4, the number of distinct suffix arrays of length 16 is $3\,614\,083\,520 = \sum_{d=0}^{3} \left\langle {16 \atop d} \right\rangle$; whereas the number of possible permutations of length 16 is $20\,922\,789\,888\,000 = 16!$, which is about 5 789 times larger. This difference increases rapidly for larger n. We achieve a lower bound on the compressibility of the whole information content of suffix arrays.

Corollary 6.2. *Let Σ^n be the set of strings of length n over an alphabet Σ of size σ. Then the lower bound for the compressibility of the respective suffix arrays in the Kolmogorov sense is* $\log(\sum_{k=0}^{\sigma-1} \binom{n}{k}(-1)^k(\sigma-k)^n)$.

Table 6.1: Number of strings of length n over alphabets of size 4 and 20, and the respective number of suffix arrays.

	Alphabet size 4		Alphabet size 20	
n	Strings	Suffix arrays	Strings	Suffix arrays
4	256	24	160 000	24
6	4 096	662	64 000 000	720
8	65 536	20 160	25 600 000	40320
10	1 048 576	504 046	$\approx 1.0 \cdot 10^{13}$	3 628 800
12	16 777 216	10 670 040	$\approx 4.1 \cdot 10^{15}$	479 001 600
14	268 435 456	202 964 470	$\approx 1.6 \cdot 10^{18}$	20087 178 291
16	4 294 967	526 614 083	$\approx 6.6 \cdot 10^{20}$	$\approx 2.1 \cdot 10^{13}$
18	68 719 476	15061 786 015	$\approx 2.6 \cdot 10^{23}$	$\approx 6.4 \cdot 10^{15}$

Proof. There are $\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle$ distinct suffix arrays. Among them, there exists at least one binary representation with Kolmogorov complexity not less than $\log \sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle$. Due to Lemma 6.1 this equals $\log \sum_{k=0}^{\sigma-1} \binom{n}{k} (-1)^k (\sigma - k)^n$. □

We pose a further question: How is the connection between the number of strings and the number of suffix arrays? For increasing string length, Table 6.1 shows the number of strings over alphabets of size 4 and 20 (DNA and amino acid alphabet size) and the respective number of suffix arrays. The first column shows the string lengths, the second column the number of strings over an alphabet of size 4, the third column the number of suffix arrays for these strings, and the fourth and the fifth column show the respective numbers for an alphabet of size 20. For a fixed alphabet of size σ and increasing string length n, the number of strings σ^n and the number of respective suffix arrays $\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle$ diverge, but we do not immediately see whether the ratio between these numbers diverges or converges. As seen below, it does, in fact, converge.

Theorem 6.3. *Let σ be fixed, then*

$$\lim_{n\to\infty} \frac{\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle}{\sigma^n} = 1.$$

Proof. We obtain

$$\begin{aligned}
\lim_{n\to\infty} \frac{\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle}{\sigma^n} &= \lim_{n\to\infty} \frac{\sum_{k=0}^{\sigma-1} \binom{n}{k} (-1)^k (\sigma - k)^n}{\sigma^n} && (6.7)\\
&= \lim_{n\to\infty} \left(\frac{\sigma^n}{\sigma^n} + \sum_{k=1}^{\sigma-1} \binom{n}{k} (-1)^k \frac{(\sigma - k)^n}{\sigma^n} \right) && (6.8)\\
&= 1 + \sum_{k=1}^{\sigma-1} (-1)^k \lim_{n\to\infty} \left(\binom{n}{k} \left(1 - \frac{k}{\sigma}\right)^n \right) && (6.9)\\
&= 1, && (6.10)
\end{aligned}$$

where equation (6.7) follows from Lemma 6.1, equations (6.8) and (6.9) from basic arithmetics, and equation (6.10) from the fact that $\lim_{n\to\infty}\left(\binom{n}{k}\left(1-\frac{k}{\sigma}\right)^n\right)=0$ for $0<\frac{k}{\sigma}<1$: The exponential term $\left(1-\frac{k}{\sigma}\right)^n$ converges to 0 and dominates the polynomial term $\binom{n}{k}$, $\binom{n}{k}\leq n^k$. □

Note that Theorem 6.3 only holds if the alphabet is of a constant size. If the alphabet size grows proportionally to the string length, it is not true anymore. For $\sigma=n$, $\lim_{n\to\infty}\frac{\sum_{d=0}^{\sigma-1}\left\langle{n\atop d}\right\rangle}{\sigma^n}=\lim_{n\to\infty}\frac{n!}{n^n}=0$.

6.2 Summation identities

We present constructive proofs for two long known summation identities of Eulerian numbers deduced by summing up the number of different suffix arrays for a fixed alphabet size and string length. We believe that our constructive proofs are simpler than previous ones.

Worpitzki's identity. The identity $\sigma^n=\sum_i\left\langle{n\atop i}\right\rangle\binom{\sigma+i}{n}$, as given in [56, eq. 6.37], was proven back in 1883 by J. Worpitzki. We prove it by summing up the number of string-distinct strings of length n over a given alphabet of size σ for each suffix array:

$$\begin{aligned}
\sigma^n &= \sum_{d=0}^{\sigma-1}\left\langle{n\atop d}\right\rangle\binom{n+\sigma-d-1}{\sigma-d-1} && (6.11)\\
&= \sum_{d=0}^{\sigma-1}\left\langle{n\atop n-1-d}\right\rangle\binom{n+\sigma-d-1}{n} && (6.12)\\
&= \sum_{i=n-\sigma}^{n-1}\left\langle{n\atop i}\right\rangle\binom{\sigma+i}{n} && (6.13)\\
&= \sum_{i\in\mathbb{N}_0}\left\langle{n\atop i}\right\rangle\binom{\sigma+i}{n}. && (6.14)
\end{aligned}$$

Equality (6.12) follows from the symmetry rule for Eulerian and binomial numbers, equality (6.13) from substituting $i=n-d-1$, and equality (6.14) from $\left\langle{n\atop i}\right\rangle=0$ for all $i\geq n$ and $\binom{\sigma+i}{n}=0$ for all $i<n-\sigma$.

Summation of Eulerian numbers to generate the Stirling numbers of the second kind. The second summation identity is the summation rule for Eulerian numbers to generate the Stirling numbers of the second kind [56, eq. 6.39]: $\kappa!\left\{{n\atop\kappa}\right\}=\sum_i\left\langle{n\atop i}\right\rangle\binom{i}{n-\kappa}$. To prove this identity, we count the $\kappa!\left\{{n\atop\kappa}\right\}$ strings composed of exactly κ different characters. Summing

up these strings for each suffix array gives

$$\begin{aligned}
\kappa!\left\{{n \atop \kappa}\right\} &= \sum_{d=0}^{\kappa-1} \left\langle {n \atop d} \right\rangle \binom{n-d-1}{\kappa-d-1} && (6.15)\\
&= \sum_{d\in\mathbb{N}_0} \left\langle {n \atop d} \right\rangle \binom{(n-\kappa)+(\kappa-d-1)}{\kappa-d-1} && (6.16)\\
&= \sum_{d\in\mathbb{N}_0} \left\langle {n \atop n-1-d} \right\rangle \binom{n-d-1}{n-\kappa} && (6.17)\\
&= \sum_{i\in\mathbb{N}_0} \left\langle {n \atop i} \right\rangle \binom{i}{n-\kappa}. && (6.18)
\end{aligned}$$

Equality (6.16) holds since $\left\langle {n \atop d} \right\rangle = 0$ for $d \geq \kappa$, equality (6.17) follows from the symmetry rule for Eulerian and binomial numbers, and equality (6.18) from substituting $i = n-d-1$.

Part II

SUFFIX ARRAY CONSTRUCTION

7 Introduction

There are several approaches to construct a suffix array. We can, for example, construct a suffix tree and derive the suffix array by traversing the constructed suffix tree "from left to right" (see [58, Section 7.14.1]). In this second part of the thesis, we mainly focus on direct suffix array construction algorithms, i.e., not taking the detour over suffix trees. We recall the suffix array construction algorithms mentioned in the introduction of the thesis. Besides the $\mathcal{O}(n \log n)$ time *prefix-doubling* algorithm of Manber and Myers [96], there are mainly three groups of algorithms: linear-time algorithms, other algorithms particularly designed for fast practical speed, and lightweight algorithms that try to minimise the auxiliary space during suffix array construction. The linear-time algorithms are the *skew* algorithm of Kärkkäinen and Sanders [71], the linear-time *odd–even* algorithm of Kim *et al.* [80], and the *smaller–larger* algorithm of Ko and Aluru [85]. Algorithms particularly designed for fast practical speed are *qsufsort* by Larsson and Sadakane [90] and the $\mathcal{O}(n \log \log n)$ time *odd–even* algorithm of Kim *et al.* [78] based on [80], but with faster practical running times. Lightweight algorithms are Itoh and Tanaka's *two-stage* algorithm [67], the *copy* and the *cache* algorithms of Seward [135], *deep–shallow* sorting of Manzini and Ferragina [102], and the *difference-cover* algorithm of Burkhardt and Kärkkäinen [31]. We created the name *smaller–larger* ourselves and took the others from the literature. The three groups of algorithms are summarised in Table 7.1.

The above mentioned suffix array construction algorithms meet some of the following requirements for practical suffix array construction:

- Fast construction for common real-life strings (small average LCP): *qsufsort* [90], *two-stage* [67], *copy* and *cache* [135], *deep–shallow* [102], and *odd–even* [78];
- Fast construction for degenerate strings (high average LCP): *prefix-doubling* [96], *qsufsort* [90], *skew* [71], *odd–even* [80], *smaller–larger* [85], *difference-cover* [31], and *odd–even* [78];
- Small space requirements: *two-stage* [67], *copy* and *cache* [135], *deep–shallow* [102], and *difference-cover* [31].

As we have mentioned in Chapter 1, we believe that further properties are required. Especially in biological sequence data, there are many long sequences with mainly small LCPs, interrupted by occasional very large LCPs. Hence, one has to build suffix arrays for strings with highly variable LCPs.

We present a new algorithm that satisfies these requirements. Before that, we review the above mentioned previous suffix array construction algorithms. These algorithms use

Table 7.1: Summary of the suffix array construction algorithms.

Suffix array construction algorithms		
linear-time	fast practical	lightweight
skew (Kärkkäinen and Sanders [71])	*qsufsort* (Larsson and Sadakane [90])	*two-stage* (Itoh and Tanaka [67])
odd–even (Kim *et al.* [80])	*odd–even* (Kim *et al.* [78])	*copy* (Seward [135])
smaller–larger (Ko and Aluru [85])		*cache* (Seward [135])
		deep–shallow (Manzini and Ferragina [102])
		difference-cover (Burkhardt and Kärkkäinen [31])

various auxiliary data structures that we define in Section 7.1. Chapter 8 classifies the techniques used and surveys the algorithms. Chapter 9 then presents our new *bucket-pointer refinement* algorithm, and Chapter 10 provides experimental results.

7.1 Definitions and notations

Let \$ be a character not contained in the alphabet Σ, and assume $\$ < c$ for all $c \in \Sigma$. We often consider the \$-padded extension $t\$^n$ of a string t of length n, which we implicitly assume in the subsequent description of the suffix array construction algorithms. Thus, if an algorithm uses a character at a position greater than n, then it is a \$.

In the following, sa denotes the not necessarily sorted suffix array $sa(t)$ of a string t of length n. That is, it is not lexicographically sorted before the completion of the suffix sorting process. A *bucket* $sa[l,r] = sa[l], sa[l+1], \ldots, sa[r]$ with $1 \leq l \leq r \leq n$ is a contiguous suffix array segment of suffixes with equal, non-empty prefix such that, for all indices $g, i, h \in \mathbb{N}$ with $1 \leq g < l \leq i \leq r < h \leq n$,

$$t[sa[g], n] < t[sa[i], n] < t[sa[h], n].$$

We disregard the order of suffixes in a bucket; buckets containing the same set of suffixes, but in a different order, are considered to be equal. An *ℓ-bucket* contains suffixes all sharing the same prefix of length ℓ, where ℓ is called the *refinement level* of the bucket. Note that ℓ is not necessarily the longest common prefix of all suffixes in an ℓ-bucket, and an ℓ-bucket is also an ℓ'-bucket for $\ell' \leq \ell$. A bucket $sa[i,j]$ is termed a *sub-bucket* of a *super-bucket* $sa[l,r]$ if $l \leq i \leq j \leq r$. *Bucket refinement* decomposes a bucket $sa[l,r]$ into a list of refined sub-buckets $sa[l_1, r_1], sa[l_2, r_2], \ldots, sa[l_\beta, r_\beta]$ for some $\beta \in [1, r-l+1]$ such

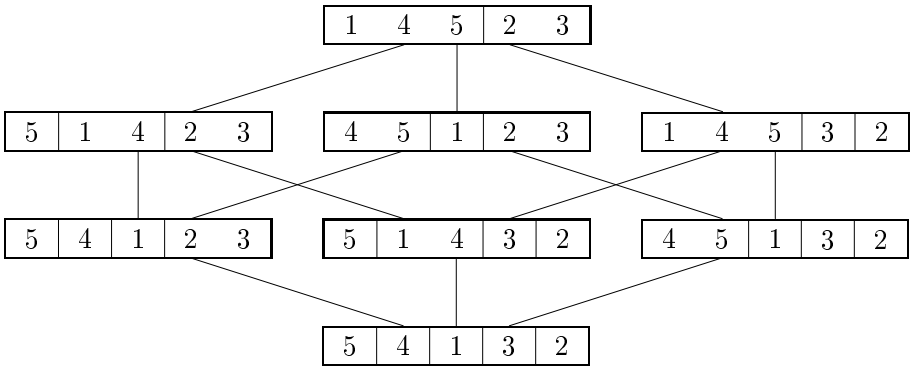

Figure 7.1: A Hasse diagram representing the partial order of the bucket segmentations for the string ABBAA, which has the suffix array $(5, 4, 1, 3, 2)$. The vertical bars between the suffix numbers denote the bucket boundaries.

that $l = l_1$, $r_\beta = r$, $l_k \leq r_k$ for all $k \in [1, \beta]$, and $r_k + 1 = l_{k+1}$ for all $k \in [1, \beta - 1]$. Likewise, a *bucket segmentation* is a decomposition of the whole suffix array into a list of buckets with refinement level $\ell > 0$, $sa[l_1, r_1], sa[l_2, r_2], \ldots, sa[l_\beta, r_\beta]$ for some $\beta \in [1, n]$, such that $1 = l_1$, $r_\beta = n$, $l_k \leq r_k$ for all $k \in [1, \beta]$, and $r_k + 1 = l_{k+1}$ for all $k \in [1, \beta - 1]$, where $sa[l_k, r_k]$ is the k^{th} bucket; k is called the *bucket number* for all suffix numbers in $sa[l_k, r_k]$. An *ℓ-bucket segmentation* consists of ℓ-buckets, $\ell > 0$.

A bucket segmentation is called *refined bucket segmentation* or, alternatively, *sub-bucket segmentation* of a *super-bucket segmentation* if each bucket of the sub-bucket segmentation is a sub-bucket of a bucket in the super-bucket segmentation. Repeated bucket refinement ultimately leads to the bucket segmentation consisting of singleton buckets only, which corresponds to the sorted suffix array.

For a given string, the *sub-bucket–super-bucket relation* defines a partial order on the set of all possible bucket segmentations. The 1-bucket segmentation is the super-bucket segmentation of every other bucket segmentation, and hence, the largest in the partial order. The bucket segmentation only consisting of singleton buckets is the smallest. Figure 7.1 shows a Hasse diagram representing the partial order of the bucket segmentations for the input string ABBAA.

The intermediate result of many suffix array construction algorithms is the sorted order of suffixes regarding their prefixes of a certain length ℓ, the *ℓ-order*. It is defined by the order relation $\leq_\ell$:

$$t[u, n] \leq_\ell t[v, n] :\Longleftrightarrow t[u, u + \ell - 1] \leq t[v, v + \ell - 1]$$

for any two suffix numbers $u, v \in [1, n]$. The relations $<_\ell$ and $=_\ell$ are defined analogously.

Some algorithms represent the ℓ-order by storing the bucket number *bnr* for each suffix. Let $sa[l_k, r_k]$ be the k^{th} bucket of a bucket segmentation into β buckets, $k \in [1, \beta]$. Recall that, for each suffix number u that is an element of the k^{th} bucket $sa[l_k, r_k]$, we have $bnr[u] := k$. More precisely,

$$bnr[sa[i]] := k \text{ for each } i \in [l_k, r_k] \text{ and for each } k \in [1, \beta]. \tag{7.1}$$

Alternatively, a *bucket pointer* $bptr[u]$ is stored for each suffix number $u \in [1, n]$. For all suffix numbers u and v in the same bucket $sa[l_k, r_k]$, we have $bptr[u] = bptr[v] = i$ for some $i \in [l, r]$. We may use the rightmost position of a bucket as bucket pointer such that

$$bptr[sa[i]] := r_k \text{ for each } i \in [l_k, r_k] \text{ and for each } k \in [1, \beta]. \tag{7.2}$$

For each suffix number u, both bucket number and bucket pointer combine the lexicographically sorted order of the respective suffix $t[u, n]$ with respect to the leading characters into a single sort key. For an ℓ-bucket segmentation, there is the following connection between the ℓ-order, bucket numbers, and bucket pointers:

$$t[u, n] \leq_\ell t[v, n] \iff bnr[u] \leq bnr[v] \iff bptr[u] \leq bptr[v]$$

for all suffix numbers $u, v \in [1, n]$. If all buckets are singletons, then the arrays bnr and $bptr$ correspond to the rank array R or, alternatively, to the inverse suffix array.

A *radix step* denotes the part of an algorithm in which strings are sorted according to the characters at a certain offset ℓ in the string; ℓ is called *radix level*. A radix step is like a single iteration of *most-significant-digit (MSD) radix sort* (see [82, Section 5.2.5]). That is, the sorting procedure orders any two suffixes $t[u, n]$ and $t[v, n]$ sharing the same prefix of length ℓ by their characters $t[u + \ell]$ and $t[v + \ell]$ (note the equality of radix level and refinement level).

The length of the longest common prefix of two strings t and t' is referred to by $\text{lcp}(t, t')$. For two suffix numbers $u, v \in [1, n]$, $\text{lcp}(u, v)$ denotes the length of the longest common prefix of $t[u, n]$ and $t[v, n]$. For a suffix array sa of a string t of length n, the LCP array lcp of length $n - 1$ is defined by the length of the longest common prefix of consecutive suffixes in the suffix array, $lcp[i] := \text{lcp}(t[sa[i], n], t[sa[i + 1], n])$ for all $i \in [1, n - 1]$. For two positions $g, h \in [1, n]$ with $g < h$, we obtain the length of the longest common prefix of the suffixes $t[sa[g], n]$ and $t[sa[h], n]$ by $\text{lcp}(sa[g], sa[h]) = \min\{lcp[i] : i \in [g, h - 1]\}$.

8 Classification and Survey of Previous Suffix Array Construction Algorithms

In the last years, many suffix array construction algorithms have been invented using various techniques. Puglisi *et al.* [120] recently categorised the suffix array construction algorithms into three different classes: prefix-doubling, recursive, and induced copying. Some algorithms, however, are not uniquely assignable to a single class and are thus classified as hybrid.

In Section 8.1, we present two new orthogonal classifications. In both, each suffix array construction algorithm surveyed is uniquely assignable to only one of two possible classes. After that, Sections 8.2 and 8.3 review the classified algorithms: we survey each algorithm, give the worst-case and expected-case time bounds, and analyse the space requirements.

8.1 Classifying suffix array construction algorithms

We categorise the suffix array construction algorithms with respect to two orthogonal classification types: The first classifies the algorithms regarding their progress in the suffix sorting process, Section 8.1.1, and the second regarding the use of dependencies among suffixes, Section 8.1.2.

8.1.1 Progression of the suffix sorting process

This classification groups the algorithms based on two questions: Which suffixes are first processed, and how does the suffix sorting process advance? The algorithms are classified into two groups: bucket refinement and reduced string sorting.

8.1.1.1 Bucket refinement

Many of the practical suffix array construction algorithms order suffixes regarding their leading characters into buckets, which are then recursively refined. These algorithms are classified as *bucket refinement algorithms*. The first type of bucket refinement techniques found in the literature is formed by string sorting methods without using the dependencies among suffixes. Most representatives of this class sort the suffixes regarding their leading characters and then refine the groups of suffixes with equal prefixes by recursively performing radix steps with increasing radix level until unique prefixes are obtained. Algorithms that fall into this category are the *MSD radix sort* implementation of McIlroy *et al.* [106] and *Multikey Quicksort* of Bentley and Sedgewick [23].

The second type of bucket refinement algorithms use the order of previously computed suffixes in the refinement phase. If two suffixes $t[u, n]$ and $t[v, n]$ share a common prefix of length ℓ, then their ordering can be derived from the ordering of their ℓ-successors $t[u + \ell, n]$ and $t[v + \ell, n]$. We further divide these algorithms into two subgroups: algorithms performing *breadth-first* refinement, as the *prefix-doubling* algorithm of Manber and Myers [96] and the *qsufsort* algorithm of Larsson and Sadakane [90], and algorithms performing *depth-first* refinement, as Itoh and Tanaka's *two-stage* algorithm [67], the *copy* and the *cache* algorithms of Seward [135], and *deep–shallow* sorting of Manzini and Ferragina [102]. The breadth-first refinement algorithms iteratively compute ℓ-bucket segmentations for an increasing ℓ such that all buckets share the same refinement level after each iteration, whereas the depth-first refinement algorithms follow the refinement scheme of methods of the first type: Before starting with the next bucket, they refine a single bucket until all its sub-buckets are singletons. Many practical algorithms that use this technique also apply methods of the first type to fall back upon if the order of suffixes at the offset ℓ is not yet available.

Figure 8.1 shows stages of the bucket refinement process for the string `AAABBABBBAAABBAB`. We represent each suffix by a vertical bar, where the length of the bar represents its relative lexicographical order: short bars for lexicographically small suffixes and long bars for lexicographically large suffixes. The top picture shows the suffixes ordered by their starting positions in the string from left to right. The pictures in the middle show a bucket segmentation after some steps of bucket refinement algorithms. The middle picture to the left shows an intermediate bucket segmentation for a breadth-first bucket refinement algorithm, and the middle picture to the right shows an intermediate bucket segmentation of a depth-first bucket refinement algorithm. The bottom picture represents the completely sorted suffix array.

8.1.1.2 Reduced string sorting

Other suffix array construction algorithms select a specific subset *sub* of suffix numbers, sort the corresponding suffixes with respect to their prefixes of a particular length, assign a sort key to each such suffix that represents the lexicographical order with respect to those prefixes, and form a *reduced string* t^{sub} of length $|sub|$ consisting of the previously assigned sort keys such that the suffix array $sa(t^{sub})$ of t^{sub} reflects the lexicographically sorted order of all suffixes in *sub*. The algorithms then construct the suffix array $sa(t^{sub})$ of t^{sub}, and derive therefrom the lexicographically sorted order of the original suffixes in *sub*. Finally, the lexicographically sorted suffixes in *sub* are used as anchors for the sorting of the remaining suffixes, and the complete suffix array is computed. Burkhardt and Kärkkäinen's *difference-cover* algorithm [31], Kärkkäinen and Sanders's *skew* algorithm [71], the *odd–even* algorithm of Kim *et al.* [80] (also [78]), and the *smaller–larger* algorithm of Ko and Aluru [85] follow this scheme. We call them *reduced string sorting algorithms*.

Figure 8.1 shows stages of a reduced string sorting algorithm, again for the string `AAABBABBBAAABBAB`. The suffixes with their relative lexicographical order are again represented by vertical bars of different lengths. The top picture shows the suffixes ordered

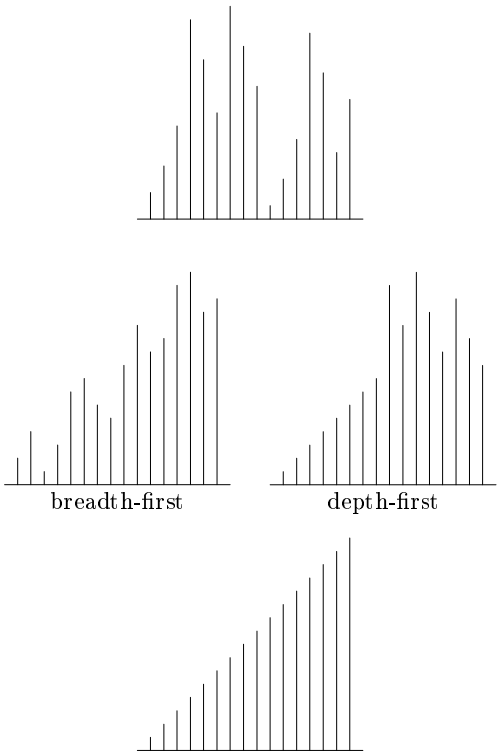

Figure 8.1: Stages of bucket refinement algorithms for the string AAABBABBBAAABBAB.

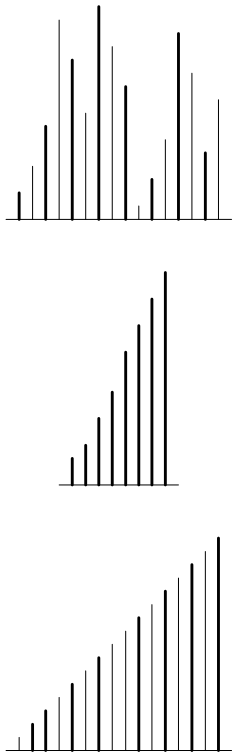

Figure 8.2: Stages of a reduced string sorting algorithm for the string AAABBABBBAAABBAB.

by their starting positions in the string from left to right, where the bars for the suffixes starting at the odd positions are printed in bold face. The middle picture represents the lexicographically sorted suffixes with odd starting position. The bottom picture again represents the completely sorted suffix array.

8.1.2 Dependency among suffixes

Another classification scheme groups the suffixes regarding their use of dependencies among suffixes. If two suffixes $t[u, n]$ and $t[v, n]$ share a common prefix of length ℓ, then their order can be derived from the order of suffixes $t[u + \ell, n]$ and $t[v + \ell, n]$. We distinguish two techniques: the *push* method and the *pull* method. The terms *push* and *pull* are adopted from the terminology of information systems: They are communication strategies between information carrier and information receiver. The push method refers to a style of communication where the information interchange originates with the information carrier. It is contrasted with the pull method, where the information receiver requests for the transmission of information.

Table 8.1: Summary of the classifications.

Suffix sorting process	Using dependencies among suffixes	
	push	pull
Bucket refinement (breadth-first)	*prefix-doubling* (Manber and Myers [96])	*qsufsort* (Larsson and Sadakane [90])
Bucket refinement (depth-first)	*two-stage* (Itoh and Tanaka [67])	*cache* (Seward [135])
	copy (Seward [135])	
	deep–shallow (Manzini and Ferragina [102])	
Reduced string sorting	*skew* (Kärkkäinen and Sanders [71])	*difference-cover* (Burkhardt and Kärkkäinen [31])
	odd–even (Kim *et al.* [80], also [78])	
	smaller–larger (Ko and Aluru [85])	

8.1.2.1 Push method

The *push* method uses the ordering of previously determined groups of suffixes (information carrier) and passes this ordering on to undetermined groups of predecessor suffixes (information receiver). This technique is used in many algorithms. Manber and Myers's *prefix-doubling* algorithm [96], Itoh and Tanaka's *two-stage* algorithm [67], Seward's *copy* algorithm [135], and *deep–shallow* sorting of Manzini and Ferragina [102] are examples of bucket refinement algorithms that use this method. It is also used in the linear-time algorithms: *skew* of Kärkkäinen and Sanders [71], *odd–even* of Kim *et al.* [80] (also [78]), and *smaller–larger* of Ko and Aluru [85].

8.1.2.2 Pull method

The *pull* method is used for the comparison-based sorting. Algorithms look up the order of successor suffixes $t[u + \ell, n]$ and $t[v + \ell, n]$ to determine the order of $t[u, n]$ and $t[v, n]$ (information request). Some representatives that use this technique are: Larsson and Sadakane's *qsufsort* [90], Seward's *cache* algorithm [135], and the *difference-cover* algorithm of Burkhardt and Kärkkäinen [31].

Table 8.1 summarises the classification of the suffix array construction algorithms that use dependencies among suffixes; ordinary string sorting algorithms are not shown. The first column shows the classes regarding the progress of the suffix sorting process. The second column shows algorithms using the push method and the third column algorithms using the pull technique. We continue with a survey of the categorised algorithms: bucket refinement algorithms in Section 8.2 and reduced string sorting algorithms in Section 8.3.

8.2 Bucket refinement algorithms

We confine ourselves to the bucket refinement algorithms utilising the dependencies among suffixes. Section 8.2.1 reviews the algorithms performing breadth-first bucket refinement and Section 8.2.2 the algorithms performing depth-first bucket refinement. The algorithms are analysed regarding their construction time and space requirements, where the expected construction times are given for a Bernoulli sequence model (i.e., symbols from the alphabet are generated independently).

8.2.1 Breadth-first bucket refinement – prefix-doubling algorithms

The prefix-doubling algorithms of Manber and Myers [96] and Larsson and Sadakane [90] both use ideas of Karp *et al.* [75]. They first sort the suffixes with respect to their leading character, producing a 1-bucket segmentation. Then they iteratively double the prefix length with respect to which the suffixes are sorted, producing a 2^i-bucket segmentation in the i^{th} iteration. The iteration loop terminates when all buckets are singletons.

At the beginning of the i^{th} iteration step, the suffixes are ℓ-ordered with $\ell = 2^{i-1}$. For any two suffixes $t[u,n]$ and $t[v,n]$ with $u, v \in [1,n]$, we obtain their relative 2ℓ-order by combining the relative ℓ-order of $t[u,n]$ and $t[v,n]$ with the relative ℓ-order of their successor suffixes $t[u+\ell,n]$ and $t[v+\ell,n]$:

$$t[u,n] \leq_{2\ell} t[v,n] \iff \begin{cases} t[u,n] <_\ell t[v,n] \text{ or} \\ t[u,n] =_\ell t[v,n] \text{ and } t[u+\ell,n] \leq_\ell t[v+\ell,n] \end{cases} \tag{8.1}$$

for $u, v \in [1, n-\ell]$. Alternatively,

$$t[u-\ell,n] \leq_{2\ell} t[v-\ell,n] \iff \begin{cases} t[u-\ell,n] <_\ell t[v-\ell,n] \text{ or} \\ t[u-\ell,n] =_\ell t[v-\ell,n] \text{ and } t[u,n] \leq_\ell t[v,n], \end{cases} \tag{8.2}$$

for $u, v \in [\ell+1, n]$.

8.2.1.1 The *prefix-doubling* algorithm of Manber and Myers

The algorithm of Manber and Myers [96] first performs a bucket sort of the suffixes according to their leading characters. Then it repeats the prefix-doubling process, which uses equivalence (8.2), until all buckets are singletons.

Each prefix-doubling iteration assumes an ℓ-bucket segmentation $sa[l_1, r_1], sa[l_2, r_2], \ldots, sa[l_\beta, r_\beta]$ with $\ell = 2^i$ for some $i \in [1, \lceil \log_2 n \rceil]$. Moreover, $front_k$ refers to the front

position of the k^{th} bucket $sa[l_k, r_k]$ for all $k \in [1, \beta]$, initially $front_k = l_k$. The algorithm scans sa bucket-wise from left to right. For each bucket $sa[l_k, r_k]$, it starts with the suffix number $sa[l_k]$, locates its ℓ-predecessor $sa[l_k] - \ell$ contained in some ℓ-bucket $sa[l_g, r_g]$, moves $sa[l_k] - \ell$ to the current front of $sa[l_g, r_g]$ (i.e., $sa[front_g] \leftarrow sa[l_k] - \ell$) and advances the front of $sa[l_g, r_g]$ by one position to the right (i.e., $front_g \leftarrow front_g + 1$). Then the algorithm continues with the next suffix number $sa[l_k + 1]$ in $sa[l_k, r_k]$, moves its ℓ-predecessor $sa[l_k + 1] - \ell$ to the front of its bucket and advances that front by one. This procedure is repeated for all suffix numbers in $sa[l_k, r_k]$ from left to right. After scanning the whole bucket $sa[l_k, r_k]$, the contiguous segments of suffixes at the leftmost positions of each ℓ-bucket that have been moved to the front during the scan form a 2ℓ-bucket. The procedure is repeated for all buckets $sa[l_k, r_k]$ with $1 \leq k \leq \beta$ in ascending order, resulting in a 2ℓ-bucket segmentation of sa.

Time and space consumption. Each prefix-doubling iteration can be performed in $\mathcal{O}(n)$ time and there are at most $\log n$ prefix-doubling iterations until the string length is reached, which together gives an $\mathcal{O}(n \log n)$ worst-case time bound for the whole algorithm. Manber and Myers further enhanced the first stage of their algorithm such that it generates a $(\log_{|\Sigma(t)|} n)$-bucket segmentation in linear time, resulting in an $\mathcal{O}(n)$ expected-case time bound.

The algorithm can be implemented using $2n$ words of space: the suffix array and an auxiliary array handling the buckets, each consuming n words. The input string needs not to be kept in main memory during the construction of the suffix array. An efficient implementation is given by McIlroy [105].

8.2.1.2 The *qsufsort* algorithm of Larsson and Sadakane

Like Manber and Myers's algorithm, the *qsufsort* algorithm of Larsson and Sadakane [90] first sorts the suffixes with respect to the leading character. After that, however, the prefix-doubling iteration of *qsufsort* uses equivalence (8.1) instead of equivalence (8.2). Each iteration again takes an ℓ-bucket segmentation and produces a 2ℓ-bucket segmentation, but here, each bucket is refined locally. The algorithm maintains a bucket pointer $bptr[u]$ for each suffix number $u \in [1, n]$ representing the relative ℓ-order of the suffixes. Let $sa[l_1, r_1], sa[l_2, r_2], \ldots, sa[l_\beta, r_\beta]$ be the current ℓ-bucket segmentation. For each $k \in [1, \beta]$, the refinement procedure sorts the suffix numbers in $sa[l_k, r_k]$ with respect to the bucket pointers of their ℓ-successors $sa[l_k] + \ell, sa[l_k + 1] + \ell, \ldots, sa[r_k] + \ell$. That is, $bptr[sa[l_k] + \ell], bptr[sa[l_k + 1] + \ell], \ldots, bptr[sa[r_k] + \ell]$ are the corresponding sort keys. Bentley and McIlroy's *Ternary-Split Quicksort* is applied to sort each ℓ-bucket. After all buckets have been processed, the algorithm computes the splitting positions between non-equal sort keys for each bucket. Together with the previous splitting positions, which have determined the ℓ-bucket segmentation, these new splitting positions determine the 2ℓ-bucket segmentation. Finally, the algorithm updates the bucket pointers. As before, the prefix-doubling process is repeated until all buckets are singletons.

Time and space consumption. As the algorithm of Manber and Myers, Larsson and Sadakane's algorithm reaches an $\mathcal{O}(n \log n)$ worst-case time bound and requires $2n$ words of space: n words for the suffix array and n words for the bucket pointer array. Nevertheless, in terms of practical running time, it is significantly faster (see Larsson and Sadakane [90, page 18] for running times of the two algorithms).

8.2.2 Depth-first bucket refinement

We begin the review of the depth-first bucket refinement algorithms with the *two-stage* algorithm of Itoh and Tanaka [67] and continue with *copy* and *cache* by Seward [135]. The former two implement the push technique and *cache* uses the pull technique. Finally, we review *deep–shallow* sorting of Manzini and Ferragina [102], which is based on the technique used by *copy*. For the analysis of these algorithms, we assume that the underlying alphabet of the input strings is of constant size σ.

8.2.2.1 The *two-stage* algorithm of Itoh and Tanaka

Itoh and Tanaka [67] classify each suffix as type s or type l ($\underline{s}$maller or $\underline{l}$arger). We instead use the notation $\preceq$ for the suffixes of type s, and $\succ$ for the suffixes of type l. A suffix $t[u, n]$ with $u \in [1, n-1]$ is of type $\preceq$ if its first character is smaller than or equal to the first character of its successor $t[u+1, n]$, $t[u] \leq t[u+1]$. Otherwise it is of type $\succ$.

The algorithm successively performs three phases. The suffixes are first bucket sorted with respect to their leading character and suffix type ($\preceq$ or $\succ$). That is, there are 2σ buckets, where a bucket $sa[l_{c,\tau}, r_{c,\tau}]$ contains all suffixes of type $\tau \in \{\preceq, \succ\}$ with leading character $c \in \Sigma$. Furthermore, the suffix number n of the last suffix $t[n, n]$ is moved to the front of its bucket.

The second phase sorts all buckets containing suffixes of type $\preceq$: Large buckets are refined by *MSD radix sort*, medium buckets are sorted by Bentley and Sedgewick's *Multikey Quicksort* [23], and small buckets by *Insertion Sort*.

The third phase determines the order of all suffixes of type $\succ$ and assigns them to their final position: The algorithm scans the suffix array sa from left to right. For each position $i \in [1, n]$, if the predecessor $t[sa[i]-1, n]$ of suffix $t[sa[i], n]$ is of type $\succ$, then the algorithm assigns $sa[i]-1$ to the current front of the bucket $sa[l_{t[sa[i]-1],\succ}, r_{t[sa[i]-1],\succ}]$ and advances the front of the bucket by one position to the right. The suffix sorting process is completed after scanning the whole suffix array sa.

Time and space consumption. The bucket sorting in phase one and the assignment of suffixes of type $\succ$ to their final positions in phase three can be performed in linear time. The most time-consuming part is the *MSD radix sort* in phase 2. Its running time is bounded by the comparison-based sorting complexity $\mathcal{O}(n \log n)$ multiplied by the maximum longest common prefix length of two suffixes of the input t, where the maximal longest common prefix length is $n-1 \in \mathcal{O}(n)$ and the expected longest common

prefix length is $\mathcal{O}(\log n)$ for different string models, a simple consequence of results by Apostolico and Szpankowski [9] and Szpankowski [139]. Hence, $\mathcal{O}(n^2 \log n)$ is the worst and $\mathcal{O}(n \log^2 n)$ the expected construction time of the *two-stage* algorithm.

The auxiliary space requirements are negligible: In addition to the suffix array and the input string, only 2σ words are required to store the bucket boundaries.

8.2.2.2 The *copy* and the *cache* algorithms of Seward

Seward [135] presented some techniques for the construction of the Burrows–Wheeler transform, which are used in the program *bzip2* [134]. These techniques can also be applied for suffix array construction, because of the equivalence to the construction of the Burrows–Wheeler transform. Here, the *copy* method, which was earlier mentioned by Burrows and Wheeler [32], and the *cache* method are reviewed.

Before applying one of these techniques, the suffixes are bucket sorted according to their leading two characters, generating a 2-bucket segmentation of the suffix array. Buckets consisting of all suffixes with the leading character b and second character c, $(b, c) \in \Sigma^2$, form the 2-bucket $sa[l_{b,c}, r_{b,c}]$, and the consecutive 2-buckets consisting of suffixes sharing the leading character b form the 1-bucket $sa[l_b, r_b]$ consisting of all suffixes with leading character b.

The *copy* algorithm. The *copy* algorithm proceeds similarly to the *two-stage* algorithm. After the initial bucket sort, *copy* performs the following steps for each 1-bucket $sa[l_c, r_c]$, $c \in \Sigma$. An ordinary string sorting algorithm sorts each 2-bucket $sa[l_{b,c}, r_{b,c}]$, $(b, c) \in \Sigma^2$, that has not yet been sorted, except for the bucket $sa[l_{c,c}, r_{c,c}]$ that consists of suffixes with equal first and second character. Let $sa[l_{b_1,c}, r_{b_1,c}], sa[l_{b_2,c}, r_{b_2,c}], \ldots, sa[l_{b_\sigma,c}, r_{b_\sigma,c}]$ be the 2-buckets of suffixes with second character c, $b_k \in \Sigma$ for all $k \in [1, \sigma]$. The algorithm passes the ordering of suffixes in $sa[l_c, r_c]$ on to the specified 2-buckets: It performs a left-to-right scan over $sa[l_c, l_{c,c} - 1]$ and over the "left part" of $sa[l_{c,c}, r_{c,c}]$, and then a right-to-left scan over $sa[r_{c,c} + 1, r_c]$ and over the "right part" of $sa[l_{c,c}, r_{c,c}]$, effectively scanning the whole 1-bucket $sa[l_c, r_c]$. For each suffix number u encountered in the left-to-right scan, if $sa[l_{t[u-1],c}, r_{t[u-1],c}]$ is not already sorted, then the predecessor suffix number $u - 1$ is assigned to the front of the bucket $sa[l_{t[u-1],c}, r_{t[u-1],c}]$, and that front is advanced by one position to the right. The left-to-right scan stops if it reaches a position of $sa[l_{c,c}, r_{c,c}]$ that has not been assigned during the current left-to-right scan, or if the rightmost position $r_{c,c}$ of $sa[l_{c,c}, r_{c,c}]$ is reached. The right-to-left scan proceeds analogously, the only difference being that the suffix numbers are assigned to the end of the buckets. Afterwards, all 2-buckets $sa[l_{b,c}, r_{b,c}]$ with $c \in \Sigma$ are correctly sorted, including $sa[l_{c,c}, r_{c,c}]$.

The *cache* algorithm. The *cache* algorithm can be used in combination with *copy*. It uses an additional *cache array* R^C of length n, which is a sort of "partial rank" of the suffix array.

The 1-buckets (or rather their sub-buckets) are sorted with an ordinary string sorting algorithm as before. After a 1-bucket $sa[l_c, r_c]$ with $v \in \Sigma$ is completely refined, R^C is

updated such that $R^C[sa[i]] := i - l_c$ for all $i \in [l_c, r_c]$. Afterwards, the relative order of any two suffixes $t[u,n]$ and $t[v,n]$ that share the same leading character c ($= t[u] = t[v]$) is represented by the order of their R^C values. That is, $t[u,n] < t[v,n]$ if and only if $R^C[u] < R^C[v]$. This property is used by the string sorting algorithm. Whenever it compares two suffixes $t[u,n]$ and $t[v,n]$ ($u, v \in [1,n]$) that share the same leading character c for which the corresponding 1-bucket $sa[l_c, r_c]$ has been previously sorted, it uses the sort key $R^C[u]$ for $t[u,n]$ and $R^C[v]$ for $t[v,n]$.

Time and space consumption. The time bounds for *cache* and *copy* are the same. The bucket sorting in phase one, the copying of suffix numbers, and the maintenance of the cache array can be performed in linear time. The most time-consuming part is the string sorting of buckets, which is bounded by the comparison-based sorting complexity $\mathcal{O}(n \log n)$ multiplied by the maximum longest common prefix length of two suffixes, which is again $\mathcal{O}(\log n)$ in the expected case and $\mathcal{O}(n)$ in the worst case. Hence, $\mathcal{O}(n \log^2 n)$ is the expected and $\mathcal{O}(n^2 \log n)$ the worst construction time.

The auxiliary space requirements of *copy* are negligible, as those of the *two-stage* algorithm: It requires σ^2 additional words for the bucket boundaries and σ words for the front positions of the respective 2-buckets during the copying.

The bucket boundaries are also used for *cache*. In addition, *cache* requires space for the n integers of the R^C array. However, only values up to the size of the largest 1-bucket have to be stored. Hence, 16 or 8 bit integers are enough if no 1-bucket exceeds the size of 2^{16} or 2^8, respectively. Even for larger 1-buckets, reduced word lengths are possible: If the word size of entries in R^C is fixed to w bits and the size of a 1-bucket $sa[l_c, r_c]$ exceeds the 2^w limit, then R^C is defined by $R^C[sa[i]] := 2^w(i - l_c)/(r_c - l_c - 1)$ for all $i \in [l_c, r_c]$.

8.2.2.3 The *deep–shallow* algorithm of Manzini and Ferragina

Manzini and Ferragina developed the *deep–shallow* algorithm [102], which improves upon Seward's *copy* algorithm [135]. The algorithm applies different sorting routines for ℓ-buckets of different size and different common prefix length ℓ, as follows. The ℓ-buckets are primarily refined by Bentley and Sedgewick's *Multikey Quicksort* if $\ell \leq L$, where L is a predefined threshold (shallow sorting). For larger ℓ ($> L$), the algorithm switches to a sorting routine for suffixes sharing a long common prefix (deep sorting). The deep sorter determines the sorting routine depending on the size of the sub-buckets. If the bucket size is smaller than a predefined threshold B, then *Blind Sort* is used, which is based on the blind trie data structure used within the String B-tree [48]. If the bucket size exceeds B, *Ternary-Split Quicksort* of Bentley and McIlroy [22] with some enhancements refines the buckets until the sub-bucket size drops below the threshold B; then *Blind Sort* is used.

A nice feature of *cache* is that some suffixes with equal prefix are not directly compared. They are rather sorted by deriving their order from previously sorted successor suffixes. The induction sort sub-procedure generalises this technique. If an ℓ-bucket $sa[l,r]$ of suffixes sharing the common prefix $p = p_1, \ldots, p_\ell$ has to be sorted, then p is searched

for the first position $k \in [1, \ell - 1]$ such that the 2-bucket of suffixes with first character p_k and second character p_{k+1} has been previously sorted. Let $sa[g, h]$ be the respective 2-bucket. Then the suffix number $sa[l] + k$ is looked up in $sa[g, h]$ and the preceding and following suffix numbers of $sa[l] + k$ in $sa[g, h]$ are scanned. Each scanned suffix number u with $(u - k)$ in $sa[l, r]$ is marked. The scanning terminates when all $r - l + 1$ k-predecessor suffixes that appear in $sa[l, r]$ have been marked. Finally, the suffix numbers in $sa[g, h]$ are scanned from left to right. For each marked suffix number u encountered, the k-predecessor $(u - k)$ is assigned to the current front of $sa[l, r]$, and that front is advanced by one position to the right.

Manzini and Ferragina employ a sparse index to efficiently determine the position of $sa[l] + k$ in $sa[g, h]$. As well as the R^C array of the *cache* method, this index can be regarded as a partial rank of the suffix array. Note that we classify this as a push method since the algorithm scans the suffixes in $sa[g, h]$ and passes their ordering on to $sa[l, r]$. The request, however, was initiated by the bucket $sa[l, r]$. Hence, this technique could be regarded as a pull method just as well.

Time and space consumption. The time bounds are the same as for the algorithms *two-stage*, *cache*, and *copy*. $\mathcal{O}(n \log^2 n)$ is the expected and $\mathcal{O}(n^2 \log n)$ the worst construction time. The auxiliary space requirements are negligible, as for the depth-first bucket refinement algorithms: Only σ^2 additional words for the bucket boundaries and some words for the sparse index are needed.

8.3 Reduced string sorting algorithms

The next four algorithms first construct a *sparse suffix array* sa^{sp} of size n^{sp} containing a particular subset of suffix numbers $sp \subset [1, n]$, $n^{sp} = |sp|$, where sa^{sp} is simply a subsequence of the lexicographically sorted complete suffix array. We transfer the concept of buckets and bucket segmentations to sparse suffix arrays: An ℓ-bucket $sa^{sp}[l, r]$ of a sparse suffix array sa^{sp} is a contiguous segment of sa^{sp} containing suffixes with an equal, non-empty prefix of length ℓ. Furthermore, an ℓ-bucket segmentation of the sparse suffix array is a decomposition of the sparse suffix array into ℓ-buckets with $sa^{sp}[l_1, r_1], sa^{sp}[l_2, r_2], \ldots, sa^{sp}[l_\beta, r_\beta]$ for some $\beta \in [1, n^{sp}]$ such that $1 = l_1$, $r_\beta = n^{sp}$, $l_k \leq r_k$ for all $k \in [1, \beta]$, and $r_k + 1 = l_{k+1}$ for all $k \in [1, \beta - 1]$, where $sa[l_k, r_k]$ is the k^{th} bucket; k is called the *sparse bucket number* for all suffix numbers in $sa^{sp}[l_k, r_k]$. The *sparse bucket number array* bnr^{sp} is accordingly defined. The *sparse rank array* R^{sp} is defined such that $R^{sp}[s] := i$ if $sa^{sp}[i] = s$. Note that the sparse bucket number array bnr^{sp} and the sparse rank array are only defined for the suffix numbers s in sa^{sp}; the other positions remain undefined: $bnr^{sp}[s] = R^{sp}[s] = \perp$ if s is not among the suffix numbers in sp. For a sparse suffix array sa^{sp}, the LCP array lcp of length $n^{sp} - 1$ is defined by $lcp^{sp}[i] := \text{lcp}(t[sa^{sp}[i], n], t[sa^{sp}[i + 1], n])$ for all $i \in [1, n^{sp} - 1]$.

8.3.1 The *difference-cover* algorithm of Burkhardt and Kärkkäinen

A set D with $D \subseteq [0, \ell-1]$ is a *difference-cover modulo* ℓ if $[0, \ell-1] = \{(d - d') \bmod \ell : (d, d') \in D^2\}$. The *difference-cover* algorithm of Burkhardt and Kärkkäinen [31] first selects an appropriate value for ℓ and computes a difference-cover D modulo ℓ with $D = \{d_1, d_2, \ldots, d_\delta\}$ of size $\delta := |D|$. Without loss of generality, we assume that the string length n is a multiple of ℓ and that $0 \notin D$. The algorithm constructs the sparse suffix array sa^D of length $n^D = n \cdot \delta/\ell$ of suffixes $s \in [1, n]$ with $s \bmod \ell \in D$. Then it uses the suffix numbers of the sparse suffix array sa^D, which represent the lexicographically sorted order of the corresponding suffixes, as anchors for the comparison-based sorting of all suffixes, yielding the complete suffix array sa.

Constructing the sparse suffix array. The sparse suffix array sa^D is constructed in three successive phases. *Multikey Quicksort* of Bentley and Sedgewick [23] first lexicographically sorts the suffixes with suffix number in sa^D with respect to their ℓ leading characters, resulting in an ℓ-bucket segmentation of sa^D. According to the ℓ-bucket segmentation, the algorithm assigns the respective sparse bucket number $bnr^D[s]$ to each suffix s in sa^D. Note that, for each suffix number s in sa^D, its bucket number $bnr^D[s]$ combines the lexicographically sorted order of $t[s, n]$ with respect to the ℓ leading characters $t[s, s+\ell-1]$ into a single sort key.

In the second phase, a reduced string t^D of length n^D is computed such that the lexicographical order of the suffixes of t^D corresponds to the lexicographical order of the suffixes contained in sa^D. The partial function μ^D bijectively maps the suffix numbers in sa^D onto the positions $[1, n^D]$ of t^D such that, for all $k \in [1, \delta]$ and for all $s \in [1, n]$,

$$\mu^D(s) \;=\; \frac{(k-1)n}{\ell} + \left\lceil \frac{s}{\ell} \right\rceil \quad \text{if } s \bmod \ell = d_k.$$

That is, the suffix numbers $s \in [1, n]$ with $s \bmod \ell = d_k$ are monotonically increasingly mapped onto a contiguous segment of natural numbers: The n/ℓ suffix numbers $d_k, d_k + \ell, d_k + 2\ell, \ldots, d_k + n - \ell$ are mapped onto $[(k-1)n/\ell + 1, k \cdot n/\ell]$ for all $k \in [1, \delta]$. Moreover, let $\mu^{D(-1)}$ be the inverse mapping, which maps the positions $[1, n^D]$ of the reduced string t^D onto the suffix numbers $s \in [1, n]$ with $s \bmod \ell \in D$.

The algorithm constructs the reduced string t^D of length n^D,

$$t^D[i] := bnr^D[\mu^{D(-1)}(i)] \quad \text{for all } i \in [1, n^D].$$

Then one of the prefix-doubling algorithms presented in Section 8.2.1 is used to compute the suffix array $sa(t^D)$ of the reduced string t^D. After that, the *difference-cover* algorithm derives the sparse suffix array sa^D from $sa(t^D)$,

$$sa^D[i] \;=\; \mu^{D(-1)}(sa(t^D)[i]) \quad \text{for all } i \in [1, n^D].$$

Constructing the complete suffix array. The complete suffix array sa is computed as follows. *Multikey Quicksort* is used to sort all suffixes according to their ℓ leading characters, generating an ℓ-bucket segmentation. Finally, a comparison-based sorting of each ℓ-bucket finishes the construction of sa: For any pair of suffix numbers $(u, v) \in [1, n]^2$, $\Delta(u, v) \in [0, \ell - 1]$ gives an offset such that $(u + \Delta(u, v)) \bmod \ell \in D$ and $(v + \Delta(u, v)) \bmod \ell \in D$. Two suffixes $t[u, n]$ and $t[v, n]$ with $u, v \in [1, n]$ are then compared by using the sort keys $R^D[u + \Delta(u, v)]$ and $R^D[v + \Delta(u, v)]$, respectively. That is, $t[u, n] < t[v, n]$ if and only if $R^D[u + \Delta(u, v)] < R^D[v + \Delta(u, v)]$.

Time and space consumption. For $\ell = \log n$ and constant alphabet size, the algorithm computes the suffix array in $\mathcal{O}(n \log n)$ time, as follows. A difference-cover of size $\mathcal{O}(\sqrt{\log n})$ is computed in sub-logarithmic time. Then the construction of the sparse suffix array requires $\mathcal{O}(n \log n)$ time: $\mathcal{O}(n \log n)$ steps for *Multikey Quicksort*, $\mathcal{O}(n)$ steps for the construction of the reduced string, again $\mathcal{O}(n \log n)$ steps for a prefix-doubling algorithm, and $\mathcal{O}(n)$ steps for deriving the sparse suffix array from the suffix array of the reduced string. The construction of the complete suffix array from the sparse suffix array also requires $\mathcal{O}(n \log n)$ time: $\mathcal{O}(n \log n)$ steps for *Multikey Quicksort*, $\mathcal{O}(\log n)$ steps for the computation of a lookup table to implement the function Δ, and again $\mathcal{O}(n \log n)$ steps for the comparison-based sorting.

The space requirements are less than for the previous $\mathcal{O}(n \log n)$ time algorithms of Manber and Myers [96] or Larsson and Sadakane [90]. The input string again requires n bytes and the suffix array n words, but the auxiliary space requirements are only $\mathcal{O}(n / \log n)$ words, which are used for the sparse suffix array, the sparse rank array, and for the construction of these data structures.

8.3.2 Suffix array construction in linear time

The development of the three linear-time algorithms seems to be inspired by different previous algorithms. The *skew* algorithm of Kärkkäinen and Sanders [71] uses a difference-cover like the *difference-cover* algorithm of Burkhardt and Kärkkäinen; the *odd–even* algorithm of Kim *et al.* [80] adopts the *odd-and-even* scheme that has been previously used by Farach and Muthukrishnan [46], Farach [45], and Farach *et al.* [47] for suffix tree construction; and the *smaller–larger* algorithm of Ko and Aluru [85] classifies each suffix as type S or L, similar to the classification of Itoh and Tanaka's *two-stage* algorithm (see Section 8.2.2.1).

All three algorithms follow different divide-and-conquer schemes, but share the basic framework. They divide the suffixes into two groups, recursively construct the suffix array of the reduced string of the first group, derive the sparse suffix array of suffixes in the first group, use that sparse suffix array to determine the sparse suffix array of the other suffixes, and finally merge the two sparse suffix arrays to obtain the total ordering of all suffixes, namely the suffix array.

8.3.2.1 The *skew* algorithm of Kärkkäinen and Sanders

The *skew* algorithm of Kärkkäinen and Sanders [71] uses a difference cover D modulo 3 with $D = \{1, 2\}$. It first constructs the sparse suffix array $sa^{(1,2)}$ of suffix numbers $s \in [1, n]$ with $s \bmod 3 \in \{1, 2\}$. Then it passes the ordering of suffixes s in $sa^{(1,2)}$ with $s \bmod 3 = 1$ on to the sparse suffix array $sa^{(0)}$ that contains the predecessor suffixes $s^{(0)}$ with $s^{(0)} \bmod 3 = 0$ (all suffixes not contained in $sa^{(1,2)}$), and finally merges $sa^{(0)}$ and $sa^{(1,2)}$. For $k \in [0, 2]$, let $n^{(k)}$ be the number of suffixes at the modulo k positions: $n^{(0)} = \lceil n/3 \rceil$, $n^{(1)} = \lceil (n-1)/3 \rceil$, and $n^{(2)} = \lceil (n-2)/3 \rceil$. The size of $sa^{(1,2)}$ is $n^{(1)} + n^{(2)}$, and the size of $sa^{(0)}$ is $n^{(0)}$.

Constructing the sparse suffix arrays. The construction of the sparse suffix array $sa^{(1,2)}$ proceeds similar to the difference cover algorithm. It first sorts the suffixes in $sa^{(1,2)}$ with respect to their three leading characters, resulting in a 3-bucket segmentation of $sa^{(1,2)}$. According to the 3-bucket segmentation, the algorithm assigns the sparse bucket number $bnr^{(1,2)}[s]$ to each suffix s in $sa^{(1,2)}$.

The reduced string $t^{(1,2)}$ of length $n^{(1)} + n^{(2)}$ is computed such that the relative lexicographical order of the suffixes of $t^{(1,2)}$ corresponds to the relative lexicographical order of the suffixes in $sa^{(1,2)}$. The partial function $\mu^{(1,2)}$ bijectively maps the suffix numbers in $sa^{(1,2)}$ onto the positions $[1, n^{(1)} + n^{(2)}]$ of $t^{(1,2)}$ such that, for all $s \in [1, n]$ with $s \bmod 3 \in \{1, 2\}$,

$$\mu^{(1,2)}(s) = \begin{cases} \dfrac{s+2}{3} & \text{if } s \bmod 3 = 1, \\ \left\lceil \dfrac{n}{3} \right\rceil + \dfrac{s+1}{3} & \text{if } s \bmod 3 = 2. \end{cases}$$

That is, the suffix numbers $s \in [1, n]$ with $s \bmod 3 = 1$ are monotonically increasingly mapped onto $[1, n^{(1)}]$, and the suffix numbers $s \in [1, n]$ with $s \bmod 3 = 2$ are monotonically increasingly mapped onto $[n^{(1)} + 1, n^{(1)} + n^{(2)}]$. Moreover, let $\mu^{(1,2)(-1)}$ be the inverse mapping, which maps the positions $[1, n^{(1)} + n^{(2)}]$ of the reduced string $t^{(1,2)}$ onto the suffix numbers $s \in [1, n]$ with $s \bmod 3 \in \{1, 2\}$.

The algorithm constructs the reduced string $t^{(1,2)}$ of length $n^{(1)} + n^{(2)}$,

$$t^{(1,2)}[i] := bnr^{(1,2)}[\mu^{(1,2)(-1)}(i)] \quad \text{for all } i \in [1, n^{(1)} + n^{(2)}].$$

That is, $t^{(1,2)} = bnr^{(1,2)}[1], bnr^{(1,2)}[4], \ldots, bnr^{(1,2)}[3n^{(1)} - 2], bnr^{(1,2)}[2], bnr^{(1,2)}[5], \ldots, bnr^{(1,2)}[3n^{(2)} - 1]$. Then it recursively constructs the suffix array $sa(t^{(1,2)})$ of the reduced string $t^{(1,2)}$ and derives the sparse suffix array $sa^{(1,2)}$ from $sa(t^{(1,2)})$,

$$sa^{(1,2)}[i] = \mu^{(1,2)(-1)}(sa(t^{(1,2)})[i]) \quad \text{for all } i \in [1, n^{(1)} + n^{(2)}].$$

The second sparse suffix array $sa^{(0)}$ is constructed in linear time by a procedure like *Counting Sort*: The suffixes $i \in [1, n]$ with $i \bmod 3 = 0$ are sorted according to the primary sort key $t[i]$ and secondary sort key $R^{(1,2)}[i+1]$, resulting in $sa^{(0)}$.

Merging both sparse suffix arrays. The two sorted sparse suffix arrays $sa^{(0)}$ and $sa^{(1,2)}$ are merged from left to right, yielding the complete suffix array sa. Let $front^{(0)}$ be the current front of $sa^{(0)}$, $front^{(1,2)}$ the current front of $sa^{(1,2)}$, and $front$ the current front of sa, initially $front^{(0)} = front^{(1,2)} = front = 1$. The merging procedure compares the suffixes that correspond to the suffix numbers $sa^{(0)}[front^{(0)}]$ and $sa^{(1,2)}[front^{(1,2)}]$, assigns the suffix number of the lexicographically smaller suffix to $sa[front]$, and advances the respective front positions. This procedure is repeated until the end of $sa^{(0)}$ or $sa^{(1,2)}$ is reached. Then the remaining suffix numbers of the other sparse suffix array are directly copied to the not yet determined positions at the end of sa.

Let $s^{(0)} = sa^{(0)}[front^{(0)}]$ and $s^{(1,2)} = sa^{(1,2)}[front^{(1,2)}]$ be the suffixes at the current front positions. The merging procedure distinguishes two cases:

(i) If $s^{(1,2)} \bmod 3 = 1$, then $t[s^{(0)}, n] < t[s^{(1,2)}, n]$ if and only if $(t[s^{(0)}], R^{(1,2)}[s^{(0)}+1]) < (t[s^{(1,2)}], R^{(1,2)}[s^{(1,2)}+1])$;

(ii) If $s^{(1,2)} \bmod 3 = 2$, then $t[s^{(0)}, n] < t[s^{(1,2)}, n]$ if and only if $(t[s^{(0)}], t[s^{(0)}+1], R^{(1,2)}[s^{(0)}+2]) < (t[s^{(1,2)}], t[s^{(1,2)}+1], R^{(1,2)}[s^{(1,2)}+2])$.

Thereby the first element of a tuple is the primary sort key, the second is the secondary sort key, and the third is the ternary sort key, where applicable.

Time and space consumption. For an integer alphabet $[1, n]$, the following steps all require linear time: the initial sorting of the suffixes with respect to their three leading characters, the assignment of the sparse bucket numbers, the construction of the reduced string $t^{(1,2)}$, the derivation of the sparse suffix array $sa^{(1,2)}$ from $sa(t^{(1,2)})$, the construction of $sa^{(0)}$ from $sa^{(1,2)}$, and the merging of $sa^{(0)}$ and $sa^{(1,2)}$. Combined with the recursive construction time of $sa(t^{(1,2)})$, we obtain $T_{\text{skew}}(n) = \mathcal{O}(n) + T_{\text{skew}}(\lceil 2n/3 \rceil)$ running time for $n \geq 3$, and $T_{\text{skew}}(n) = \mathcal{O}(1)$ for $n < 3$. This recursion can be solved to $T_{\text{skew}}(n) = \mathcal{O}(n)$.

Kärkkäinen and Sanders's implementation of the skew algorithm [72] requires a significant amount of working space. The input sequence is a string over an integer alphabet. It requires n words, instead of n bytes for a standard ASCII input. Additionally, in each recursive call, two auxiliary arrays of length $2n/3$ are allocated, one for the reduced string $t^{(1,2)}$ and one for the sparse suffix array $sa^{(1,2)}$. The other auxiliary data structures are only used temporarily; their space requirements are negligible compared to the recursively collected space. Therefore, the algorithm accumulates up to $S_{\text{skew}}(n) = 2n + S_{\text{skew}}(\lceil 2n/3 \rceil)$ words of working space for $n \geq 3$, and $S_{\text{skew}}(n) = \mathcal{O}(1)$ for $n < 3$. We unroll this recursion and observe that it terminates after at most $\log_{3/2} n$ recursive calls. This implies a maximum space consumption of $S_{\text{skew}}(n) = \sum_{i=0}^{\log_{3/2} n} 2n(2/3)^i$ words. For large n, this can be approximated by $S_{\text{skew}}(n) \approx 2n \sum_{i=0}^{\infty} (2/3)^i$. Since $0 \leq 2/3 < 1$, we can use $\sum_{i=0}^{\infty} x^i = 1/(1-x)$, a common equation for the geometric series, and obtain $S_{\text{skew}}(n) \approx 2n \sum_{i=0}^{\infty} (2/3)^i = 2n/(1-2/3) = 6n$. Therefore, the total space requirements are up to $6n$ words.

Na [111], however, presented a variant of the *skew* scheme that allows the linear-time construction of suffix arrays in $o(n \log n)$ bits of auxiliary space.

8.3.2.2 The *odd–even* algorithm of Kim *et al.*

The *odd–even* algorithm of Kim *et al.* [80] first constructs the sparse suffix array of the odd suffix numbers, passes the ordering of the *odd suffixes* onto the sparse suffix array of the predecessor suffixes starting at the even positions, and finally merges both sparse suffix arrays.

We first present some notations and tools for the implementation of the algorithm. The sparse *odd suffix array* sa^{o} of length $n^{\mathrm{o}} = \lceil n/2 \rceil$ represents the lexicographically ordered suffixes starting at the odd positions, and the corresponding LCP array lcp^{o} of length $n^{\mathrm{o}} - 1$ contains the longest common prefix information of consecutive suffixes in sa^{o}. The sparse *even suffix array* sa^{e} of length $n^{\mathrm{e}} = \lfloor n/2 \rfloor$ analogously represents the ordered suffixes starting at the even positions, and lcp^{e} is the respective LCP array of length $n^{\mathrm{e}} - 1$. Let $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}])$ denote the length of the longest common prefix of all suffixes $t[sa^{\mathrm{o}}[i], n]$ with $i \in [l^{\mathrm{o}}, r^{\mathrm{o}}]$ and $\mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$ analogously the length of the longest common prefix of all suffixes $t[sa^{\mathrm{e}}[j], n]$ with $j \in [l^{\mathrm{e}}, r^{\mathrm{e}}]$. Let $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$ denote the length of the longest common prefix of all suffixes with a suffix number in one of the two buckets $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ or $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$, $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]) = \min\{\mathrm{lcp}(sa^{\mathrm{o}}[i], sa^{\mathrm{e}}[j]) : i \in [l^{\mathrm{o}}, r^{\mathrm{o}}] \text{ and } j \in [l^{\mathrm{e}}, r^{\mathrm{e}}]\}$.

An important tool for the *odd–even* algorithm is a data structure that supports constant time *range minimum queries*. Given an array A of size n whose elements are integers in $[0, n-1]$ and any two indices $l, r \in [1, n]$ with $l \leq r$, then a range minimum query $\mathrm{rangeMinAt}(A, l, r)$ finds the smallest index i such that $A[i] = \min_{l \leq j \leq r} A[j]$. This can also be used to find the smallest value $\mathrm{rangeMin}(A, l, r)$ within a given range $[l, r]$ of A, $\mathrm{rangeMin}(A, l, r) = A[\mathrm{rangeMinAt}(A, l, r)]$. The *odd–even* algorithm uses the range minimum query to compute the length of the longest common prefix for a range of suffixes in the odd or, alternatively, in the even suffix array in constant time, $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]) = \mathrm{rangeMin}(lcp^{\mathrm{o}}, l^{\mathrm{o}}, r^{\mathrm{o}} - 1)$ and $\mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]) = \mathrm{rangeMin}(lcp^{\mathrm{e}}, l^{\mathrm{e}}, r^{\mathrm{e}} - 1)$.

A simple solution for the range minimum query problem was given by Bender and Farach-Colton [18], and Sadakane [124] uses range minimum queries to compute longest common prefixes of suffixes in compressed suffix arrays. For the *odd–even* algorithm, Kim *et al.* [80] use a modification of the range minimum algorithm of Berkman and Vishkin [24]. For an in-depth study, we refer to Alstrup *et al.*'s survey of the least common ancestor problem [5], which is intimately connected with the range minimum problem.

Constructing the odd and the even suffix array. The odd suffix array is recursively constructed. The algorithm first sorts the suffixes of sa^{o} with respect to their two leading characters, resulting in a 2-bucket segmentation of sa^{o}. According to the 2-bucket segmentation, it assigns the sparse bucket number $bnr^{\mathrm{o}}[s]$ to each suffix $s \in [1, n]$ with $s \bmod 2 = 1$.

The reduced string t^{o} of length n^{o} is computed such that the relative lexicographical order of the suffixes of t^{o} corresponds to the relative lexicographical order of the suffixes in sa^{o}. The partial function μ^{o} bijectively maps the suffix numbers in sa^{o} onto the positions

$[1, n^{\mathrm{o}}]$ of the reduced string t^{o} such that

$$\mu^{\mathrm{o}}(s) \;=\; \frac{s+1}{2} \quad \text{for all } s \in [1,n] \text{ with } s \bmod 2 = 1.$$

That is, the odd suffix numbers are monotonically increasingly mapped onto $[1, n^{\mathrm{o}}]$. Moreover, let $\mu^{\mathrm{o}(-1)}$ be the inverse mapping, which maps the positions $[1, n^{\mathrm{o}}]$ of the reduced string t^{o} onto the suffix numbers $s \in [1, n]$ with $s \bmod 2 = 1$.

The algorithm constructs the reduced string t^{o} of length n^{o}:

$$t^{\mathrm{o}}[i] := bnr^{\mathrm{o}}[\mu^{\mathrm{o}(-1)}(i)] \;\; (= bnr^{\mathrm{o}}[2\,i-1]) \quad \text{for all } i \in [1, n^{\mathrm{o}}].$$

That is, $t^{\mathrm{o}} = bnr^{\mathrm{o}}[1], bnr^{\mathrm{o}}[3], \ldots, bnr^{\mathrm{o}}[2n^{\mathrm{o}}-1]$. Then it recursively constructs the suffix array $sa(t^{\mathrm{o}})$ of the reduced string and the corresponding LCP array $lcp(t^{\mathrm{o}})$, and finally derives sa^{o} from $sa(t^{\mathrm{o}})$ and lcp^{o} from $lcp(t^{\mathrm{o}})$ such that, for all $i \in [1, n^{\mathrm{o}}]$,

$$sa^{\mathrm{o}}[i] \;=\; \mu^{\mathrm{o}(-1)}(sa(t^{\mathrm{o}})[i]) \;\; (= 2(sa(t^{\mathrm{o}})[i]) - 1)$$

and, for all $i \in [1, n^{\mathrm{o}} - 1]$,

$$lcp^{\mathrm{o}}[i] = \begin{cases} 2\,lcp(t^{\mathrm{o}})[i] + 1 & \text{if } t[sa^{\mathrm{o}}[i] + 2\,lcp(t^{\mathrm{o}})[i]] = t[sa^{\mathrm{o}}[i+1] + 2\,lcp(t^{\mathrm{o}})[i]] \\ 2\,lcp(t^{\mathrm{o}})[i] & \text{otherwise.} \end{cases}$$

Finally, sa^{e} and lcp^{e} are constructed from sa^{o} and lcp^{o}. The suffixes $s^{\mathrm{e}} \in [1, n]$ with even suffix number, $s^{\mathrm{e}} \bmod 2 = 0$, are sorted according to the primary sort key $t[s^{\mathrm{e}}]$ and secondary sort key $R^{\mathrm{o}}[s^{\mathrm{e}}+1]$, resulting in sa^{e}. Afterwards, the corresponding LCP array lcp^{e} of length $n^{\mathrm{e}} - 1$ is computed:

$$lcp^{\mathrm{e}}[i] = \begin{cases} 0 & \text{if } t[sa^{\mathrm{e}}[i]] \neq t[sa^{\mathrm{e}}[i+1]] \\ 1 + \mathrm{lcp}(t[sa^{\mathrm{e}}[i]+1, n], t[sa^{\mathrm{e}}[i+1]+1, n]) & \text{otherwise,} \end{cases}$$

for all $i \in [1, n^{\mathrm{e}} - 1]$, where $sa^{\mathrm{e}}[i] + 1$ and $sa^{\mathrm{e}}[i+1] + 1$ are odd suffix numbers. Let $g^{\mathrm{o}} = R^{\mathrm{o}}[sa^{\mathrm{e}}[i]+1]$ and $h^{\mathrm{o}} = R^{\mathrm{o}}[sa^{\mathrm{e}}[i+1]+1]$ be the positions of these suffix numbers in sa^{o}, then the algorithm computes $\mathrm{lcp}(t[sa^{\mathrm{e}}[i]+1, n], t[sa^{\mathrm{e}}[i+1]+1, n]) = \mathrm{lcp}(sa^{\mathrm{o}}[g^{\mathrm{o}}, h^{\mathrm{o}}])$ by a range minimum query on lcp^{o}, $\mathrm{lcp}(sa^{\mathrm{o}}[g^{\mathrm{o}}, h^{\mathrm{o}}]) = \mathrm{rangeMin}(lcp^{\mathrm{o}}, g^{\mathrm{o}}, h^{\mathrm{o}} - 1)$.

Merging the odd and the even suffix array. A brief explanation of the general merging strategy can be given based on the *lcp-interval trees* of Abouelhoda *et al.* [1, 2]: The merging of the two sparse suffix arrays is a kind of breadth-first merging of their implicit *lcp-interval trees*.

The *odd–even* algorithm only processes *non-extendable buckets*. A *non-extendable ℓ-bucket* $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ of the odd suffix array contains *all* odd suffix numbers $s \in [1, n]$ with $t[s, s+\ell-1] = t[sa^{\mathrm{o}}[l^{\mathrm{o}}], sa^{\mathrm{o}}[l^{\mathrm{o}}] + \ell - 1]$, and a *non-extendable ℓ-bucket* $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$ of the even suffix array contains *all* even suffix numbers $s \in [1, n]$ with $t[s, s+\ell-1] = t[sa^{\mathrm{e}}[l^{\mathrm{e}}], sa^{\mathrm{e}}[l^{\mathrm{e}}]+\ell-1]$. The non-extendable ℓ-buckets $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ and $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$ are *ℓ-coupled* if all suffixes of both buckets share the same prefix of length ℓ; $(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$ is

called an *ℓ-coupled pair*. Otherwise the buckets are *ℓ-uncoupled*. If $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ and $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$ are ℓ-coupled, then their suffix numbers form an ℓ-bucket $sa[l^{\mathrm{o}} + l^{\mathrm{e}} - 1, r^{\mathrm{o}} + r^{\mathrm{e}}]$ of the complete suffix array. The length of the longest common prefix of all suffixes in an ℓ-coupled pair $(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$ is denoted by $\lambda := \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$. Moreover, $\theta := \min\{\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]), \mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])\}$ is an upper bound for λ, called the LCP limit of the coupled pair $(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$. Note that $\ell \leq \lambda \leq \theta$.

There are two further auxiliary data structures: the array ptr^{o} of length n^{o} and the array ptr^{e} of length n^{e}. For each $i^{\mathrm{o}} \in [1, n^{\mathrm{o}}]$, $ptr^{\mathrm{o}}[i^{\mathrm{o}}]$ is defined if $sa^{\mathrm{o}}[i^{\mathrm{o}}]$ is an entry of an uncoupled bucket or the last entry of a coupled bucket:

- If $sa^{\mathrm{o}}[i^{\mathrm{o}}]$ is an entry of an uncoupled bucket $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$, $i^{\mathrm{o}} \in [l^{\mathrm{o}}, r^{\mathrm{o}}]$, then $ptr^{\mathrm{o}}[i^{\mathrm{o}}]$ stores the rightmost position r^{e} of a bucket $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$ such that

  $$\mathrm{lcp}(sa^{\mathrm{o}}[i^{\mathrm{o}}], sa^{\mathrm{e}}[r^{\mathrm{e}}]) \geq \mathrm{lcp}(sa^{\mathrm{o}}[i^{\mathrm{o}}], sa^{\mathrm{e}}[j^{\mathrm{e}}]) \text{ for all } j^{\mathrm{e}} \in [1, n^{\mathrm{e}}].$$

  Among all suffixes in the even suffix array, $t[sa^{\mathrm{e}}[r^{\mathrm{e}}], n]$ shares the longest common prefix with $t[sa^{\mathrm{o}}[i^{\mathrm{o}}], n]$.

- If $sa^{\mathrm{o}}[i^{\mathrm{o}}]$ is the last entry of a bucket $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ ($i^{\mathrm{o}} = r^{\mathrm{o}}$) coupled with $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$, then $ptr^{\mathrm{o}}[i^{\mathrm{o}}] := r^{\mathrm{e}}$.

The array ptr^{e} is analogously defined.

The merging procedure. For each position $i^{\mathrm{o}} \in [1, n^{\mathrm{o}}]$, the correct target position i of the suffix number $sa^{\mathrm{o}}[i^{\mathrm{o}}]$ in the lexicographically sorted complete suffix array sa is computed. That is, i is the target position of i^{o} if and only if $sa^{\mathrm{o}}[i^{\mathrm{o}}] = sa[i]$. The target positions of sa^{e} are analogously defined. In fact, the algorithm determines the target positions for complete uncoupled buckets. Coupled buckets are repeatedly subdivided according to larger common prefixes until the sub-buckets become uncoupled such that the targets can be determined.

The algorithm successively performs up to n stages until the complete suffix array is constructed. In stage θ, it processes all coupled pairs with LCP limit θ. It starts with the coupled pair $(sa^{\mathrm{o}}[1, n^{\mathrm{o}}], sa^{\mathrm{e}}[1, n^{\mathrm{e}}])$, formed of the complete odd and even suffix array, in stage 0. For an ℓ-coupled pair $(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$ with LCP limit θ and longest common prefix of length λ, the algorithm determines the target positions in sa, where λ is computed in constant time, as we will show later. The algorithm distinguishes two cases:

(i) If $\lambda < \theta$, then all suffixes with a suffix number in $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ are lexicographically smaller than the suffixes with a suffix number in $sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$, or vice versa. The buckets are uncoupled.

  (i.1) If $t[sa^{\mathrm{o}}[l^{\mathrm{o}}] + \lambda] < t[sa^{\mathrm{e}}[l^{\mathrm{e}}] + \lambda]$, then $sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ contains the smaller suffixes. The respective target segments of the complete suffix array are determined by $sa[l^{\mathrm{o}} + l^{\mathrm{e}} - 1, r^{\mathrm{o}} + l^{\mathrm{e}} - 1] = sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]$ and $sa[r^{\mathrm{o}} + l^{\mathrm{e}}, r^{\mathrm{o}} + r^{\mathrm{e}}] = sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]$. The

corresponding segment in the LCP array is determined by $lcp[l^o+l^e-1, r^o+l^e-2] = lcp^o[l^o, r^o-1]$, $lcp[r^o+l^e-1] = \lambda$, and $lcp[r^o+l^e, r^o+r^e-1] = lcp^e[l^e, r^e-1]$. The algorithm also assigns r^e to $ptr^o[i^o]$ for all $i^o \in [l^o, r^o]$ and r^o to $ptr^e[j^e]$ for all $j^e \in [l^e, r^e]$.

(i.2) If $t[sa^o[l^o] + \lambda] > t[sa^o[l^e] + \lambda]$, then the targets are determined analogously.

(ii) If $\lambda = \theta$, then the λ-coupled buckets $sa^o[l^o, r^o]$ and $sa^e[l^e, r^e]$ are subdivided into $(\lambda+1)$-buckets. The right boundaries of the sub-buckets of $sa^o[l^o, r^o]$ are the positions $i^o \in [l^o, r^o - 1]$ with $lcp^o[i^o] = \lambda$, and the right boundaries of the sub-buckets of $sa^e[l^e, r^e]$ are the positions $j^e \in [l^e, r^e - 1]$ with $lcp^e[j^e] = \lambda$. The positions are computed by range minimum queries on the respective LCP arrays. Let $sa^o[l^o_1, r^o_1], sa^o[l^o_2, r^o_2], \ldots, sa^o[l^o_\beta, r^o_\beta]$ be the respective sub-buckets of $sa^o[l^o, r^o]$ ($lcp^o[r^o_g] = \lambda$ for all $g \in [1, \beta-1]$), and let $sa^e[l^e_1, r^e_1], sa^e[l^e_2, r^e_2], \ldots, sa^e[l^e_\gamma, r^e_\gamma]$ be the respective sub-buckets of $sa^e[l^e, r^e]$ ($lcp^e[r^e_h] = \lambda$ for all $h \in [1, \gamma-1]$). For all $g \in [1, \beta]$, let c^o_g be the $(\lambda+1)^{st}$ character of all suffixes in $sa^o[l^o_g, r^o_g]$, and let c^o_h be the $(\lambda+1)^{st}$ character of all suffixes in $sa^e[l^e_h, r^e_h]$ for all $h \in [1, \gamma]$.

The algorithm merges the lists of odd and even sub-buckets from left to right starting with $sa^o[l^o_1, r^o_1]$ and $sa^e[l^e_1, r^e_1]$. We describe a step of the merging procedure, which is iterated until one sub-bucket list becomes empty. Let the buckets $sa^o[l^o_g, r^o_g]$ with $g \in [1, \beta]$ and $sa^e[l^e_h, r^e_h]$ with $h \in [1, \gamma]$ be the current heads of the sub-bucket lists. The algorithm compares c^o_g and c^e_h.

(ii.1) If $c^o_g = c^e_h$, then the pair of buckets $(sa^o[l^o_g, r^o_g], sa^e[l^e_h, r^e_h])$ is $(\lambda+1)$-coupled and its target processing is postponed to stage $\theta_{g,h}$, where $\theta_{g,h}$ is the LCP limit of $(sa^o[l^o_g, r^o_g], sa^e[l^e_h, r^e_h])$. The algorithm assigns λ to $lcp[r^o_g + r^e_h]$ if $g < \beta$ or $h < \gamma$, r^e_h to $ptr^o[r^o_g]$ if $g < \beta$, and r^o_g to $ptr^e[r^e_h]$ if $h < \gamma$. The bucket $sa^o[l^o_g, r^o_g]$ is removed from the list of odd sub-buckets and $sa^e[l^e_h, r^e_h]$ from the list of even sub-buckets.

(ii.2) If $c^o_g < c^e_h$, then $sa^o[l^o_g, r^o_g]$ is $(\lambda+1)$-uncoupled and $sa[l^o_g + l^e_h - 1, r^o_g + l^e_h - 1] = sa^o[l^o_g, r^o_g]$. The corresponding LCP values are $lcp[l^o_g + l^e_h - 1, r^o_g + l^e_h - 2] = lcp^o[l^o_g, r^o_g - 1]$ and $lcp[r^o_g + l^e_h - 1] = \lambda$. The algorithm also assigns r^e_h to $ptr^o[i^o]$ for all $i^o \in [l^o_g, r^o_g]$. The bucket $sa^o[l^o_g, r^o_g]$ is removed from the list of odd sub-buckets.

(ii.3) If $c^o_g > c^e_h$, then $sa[l^e_h + l^o_g - 1, r^e_h + l^o_g - 1] = sa^e[l^e_h, r^e_h]$. The corresponding LCP values are $lcp[l^e_h + l^o_g - 1, r^e_h + l^o_g - 2] = lcp^e[l^e_h, r^e_h - 1]$ and $lcp[r^e_h + l^o_g - 1] = \lambda$. The algorithm also assigns r^o_g to $ptr^e[j^e]$ for all $j^e \in [l^e_h, r^e_h]$. The bucket $sa^e[l^e_h, r^e_h]$ is removed from the list of even sub-buckets.

If one sub-bucket list becomes empty, then the merging procedure stops and the algorithm copies the remaining buckets in the non-empty list to the respective target segment of sa.

The longest common prefix of a coupled pair. We now show how the algorithm computes the longest common prefix λ of all suffixes in a coupled pair $(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$ with LCP limit θ in stage θ. We have two base cases: $\theta = 0$ implies $\lambda = 0$, and $\theta = 1$ implies $\lambda = 1$ if the input string t is composed of at least two distinct characters. For $\theta > 1$, according to the definition of the LCP limit ($\theta := \min\{\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}]), \mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])\}$), we have $\theta = \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}])$ or $\theta = \mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}])$. Without loss of generality, we assume $\theta = \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}])$.

The key of the algorithm is to reduce the computation of λ to the computation of the length of the longest common prefix of two suffixes in sa^{o}, which is then performed in constant time by a range minimum query:

$$\begin{aligned}
\lambda &= \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]) && (8.3)\\
&= \min(\theta, \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}], sa^{\mathrm{e}}[r^{\mathrm{e}}])) && (8.4)\\
&= \min(\theta - 1, \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}] + 1, sa^{\mathrm{e}}[r^{\mathrm{e}}] + 1)) + 1 && (8.5)\\
&= \min(\theta - 1, \mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{o}+}], sa^{\mathrm{o}}[r^{\mathrm{e}+}])) + 1, && (8.6)
\end{aligned}$$

where $l^{\mathrm{o}+} = R^{\mathrm{e}}[sa^{\mathrm{o}}[l^{\mathrm{o}}] + 1]$ is the position of $sa^{\mathrm{o}}[l^{\mathrm{o}}] + 1$ in sa^{e}, and $r^{\mathrm{e}+} = R^{\mathrm{o}}[sa^{\mathrm{e}}[r^{\mathrm{e}}] + 1]$ is the position of $sa^{\mathrm{e}}[r^{\mathrm{e}}]+1$ in sa^{o}. Equality (8.3) holds from the definition of λ, equality (8.4) since $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}], sa^{\mathrm{e}}[r^{\mathrm{e}}]) < \theta$ implies $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]) = \mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}], sa^{\mathrm{e}}[r^{\mathrm{e}}])$ and $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}], sa^{\mathrm{e}}[r^{\mathrm{e}}]) \geq \theta$ implies $\mathrm{lcp}(sa^{\mathrm{o}}[l^{\mathrm{o}}, r^{\mathrm{o}}], sa^{\mathrm{e}}[l^{\mathrm{e}}, r^{\mathrm{e}}]) = \theta$, equality (8.5) since the suffixes in the coupled pair share at least the first character, and equality (8.6) from $sa^{\mathrm{o}}[l^{\mathrm{o}}] + 1 = sa^{\mathrm{e}}[l^{\mathrm{o}+}]$ and $sa^{\mathrm{e}}[r^{\mathrm{e}}] + 1 = sa^{\mathrm{o}}[r^{\mathrm{e}+}]$. Note that $R^{\mathrm{e}}[sa^{\mathrm{o}}[.] + 1]$ defines a kind of sparse ${}^{+}R$-array (see Definition 3.1): For each suffix number $sa^{\mathrm{o}}[i^{\mathrm{o}}]$ in the odd suffix array, $i^{\mathrm{o}} \in [1, n^{\mathrm{o}}]$, $R^{\mathrm{e}}[sa^{\mathrm{o}}[i^{\mathrm{o}}] + 1]$ stores the position of the successor suffix number $sa^{\mathrm{o}}[i^{\mathrm{o}}] + 1$ in the even suffix array. It is a cross-link between the two sparse suffix arrays. The statement analogously holds for $R^{\mathrm{o}}[sa^{\mathrm{e}}[.] + 1]$.

For $sa^{\mathrm{e}}[l^{\mathrm{o}+}]$, the algorithm finds a position ϕ of sa^{o} such that the suffixes $sa^{\mathrm{e}}[l^{\mathrm{o}+}]$ and $sa^{\mathrm{o}}[\phi]$ share a sufficiently long common prefix such that

$$\min(\theta - 1, \mathrm{lcp}(sa^{\mathrm{e}}[l^{\mathrm{o}+}], sa^{\mathrm{o}}[r^{\mathrm{e}+}])) = \min(\theta - 1, \mathrm{lcp}(sa^{\mathrm{o}}[\phi], sa^{\mathrm{o}}[r^{\mathrm{e}+}])). \qquad (8.7)$$

Let $sa^{\mathrm{e}}[x^{\mathrm{e}}, y^{\mathrm{e}}]$ be the bucket containing $sa^{\mathrm{e}}[l^{\mathrm{o}+}]$ after stage $\theta - 1$, $l^{\mathrm{o}+} \in [x^{\mathrm{e}}, y^{\mathrm{e}}]$. Then $\phi = ptr^{\mathrm{e}}[y^{\mathrm{e}}]$ satisfies equation (8.7) (see [80] for a proof). The algorithm computes

$$\mathrm{lcp}(sa^{\mathrm{o}}[\phi], sa^{\mathrm{o}}[r^{\mathrm{e}+}]) = \begin{cases} \mathrm{rangeMin}(lcp^{\mathrm{o}}, \phi, r^{\mathrm{e}+} - 1) & \text{if } \phi < r^{\mathrm{e}+} \\ n - r^{\mathrm{e}+} + 1 & \text{if } \phi = r^{\mathrm{e}+} \\ \mathrm{rangeMin}(lcp^{\mathrm{o}}, r^{\mathrm{e}+}, \phi - 1) & \text{if } \phi > r^{\mathrm{e}+}. \end{cases}$$

Finally, according to equations (8.3)–(8.7), we obtain $\lambda = \mathrm{lcp}(sa^{\mathrm{o}}[\phi], sa^{\mathrm{o}}[r^{\mathrm{e}+}]) + 1$.

The algorithm finds the rightmost position y^{e} of the bucket $sa^{\mathrm{e}}[x^{\mathrm{e}}, y^{\mathrm{e}}]$ containing the suffix number $sa^{\mathrm{e}}[l^{\mathrm{o}+}]$ in constant time. The complete merging procedure runs in linear time since the algorithm processes at most n coupled buckets and n suffixes.

Time and space consumption. For the construction of the suffix array of an input string of length n over an integer alphabet $[1, n]$, the running time $T_{\text{odd–even}}(n)$ of the algorithm is composed of the $\mathcal{O}(n) + T_{\text{odd–even}}(n/2)$ construction time of the odd suffix array, the linear-time construction of the even suffix array, and the linear-time merge of the two sparse suffix arrays. This together leads to $T_{\text{odd–even}}(n) = \mathcal{O}(n) + T_{\text{odd–even}}(n/2) = \mathcal{O}(n)$ for the complete suffix array construction.

The space requirements of the algorithm depend on the implementation. Besides n bytes for the input string and n words for the suffix array, a straightforward implementation would require auxiliary space for the arrays sa^{o}, sa^{e}, lcp^{o}, lcp^{e}, ptr^{e}, for the reduced string t^{o}, and for the data structure providing constant time range minimum computations. There are, however, more space-efficient implementations of the odd-even suffix array construction scheme. Kim *et al.*'s [78] approach works on fixed-sized alphabets and requires less space, but $\mathcal{O}(n \log\log n)$ construction time. In practice, though, it is faster than the linear-time *odd–even* algorithm. Moreover, Hon *et al.* [63] manage the suffix array construction with the *odd–even* scheme using only $\mathcal{O}(n)$ auxiliary bits.

8.3.2.3 The *smaller–larger* algorithm of Ko and Aluru

The *smaller–larger* algorithm of Ko and Aluru [85] also classifies the set of suffixes into two types, like the *skew* algorithm and the *odd–even* algorithm. The *smaller–larger* algorithm, however, partitions the suffixes based on the relative order of consecutive suffixes and not based on their starting positions. Similar to the *two-stage* algorithm of Itoh and Tanaka [67], which classifies the suffixes as type s or type l, the *smaller–larger* algorithm classifies the suffixes either as type S or type L. Alternatively, the suffix numbers are classified either as type S or type L.

Let $S := \{s \in [1, n-1] : t[s, n] < t[s+1, n]\}$ of size $n^S = |S|$ be the set of suffix numbers of type S that contains each suffix number $s \in [1, n-1]$ if and only if the suffix $t[s, n]$ is lexicographically smaller than its successor suffix $t[s+1, n]$. Let $L := [1, n] \setminus S$ of size $n^L = |L|$ be the set of suffix numbers of type L containing the suffix number of each suffix that is lexicographically larger than its successor. The algorithm uses a local property to efficiently determine the type of each suffix: A suffix number $s \in [1, n]$ is of type S if $t[s] < t[s+1]$ or if $t[s] = t[s+1]$ and the successor suffix number $s+1$ is of type S; otherwise it is of type L. The algorithm uses this property to assign all suffix numbers to either S or L by a right-to-left scan of the string.

Let sa^S be the sparse suffix array of size n^S of all suffix numbers of type S, and let sa^L be the sparse suffix array of size n^L of all suffix numbers of type L. The algorithm first constructs the smaller of the two sparse suffix arrays. Without loss of generality, we assume that there are fewer type S suffixes than type L suffixes, or rather, $n^S \leq n^L$. The algorithm first constructs the sparse suffix array sa^S and then the complete suffix array sa from sa^S.

Constructing the sparse suffix array of type *S* suffixes. Let $s_1, s_2, \ldots, s_{(n^S)}$ be the sorted list of type S suffix numbers with $s_1 < s_2 < \ldots < s_{(n^S)}$ (sorted with respect

to the numbers, not lexicographically). For each such suffix number s_i of type S with $i \in [1, n^S - 1]$, the prefix $t[s_i, s_{i+1}]$ is called the *S-prefix* of s_i and $t[s_{(n^S)}, n]$ the *S-prefix* of $s_{(n^S)}$. The algorithm sorts the type S suffixes with respect to their S-prefixes, resulting in a bucket segmentation of sa^S such that two type S suffixes $s_i, s_j \in S$ with $i, j \in [1, n^S]$ are element of the same ℓ-bucket if and only if they share the same S-prefix of length ℓ, $t[s_i, s_{i+1}] = t[s_j, s_{j+1}]$ with $\ell = s_{i+1} - s_i + 1 = s_{j+1} - s_j + 1$. Note that the bucket segmentation contains ℓ-buckets for different ℓ. We will show later how the S-prefixes are sorted. According to the bucket segmentation of sa^S, the algorithm assigns the sparse bucket number $bnr^S[s]$ to each suffix $s \in S$, representing the relative order of type S suffixes with respect to their S-prefixes: $t[s_i, s_{i+1}] \leq t[s_j, s_{j+1}]$ if and only if $bnr^S[s_i] \leq bnr^S[s_j]$ for all $i, j \in [1, n^S]$.

Then a reduced string t^S of length n^S is computed such that the relative lexicographical order of the suffixes of t^S corresponds to the relative lexicographical order of the suffixes in sa^S. The partial function μ^S bijectively maps the suffix numbers in S onto the positions $[1, n^S]$ of the reduced string t^S such that

$$\mu^S(s_i) \quad = \quad i \quad \text{for all } i \in [1, n^S].$$

That is, the suffix numbers of type S are monotonically increasingly mapped onto $[1, n^S]$. Moreover, let $\mu^{S(-1)}$ be the inverse mapping. The algorithm constructs the reduced string t^S of length n^S:

$$t^S[i] := bnr^S[\mu^{S(-1)}(i)] \ (= bnr^S[s_i]) \quad \text{for all } i \in [1, n^S].$$

That is, $t^S = bnr^S[s_1], bnr^S[s_2], \ldots, bnr^S[s_{(n^S)}]$.

Then it recursively constructs the suffix array $sa(t^S)$ of the reduced string t^S and derives the sparse suffix array sa^S from $sa(t^S)$,

$$sa^S[i] = \mu^{S(-1)}(sa(t^S)[i]) \quad \text{for all } i \in [1, n^S].$$

Sorting the *S*-prefixes. The algorithm sorts the S-prefixes in three phases, using a procedure similar to *MSD radix sort*.

1. First of all, the *S-distance* $\text{dist}^S(u)$ of a suffix number u is the distance to the closest predecessor suffix number of type S, $\text{dist}^S(u) := \min\{u - s \ : \ s < u \text{ and } s \in S\}$. The algorithm computes the S-distance for each suffix number $u \in [1, n]$, leaving it undefined if there is no type S suffix number smaller than or equal to u, $\text{dist}^S(u) := \perp$ for $u \in [1, s_1]$.

2. For each encountered S-distance Δ, a list $list_\Delta$ stores the suffix numbers $u \in [1, n]$ with $\text{dist}^S(u) = \Delta$. Each list is ordered by the first character of the respective suffixes.

3. The algorithm starts with the sparse suffix array $sa^S = s_1, s_2, \ldots, s_{(n^S)}$. It repeatedly performs bucket refinement steps for each S-distance Δ, starting from 1 up to the

maximal S-distance. In the Δ^{th} bucket refinement step, it scans $list_\Delta$ from left to right. For each suffix number u encountered, it moves the Δ-predecessor $u - \Delta$ to the front of its bucket and advances the front by one. After scanning $list_\Delta$, the suffixes of type S with the same prefix of length Δ are grouped together, resulting in a Δ-bucket segmentation of sa^S. After processing all lists, we obtain the desired bucket segmentation of sa^S, representing the order of the type S suffixes with respect to their S-prefixes.

Constructing the complete suffix array from the sparse suffix array of type *S* suffixes. Ko and Aluru construct the complete suffix array sa from sa^S in three phases, as follows.

1. All suffixes are first sorted according to their leading character, producing a 1-bucket segmentation of the suffix array sa. Furthermore, the suffix number n of the last suffix $t[n, n]$ is moved to the front of its bucket.

2. The sparse suffix array sa^S is scanned from right to left. For each suffix in sa^S, the algorithm moves its counterpart in sa to the current end of its bucket and shifts the current end by one position to the left. After scanning sa^S, all suffixes of type S are in their final positions.

3. The third phase determines the order of the L suffixes and moves them to their final position. The suffix array sa is scanned from left to right. For each suffix number $v \in [1, n]$ of type L, the algorithm moves the predecessor $v - 1$ to the current front of its bucket and advances the front by one position to the right. The suffix sorting process is completed after scanning the whole suffix array sa.

So far, we have shown how to build the complete suffix array sa via the sparse suffix array sa^S of the suffixes of type S. If the suffixes of type S are fewer than the suffixes of type L, however, the sparse suffix array sa^L of the type L suffix numbers and finally the complete suffix array is constructed using a symmetric procedure.

Time and space consumption. Let $T_{\mathrm{SL}}(n)$ denote the total running time of the *smaller–larger* algorithm for input strings of length n over an integer alphabet $[1, n]$. $T_{\mathrm{SL}}(n)$ decomposes into the running time of the separate phases. The following steps all require linear time: the computation of the suffixes of type S, the sorting with respect to their S-prefixes, and the mapping to the reduced string t^S. In addition, the recursive construction of the suffix array $sa(t^S)$ takes $T_{\mathrm{SL}}(n^S) \leq T_{\mathrm{SL}}(\lfloor n/2 \rfloor)$ time. The derivation of the sparse suffix array sa^S from $sa(t^S)$ and the three phases for the construction of the complete suffix array from $sa(t^S)$ again require linear time. Altogether, this leads to $T_{\mathrm{SL}}(n) \leq \mathcal{O}(n) + T_{\mathrm{SL}}(\lceil n/2 \rceil)$ for $n \geq 2$ and $T_{\mathrm{SL}}(n) = \mathcal{O}(1)$ for $n < 2$, which can be solved to $T_{\mathrm{SL}}(n) = \mathcal{O}(n)$.

The algorithm has different space requirements for the separate sub-procedures. Among all mentioned subroutines, the sorting of the S-prefixes, particularly the construction of the S-distance lists, is the most space-consuming part of Ko and Aluru's implementation.

Therefore, the space analysis concentrates on that sub-procedure, as follows. For an integer alphabet with $\Sigma = [1, n]$, the construction of the S-distance lists requires $3n$ words: the integer array for the S-distances, an integer array for the S-distance lists, and a temporary array for a stable counting sort of the lists, each consume n words. Moreover, Ko and Aluru suggest to use bit arrays to mark the bucket boundaries and the suffix numbers of type S: two bit arrays of size n and one of size $n/2$. Hence, the overall space requirements are $3n$ words plus $5n/2$ bits. This could be further reduced to $3n$ words if the most significant bit of the integer words is used for the marker bits. Moreover, for a small fixed-sized alphabet, Ko and Aluru reduce the space requirements to $2n$ words and $1.25n$ bits or, alternatively, to $2n$ words if the most significant bit of each integer word can be used as a marker bit.

9 The Bucket-Pointer Refinement Algorithm

We observed that the bucket refinement algorithms, in particular the *deep–shallow* algorithm, show faster practical running times for common real-world strings than the reduced string sorting algorithms (see also [7, 119]). For degenerated strings with large LCPs, however, *deep–shallow* performs poorly (see [31]).

Our aim was to design a new algorithm that is fast for common strings with small LCPs and for strings with highly variable LCPs, but it should also construct suffix arrays of degenerated strings in reasonable time. Our algorithm follows the depth-first bucket refinement scheme, which proved its efficiency for common strings, and combines it with a *pull* technique (see Chapter 8.1) using the following fact for an input string t of length n:

$$(t[sa[i], n] =_\ell t[sa[j], n] \wedge bptr[sa[i] + \ell] < bptr[sa[j] + \ell]) \Longrightarrow t[sa[i], n] < t[sa[j], n]$$

for $i, j, \ell \in [1, n]$. That is, if two suffixes with the same ℓ-length prefix are contained in the same ℓ-bucket, then their order is determined by the order of their ℓ-successors. Our strategy is to use the information of subdivided buckets as early as possible. We alternate refinement steps and updates of the bucket pointers such that the information about the subdivided buckets is used in the bucket refinement process as soon as this information becomes available.

In Section 9.1, we describe the basic algorithm, which is analysed regarding asymptotic running time complexity in Section 9.2. In Section 9.3, we present the implementation details including an advanced push method that enhances the basic algorithm. Section 9.4 contains use cases of our algorithm.

9.1 The basic algorithm

Our new *bucket-pointer refinement* (*bpr*) algorithm mainly consists of two simple phases. Given a parameter q (usually less than $\log n$), the suffixes are lexicographically sorted in the first phase, so that suffixes with the same q-length prefix are grouped together, forming a q-bucket segmentation $sa[l_1, r_1], sa[l_2, r_2], \ldots, sa[l_\beta, r_\beta]$ for some $\beta \in [1, n]$. Before entering the second phase, a pointer to its bucket $bptr[i]$ is computed for each suffix with suffix number $i \in [1, n]$, such that suffixes with the same q-length prefix share the same bucket pointer. In our descriptions and in our implementation, we use the position of the rightmost suffix in each bucket as bucket pointer. Recall the definition of bucket pointers

from Chapter 7.1, equation (7.2). We have

$$bptr[sa[i]] = r_k \quad \text{for each } i \in [l_k, r_k] \text{ and for each } k \in [1, \beta]. \tag{9.1}$$

In the second phase, the buckets containing suffixes with equal prefix are recursively refined. Let $sa[l, r]$ be an ℓ-bucket of the suffix array sa. Then the refinement procedure applies the ternary partitioning scheme of Bentley and McIlroy's *Ternary-Split Quicksort* [22]. The bucket $sa[l, r]$ is partitioned into three sub-buckets according to the bucket pointers at offset ℓ: a left, a middle, and a right sub-bucket. That is, for each suffix $sa[i]$ with $i \in [l, r]$, $bptr[sa[i] + \ell]$ is used as the sort key. The refinement procedure first selects a pivot sort key $p = bptr[sa[j] + \ell]$ for some $j \in [l, r]$. Then the suffixes $sa[i]$ in $sa[l, r]$ with smaller sort key, $bptr[sa[i] + \ell] < p$ with $i \in [l, r]$, are assigned to the left sub-bucket $sa[l_<, r_<]$, the suffixes with sort key equal to the pivot, $bptr[sa[i] + \ell] = p$, to the middle sub-bucket $sa[l_=, r_=]$, and the suffixes with larger sort key, $bptr[sa[i] + \ell] > p$, to the right sub-bucket $sa[l_>, r_>]$ ($l = l_<$, $r_< + 1 = l_=$, $r_= + 1 = l_>$, and $r_> = r$).

After partitioning the suffixes of $sa[l, r]$, the bucket pointers for the suffixes in $sa[l, r]$ are updated to conform with the refined bucket segmentation. For each suffix $sa[i]$ with $i \in [l, r]$, *bpr* assigns the right-most position of its refined sub-bucket to its bucket pointer $bptr[sa[i]]$, such that

$$bptr[sa[i]] = \begin{cases} r_< \text{ for all } i \in [l_<, r_<] \\ r_= \text{ for all } i \in [l_=, r_=] \\ r_> \text{ for all } i \in [l_>, r_>]. \end{cases}$$

Then each of the three sub-buckets that is not empty or singleton is partitioned recursively by calling the refinement procedure. We use the unmodified offset ℓ for the left and for the right sub-bucket since both remain ℓ-buckets, but use the increased offset $\ell + q$ for the middle sub-bucket $sa[l_=, r_=]$ since its suffixes share a common prefix of length $(\ell + q)$ and thus form an $(\ell + q)$-bucket. After termination of the algorithm, all buckets are singletons, sa is the lexicographically sorted suffix array, and *bptr* reflects the rank array or, alternatively, the inverse suffix array.

An example of the refinement procedure for the string $t =$ `DEBDEBDEA` with parameter $q = 2$ is shown in Figure 9.1. The top of the figure, below the input string, shows the suffix array sa segmented into buckets and the bucket pointer array *bptr* after phase 1 and after each further refinement step. The vertical lines in sa denote the bucket boundaries. The bucket that is going to be refined in the next step is overlined, and the bucket pointers that are used as sort keys during that next refinement step are drawn in bold face. Initially, there are three non-singleton buckets, which are then refined from left to right: the bucket $sa[2, 3]$ containing the suffix numbers of suffixes with the prefix `BD`, $sa[4, 6]$ containing the suffix numbers of suffixes with the prefix `DE`, and $sa[8, 9]$ containing the suffix numbers of suffixes with the prefix `EB`. We first refine the bucket $sa[2, 3]$ containing the suffix numbers 3 and 6 with respect to $\ell = 2$. The sort keys (drawn in bold face) are $sortkey(3) = bptr[3 + 2] = 9$ and $sortkey(6) = bptr[6 + 2] = 7$, where the sort key 9 is selected as pivot. After the partitioning, the bucket pointer for the suffix 3 is updated to

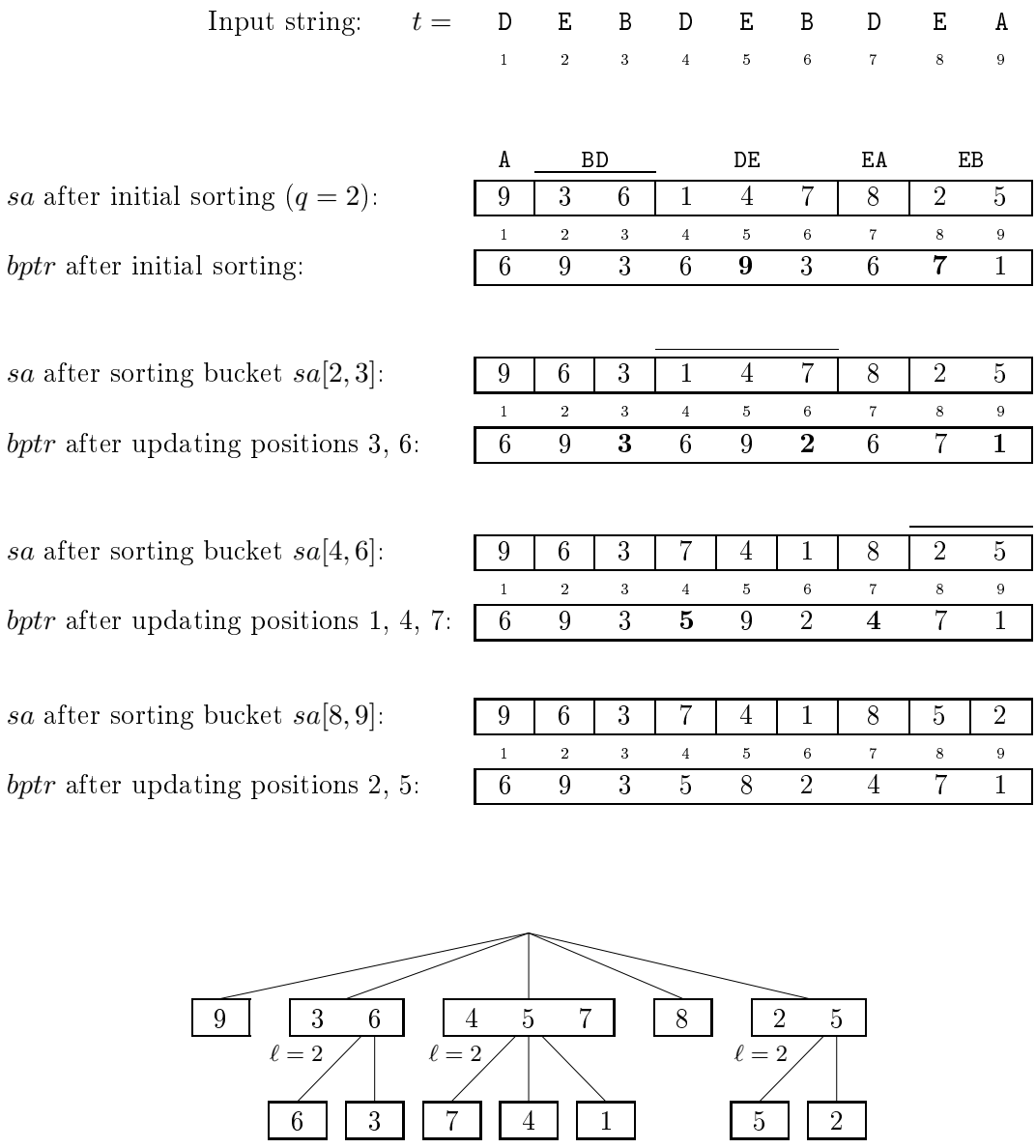

Figure 9.1: The bucket segmentation of the suffixes of the input string DEBDEBDEA and the respective bucket pointer array *bptr* after the initial sorting of the suffixes regarding prefixes of length $q = 2$ (2-bucket segmentation) and after each refinement step (top). Moreover, the corresponding ternary recursion tree (bottom).

$bptr[3] = 3$ and the bucket pointers for the suffix 6 to $bptr[6] = 2$. Then the refinement of the buckets $sa[4, 6]$ and $sa[8, 9]$ follows. The pivot is always the median sort key if the buckets are of odd size, or the next larger sort key if the buckets are of even size.

The bottom of Figure 9.1 shows the ternary recursion tree corresponding to the complete bucket refinement process. The inner nodes of the tree are the non-singleton buckets that have to be refined. The children of each such bucket correspond to the sub-buckets after a refinement step: a left child corresponds to a left sub-bucket, a middle child to a middle sub-bucket, and a right child to a right sub-bucket. Note that the first level of the recursion tree corresponds to the 2-bucket segmentation after the initial sorting of the suffixes regarding their prefixes of length $q = 2$.

Properties. The main improvement of our algorithm, compared to earlier algorithms performing bucket refinements, is that it benefits from the immediate use of subdivided bucket pointers after each refinement step. With increasing number of subdivided buckets, it becomes more and more likely that different bucket pointers can be used as sort keys during the refinement steps, such that the expected recursion depth decreases for the buckets refined later. The final position of a suffix number u in the current bucket is reached at the latest when $bptr[u + \ell]$ is unique for the current offset, that is, when the suffix number $u + \ell$ is contained in a singleton bucket $sa[bptr[u + \ell], bptr[u + \ell]]$ and thus has reached its final position.

Another improvement of our algorithm is that, in each recursive refinement step of a middle sub-bucket, ℓ can be increased by q. Hence, the recursion depth decreases by a factor of q, compared to algorithms performing characterwise radix steps.

Note that the algorithm can be applied to arbitrary ordered alphabets since it just uses comparisons to perform suffix sorting.

9.2 Analysis

So far we were not able to determine tight time bounds for our algorithm. The problem is that the algorithm quite arbitrarily uses the dependencies among suffixes. Hence, we only present lower and upper limits for the worst-case and expected-case time bounds.

The first phase of the algorithm can simply be performed in optimal linear time (see Section 9.3 for more details). For the second phase, we assume throughout the analysis that the algorithm finds the true median sort key in linear time, which can be performed by algorithms of Blum *et al.* [26], Schönhage *et al.* [126], or Dor and Zwick [43]. These methods, however, are not desirable for practical implementations since they increase the constant running time factors. Our implementation rather uses a pivot choice method that is directed to fast practical running time, instead of good worst-case time complexity.

9.2.1 Worst-case time bound

We first neglect that the expected recursion depth decreases for the buckets refined later.

Theorem 9.1. *Let t be a string of length n, and let q with $q \leq \log n$ be the common prefix length with respect to which our algorithm sorts the suffixes in phase 1. Then our algorithm constructs the suffix array of t in $\mathcal{O}(n^2/q)$ time.*

Proof. We assume that phase 1 is computed in linear time. The recursive refinement in phase 2 defines an implicit ternary recursion tree similar to the ternary search tree of Bentley and Sedgewick [23], which they used for the analysis of their string sorting algorithm. In the strict sense, we have one ternary recursion tree for each bucket generated by phase 1, but we include phase 1 to have only one recursion tree. Hence, the root is the only inner node that may have more than three children; it has as many children as there are buckets generated by phase 1 (see Figure 9.1). The refinement procedure starts with the offset $\ell = q$ for each bucket generated by phase 1. The ternary recursion tree branches into a left child for a left sub-bucket, a middle child for a middle sub-bucket, and a right child for a right sub-bucket. The middle child exists for each internal node since the corresponding middle sub-bucket contains at least the suffix that has the pivot sort key, but the left or the right child may not exist: The left child is empty if the corresponding left sub-bucket is empty, and the right child is empty if the corresponding right sub-bucket is empty.

We present a limit for the recursion depth by counting the number of edges, or branches, to child nodes on a path from the root to any leaf, where we distinguish between the middle branches and the left or right branches. Middle branches correspond to recursive refinements of middle sub-buckets, while the offset ℓ is incremented by q in each recursive call, starting with $\ell = q$. Recall that ℓ reflects the length of a common prefix of all suffixes in an ℓ-bucket, which is bounded by $n-1$. That is, ℓ has reached its maximum $n-1$ after encountering at most $\lceil n/q \rceil$ middle branches on the path from the root to any leaf, $n-1 < \lceil n/q \rceil q$. For each left or right branch, we observe that the size of its corresponding sub-bucket is at most half of the size of its father's bucket since the suffixes with the median sort key fall into the middle sub-bucket. Hence, the buckets are split up into singleton buckets after at most $\lceil \log_2 n \rceil$ left or right branches. Together, the total length of a path from the root to any leaf is bounded by $\lceil n/q \rceil + \lceil \log_2 n \rceil \in \mathcal{O}(n/q)$.

Moreover, the partitioning of a bucket takes linear time in the size of the bucket, and the buckets at any depth of the tree sum up to at most n since each suffix appears at most once in a bucket at any depth of the recursion tree. We multiply the linear partitioning time at any depth of the recursion tree by the maximum recursion depth of $\mathcal{O}(n/q)$ and add the linear computation time of phase 1 to get the $\mathcal{O}(n^2/q)$ worst-case time bound. □

Now, we focus on especially bad instances for our algorithm, in particular, strings maximising the recursion depth. Since the recursion depth is limited by the LCPs of suffixes to be sorted, periodic strings maximising the average LCP are especially hard strings for our algorithm.

A string $\mathtt{A}^n$ consisting of one repeated character maximises the average LCP and is therefore analysed as a particularly difficult input string. In the first phase of our algorithm the last $q-1$ suffixes $\{\mathtt{A}^{q-1}, \mathtt{A}^{q-2}, \ldots, \mathtt{AA}, \mathtt{A}\}$ are mapped to singleton buckets. One large

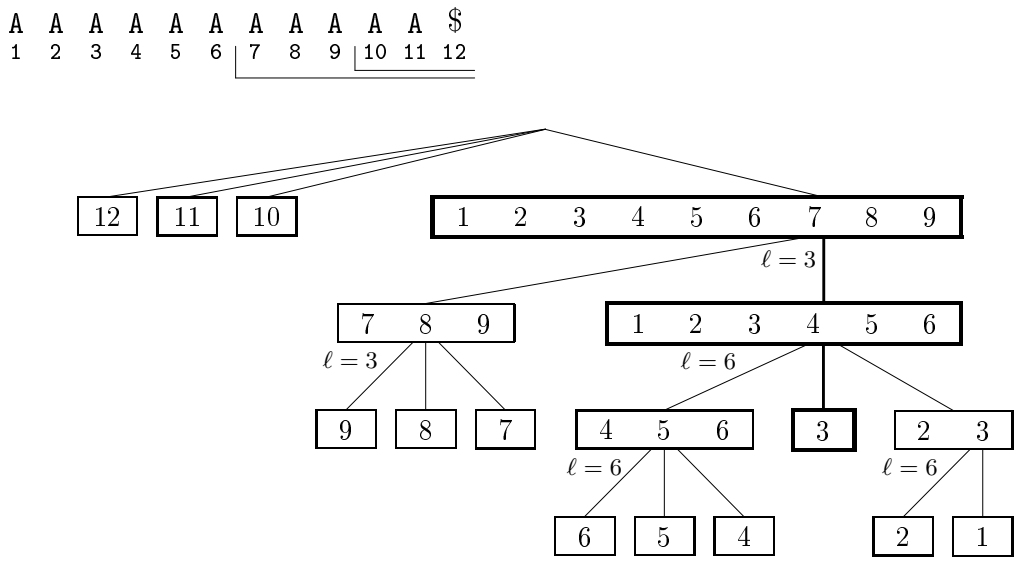

Figure 9.2: Recursion tree of the bucket refinements for the \$ extended input string AAAAAAAAAAA\$.

bucket containing all the other suffixes with prefix $\mathtt{A}^q$ remains to be refined. We assume that after partitioning in phase 2 the three sub-buckets are refined in ascending order of their size. In a recursive refinement step with offset ℓ, if the remaining large bucket contains at least 2ℓ suffixes, then it is subdivided into a left sub-bucket of size ℓ containing only suffixes with unique sort keys and into one larger middle sub-bucket containing the other suffixes with prefix $\mathtt{A}^{\ell+q}$, while ℓ is incremented by q for the recursive refinement of the middle sub-bucket, starting with $\ell = q$. If the remaining large bucket is of size *bsize* with $bsize < 2\ell$, then it is subdivided into a left sub-bucket of size $\lfloor bsize/2 \rfloor$, a singleton middle sub-bucket, and a right sub-bucket of size $\lceil bsize/2 \rceil - 1$. We assume that the left sub-bucket is recursively refined before the middle sub-bucket (small sub-buckets are refined first) such that, before the i^{th} recursive refinement of the middle sub-bucket, $\ell = q \cdot i$ suffixes are partitioned into a left sub-bucket and further into singleton buckets.

For $q = 3$, Figure 9.2 shows the ternary recursion tree of the refinement process for the string $\mathtt{A}^{11}$ extended with \$. Here, \$ belongs to the string. The suffixes 10, 11, and 12 are mapped to singleton buckets by the initial sorting and thus have unique bucket pointers: $bptr[10] = 3$, $bptr[11] = 2$, and $bptr[12] = 1$. Hence, for the offset $\ell = 3$, the suffixes 7, 8, and 9 have unique sort keys after the initial sorting: $sortkey(7) = bptr[7 + 3] = 3$, $sortkey(8) = bptr[8 + 3] = 2$, and $sortkey(9) = bptr[9 + 3] = 1$. Both groups are marked in the string. The bucket containing the suffixes $[1, 9]$ is then refined into the left sub-bucket of suffixes 7, 8, 9 and the middle sub-bucket of suffixes $[1, 6]$. Then, in one further refinement step, the suffixes 7, 8, and 9 are subdivided into singleton buckets. The remaining large bucket of suffixes $[1, 6]$ is refined with respect to the offset $\ell = 6$ such that the respective sort keys are the bucket pointers of the suffixes $[7, 12]$, which are unique:

$sortkey(k) = bptr[k+6] = 7-k$ for all $k \in [1,6]$. Finally, the two remaining non-singleton sub-buckets are refined: the left sub-bucket of suffixes 4, 5, 6 and the right sub-bucket of suffixes 1 and 2.

In the following, we separately analyse the so called *middle refinement thread* corresponding to the path of the recursion tree that always follows the middle sub-bucket until it is singleton, and the threads branching from the middle refinement thread. In Figure 9.2, the middle refinement thread is drawn in bold face. Without loss of generality, we assume that in the i^{th} recursive refinement of a middle sub-bucket on the middle refinement thread $\ell = q \cdot i$ suffixes are partitioned into a left sub-bucket. The repeated recursive refinement of the middle sub-buckets on the middle refinement thread proceeds until all suffixes are split off into left sub-buckets or until the middle sub-bucket is singleton, that is, until a recursion depth *recdepth* for the middle sub-buckets is reached, such that $n \leq q-1+\sum_{i=1}^{recdepth} q \cdot i = q-1+q(recdepth(recdepth+1)/2)$. Therefore, for the string $\mathtt{A}^n$, the recursion depth *recdepth* of the repeated middle sub-bucket refinement on the middle refinement thread is in $\Theta(\sqrt{n/q})$. Immediately after branching from the middle refinement thread, all sort keys of the suffixes in the corresponding sub-buckets are unique. Since the bucket size is limited by n, these buckets are split up into singleton buckets after at most $\lceil \log_2 n \rceil$ further branches in the recursion tree. Together, the total length of a path from the root to any leaf in this recursion tree is bounded by $\Theta(\sqrt{n/q}) + \lceil \log_2 n \rceil = \Theta(\sqrt{n/q})$ for $q \leq \log n$. We multiply the $\mathcal{O}(n)$ time for the refinement at any depth of the recursion tree by the recursion depth $\Theta(\sqrt{n/q})$ and add the linear time complexity of phase 1 to get the time bound $\Theta(n\sqrt{n/q})$ of our algorithm for the string $\mathtt{A}^n$. By setting $q = \log n$, we achieve a running time of $\mathcal{O}(n\sqrt{n/\log n}) = \mathcal{O}(n^{3/2}/\sqrt{\log n})$.

In general, since the partitioning time of a bucket is linear in its size, the running time of our algorithm is essentially given by summing up the sizes of the different non-singleton buckets that appear in the whole refinement computation. We identify two main parameters of the input strings that influence this sum: the initial distribution of q-length substrings (*q-gram profile*) and the average LCP. The initial distribution of q-length substrings influences the size and the number of buckets at the lower refinement levels with small offset $\ell = q$, where a few large buckets increase the requirement of further refinements. The average LCP is an indicator for the average recursion depth and thus for the total number of accumulated non-singleton buckets. The string $\mathtt{A}^n$ maximises both, the size of the initial buckets and the average LCP. Hence, we believe that the worst-case time bound for $\mathtt{A}^n$ also holds for all other strings.

Conjecture 9.2. *Let t be a string of length n, and let q with $q \leq \log n$ be the common prefix length with respect to which our algorithm sorts the suffixes in phase 1. Then our algorithm constructs the suffix array of t in $\mathcal{O}(n^{3/2}/\sqrt{q})$ time.*

9.2.2 Expected-case time bound

In practice, worst-case strings like $\mathtt{A}^n$ rarely appear. We are rather interested in the average construction time of our algorithm. Therefore, we analyse its expected construction time

for strings that are generated according to a Bernoulli model (i.e., symbols from the alphabet are generated independently) or a first order Markov model (i.e., the next symbol depends in a probabilistic sense only on the previous one).

Theorem 9.3. *Let t be a string of length n generated according to a Bernoulli model or according to a first order Markov model, and let q with $q \leq \log n$ be the common prefix length with respect to which our algorithm sorts the suffixes in phase 1. Then our algorithm constructs the suffix array of t in $\mathcal{O}(n \log n)$ expected time.*

Proof. We again use the implicit ternary recursion tree and follow the same line of argument as the proof of Theorem 9.1. The number of left or right branches on a path from the root to any leaf in the recursion tree is again bounded by $\log n$. Recall further that the number of middle branches is bounded by the maximal length of the common prefix of two suffixes of the input string divided by the parameter q. A simple consequence of a result by Apostolico and Szpankowski [9] and Szpankowski [139] is that the expected maximal length of such a longest common prefix is bounded by $\mathcal{O}(\log n)$. Hence, ℓ has reached its expected maximum after at most $\mathcal{O}(\log n/q)$ middle branches. Altogether, the expected maximal recursion depth is bounded by $\mathcal{O}(\log n)$: $\mathcal{O}(\log n)$ left or right branches and $\mathcal{O}(\log n)$ middle branches. We multiply the $\mathcal{O}(n)$ time for the refinement at any depth of the recursion tree by the expected maximal recursion depth of $\mathcal{O}(\log n)$ and add the linear computation time of phase 1 to get the $\mathcal{O}(n \log n)$ expected-case time bound of our algorithm, independent of the parameter q. □

We further choose $q = \log_{|\Sigma|} n$. There exist $|\Sigma|^q = n$ potential buckets, one for each possible prefix of length $\log_{|\Sigma|} n$ over the alphabet Σ. If we assume that the suffixes are independently assigned to the n buckets, then an expected-case analysis analogous to the analysis of bucket sort in [38, Section 8.4] would give a linear expected construction time for the Bernoulli model. The suffixes of a string are, however, not independent. Nevertheless, we believe that the expected construction time is linear for $q = \log n$.

Conjecture 9.4. *Let t be a string of length n over an alphabet Σ of constant size σ generated according to a Bernoulli model or a first order Markov model, and let $q = \log_\sigma n$ be the common prefix length with respect to which our algorithm sorts the suffixes in phase 1. Then our algorithm constructs the suffix array of t in $\mathcal{O}(n)$ expected time.*

9.2.3 Space requirements

Bpr requires more space than the lightweight algorithms *deep–shallow*, *cache*, *copy*, and *difference-cover*. The suffix array and the bucket pointer array each consume n integer words, and the input string n bytes. For an alphabet Σ of size σ, σ^q additional integer words are used for the bucket pointers of the initial bucket sort. Hence, for reasonable q, the total space requirements of *bpr* are between $9n$ and $10n$ bytes on computers with 4 byte integer words. However, for certain applications, such as the computation of the Burrows–Wheeler transform [32], the construction of the suffix array is just a byproduct, and the complete suffix array does not need to remain in memory.

9.3 Engineering and implementation for fast speed

In this section, we present more detailed descriptions of the two phases of the algorithm and enhance the second phase with a push method that is used in combination with the recursive refinement procedure.

9.3.1 Computing the initial bucket segmentation

We first define two specific terms: *range reduction* and *multiple character encoding*. Let t be a string of length n with character set Σ of size σ. *Range reduction* realises an order-preserving character mapping rk onto a contiguous segment of natural numbers. It is a monotone, bijective function, $\mathrm{rk} : \Sigma \to [0, \sigma - 1]$. The range reduced string $\mathrm{rk}(t)$ is defined by $\mathrm{rk}(t) := \mathrm{rk}(t[1]), \mathrm{rk}(t[2]), \ldots, \mathrm{rk}(t[n])$. A *multiple character encoding* for strings of length q is a monotone bijective function $code_q : \Sigma^q \to [0, \sigma^q - 1]$ such that for two strings w and w' of length q, $code_q(w) < code_q(w')$ if and only if w is lexicographically smaller than w'. For a given range reduction, such an encoding can easily be defined as $code_q(w) := \sum_{i=1}^{q} \sigma^{q-i} \mathrm{rk}(w[i])$. The encoding can be generalised to strings of length greater than q, by just encoding the first q characters. Given the encoding $code_q(u)$ for the suffix $t[u, n]$, $1 \le u < n$, the encoding for the successor suffix $t[u+1, n]$ can be derived by shifting away the first character of $t[u]$ and adding the range reduced value $\mathrm{rk}(t[u+q])$ of character $t[u+q]$:

$$code_q(u+1) = \sigma \left(code_q(u) \bmod \sigma^{q-1} \right) + \mathrm{rk}(t[u+q]). \tag{9.2}$$

We are now prepared to formulate phase 1. Our algorithm performs the initial sorting regarding the q-length prefixes of the suffixes by bucket sort, using $code_q(u)$ as the sort key for suffix $u \in [1, n]$ (assuming that t is extended with multiple \$s).

The bucket sorting is performed using two scans of the sequence, thereby successively computing $code_q(u)$ for each suffix using equation (9.2), or rather, the equivalent equation

$$code_q(u+1) = \sigma \left(code_q(u) - \sigma^{q-1} \cdot \mathrm{rk}(t[u]) \right) + \mathrm{rk}(t[u+q]) \tag{9.3}$$

to avoid the modulo operations, which are possibly time consuming.

There are σ^q buckets, one for each possible $code_q$. In the first scan, the size of each bucket is determined by counting the number of suffixes for each possible $code_q$. The outcome of this is used to compute the starting position for each bucket. These positions are stored in the array bkt, which is of size σ^q. During the second scan, the suffix numbers are mapped to the buckets, where suffix number u is mapped to bucket number $code_q(u)$.

After the bucket sort, the bucket pointer table $bptr$ can be computed by another scan of the sequence. Recall our definition of bucket pointers, equation (9.1). For each suffix $u \in [1, n]$, the bucket pointer $bptr[u]$ is simply the rightmost position of the bucket containing u, $bptr[u] = bkt[code_q(u) + 1] - 1$.

9.3.2 Recursively refining the buckets

We now give a more in-depth description of the three steps of the refinement procedure and present improvements to the basic approach.

Partitioning. In the refinement procedure, the suffixes are first partitioned with respect to a certain offset ℓ using the bucket pointer $bptr[u + \ell]$ as the sort key for the suffix number u. Our ternary partitioning algorithm is adapted from Lomuto's binary partitioning scheme [21, Column 10] (see also [38, Section 7.1]). We further tried other ternary partitioning algorithms that were suggested by Kiwiel [81], but ours performs best. Algorithm 9.1 (TernaryPartition) shows our partitioning procedure for an ℓ-bucket $sa[l, r]$ around the pivot p. The algorithm partitions the suffixes into three segments: a left, a middle, and a right segment. The suffixes with sort key equal to the pivot p are first moved to the middle segment and then further to the left segment, the suffixes with sort key smaller than p to the middle segment, and the suffixes with sort key larger than p to the right segment. The numbers $end_=$, $end_<$, i refer to the rightmost positions of the respective segments and are appropriately updated when the suffixes are moved: $end_=$ refers to the rightmost position of the left segment, $end_<$ to the rightmost position of the middle segment, and i to the rightmost position of the right segment. The movements are performed by swapping the suffixes as in the original *Quicksort*. Finally, VectorSwap (Algorithm 9.2) moves the suffixes of the left segment, with sort key equal to the pivot, to their final position by swapping them with the rightmost suffixes of the middle segment, ultimately producing the desired three sub-buckets. Figure 9.3 sketches the segments of the array immediately before and after the movement of suffixes by VectorSwap.

Our VectorSwap procedure improves upon the *vector swap* used by Bentley and McIlroy [22] for the ternary partitioning. Their procedure swaps the elements of two arrays A and B, each of length m, elementwise for each position $i \in [1, m]$: It assigns $A[i]$ to an auxiliary variable tmp, $B[i]$ to $A[i]$, and tmp to $B[i]$, altogether performing $3m$ assignment operations. The ordering of the elements is kept during the vector swap. Our vector swap reduces the number of assignment operations. Although it is quite simple, we have not seen that it has been previously used for the ternary partitioning. It first assigns the last element of the second array $B[m]$ to tmp. Then it performs the following steps for each $i \in [2, m]$ from m down to 2: $A[i]$ is assigned to $B[i]$ and $B[i-1]$ to $A[i]$. Finally, $A[1]$ is assigned to $B[1]$ and tmp to $A[1]$. Our vector swap keeps the order of elements that are moved from A to B, but alters the order of elements that are moved from B to A: The last element $B[m]$ is moved to the first position $A[1]$. The number of assignment operations, however, is only $2m + 1$, instead of $3m$ for Bentley and McIlroy's vector swap.

Moreover, we want to find a pivot sort key, hopefully near the true median, in constant time. Hoare [59] proposed using the median of a small sample of sort keys. We choose the pivot to be the median of nine sort keys for buckets larger than 10 000 suffixes and the median of three sort keys for smaller buckets. The median of three was proposed by Singleton [137], who suggested the median of the leftmost, the middle, and the rightmost element. We, however, observed that his selection sometimes causes a significant increase

Algorithm 9.1.
TernaryPartition(sa, ℓ, l, r, p)
  $i \leftarrow end_= \leftarrow end_< \leftarrow l - 1$
  while $i < r$ **do**
    $i \leftarrow i + 1$
    $sortkey \leftarrow bptr[sa[i] + \ell]$
    if $sortkey \leq p$ **then**
      $end_< \leftarrow end_< + 1$
      $tmp \leftarrow sa[i]$
      $sa[i] \leftarrow sa[end_<]$
      $sa[end_<] \leftarrow tmp$
      if $sortkey = p$ **then**
        $end_= \leftarrow end_= + 1$
        $sa[end_<] \leftarrow sa[end_=]$
        $sa[end_=] \leftarrow tmp$
      end if
    end if
  end while
  $swapsize \leftarrow \min\{end_= + 1 - l, end_< - end_=\}$
  VectorSwap$(l, l + swapsize - 1, end_<)$

Algorithm 9.2.
VectorSwap(g, h, z)
  $tmp \leftarrow sa[z]$
  while $g < h$ **do**
    $sa[z] \leftarrow sa[h]$
    $z \leftarrow z - 1$
    $sa[h] \leftarrow sa[z]$
    $h \leftarrow h - 1$
  end while
  $sa[z] \leftarrow sa[h]$
  $sa[h] \leftarrow tmp$

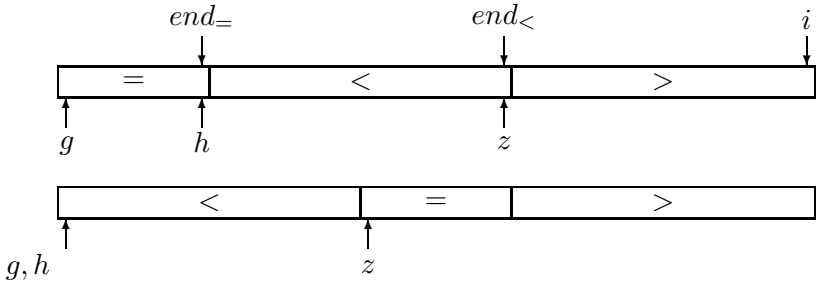

Figure 9.3: Partitioning suffixes before and after vector swap.

in running time for inputs with long repeated substrings. This is presumably due to the dependencies among suffixes in the refinement process such that the same suffixes are repeatedly encountered at the leftmost or rightmost bucket positions in successive refinement steps and are thus overrepresented in the choice of the median sort key. Hence, we choose the sort key of the middle element and the sort keys of the two elements that are one-fourth of the bucket size away from the bucket boundaries, preventing the mentioned effect. For the median of nine, we choose the sort keys analogously: at positions every one-tenth of the bucket size away from each other and away from the bucket boundaries.

For small buckets, our refinement algorithm falls back upon simple sorting routines: Buckets of size 2 or 3 are refined into singleton buckets by directly comparing the sort

keys, while ℓ is incremented by q. *Insertion Sort* is used for buckets of size up to 15.

Updating bucket pointers. The used procedure for the bucket pointer update depends on the refinement algorithm. If the suffixes in a small bucket of size 2 or 3 are directly compared and refined into singleton buckets, then the updated bucket pointer of a suffix $sa[i]$ is simply a backward link, $bptr[sa[i]] = i$.

After sorting the suffixes via *Insertion Sort*, the update is performed by a right-to-left scan of the current bucket. As long as the sort keys of consecutive suffixes are equal, they are located in the same refined bucket, and the bucket pointer is set to the rightmost position of the refined bucket. Note that the refined bucket positions are implicitly contained in the bucket pointer table $bptr$. The left pointer l of a bucket is the right pointer of the bucket directly to the left increased by one, and the right pointer r is simply the bucket pointer for the suffix $sa[l]$ at position l, $r = bptr[sa[l]]$, since the bucket pointer $bptr[u]$ of each suffix u points to the rightmost position of its bucket.

The ternary partitioning generates the sub-buckets including the leftmost and rightmost position of each such bucket. The strategy that we would follow to meet the best asymptotic running time is the following: In one scan of *each* of the three sub-buckets, the update procedure assign the rightmost position to the bucket pointers of all contained suffixes. During the practical engineering of our algorithm, however, we observed that the memory references to the bucket pointer array follow a quite arbitrary access pattern, resulting in many cache misses. Especially the write operations during the updates cause a delay in data access. Hence, in our practical implementation that refines the sub-buckets from left to right, we postpone the update of bucket pointers of suffixes in the left or right sub-buckets until they are singletons. We update the respective bucket pointers for the middle sub-bucket after the left sub-bucket has been completely refined.

Recursive Refinement. The recursive refinement procedure is usually called with an incremented offset $\ell + q$ for the middle sub-bucket. Note that, for a middle sub-bucket $sa[l_=, r_=]$ of $sa[l, r]$ containing each suffix $t[sa[i], n]$, $i \in [l_=, r_=]$, for which the ℓ-successor suffix $t[sa[i] + \ell, n]$ is also contained in $sa[l, r]$, the offset can be doubled. This is so because all suffixes contained in $sa[l, r]$ share a common prefix of length ℓ, and for each suffix $t[sa[i], n]$ in the middle sub-bucket, $i \in [l_=, r_=]$, there is also the ℓ-successor suffix $t[sa[i] + \ell, n]$ in its super-bucket $sa[l, r]$. Hence, all suffixes contained in $sa[l_=, r_=]$ share a prefix of length 2ℓ.

We add a further heuristic to avoid the unnecessary repeated sorting of buckets. For a bucket consisting of suffixes that all share a common prefix much larger than the current offset, many refinement steps may be performed without actually refining the bucket. This may continue until ℓ reaches the length of the common prefix. Therefore, if a bucket is not refined during a recursion step, we search for the lowest offset dividing the bucket. This is performed by just iteratively scanning the bucket pointers of the contained suffixes with respect to ℓ and incrementing ℓ by q after each run. As soon as a bucket pointer different from the others is met, the current ℓ is used to call the refinement procedure.

9.3.3 Double pushing

We use a *push* technique in combination with the recursive refinement procedure. Our *double push* method that we present in this section is based upon Seward's *copy* technique (see Section 8.2.2.2). It is used in combination with the previously described recursive partitioning of the buckets after the initial sorting in the first phase. Recall that the *copy* method passes the order of suffixes in a 1-bucket on to the order of the corresponding predecessor suffixes in some 2-buckets (pushing once). Double push further passes the sorted order of these just copied suffixes on to predecessor suffixes in some 3-buckets (pushing twice).

We assume a fixed, small alphabet Σ of size σ. For all $(a, b, c) \in \Sigma^3$, we denote a 3-bucket containing all suffixes with prefix a, b, c by $sa[l_{a,b,c}, r_{a,b,c}]$, a 2-bucket containing all suffixes with prefix a, b by $sa[l_{a,b}, r_{a,b}]$, and a 1-bucket containing all suffixes with prefix a by $sa[l_a, r_a]$. Note that consecutive 3-buckets consisting of suffixes sharing the prefix a, b form a 2-bucket $sa[l_{a,b}, r_{a,b}]$ and that consecutive 2-buckets of suffixes with leading character a form a 1-bucket $sa[l_a, r_a]$.

After the first phase of our algorithm that generates a q-bucket segmentation for $q \geq 3$, our program processes the 1-buckets $sa[l_c, r_c]$, $c \in \Sigma$, in ascending order with respect to the number of suffixes, $|sa[l_c, r_c]| - |sa[l_{c,c}, r_{c,c}]| = r_c - l_c - (r_{c,c} - l_{c,c})$. The recursive refinement procedure, described in Section 9.3.2, sorts all sub-buckets of $sa[l_c, r_c]$ that have not yet been sorted, except for the buckets with equal first and second character c. Then the *copy* algorithm of Seward [135] passes the ordering of suffixes in $sa[l_c, r_c]$ on to the not previously refined buckets among $sa[l_{b_1,c}, r_{b_1,c}], sa[l_{b_2,c}, r_{b_2,c}], \ldots, sa[l_{b_\sigma,c}, r_{b_\sigma,c}]$, where $b_k \in \Sigma$ is the k^{th} character of the alphabet, $k \in [1, \sigma]$. Finally, the suffixes in each of these 2-buckets are *pushed* further. Let $sa[l_{b_k,c}, r_{b_k,c}]$ with $k \in [1, \sigma]$ be any of these 2-buckets and $sa[l_{a_1,b_k,c}, r_{a_2,b_k,c}], sa[l_{a_2,b_k,c}, r_{a_2,b_k,c}], \ldots, sa[l_{a_\sigma,b_k,c}, r_{a_\sigma,b_k,c}]$ the buckets of suffixes with first character $a_j \in \Sigma$ ($j \in [1, \sigma]$), second character b_k, and third character c. Then $sa[l_{b_k,c}, r_{b_k,c}]$ is scanned from left to right. For each suffix number $sa[i]$ with $i \in [l_{b_k,c}, r_{b_k,c}]$ and $sa[i] > 1$, encountered in the scan, if the buckets of suffixes with the first character $t[i-1]$ are not already refined, then the predecessor suffix number $sa[i] - 1$ is assigned to the front of the bucket $sa[l_{t[i-1],b_k,c}, r_{t[i-1],b_k,c}]$, and the front is advanced by one.

Figure 9.4 shows an example of the double push procedure for the input string $t =$ `CEBDEBDEBDEA`. The topmost part below the input string shows the bucket segmentation of the suffix array sa before applying the double push procedure to the bucket of suffixes with leading character `B`. All shown suffix numbers are already in their final position. The double push procedure applied to the bucket of suffixes with leading character `A`, which only contains the suffix number 12, has previously assigned the suffix numbers 11 (predecessor of 12) and 10 (predecessor of 11) to their final positions. The buckets that are going to be determined by the current double push are left empty (buckets of suffixes with second or third character `B`). For each suffix that is involved in the current pushing procedure, the first character of its predecessor suffix (the character to the "left") is printed below its suffix number. `E` is, for example, the character at the positions 8, 5, and 2 to

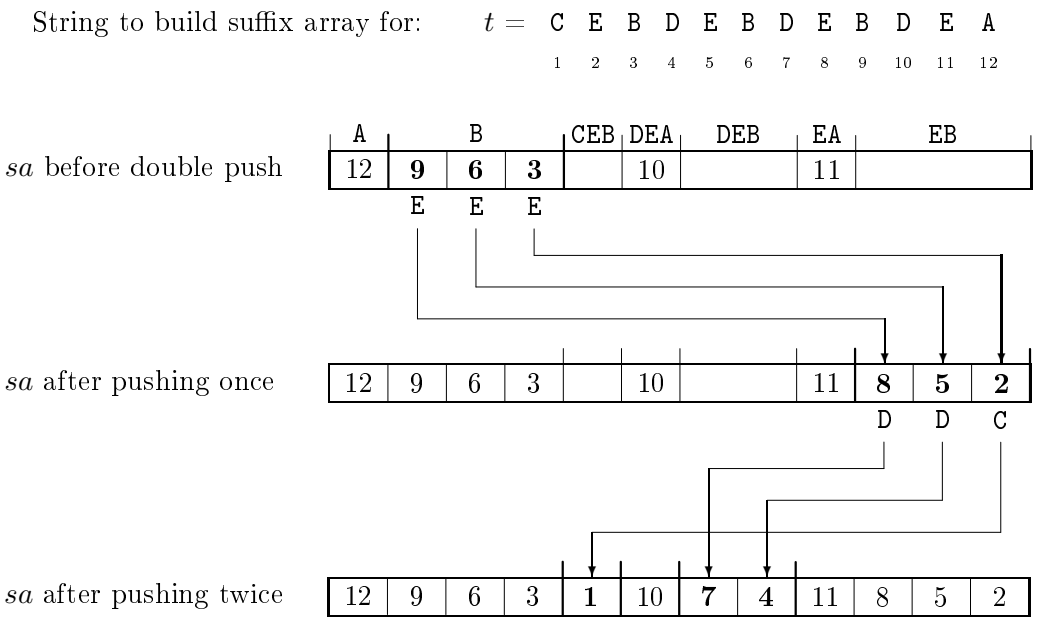

Figure 9.4: Double pushing the bucket of lexicographically sorted suffixes with leading character `B` of the string `CEBDEBDEBDEA`.

the left of 9, 6, and 3. The middle part of the figure shows *sa* after pushing once, and the bottom part shows *sa* after the complete double pushing. We first *push* the bucket of suffix numbers 9, 6, and 3. Their order is passed on to the bucket of predecessor suffix numbers 8, 5, and 2 for which the corresponding suffixes share the prefix `EB`. Then the order of the suffix numbers 8, 5, and 2 is further passed on to the buckets of suffixes with third character `B`. The suffix numbers 7 and 4, which correspond to suffixes with common prefix `DEB`, form a bucket, and the suffix number 1, which corresponds to a suffix with prefix `CEB`, forms another bucket.

9.4 Use cases

A previous version of the here presented bucket-pointer refinement algorithm is published in [132] and [133]. Its implementation proved its efficiency in several bioinformatics applications. Paarmann [116] as well as Twardziok and Schwientek [140] integrated *bpr* in their tools for the design of oligo nucleotides (see also [121]). They applied *bpr* for the construction of their suffix-array-based index, which is then processed further. Kemena [77] and Holthaus [62] use *bpr* for the construction of Abouelhoda *et al.*'s enhanced suffix array [1, 2], upon which they implemented several query algorithms. Moreover, Husemann [64] applied *bpr* for text compression. He implemented Manzini and Ferragina's compression boosting scheme [51] based on suffix arrays.

10 Experimental Results

In this chapter, we investigate the practical construction times and the space requirements of our algorithm and compare it to the fastest previous suffix array construction algorithms. Section 10.1 contains the settings of the experiments. In Section 10.2, we present the results of the experiments and discuss them in Section 10.3.

10.1 Description of the experiments

10.1.1 Implementation of the algorithms

We compared our *bpr* implementation [127, version 2.0.0] to eight other practical implementations: *deep–shallow* by Manzini and Ferragina [102], *cache* and *copy* by Seward [135], *qsufsort* by Larsson and Sadakane [90], *difference-cover* by Burkhardt and Kärkkäinen [31], *odd–even* by Kim *et al.* [78], and *skew* by Kärkkäinen and Sanders [71]. We retrieved the implementations of *deep–shallow*, *cache*, *copy*, and *qsufsort* from Manzini's homepage [100], the code for *difference-cover* and *skew* via Kärkkäinen's homepage [69], and the implementation of *odd–even* was kindly provided by Dong Kyue Kim. We further added the recent *msufsort* implementation of Maniscalco (version 2.0.1), which we retrieved from his homepage [97] (see also [98, 118, 120]). Maniscalco's *msufsort*, however, only constructs the inverse suffix array, although Puglisi *et al.* [120] stated that the suffix array is constructed from the inverse suffix array in-place. Hence, we added a procedure that derives the suffix array by a single scan of the inverse suffix array, but not in-place. The *msufsort* procedure follows the depth-first bucket refinement scheme and uses a pull technique. The general framework is quite similar to our *bpr* algorithm: Similar to our bucket pointer array, *msufsort* uses an array that stores the lexicographical order of previously sorted suffixes in the suffix sorting process. This array ultimately becomes the inverse suffix array (as our bucket pointer array). Beyond that, *msufsort* manages to store further information in the same array: For each non-singleton bucket, it stores a chain of all suffixes located in the bucket. Hence, *msufsort* does not need the suffix array. It is thus more space efficient than *bpr*. Furthermore, *msufsort* uses a tandem repeat detection for suffixes with equal prefix. Once such a tandem repeat is detected, the suffixes can be directly sorted (see [118] for a detailed explanation).

Table 10.1 shows the worst-case asymptotic time complexities of the investigated algorithms.

Table 10.1: Worst-case time complexities of the investigated suffix array construction algorithms.

bpr	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
$\mathcal{O}\left(\frac{n^2}{\log n}\right)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log \log n)$	$\Theta(n)$

10.1.2 Methods

The experiments were performed on four different computers: three computers with x86 architecture and one Sun UltraSPARC computer. We refer to the x86 computers by *Small Scale x86*, *Medium Scale x86*, and *Large Scale x86* regarding their main memory size, and to the Sun UltraSPARC computer by *UltraSPARC*:

- *Small Scale x86* – A 1.3 GHz Intel Pentium™ M (Klamath) processor, running a GNU/Linux operating system. The memory hierarchy is composed of separate L1 instruction and data cache, each of size 32 Kbyte and 3 cycles latency, a 1 Mbyte L2 cache with 10 cycles latency, and 512 Mbytes of main memory. Each cache is 8-way associative with 64 byte line size.

- *Medium Scale x86* – A SunFire V20z with two 1.6 GHz AMD Opteron™ 242 processors running the Solaris 10 operating system. The memory hierarchy is composed of separate L1 instruction and data cache, each of size 64 Kbyte, a 1 Mbyte L2 cache, and 2 Gbytes of main memory. The L1 caches are 2-way associative, and the L2 cache is 16-way associative.

- *Large Scale x86* – A Xen-DomU with three virtual CPUs (mapped onto 3 real Opteron cores) running a GNU/Linux operating system. The real hardware is a SunFire X4100 with two 2.6 GHz AMD Dual-Core Opteron™ 285 SE processors running a GNU/Linux operating system. The memory hierarchy is composed of separate L1 instruction and data cache, each of size 64 Kbyte and 3 cycles latency, a 1 Mbyte L2 cache with 12 cycles latency, and 8 Gbytes of main memory. The L1 caches are 2-way associative with 64 byte line size, and the L2 cache is 8-way associative with 64 byte line size.

- *UltraSPARC* – A SunFire V440 with four 1.3 GHz UltraSPARC IIIi processors running the Solaris 10 operating system. The memory hierarchy is composed of separate L1 instruction and data cache, the instruction cache of size 32 Kbyte and the data cache of size 64 Kbyte, a 1 Mbyte L2 cache, and 16 Gbytes of main memory.

All programs were compiled with the *gcc* compiler, respectively *g++* compiler, with optimisation options '-O3 -fomit-frame-pointer -funroll-loops'. For *Small Scale x86* and *Large Scale x86* both running a GNU/Linux operating system, we used the same executable that was generated with the *gcc* compiler version 3.3.6. For the *Medium Scale x86* and for the *UltraSPARC*, we used the *gcc* compiler version 4.1.1.

10.1.3 Investigated sequence data

We encounter two main types of sequences that are indexed by full-text indices: DNA sequences and other common real-world strings, like natural language texts or software source code. In the analysis of genomes, for example, individual DNA sequences or, alternatively, concatenations of similar DNA sequences are indexed to find repeats, unique regions, and common subsequences (see, for example, [58, 87, 89]). Moreover, Joy and Luck [68] observed that in programming courses, where the assessment is often carried out by means of programming assignments, there is a temptation among some students to copy and modify the work of others. Baker [13] and Mozgovoy *et al.* [110], for example, use full-text indices to detect such plagiarism in program source codes as well as in natural language texts.

Hence, our test data set consists of two major groups of sequences: DNA sequences and common real-world strings. Beyond that, we investigated a third group of artificially generated sequences, mainly to examine degenerated strings with large LCPs. The maximum LCP of a string is a good indicator for the recursion depth of bucket refinement algorithms, and the average LCP further incorporates information of the sizes of the buckets at different refinement levels: Many large ℓ-buckets for a high refinement level ℓ imply a high average LCP. The investigated data files are listed in Table 10.2 and are basically ordered by average LCP. The columns show the name of the sequence, the average and maximum values in the respective LCP array, the length of the sequence, its character set, and a short description of the content. Due to the memory constraints of our *Small Scale x86* test computer, several of the investigated algorithms could not construct suffix arrays for text files that exceeded the 50 million character limit. Hence, we took the last 50 million characters of those text files and added them to our collection of common real-world strings. These truncated sequences are annotated with *50M*. The complete test data set is available through the *bpr* homepage [127, bpr-strings.tar.bz2].

DNA sequences. For the DNA sequences, we selected genomic DNA from different species: the whole genome of the bacteria *Escherichia coli* (*E. coli*), the fourth chromosome of the flowering plant *Arabidopsis thaliana* (*A. thaliana*), the first chromosome of the nematode *Caenorhabditis elegans* (*C. elegans*), and the human (*H. sapiens*) chromosome 22. Moreover, we investigated the construction times for different concatenated DNA sequences of certain families. For this we used six *Streptococcus* genomes, four genomes of the *Chlamydophila* family, and three different *E. coli* genomes. We retrieved the *Escherichia coli* sequence from the *Canterbury Large Corpus* [10, 17], the human chromosome 22 from the corpus of test files provided by Manzini and Ferragina [102, 100], and the other sequences from *GenBank* [20, 115].

Text. For the evaluation of common real-world strings, we used the King James bible (*bible*) and the CIA world fact book (*world*), both from the *Canterbury Large Corpus* [10, 17], and the suite of test files provided by Manzini and Ferragina [102, 100]. The strings of Manzini and Ferragina's corpus are usually concatenations of text files or, alternatively, *tar*

Table 10.2: Description of the data set.

Data set	LCP average	LCP maximum	String length	Alphabet size	Description
E. coli genome	17	2 815	4 638 690	4	*Escherichia coli* genome
A. thaliana chr. 4	58	30 319	12 061 490	7	*A. thaliana* chromosome 4
H. sapiens chr. 22	1 979	199 999	34 553 758	5	*H. sapiens* chromosome 22
C. elegans chr. 1	3 181	110 283	14 188 020	5	*C. elegans* chromosome 1
6 *Streptococci*	131	8 091	11 635 882	5	6 *Streptococcus* genomes
4 *Chlamydophila*	1 555	23 625	4 856 123	6	4 *Chlamydophila* genomes
3 *E. coli*	68 061	1 316 097	14 776 363	5	3 *E. coli* genomes
bible	13	551	4 047 392	63	King James bible
world	23	559	2 473 400	94	CIA world fact book
sprot	89	7 373	109 617 186	66	SwissProt database
rfc	93	3 445	116 421 901	120	Texts from the RFC project
howto	267	70 720	39 422 105	197	Linux Howto files
reuters	282	26 597	114 711 151	93	Reuters news in XML
linux	478	136 035	116 254 720	256	Linux kernel source files
jdk	678	37 334	69 728 899	113	JDK 1.3 doc files
etext	1 108	286 352	105 277 340	146	Project Gutenberg texts
gcc	8 603	856 970	86 630 400	150	*gcc* 3.0 source files
w3c	42 299	990 053	104 201 579	256	HTML files of `www.w3c.org`
sprot 50M	91	2 665	50 000 000	66	SwissProt database
rfc 50M	87	3 445	50 000 000	110	Texts from the RFC project
reuters 50M	280	24 449	50 000 000	91	Reuters news in XML
linux 50M	766	136 035	50 000 000	256	linux kernel source files
jdk 50M	654	34 557	50 000 000	110	JDK 1.3 doc files
etext99 50M	1 845	286 352	50 000 000	120	Project Gutenberg texts
gcc 50M	14 745	856 970	50 000 000	121	*gcc* 3.0 source files
w3c 50M	478	29 752	50 000 000	255	HTML files of `www.w3c.org`
random	4	9	20 000 000	26	Bernoulli string
period 500 000	9 506 251	19 500 000	20 000 000	26	Repeated Bernoulli string
period 1000	9 999 001	19 999 000	20 000 000	26	Repeated Bernoulli string
period 20	9 999 981	19 999 980	20 000 000	17	Repeated Bernoulli string
Fibonacci	5 029 840	10 772 535	20 000 000	2	*Fibonacci string*

archives: the *Swiss prot database* version 34.0 in flat file format (*sprot*), HTML files from the *Request for Comments* database (*rfc*), text files of the *Linux Howto* (*howto*), Reuters news in XML format (*reuters*), the C source code of the Linux kernel 2.4.5 (*linux*), *javadoc* pages consisting of HTML and Java files for JDK 1.3 (*jdk*), text files from the Project Gutenberg (*etext*), source code of the GNU Compiler Collection version 3.0 (*gcc*), and HTML files from the homepage of the World Wide Web consortium (*w3c*).

Table 10.3: Suffix array construction times for different DNA sequences and generalised DNA sequences by different algorithms on the *Large Scale x86* computer, with $q = 7$ for *bpr*. The programs were compiled with the *gcc* compiler version 3.3.6.

	Construction time (s)								
DNA sequences	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
E. coli genome	**1.00**	1.57	1.14	2.08	1.73	1.51	2.47	4.07	8.58
A. thaliana chr. 4	**3.00**	4.57	3.51	6.99	5.99	4.63	7.87	12.17	25.26
H. sapiens chr. 22	**9.88**	14.36	11.76	24.64	20.35	16.31	27.49	39.95	80.91
C. elegans chr. 1	**3.52**	15.69	4.51	11.84	9.80	7.76	10.58	14.37	28.64
6 *Streptococci*	**3.25**	6.28	4.86	8.98	7.45	8.32	9.21	12.04	25.38
4 *Chlamydophila*	**1.32**	8.09	2.44	8.32	8.28	4.85	3.52	4.70	9.82
3 *E. coli*	**4.01**	782.43	9.79	234.04	675.04	24.24	13.55	16.54	34.28

Artificial strings. The artificial files were generated as described by Burkhardt and Kärkkäinen [31]: a random string made out of Bernoulli-distributed characters and periodic strings composed of an initial random string that is repeated until a length of 20 million characters is reached. We used initial random strings of length 20, 1000 and 500 000 to generate the periodic strings. We also investigated a string consisting of the first 20 million characters of a Fibonacci string (see [25]). Fibonacci strings have the reputation for being particularly bad instances for non-linear suffix tree construction algorithms (see, for example, [54, 129, 122]) since they have many long repeats (see [65]).

10.2 Results

The complete running time results on the four different computers are shown in the appendix, Tables A.1–A.5. In this section, we particularly examine the results on the *Large Scale x86* computer. The suffix array construction times are given in Tables 10.3–10.5. Table 10.3 contains the construction times for the DNA sequences. Our *bpr* algorithm is the fastest suffix array construction algorithm for all investigated DNA sequences. The running times of the second fastest algorithm, *deep–shallow*, are by a factor between 1.14 and 2.44 greater than the running times of *bpr*. The other investigated depth-first bucket refinement algorithms, *msufsort*, *cache*, and *copy*, show greater but still reasonable running times if the average LCP is relatively small. For the concatenated sequence of three *E. coli* genomes with average LCP 68 061, however, their running times are significantly greater than the running times of the other algorithms. The breadth-first bucket refinement algorithm *qsufsort* is more stable regarding variations of the average LCP. Nevertheless, the difference between the running time of *bpr* and *qsufsort* is again maximal for the concatenated sequence of the *E. coli* genomes (a factor of 6.04). The reduced string sorting algorithms are slower than all bucket refinement algorithms if the average LCP is small, but significantly faster than the depth-first bucket refinement algorithms *msufsort*, *cache*,

Table 10.4: Suffix array construction times for various texts by different algorithms on the *Large Scale x86* computer, with $q = 3$ for *bpr*. The programs were compiled with the *gcc* compiler version 3.3.6.

	Construction time (s)								
Text	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
bible	**0.90**	1.12	0.93	1.57	1.29	1.72	2.07	4.10	7.44
world	0.55	0.73	**0.48**	0.84	0.66	1.12	1.30	2.56	4.41
sprot	**41.06**	56.66	59.16	111.89	97.84	108.26	145.42	200.61	335.47
rfc	**40.93**	56.23	55.15	100.25	84.06	115.24	125.82	204.20	350.76
howto	**11.87**	15.68	15.02	22.83	25.63	27.54	30.27	62.43	110.32
reuters	**46.26**	66.89	110.99	189.74	212.52	136.83	212.49	217.17	342.13
linux	**37.23**	48.61	48.69	106.21	120.04	99.43	114.86	187.28	345.21
jdk	**24.19**	39.71	63.30	110.25	183.89	83.64	130.65	114.71	186.37
etext	**41.74**	51.36	60.28	101.44	221.22	110.94	106.60	217.79	397.12
gcc	**29.62**	35.33	60.75	1148.78	7153.44	72.67	84.29	123.56	237.12
w3c	**38.31**	55.32	94.65	124.41	3618.65	148.65	143.83	176.27	285.70
sprot 50M	**15.59**	23.31	23.16	41.74	39.67	41.33	55.20	79.80	129.96
rfc 50M	**15.34**	21.35	20.16	34.91	32.22	40.49	45.81	76.99	128.37
reuters 50M	**17.20**	25.64	40.21	66.69	81.86	50.26	76.55	83.44	129.90
linux 50M	**15.83**	19.06	18.18	29.59	47.28	42.27	42.17	71.84	130.25
jdk 50M	**15.46**	24.65	35.54	59.02	112.07	49.34	75.48	77.00	129.07
etext 50M	**17.15**	21.47	25.23	41.05	119.60	43.88	41.27	88.00	141.30
gcc 50M	**17.77**	18.88	49.39	1402.91	7756.83	39.83	47.36	60.55	118.93
w3c 50M	**15.95**	23.42	40.77	49.77	75.31	46.39	66.37	76.41	121.65

and *copy* for the concatenated sequence of the *E. coli* genomes. The running times of *bpr*, however, are as stable as the running times of the *quasi-linear odd–even* algorithm: *bpr* is continuously around 4 times faster than *odd–even* for every DNA sequence.

For the other real-world strings, the running times of the investigated algorithms are shown in Table 10.4. Our *bpr* is the fastest suffix array construction algorithm for all but one string: *deep–shallow* is faster for the CIA world fact book (*world*). The depth-first bucket refinement algorithms *deep–shallow* and *msufsort* show the next best running times: *deep–shallow* is often faster than *msufsort* for strings with small average LCP, but slower for strings with large average LCP. The other depth-first bucket refinement algorithms *cache* and *copy* are only competitive for strings with small average LCP. For such strings, they are faster than the breadth-first bucket refinement algorithm *qsufsort* and the reduced string sorting algorithms *difference-cover*, *odd–even*, and *skew*. For strings with large average LCP, however, they are significantly slower than *qsufsort* and the reduced string sorting algorithms. A strange result is that the running times of *cache* and *copy* for the string *gcc* are less than the running times for its shorter suffix *gcc 50M*. For the strings consisting of exactly 50 million characters, we observe that the running times of *bpr* and *msufsort* as well as the running times of *qsufsort*, *difference-cover*, *odd–even*,

Table 10.5: Suffix array construction times for artificial strings by different algorithms on the *Large Scale x86* computer, with $q = 3$ for *bpr*. The programs were compiled with the *gcc* compiler version 3.3.6.

	Construction time (s)								
Artificial strings	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
random	**5.60**	7.08	6.73	9.23	7.88	8.08	13.30	27.01	36.19
period 500 000	**6.95**	224.85	562.60	43 370.14	—	47.32	29.89	21.20	43.94
period 1000	**7.98**	15.21	651.68	20 998.25	—	50.83	55.16	13.00	35.01
period 20	4.71	**3.36**	31 807.89	—	—	39.14	35.14	6.10	36.78
Fibonacci	**15.75**	232 585.62	547.49	—	176 968.97	44.01	48.44	21.71	27.08

and *skew* are quite stable regarding varying average LCP, although the stated worst-case time complexities of *bpr* and *msufsort* are considerably worse than those of *qsufsort* and the reduced string sorting algorithms.

The construction times for the artificial strings are shown in Table 10.5. Wherever an algorithm used more than 6 days of computation time, we stopped the computation. This is indicated by a dash in the table. For the random string with small average LCP, the bucket refinement algorithms are faster than the reduced string sorting algorithms. For the periodic strings, however, the depth-first bucket refinement algorithms *deep–shallow*, *cache*, and *copy* are significantly slower than the other algorithms. Here, *bpr* performs very well, even compared to *msufsort*, which has a tandem repeat detection, and compared to the algorithms *qsufsort*, *difference-cover*, *odd–even*, and *skew* with good worst-case time complexities. Our algorithm is by far the fastest algorithm for strings with period 1000 and 500 000. For strings with period 20, *msufsort* with its repeat detection is slightly faster. The repeat detection of *msufsort*, however, seems only to work for "simple" short repeats. For the suffix array construction of the repetitive Fibonacci string, *msufsort* needs almost 3 days. Here, *bpr* is the fastest algorithm. It is even faster than the linear-time *skew* algorithm and the quasi-linear *odd–even* algorithm.

Puglisi *et al.* [120] presented an experimental study of different suffix array construction algorithms, including *msufsort*, *deep–shallow*, and our first version of *bpr*. In their evaluation, *msufsort* is always faster than *bpr*, and *deep-shallow* is in most cases faster than *bpr*. These results seem to contradict previous results that we have presented in [132] and [133]. Thus, we performed experiments on computers of different scale and observed ourselves that the relative running time of the first version of *bpr* compared to the running time of other suffix array construction algorithms depend on the used computer with its particular cache and even on the version of the *gcc* compiler. The improved *bpr* algorithm that we investigate in this thesis is much faster than the first version, but the running times compared to the other algorithms still depend on the used computer and on the used compiler. Table 10.6 shows the running times of the investigated suffix array construction algorithms for the string *jdk 50M* on the four different computers. The *msufsort*

Table 10.6: Suffix array construction times for the string *jdk 50M* by different algorithms on four different computers, with $q = 3$ for *bpr*. The programs were compiled with different *gcc* compiler versions.

	Construction time (s)								
Computer	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
Small Scale x86	**19.73**	35.10	45.55	85.34	206.63	75.36	98.20	98.29	162.65
Medium Scale x86	33.49	**28.55**	56.02	117.31	146.06	69.42	82.92	82.37	147.82
Large Scale x86	**15.46**	24.65	35.54	59.02	112.07	49.34	75.48	77.00	129.07
UltraSPARC	**36.84**	—	85.31	145.99	344.27	123.18	192.99	137.53	247.80

implementation aborts unexpectedly for every input on the *UltraSPARC* computer. This is indicated by a dash in the table. All programs have the shortest running time on the *Large Scale x86* computer, but *bpr*, *ds*, and *cache* run faster on the *Small Scale x86* computer than on the *Medium Scale x86* computer, while the other programs run faster on the *Medium Scale x86* computer than on the *Small Scale x86* computer. Also, the relative running times between the algorithms vary greatly: On the *Small Scale x86* computer, for example, the running time of *msufsort* is by a factor of 1.78 greater than the running time of *bpr*, but by a factor of 0.85 smaller on the *Medium Scale x86* computer.

In addition, we run experiments on the *Large Scale x86* computer, where the algorithms were compiled with the *gcc* compiler version 4.0.3, instead of version 3.3.6. The results are shown in the appendix, Table A.4. Here, *bpr* is still the fastest algorithm for the DNA sequences and among the two fastest algorithms for the other sequences, but *bpr*'s advantage decreases. The *gcc* compiler version 4.0.3, however, generates code that uses 64-bit pointers, and we were not able to turn that off. Unlike the other investigated programs, which mainly use integer values for most of their data structures, the bucket pointer array used by *bpr* is based on real C pointers. It thus requires twice as much space as would be necessary for a 32-bit implementation. This certainly leads to more cache misses. Hence, the running times of these compiled programs are not directly comparable.

In summary, one can say that *bpr* is always among the two fastest of the investigated algorithms on every of the four investigated computers. In most cases, and specifically for all DNA sequences, it is the fastest algorithm. Unlike the other depth-first bucket refinement algorithms, it shows stable running times for all investigated sequences, regardless of the average LCP. Even for the Fibonacci string, *bpr* performs well compared to the algorithms *qsufsort*, *difference-cover*, and *odd–even* with good worst-case time complexity, whereas the construction times for *msufsort*, *deep–shallow*, *cache*, and *copy* escalate. The running times of the different algorithms, however, also depend on the used computer and on the used compiler. We should thus be careful with general statements regarding the practical performance of the different algorithms.

Table 10.7: Description of the genomic DNA sequences and the suffix array construction times for these sequences by *bpr*, with $q = 7$.

Genomes	LCP average	LCP maximum	String length	Alphabet size	*bpr* construction time (s)
Human (*H. Sapiens*)	518 611	29 999 999	3 096 521 113	7	4978.11
Mouse (*M. musculus*)	37 338	3 049 999	2 482 869 215	5	3968.57
Dog (*C. lupus*)	69 485	3 000 010	2 531 673 953	5	3856.99

10.2.1 Performance on very large-scale data sets

In a separate experiment, we took the construction times for the human [36], mouse [37] and dog genome [35] (all downloaded from [11]) on a Sun Fire V1280 server running twelve 900 MHz UltraSparc-III processors. Its memory hierarchy is composed of 32 Kbyte L1 instruction and 64 Kbyte L1 data cache, 8 Mbyte L2 cache, and 96 Gbyte main memory. The genomes are concatenated DNA sequences of all their chromosomes where the human genome consists of about 3.09 billion nucleotides, the mouse genome of about 2.48 billion, and the dog genome of about 2.53 billion, in total. The three genome sequences are available through the *bpr* homepage [127, bpr-genomes.tar.bz2]. We compiled the implementations of suffix array construction algorithms with the *gcc* compiler version 4.1.1 and further 64-bit options '-m64 -mptr64'.

Bpr with $q = 7$ needs about 1 h 23 min for the human genome, 1 h 6 min for the mouse genome, and 1 h 4 min for the dog genome. The other algorithms abort unexpectedly. It seems that their particular implementations are limited to 32 bit address space. Note that, at the time we were performing the experiments, the server ran multiple concurrent processes, such that the times may vary in different runs.

10.2.2 Space consumption

Besides the running times, we measured the space consumptions of the different suffix array construction algorithms over all data files. We used *memtime* [19] to get the peak virtual memory consumption traced by the linux operating system. Table 10.8 shows the results in average number of bytes per character of the used input sequences. The given virtual memory consumption of *msufsort* includes only the space for the construction of the inverse suffix array, not the additional space that we used for deriving the suffix array from its inverse.

With $5.04n$ to $6.04n$ bytes, the lightweight algorithms *copy*, *deep–shallow*, *msufsort*, *difference-cover*, and *cache* use slightly more space than the theoretical minimum of $5n$ bytes, consisting of $4n$ bytes for the suffix array and n bytes for the input string. *Qsufsort's* $8.03n$ and *bpr's* $9.30n$ bytes are still under the limit of $10n$ bytes, while *odd–even* and *skew* using $16.03n$ and $23.92n$ bytes, respectively, consume significantly more space.

Table 10.8: Average virtual memory space consumption per input character for the different suffix array construction algorithms.

Bytes per input character								
bpr	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
9.30	5.29	5.06	6.04	5.04	8.03	5.93	16.03	23.92

10.2.3 Detailed runtime analysis

For a more detailed performance analysis of the suffix array construction algorithms, we used the profiler and cache simulator *valgrind* [136, 114] to count the number of executed instructions and to simulate the caching behaviour on the *Large Scale x86* computer. The programs were compiled with the *gcc* compiler version 3.3.6.

The number of executed instructions per input character of the different algorithms is shown in Table 10.9, the L1 data references per input character in Table 10.10, the L1 misses or, alternatively, L2 references per input character in Table 10.11, and the number of L2 misses per input character in Table 10.12. We stopped the computation whenever a simulation used more than 24 hours. This is indicated by a dash in the tables. In addition, Figures 10.1 and 10.2 exemplarily show bar charts for *H. sapiens* chromosome 22 and the *linux* source code. Note that, besides the instructions and cache references of the pure suffix array construction algorithms, *valgrind* also counts those of the different IO routines for reading the input strings from the disk.

It is impressive that the instruction counts for *bpr* clearly outperform all other algorithms for all strings. For real-world strings, the second best algorithm, *msufsort*, executes on average more than twice as many instructions. For the Fibonacci string, *msufsort* executes an enormous number of instructions, although it shows reasonable instruction counts for the artificial strings with shorter periods. In contrast, the instruction counts of *bpr* are stable with respect to strings of varying average LCP. Even for the Fibonacci string, the average instruction count of *bpr* (345 instructions per input symbol) is comparable with the linear-time algorithm *skew* (396 instructions per input symbol) and the quasi-linear *odd–even* algorithm (533 instructions per input symbol).

We additionally counted the executed instructions for the algorithms on the *Large Scale x86* computer compiled with the *gcc* compiler version 4.0.3, instead of version 3.3.6. The results are shown in the appendix (Table A.6). Here, the instruction counts for *bpr* still outperform all the other algorithms for all but one string, the string *gcc 50M* for which *msufsort* takes fewer instructions. The difference to *msufsort*, however, is not as large as for the algorithms compiled with the *gcc* compiler version 3.3.6.

The caching behaviour of *bpr* is also quite good. The number of L1 cache references is correlated with the number of executed instructions, which can be seen in Figures 10.1 and 10.2. Thus, *bpr* takes the smallest number of L1 cache references for all strings. Its inferior miss ratio, however, often leads to more cache misses. For all DNA sequences, *bpr*

Table 10.9: Number of executed instructions on the *Large Scale x86* computer (*gcc* compiler version 3.3.6).

		Executed instructions per input character								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	138	404	231	678	603	304	798	382	397
	A. thaliana chr. 4	149	480	236	879	865	334	856	383	406
	H. sapiens chr. 22	152	414	247	849	749	337	987	391	409
	C. elegans chr. 1	144	2879	302	1749	1727	405	1054	395	406
	6 *Streptococci*	151	809	401	1161	926	428	953	386	401
	4 *Chlamydophila*	156	4457	918	5710	5092	538	978	384	404
	3 *E. coli*	169	150 398	1280	54 382	169 118	701	1029	386	408
Text	*bible*	160	316	248	635	582	364	839	415	378
	world	161	331	253	603	624	348	979	414	378
	sprot	178	406	471	1589	1937	445	1329	440	400
	rfc	171	382	420	1077	1252	470	1171	460	395
	howto	171	377	347	744	1590	421	928	430	412
	reuters	186	459	1077	3281	5599	487	1530	472	400
	linux	167	379	412	2055	3429	454	1144	447	409
	jdk	185	488	1107	2889	10215	491	1680	475	397
	etext	178	385	459	1087	7206	466	925	438	412
	gcc	281	386	1574	—	—	459	1250	451	410
	w3c	185	600	1839	2178	—	606	1557	466	405
	sprot 50M	173	396	466	1369	1995	427	1298	433	395
	rfc 50M	169	371	381	962	1399	446	1129	453	396
	reuters 50M	180	447	969	2525	5101	470	1469	464	401
	linux 50M	167	376	403	942	3298	486	1109	439	409
	jdk 50M	182	476	971	2341	9548	478	1617	468	398
	etext 50M	174	381	449	947	9365	454	901	432	413
	gcc 50M	359	387	3457	—	—	468	1312	452	409
	w3c 50M	184	452	1724	1766	5770	474	1583	465	399
Artificial	random	153	267	263	521	464	250	667	332	291
	period 500 000	211	18 600	100 750	—	—	785	2070	335	395
	period 1 000	176	452	149 789	—	—	794	2214	349	398
	period 20	201	275	—	—	—	880	2467	418	384
	Fibonacci	345	—	83 378	—	—	815	2469	533	386

Table 10.10: Number of L1 cache references on the *Large Scale x86* computer (*gcc* compiler version 3.3.6).

		L1 data cache references per input character								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	81.80	200.17	91.27	306.28	257.24	138.75	405.41	243.11	250.53
	A. thaliana chr. 4	87.76	241.33	93.01	385.14	344.14	149.60	436.21	243.26	255.83
	H. sapiens chr. 22	89.36	203.32	97.16	375.40	311.57	151.78	514.01	243.99	257.62
	C. elegans chr. 1	85.08	1610.44	119.59	723.83	618.63	187.99	552.16	244.86	255.68
	6 *Streptococci*	89.02	428.45	152.74	495.16	362.49	200.00	474.67	244.37	252.58
	4 *Chlamydophila*	91.77	2499.99	351.48	2283.86	1719.18	261.70	476.53	242.92	254.49
	3 *E. coli*	98.13	85601.40	495.72	21418.43	55169.62	348.70	505.13	244.21	257.19
Text	*bible*	93.26	155.92	100.69	291.71	250.05	164.65	423.21	237.54	238.95
	world	94.75	165.00	103.57	275.14	257.33	160.80	507.66	238.15	238.48
	sprot	102.50	200.41	183.82	668.76	697.61	199.57	717.39	242.69	252.12
	rfc	98.96	190.21	171.27	465.55	472.15	209.90	612.82	246.73	249.13
	howto	99.51	187.84	136.47	338.20	583.15	190.20	470.43	241.63	259.80
	reuters	106.35	224.90	459.09	1336.21	1900.02	219.46	840.07	248.85	251.75
	linux	97.62	188.98	161.70	853.01	1183.30	205.22	594.99	244.61	257.71
	jdk	106.29	241.72	479.19	1179.01	3398.94	225.70	935.72	248.94	249.67
	etext	102.60	190.29	181.35	478.35	2424.53	207.44	471.28	243.02	260.07
	gcc	151.13	191.54	611.68	—	—	208.31	662.31	245.14	258.19
	w3c	106.21	305.88	814.64	899.71	—	287.50	846.47	247.78	254.55
	sprot 50M	100.17	195.87	183.61	580.71	713.44	193.05	700.57	241.59	248.66
	rfc 50M	97.94	184.69	153.55	419.28	517.28	200.49	590.30	245.54	249.53
	reuters 50M	103.41	219.93	407.00	1037.04	1733.24	213.46	804.76	247.32	252.24
	linux 50M	98.26	189.20	157.92	413.92	1137.07	219.92	574.32	243.31	258.13
	jdk 50M	104.77	236.34	411.98	962.88	3179.78	220.26	896.54	247.72	250.25
	etext 50M	100.14	188.98	177.90	422.47	3125.29	203.95	458.69	241.74	260.51
	gcc 50M	187.43	192.45	1286.20	—	—	213.68	702.39	245.22	257.63
	w3c 50M	106.33	224.87	846.42	735.18	1945.90	218.90	868.91	247.36	251.17
Artificial	random	89.35	129.86	103.61	255.23	213.68	119.41	333.19	214.56	185.25
	period 500 000	114.87	10251.72	38942.08	—	—	371.74	1191.01	218.09	247.45
	period 1 000	100.02	224.44	52875.36	—	—	365.66	1275.53	221.70	248.02
	period 20	115.32	129.62	—	—	—	393.94	1426.58	260.75	240.05
	Fibonacci	185.17	—	32225.18	—	—	387.25	1425.79	314.04	241.49

Table 10.11: Number of L1 cache misses (L2 cache references) on the *Large Scale x86* computer (*gcc* compiler version 3.3.6).

		L1 cache misses per input character								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	3.10	5.22	4.54	7.03	5.41	6.59	12.84	15.32	31.54
	A. thaliana chr. 4	4.78	5.31	4.99	7.72	5.96	7.82	15.12	15.59	32.38
	H. sapiens chr. 22	4.49	5.25	5.72	8.81	6.89	8.21	17.19	15.39	31.78
	C. elegans chr. 1	4.01	8.39	4.87	13.70	10.45	9.82	14.91	14.47	29.94
	6 *Streptococci*	4.75	5.83	5.95	10.04	7.23	11.75	16.00	15.73	31.72
	4 *Chlamydophila*	5.87	11.44	7.69	29.90	21.65	17.43	14.87	15.41	31.72
	3 *E. coli*	6.63	218.99	11.23	336.98	835.38	23.02	17.24	15.90	32.50
Text	*bible*	3.74	4.44	4.34	6.61	4.98	9.42	12.20	18.81	30.27
	world	4.03	4.75	3.61	5.55	4.01	9.16	11.00	18.76	29.72
	sprot	5.91	6.11	8.60	11.73	8.84	15.79	26.08	21.50	32.06
	rfc	5.03	5.62	6.58	10.70	7.99	15.86	20.82	20.93	31.15
	howto	4.77	5.30	5.87	8.26	10.09	12.74	15.83	21.27	34.01
	reuters	6.24	7.60	16.22	18.18	16.21	19.22	35.01	22.35	31.40
	linux	5.66	5.08	6.14	11.53	12.55	14.33	18.23	20.39	32.38
	jdk	5.93	7.89	16.69	18.84	25.77	19.22	34.20	21.36	30.18
	etext	5.42	5.36	8.22	11.70	32.34	15.74	18.54	21.81	34.77
	gcc	11.37	5.26	19.18	—	—	15.25	19.75	19.13	31.85
	w3c	6.83	6.79	12.49	16.07	—	23.43	28.34	22.43	31.60
	sprot 50M	5.53	6.03	7.92	11.48	8.16	14.60	23.87	21.11	31.42
	rfc 50M	4.76	5.49	6.17	9.50	7.54	14.40	18.90	20.39	31.29
	reuters 50M	5.92	7.34	14.01	17.46	14.41	17.88	32.67	21.72	31.46
	linux 50M	5.55	4.90	5.47	8.62	15.41	15.18	16.29	19.79	32.42
	jdk 50M	5.55	7.50	13.82	16.28	24.17	18.22	31.31	20.83	30.36
	etext 50M	5.06	5.18	7.65	10.78	43.51	14.76	17.10	21.37	34.78
	gcc 50M	15.87	5.16	28.81	—	—	15.46	19.47	17.95	30.84
	w3c 50M	5.99	6.71	10.74	13.77	16.55	17.51	29.60	22.17	30.74
Artificial	random	3.79	4.51	5.30	6.87	5.81	6.81	14.02	19.72	24.33
	period 500 000	5.61	63.94	604.75	—	—	46.53	23.37	17.21	30.85
	period 1 000	9.12	12.64	720.25	—	—	51.35	62.05	13.36	26.19
	period 20	7.44	4.01	—	—	—	56.17	34.54	5.19	24.40
	Fibonacci	22.61	—	550.58	—	—	47.32	40.78	10.89	21.37

Table 10.12: Number of L2 cache misses on the *Large Scale x86* computer (*gcc* compiler version 3.3.6).

		L2 cache misses per input character								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	2.10	3.76	2.45	4.00	2.63	4.88	5.75	11.63	27.58
	A. thaliana chr. 4	2.22	4.01	3.01	4.91	3.41	5.18	8.06	13.00	29.80
	H. sapiens chr. 22	2.24	4.15	3.67	5.92	4.37	6.11	10.16	13.56	30.06
	C. elegans chr. 1	2.28	5.69	3.04	8.02	4.89	7.95	8.36	12.14	27.46
	6 *Streptococci*	2.69	4.49	3.83	6.94	4.42	9.19	8.89	13.18	29.32
	4 *Chlamydophila*	3.44	8.27	4.62	20.72	11.94	14.98	7.26	11.69	27.35
	3 *E. coli*	4.36	170.12	8.40	299.50	724.25	21.21	10.28	13.57	30.06
Text	*bible*	2.08	2.92	1.81	3.11	1.88	5.67	5.03	13.59	26.78
	world	2.00	2.71	1.16	2.15	1.15	5.89	3.60	12.15	24.04
	sprot	3.08	4.05	4.36	7.09	4.86	10.46	14.84	17.60	30.18
	rfc	3.01	3.81	4.06	6.30	4.71	10.74	12.57	17.15	29.70
	howto	2.70	3.38	3.28	4.90	4.17	8.11	9.10	17.54	32.01
	reuters	3.75	5.25	8.40	11.30	8.78	14.06	24.70	18.37	29.87
	linux	3.14	3.25	3.47	5.28	4.68	9.50	10.89	16.35	30.64
	jdk	3.32	4.74	5.78	9.62	9.79	14.18	17.16	16.57	28.56
	etext	3.10	3.77	5.32	7.84	16.19	10.58	11.80	18.95	33.29
	gcc	2.88	3.42	6.86	—	—	10.48	11.28	15.05	30.19
	w3c	3.76	4.27	7.51	8.67	—	18.18	15.89	17.09	29.87
	sprot 50M	2.79	3.85	3.59	5.94	4.05	9.66	12.28	16.82	29.37
	rfc 50M	2.74	3.55	3.42	5.32	3.98	9.46	10.58	16.45	29.60
	reuters 50M	3.30	4.97	6.82	9.07	7.13	12.85	20.47	17.52	29.74
	linux 50M	3.04	3.03	2.90	4.33	4.23	10.35	9.22	15.71	30.35
	jdk 50M	3.09	4.33	4.95	7.97	8.04	13.30	14.67	16.03	28.63
	etext 50M	2.79	3.63	4.71	6.70	17.14	9.47	10.43	18.41	33.11
	gcc 50M	2.84	3.31	5.45	—	—	10.70	10.72	13.64	28.95
	w3c 50M	3.28	4.09	5.04	6.75	5.87	12.52	14.48	16.25	28.77
Artificial	random	2.10	3.40	2.30	3.49	2.46	5.74	8.13	17.55	21.69
	period 500 000	4.08	44.54	318.75	—	—	44.76	17.10	14.81	28.42
	period 1 000	7.14	11.00	694.15	—	—	45.17	46.25	11.00	24.67
	period 20	5.93	3.57	—	—	—	51.72	29.47	5.13	24.10
	Fibonacci	16.25	—	544.01	—	—	44.54	33.99	10.02	20.83

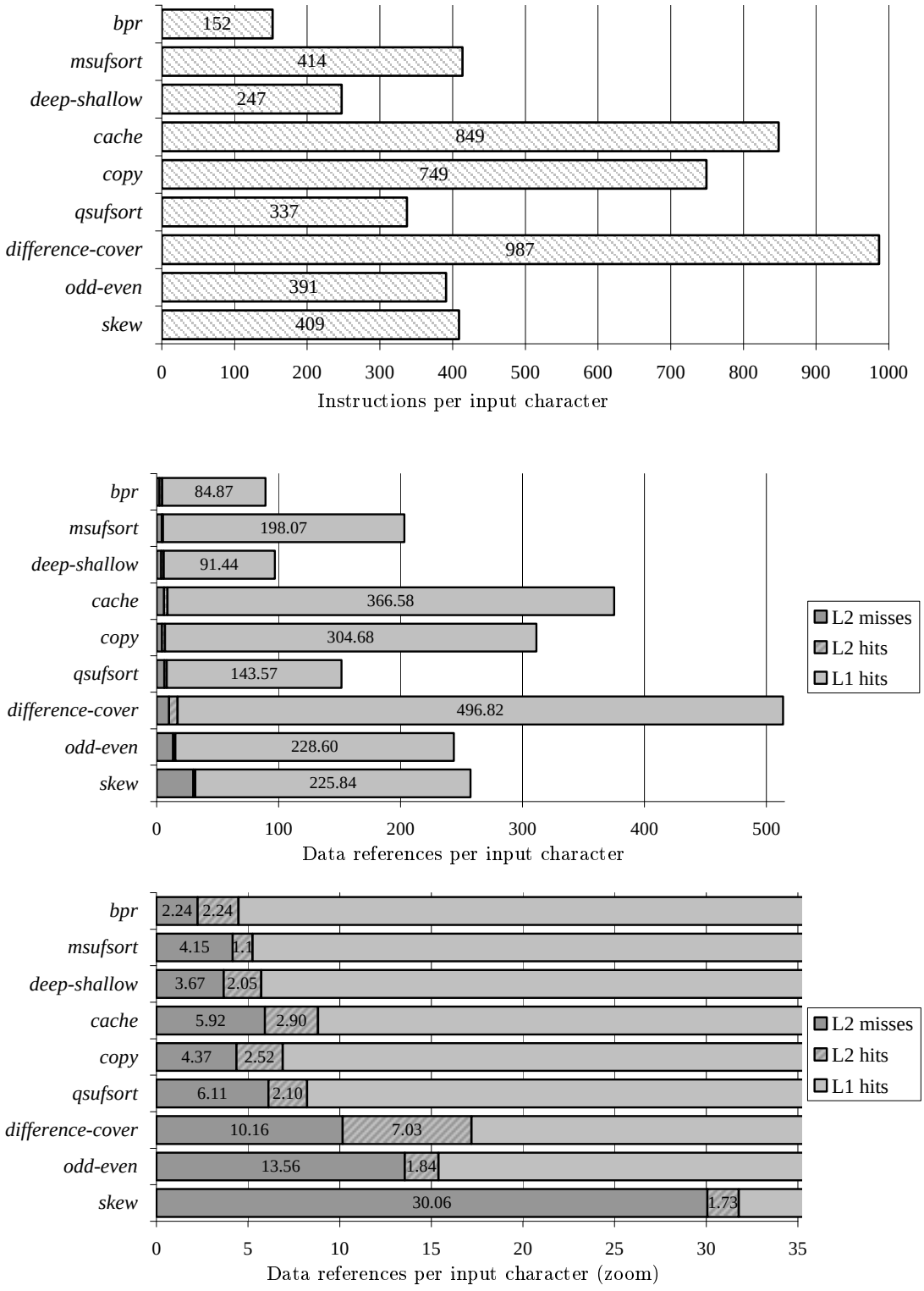

Figure 10.1: Instruction counts and cache references for *H. sapiens chr. 22*, with $q = 7$ for *bpr*.

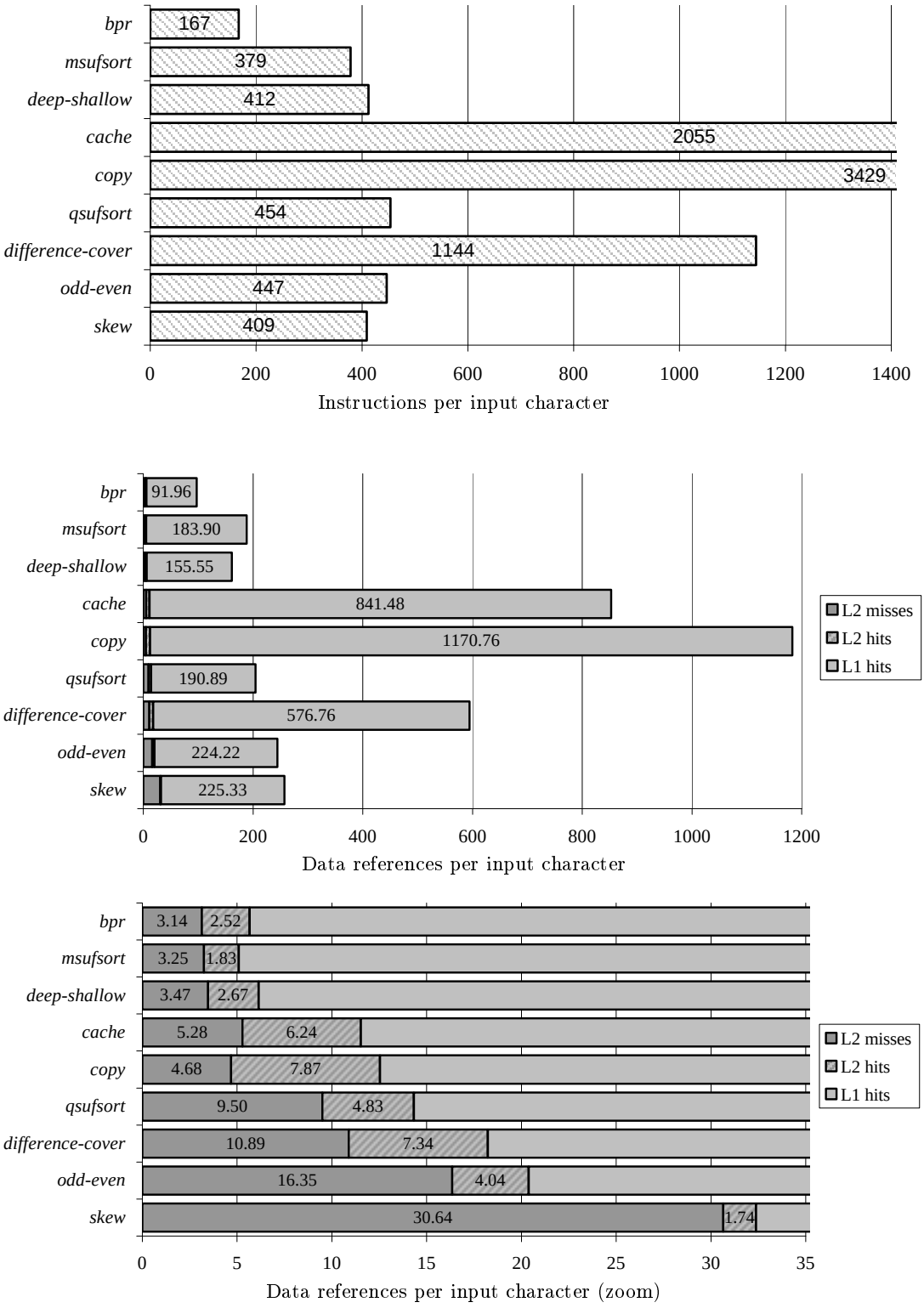

Figure 10.2: Instruction counts and cache references for the *linux* file, with $q = 3$ for *bpr*.

still has the fewest L1 and L2 cache misses, but for other real-world strings, *msufsort* often has less L1 cache misses, and *deep-shallow* has sometimes less L2 cache misses. Although the L1 cache miss ratio of *bpr* is often worse than *msufsort*'s, its L2 cache miss ratio is usually better. The reason is probably the different granularity of the respective data access patterns.

For degenerated strings, the caching behaviour of *bpr* is also quite good. It takes the fewest cache misses for strings with periods of length 500 000 and 1 000. For the string with a period of length 20 and for the Fibonacci string, however, other algorithms have fewer cache misses, but *bpr* is still among the three algorithms with the fewest number of cache misses.

10.3 Discussion of the experimental results

We first believed that the practical speed of our algorithm was mainly due to the combination of different techniques with good locality behaviour. However, the simulations showed that, compared to the other suffix array construction algorithms, *bpr* mainly gains its fast running time from the fewer executed instructions rather than from its good locality behaviour. With respect to the number of executed instructions, *bpr* is the algorithmically best algorithm.

The few executed instructions are apparently due to the different strategies of the two phases of the *bpr* algorithm. First of all, if the q-length substrings are uniformly distributed, phase 1 equally divides all suffixes into small buckets by just scanning the input string twice. This, however, does not explain its speed for the periodic strings. Here, the suffixes are just partitioned into a few large buckets. For such strings, our algorithm basically benefits from the use of relations among the suffixes in phase 2. By using the bucket pointers as sort keys, the method incorporates information about the subdivided buckets into the bucket refinement process as soon as this information becomes available. In the bucket-refinement process, each bucket is refined recursively until it consists of singleton sub-buckets. This technique of dividing suffixes from small to smaller buckets is similar to *Quicksort* for original sorting, which is known to be fast in practice. The combination of these techniques and further heuristics in the refinement procedure (Section 9.3), in particular the double push method (Section 9.3.3), results in the final low instruction count. This stably low instruction count also supports Conjecture 9.2, which assumes a subquadratic worst-case time bound of the *bpr* algorithm.

In our first assumption that the good locality behaviour was mainly responsible for the speed of *bpr*, we were misled by some elements of the algorithm that have good locality behaviour with respect to the data structure, but this is not always the case. The data structure can be divided into four parts: the input string, the suffix array, the bucket pointer array, and the bucket array storing the boundaries for all buckets. Phase 1, for example, just scans the sequence twice. It has a good locality of memory access with respect to the input string and the bucket pointer array, whereas the bucket array and the suffix array are arbitrarily accessed. In contrast, phase 2 has a good locality of memory

access with respect to the bucket array and the suffix array. The bucket array is accessed from left to right and the suffix array is divided into increasingly smaller buckets. The bucket pointer array is again arbitrarily accessed. Therefore, *bpr*'s cache miss ratio is often worse than that of the other depth-first bucket refinement algorithms *msufsort*, *deep–shallow*, *cache*, and *copy*. Nevertheless, thanks to its fewer total cache accesses and its fewer executed instructions, *bpr* is generally faster than the other algorithms.

The instruction counts for the different real-world strings reveal further interesting facts. The linear-time *skew*, the quasi-linear *odd–even*, and the $\mathcal{O}(n \log n)$ time *qsufsort* algorithms show little variance of instruction counts, indicating little dependence on the sequence structure. In contrast, the instruction counts of *msufsort*, *deep–shallow*, *cache*, and *copy* vary greatly. *Deep–shallow*, for example, executes less than 400 instructions per input character for the *howto* and the *rfc 50M* files, but more than 1500 instructions per input character for the *w3c* and the *gcc* files. For the *gcc* files and for the longer *w3c* file, the very high average and maximum LCP values account for the high instruction count, whereas for *w3c 50M* this is not so. The string has even lower LCP values than the *linux 50M* string, nevertheless, *deep–shallow* needs more than four times the number of executed instructions. Therefore, other structural properties of the text also seem to be important for the instruction count and thus for the performance of those algorithms. *Msufsort*, for example, shows worse instruction counts for the DNA sequences than for the other real-world strings, even if the average and maximum LCPs of the DNA sequences are smaller. One reason could be the particular structure of the DNA sequences with highly variable LCPs, or simply the relatively small DNA alphabet. Apart from that, *msufsort* shows relatively low instruction counts for the strings with periods of length 1000 and 20, which is presumably due to its repeat detection. The efficiency of their repeat detection, however, decreases with increasing period length since *msufsort* detects a period of length ℓ not until the bucket refinement process has reached the refinement level ℓ. Hence, the instruction count is very high for the string with a period of length 500 000 and for the Fibonacci string.

Comparing the instruction counts for the real-world strings shows that *deep–shallow* often executes many more instructions than, and *msufsort* often about as many as, *qsufsort*, *odd–even*, or *skew*, even though the execution times of *deep–shallow* and *msufsort* are always significantly faster. The higher number of L2 cache misses for *qsufsort*, *odd–even*, and in particular *skew* reveal that the fragmented memory access slows down their suffix array construction. Therefore, the practically fastest algorithm does not need to have the lowest instruction count or the lowest number of cache misses, but as with *bpr*, it must possess the optimal combination of both properties.

Bpr is generally the fastest among the investigated suffix array construction algorithms on the four different computers, but the relative running times between the algorithms vary greatly. Responsible for that are mainly the different compiler versions and the different memory facilities of the computers with their multiple levels of cache and their main memory. The used compiler is mainly responsible for the number of executed instructions. Different compilers, respectively different compiler versions, may generate machine code of different quality (e.g., "faster" or "slower") depending on the computer architecture,

the used processor, and the implementation of the algorithm. The particular memory hierarchy is responsible for the number of cache misses at different cache levels and for the cache latencies. The performance of a cache is mainly determined by three parameters: cache size, line size, and degree of associativity. Note that the cache miss ratio is usually negatively correlated with the cache latency: A larger cache usually leads to a lower cache miss ratio, but a higher latency. Moreover, on modern computers, a "clever" compiler can insert prefetch instructions to request the data before they are needed to avoid cache misses (compiler prefetching), and there are several further techniques to improve the caching behaviour (see, for example, [117, Chapter 5]). Therefore, we should be careful with general statements regarding the practical performance of our algorithm, even though it is the fastest suffix array construction algorithm on our four test computers.

However, the space requirements of *bpr* are higher than the space requirements for *msufsort*, *deep–shallow*, *cache*, and *copy*. In practice, *bpr* takes between $9n$ and $10n$ bytes, the suffix array and the bucket pointer table each consume $4n$ bytes, and the input string n bytes. Additional space is used for the bucket pointers of the initial bucket sort and for the recursion stack, even though the recursion depth decreases by a factor of q.

Therefore, if one is concerned about space, the *msufsort* algorithm or the *deep–shallow* algorithm might be the best choice. If there are no major space limitations, we believe that the *bpr* algorithm is an attractive alternative. Maniscalco and Puglisi [99], however, recently presented a suffix array construction algorithm that seems to be faster than the version of *msufsort* that we analysed in this thesis (see [120]), but that algorithm was not available when we performed our experiments. Its practical running time should be investigated further.

11 Conclusion

We have discussed two major aspects of suffix arrays, namely their combinatorics and their construction. We have been the first presenting an in-depth study on the combinatorics of suffix arrays. Our work dealt with the classical combinatorial counting problem and with the related algorithmical enumeration problem: We have presented constructive proofs to count the strings sharing the same suffix array as well as the distinct suffix arrays for fixed size alphabets. Beyond that, based on the construction schemes used in the proofs, we developed efficient algorithms to enumerate those strings and those suffix arrays, respectively. For alphabets of size σ, $\binom{n+\sigma-d-1}{\sigma-d-1}$ strings of length n share the same suffix array (with d ${}^+R$-descents) among which $\binom{n-d-1}{\sigma-d-1}$ are composed of exactly σ distinct characters. For these strings, we have given a bijection into the set of non-decreasing sequences over $\sigma-d$ integers and presented optimal-time enumeration algorithms. The number of distinct suffix arrays is $\sum_{d=0}^{\sigma-1} \left\langle {n \atop d} \right\rangle = \sum_{k=0}^{\sigma-1} \binom{n}{k}(-1)^k(\sigma-k)^n$. This has yielded lower bounds for the compressibility of such suffix arrays. Moreover, summing up the number of strings for each suffix array yields constructive proofs for Worpitzki's identity and for the summation rule of Eulerian numbers to generate the Stirling numbers of the second kind. One could also say the number of suffix arrays and their strings form a particular instance of these identities.

Unlike the combinatorics of suffix arrays, their efficient construction has been widely studied before. We have introduced new classifications of suffix array construction algorithms and have surveyed the previous algorithms. On the one hand, we have classified the suffix array construction algorithms regarding their progress in the suffix sorting process: either bucket refinement or reduced string sorting. On the other hand, we have classified them regarding the use of dependencies among suffixes: either the push method or the pull method. We have presented our new bucket-pointer refinement algorithm, proved an $\mathcal{O}(n^2/\log n)$ worst-case time bound and an $\mathcal{O}(n \log n)$ expected-case time bound, and enhanced the basic algorithm with some further techniques for fast practical suffix array construction. Due to its simple structure, it is easy to implement. Finally, we have extensively evaluated the practical performance of our algorithm and other suffix array construction algorithms for real-world input sequences of different type and for degenerated input sequences that were artificially generated. The results show that our bucket-pointer refinement algorithm is usually the fastest among all investigated suffix array construction algorithms, even for worst-case strings. Therefore, we believe that it can be widely used in all kinds of suffix array applications.

Open problems

Some problems regarding the combinatorics and the construction of suffix arrays remain unsolved or have been opened up by the thesis.

For the lower bound of the compressibility of the information content of suffix arrays in the Kolmogorov sense, we have counted all possible suffix arrays for strings over a fixed sized alphabet. The Kolmogorov complexity considers the information content of a sequence independent of any particular probability model, but if the underlying probability model for a sequence is known, Shannon entropy is often used as a measure of the information content. In terms of Shannon entropy, however, we are so far not able to give such lower bounds for the compressibility of suffix arrays.

Moreover, the running time of our enumeration algorithm for the suffix arrays or, alternatively, for the corresponding equivalence classes of strings sharing the same suffix array could possibly be reduced further. The running time of our algorithm is $\mathcal{O}(\log n)$ multiplied by the number of enumerated suffix arrays. The $\mathcal{O}(\log n)$ factor is used for the update of the dynamic auxiliary data structure for the implementation of the Burrows-Wheeler transform, or rather for the corresponding Last-to-First mapping. With a more advanced dynamic data structure it could possibly be reduced to a constant factor. As we mentioned, our right-to-left extension scheme for the enumeration can also be used for the suffix array construction or for the construction of the Burrows-Wheeler transform. Hence, with a dynamic data structure that would allow the constant time extension to the left, we could solve two problems at once: the optimal-time enumeration of suffix arrays and the optimal linear-time right-to-left online construction of suffix arrays. For suffix tree construction, there is Weiner's optimal linear-time algorithm that also adds the suffixes of the input string from right to left. So maybe we can use some of Weiner's techniques. A straightforward approach could use his algorithm for the construction of suffix trees and keep track of the sorted list of suffixes at the leaves of the suffix tree. Weiner's algorithm, however, requires quite a bit of working space, which we would like to save. Hence, we would not like to simply port that algorithm to suffix arrays.

For the right-to-left online construction of suffix arrays or, alternatively, for the construction of the Burrows-Wheeler transform, a practical approach could abandon the optimal time criterion and search for the proper insertion positions of the new suffix into the suffix array in another way. Table 5.2 shows, for example, how the Burrows-Wheeler transform is updated when the character `A` is added to the front of the string `ABBAA`. In the Burrows-Wheeler transform, the character \$ is simply replaced by the new character `A`. The crucial and also most time-consuming part is to find the insertion position of the `A` in the corresponding First sequence. We could simply search for the first `A` preceding the newly inserted `A` in the Burrows-Wheeler transform and follow a link (corresponding to the LF-mapping, described by the dashed line in the table) to the corresponding `A` in the First sequence. The correct insertion position in the First sequence is then directly behind this `A`, which is also the new position of the \$ in the Burrows-Wheeler transform. This method works for every front extension of the input string. The preceding character in the Burrows-Wheeler transform that equals the new character at the front of the string

is, however, possibly far away. Moreover, such an algorithm would require traversals of dynamic lists and links between these lists, which usually has a bad locality of memory reference. Hence, we doubt that such an algorithm performs well in practice.

Also questions regarding our bucket-pointer refinement algorithm remain. We were so far unable to prove a better worst-case time complexity than $\mathcal{O}(n^2/\log n)$ while at the same time we are not aware of an example showing that this bound is tight. For certain periodic strings, we verified an $\mathcal{O}\left(n^{\frac{3}{2}}/\sqrt{\log n}\right)$ time bound, but for general strings finding a non-trivial upper bound seems to be hard since our algorithm quite arbitrarily uses the dependence among suffixes. We have further proved an $\mathcal{O}(n \log n)$ expected time bound, but suppose that it is linear.

Beyond the construction of the complete suffix array, we may be interested in sparse suffix arrays that only contain a particular subset of suffixes. There are sparse suffix trees [12, 74] with linear time construction algorithms [6, 66] using space proportional to the number of suffixes in the sparse index. To the best of our knowledge, linear-time construction algorithms using space proportional to the number of suffixes in the sparse suffix array do not exist. A promising approach to solve that problem could be to modify one of the reduced string sorting algorithms since these algorithms also use sparse suffix arrays in intermediate steps.

A Appendix

Tables A.1–A.5 contain the running times of the different suffix array construction programs on the four different test computers: Table A.1 for the *Small Scale x86* computer (*gcc* compiler version 3.3.6), Table A.2 for the *Medium Scale x86* computer (*gcc* compiler version 4.1.1), Table A.3 for the *Large Scale x86* computer where the programs were compiled with the *gcc* compiler version 3.3.6, Table A.4 for the *Large Scale x86* computer where the programs were compiled with the *gcc* compiler version 4.0.3, and Table A.5 for the *UltraSPARC* computer (*gcc* compiler version 4.1.1). A dash in a table denotes that the running time experiment of the respective algorithm could not be carried out successfully for the corresponding string: a dash for *cache* and *copy* denotes that we terminated the experiment after 6 days of computation, a dash for *odd–even* or *skew* in Table A.2 denotes that the programs aborted with a memory allocation error on the *Medium Scale x86* computer, and a dash for *msufsort* in Table A.5 denotes that the program aborts unexpectedly on the *UltraSPARC* computer.

The number of executed instructions per input character of the different algorithms on the *Large Scale x86* computer compiled with the *gcc* compiler version 4.0.3 is shown in Table A.6. We stopped the computation whenever a simulation used more than 24 hours, which is indicated by a dash in the table. Note that Table 10.9 shown in Section 10.2.3 shows the respective results on the same computer, but the programs were compiled with the *gcc* compiler version 3.3.6.

Table A.1: Suffix array construction times on the *Small Scale x86* computer (*gcc* compiler version 3.3.6).

		Construction time (s)								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	**1.40**	3.00	1.69	3.65	2.81	3.03	4.53	6.10	14.09
	A. thaliana chr. 4	**4.08**	8.62	5.02	11.72	9.37	9.17	14.32	17.54	40.09
	H. sapiens chr. 22	**11.77**	24.03	15.63	37.67	28.97	27.60	46.18	52.51	112.59
	C. elegans chr. 1	**4.38**	26.25	5.87	17.85	15.21	13.36	17.06	18.60	40.85
	6 *Streptococci*	**4.07**	10.18	5.94	13.11	9.69	14.16	14.71	16.54	36.72
	4 *Chlamydophila*	**1.82**	12.66	3.63	13.96	11.65	8.58	5.80	6.41	14.65
	3 *E. coli*	**6.27**	1069.00	14.62	327.68	977.04	36.67	21.24	22.12	48.97
Text	*bible*	**1.25**	2.06	1.37	2.81	2.16	3.31	3.82	6.42	11.72
	world	0.75	1.23	**0.72**	1.40	1.15	1.94	2.29	3.57	6.40
	howto	**14.75**	23.98	18.85	35.61	41.74	42.44	48.97	84.20	141.62
	sprot 50M	**19.79**	32.98	28.42	61.86	62.06	61.80	82.30	104.36	169.87
	rfc 50M	**18.94**	31.00	25.29	50.74	49.15	59.73	71.58	101.74	169.30
	reuters 50M	**20.89**	37.48	50.01	94.04	128.70	74.67	109.27	108.54	170.47
	linux 50M	**18.89**	27.61	23.41	45.05	81.54	62.94	66.30	99.17	173.70
	jdk 50M	**19.73**	35.10	45.55	85.34	206.63	75.36	98.20	98.29	162.65
	etext 50M	**20.10**	32.49	31.61	59.00	212.29	62.98	66.58	110.68	190.97
	gcc 50M	**23.10**	28.11	87.62	2122.08	16256.37	61.07	74.10	84.52	162.53
	w3c 50M	**20.75**	32.39	61.61	70.78	133.99	70.73	96.53	106.23	163.67
Artificial	random	**6.52**	11.07	7.76	13.75	10.78	14.76	20.36	36.96	46.92
	period 500 000	**8.95**	275.17	870.52	86242.11	—	88.93	46.45	30.58	60.81
	period 1000	**11.80**	18.70	1282.91	35681.56	—	86.95	78.78	21.78	52.36
	period 20	7.28	**5.74**	62886.31	—	—	74.97	59.10	10.12	43.71
	Fibonacci	23.42	322977.74	784.41	—	265061.12	82.41	70.02	**22.05**	38.29

Table A.2: Suffix array construction times on the *Medium Scale x86* computer (*gcc* compiler version 4.1.1).

		Construction time (s)								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	**1.37**	2.01	1.59	3.00	2.33	2.21	3.55	4.30	10.39
	A. thaliana chr. 4	**4.20**	6.00	4.88	10.73	7.83	7.09	11.06	13.20	30.14
	H. sapiens chr. 22	**15.48**	20.85	16.15	33.89	25.05	27.33	37.61	44.11	113.19
	C. elegans chr. 1	**4.71**	19.84	5.78	24.74	12.28	11.04	13.82	15.06	33.43
	6 *Streptococci*	**4.19**	7.46	5.79	14.60	8.27	10.12	11.75	13.07	28.58
	4 *Chlamydophila*	**1.64**	9.23	3.36	26.44	8.62	5.58	4.55	4.66	11.07
	3 *E. coli*	**6.33**	841.82	13.36	753.82	729.93	26.18	16.86	17.38	39.26
Text	*bible*	**1.18**	1.38	1.36	2.23	1.79	2.54	2.95	4.54	8.79
	world	0.71	0.86	**0.69**	1.21	0.93	1.53	1.76	2.65	4.94
	sprot	**58.97**	66.99	84.44	199.56	120.51	154.95	202.64	—	—
	rfc	**58.65**	66.92	73.97	149.20	105.49	154.56	172.79	—	—
	howto	**16.74**	18.85	20.82	30.68	33.29	39.19	40.07	71.74	134.75
	reuters	**65.05**	73.75	169.44	407.40	260.03	184.71	277.86	—	—
	linux	**52.82**	56.02	65.27	228.17	155.03	132.80	150.67	—	—
	jdk	**33.49**	43.97	94.93	205.34	220.26	105.72	136.61	123.83	—
	etext	**61.47**	65.13	88.57	150.30	292.95	163.26	151.30	—	—
	gcc	41.61	**41.41**	82.73	4199.94	9786.24	100.18	111.30	144.02	—
	w3c	**51.93**	61.47	168.88	234.28	5062.09	191.42	184.89	—	—
	sprot 50M	**23.82**	28.12	31.68	72.34	50.02	59.75	71.80	89.43	170.44
	rfc 50M	**21.43**	25.61	27.01	51.67	41.02	57.09	60.30	86.07	170.11
	reuters 50M	**24.72**	30.02	61.45	134.48	99.08	73.00	93.04	94.12	163.63
	linux 50M	**21.55**	22.20	24.25	45.72	62.26	57.54	54.44	82.45	164.35
	jdk 50M	33.49	**28.55**	56.02	117.31	146.06	69.42	82.92	82.37	147.82
	etext 50M	**25.53**	28.70	34.18	58.59	161.77	66.90	56.36	95.99	196.47
	gcc 50M	25.64	**21.85**	68.29	4724.60	10895.42	53.51	64.53	69.64	148.40
	w3c 50M	**22.41**	26.09	92.99	88.78	98.23	64.34	81.29	86.52	148.87
Artificial	random	**8.01**	9.07	8.52	10.46	9.92	11.84	16.87	30.24	50.55
	period 500 000	**9.10**	170.16	548.25	202410.10	—	78.69	37.29	22.45	53.76
	period 1000	**10.44**	14.11	873.97	79408.46	—	71.86	62.50	15.36	45.45
	period 20	5.82	**4.11**	44082.23	—	—	50.39	44.22	8.34	32.68
	Fibonacci	25.00	296780.74	528.70	—	201211.44	78.99	52.21	**24.40**	34.86

Table A.3: Suffix array construction times on the *Large Scale x86* computer (*gcc* compiler version 3.3.6).

		Construction time (s)								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	**1.00**	1.57	1.14	2.08	1.73	1.51	2.47	4.07	8.58
	A. thaliana chr. 4	**3.00**	4.57	3.51	6.99	5.99	4.63	7.87	12.17	25.26
	H. sapiens chr. 22	**9.88**	14.36	11.76	24.64	20.35	16.31	27.49	39.95	80.91
	C. elegans chr. 1	**3.52**	15.69	4.51	11.84	9.80	7.76	10.58	14.37	28.64
	6 *Streptococci*	**3.25**	6.28	4.86	8.98	7.45	8.32	9.21	12.04	25.38
	4 *Chlamydophila*	**1.32**	8.09	2.44	8.32	8.28	4.85	3.52	4.70	9.82
	3 *E. coli*	**4.01**	782.43	9.79	234.04	675.04	24.24	13.55	16.54	34.28
Text	*bible*	**0.90**	1.12	0.93	1.57	1.29	1.72	2.07	4.10	7.44
	world	0.55	0.73	**0.48**	0.84	0.66	1.12	1.30	2.56	4.41
	sprot	**41.06**	56.66	59.16	111.89	97.84	108.26	145.42	200.61	335.47
	rfc	**40.93**	56.23	55.15	100.25	84.06	115.24	125.82	204.20	350.76
	howto	**11.87**	15.68	15.02	22.83	25.63	27.54	30.27	62.43	110.32
	reuters	**46.26**	66.89	110.99	189.74	212.52	136.83	212.49	217.17	342.13
	linux	**37.23**	48.61	48.69	106.21	120.04	99.43	114.86	187.28	345.21
	jdk	**24.19**	39.71	63.30	110.25	183.89	83.64	130.65	114.71	186.37
	etext	**41.74**	51.36	60.28	101.44	221.22	110.94	106.60	217.79	397.12
	gcc	**29.62**	35.33	60.75	1148.78	7153.44	72.67	84.29	123.56	237.12
	w3c	**38.31**	55.32	94.65	124.41	3618.65	148.65	143.83	176.27	285.70
	sprot 50M	**15.59**	23.31	23.16	41.74	39.67	41.33	55.20	79.80	129.96
	rfc 50M	**15.34**	21.35	20.16	34.91	32.22	40.49	45.81	76.99	128.37
	reuters 50M	**17.20**	25.64	40.21	66.69	81.86	50.26	76.55	83.44	129.90
	linux 50M	**15.83**	19.06	18.18	29.59	47.28	42.27	42.17	71.84	130.25
	jdk 50M	**15.46**	24.65	35.54	59.02	112.07	49.34	75.48	77.00	129.07
	etext 50M	**17.15**	21.47	25.23	41.05	119.60	43.88	41.27	88.00	141.30
	gcc 50M	**17.77**	18.88	49.39	1402.91	7756.83	39.83	47.36	60.55	118.93
	w3c 50M	**15.95**	23.42	40.77	49.77	75.31	46.39	66.37	76.41	121.65
Artificial	random	**5.60**	7.08	6.73	9.23	7.88	8.08	13.30	27.01	36.19
	period 500 000	**6.95**	224.85	562.60	43 370.14	—	47.32	29.89	21.20	43.94
	period 1000	**7.98**	15.21	651.68	20 998.25	—	50.83	55.16	13.00	35.01
	period 20	4.71	**3.36**	31 807.89	—	—	39.14	35.14	6.10	36.78
	Fibonacci	**15.75**	232 585.62	547.49	—	176 968.97	44.01	48.44	21.71	27.08

Table A.4: Suffix array construction times on the *Large Scale x86* computer (*gcc* compiler version 4.0.3).

		Construction time (s)								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	**1.09**	1.37	1.17	1.98	1.67	1.44	2.37	3.72	8.35
	A. thaliana chr. 4	**3.09**	3.64	3.21	6.21	5.38	4.05	7.26	9.79	21.02
	H. sapiens chr. 22	**10.92**	12.24	11.76	22.24	18.70	14.55	25.30	33.70	70.75
	C. elegans chr. 1	**3.55**	9.81	3.77	10.64	8.88	5.95	8.91	11.05	23.25
	6 *Streptococci*	**3.24**	4.29	3.75	7.58	5.70	5.83	7.67	9.62	20.16
	4 *Chlamydophila*	**1.35**	4.56	2.00	9.38	6.75	3.46	3.00	3.58	7.89
	3 *E. coli*	**5.09**	363.95	7.77	245.85	566.67	15.23	10.94	12.57	26.52
Text	*bible*	**0.88**	0.93	**0.88**	1.40	1.16	1.53	1.84	3.59	6.80
	world	0.54	0.57	**0.43**	0.74	0.62	0.90	1.08	2.02	3.75
	sprot	**41.33**	43.36	51.40	95.70	88.00	85.33	119.00	162.27	275.91
	rfc	**39.92**	41.39	46.19	81.45	71.77	89.49	104.80	160.69	282.59
	howto	11.73	**11.61**	12.49	19.66	24.17	21.76	25.04	52.48	85.18
	reuters	**45.76**	50.76	92.18	170.67	196.67	105.41	168.87	176.44	274.91
	linux	**36.08**	36.37	39.83	99.21	116.30	74.97	92.59	148.22	275.98
	jdk	**22.49**	26.55	49.02	88.53	176.79	55.69	82.03	86.34	141.40
	etext	41.70	**39.28**	54.52	90.70	226.60	86.36	90.68	156.72	290.30
	gcc	30.68	**25.91**	52.60	1342.53	8180.68	55.43	69.62	97.34	194.78
	w3c	**37.19**	38.15	75.19	105.59	4033.96	107.74	116.38	140.99	228.18
	sprot 50M	**16.76**	17.78	19.78	37.29	36.83	32.89	44.26	64.00	106.56
	rfc 50M	**15.29**	15.72	16.90	29.02	28.86	32.63	37.91	61.08	104.13
	reuters 50M	**17.33**	19.24	33.69	59.88	74.53	41.32	60.20	65.51	104.54
	linux 50M	14.86	**13.75**	14.86	25.04	46.74	32.11	34.11	59.18	104.20
	jdk 50M	**14.98**	17.18	30.39	53.82	117.54	36.09	51.85	57.62	97.18
	etext 50M	17.80	**16.40**	21.64	34.09	123.63	34.28	35.62	66.50	117.83
	gcc 50M	18.98	**13.93**	40.46	1605.19	9178.22	29.93	38.27	50.04	95.27
	w3c 50M	**16.31**	17.06	35.22	44.30	76.75	35.60	51.95	61.81	97.97
Artificial	random	5.46	**5.44**	5.72	7.38	6.45	6.39	10.60	21.02	28.78
	period 500 000	**7.57**	85.39	306.35	51 315.47	—	38.70	22.87	17.60	39.76
	period 1000	**8.01**	10.58	420.82	24 910.40	—	39.54	43.74	10.96	32.69
	period 20	3.87	**2.93**	19 913.87	—	—	32.16	29.99	5.35	24.90
	Fibonacci string	**16.81**	161 230.81	280.57	347 384.66	153 150.26	35.22	36.24	16.89	22.57

Table A.5: Suffix array construction times on the *UltraSPARC* computer (*gcc* compiler version 4.1.1).

		Construction time (s)								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	**1.86**	—	2.83	4.84	3.63	4.01	7.43	9.61	21.93
	A. thaliana chr. 4	**6.60**	—	8.70	16.26	12.75	11.54	24.02	26.99	59.96
	H. sapiens chr. 22	**19.37**	—	29.24	51.40	39.74	39.75	82.62	78.85	174.85
	C. elegans chr. 1	**7.92**	—	10.63	36.22	25.97	20.41	32.03	30.77	67.26
	6 *Streptococci*	**7.27**	—	10.93	21.93	15.13	17.89	25.85	25.93	57.19
	4 *Chlamydophila*	**3.60**	—	7.36	35.10	22.87	10.94	9.50	9.93	23.56
	3 *E. coli*	**13.96**	—	34.46	1258.88	3018.04	48.68	37.58	34.27	74.52
Text	*bible*	**2.24**	—	2.28	3.79	2.70	4.71	6.11	9.32	18.53
	world	1.37	—	**1.10**	1.74	1.44	2.78	3.47	5.15	10.92
	sprot	**80.10**	—	123.90	245.51	208.65	215.59	386.66	323.03	557.44
	rfc	**82.10**	—	124.64	202.03	174.95	238.48	347.95	335.17	587.38
	howto	**26.41**	—	33.40	49.62	64.97	62.93	87.68	112.18	212.39
	reuters	**97.34**	—	277.61	471.15	531.25	295.02	618.86	352.24	586.55
	linux	**76.76**	—	109.32	264.33	320.29	213.51	323.59	317.26	603.28
	jdk	**54.54**	—	143.32	247.57	525.92	180.32	299.68	197.36	342.28
	etext	**77.70**	—	136.47	208.00	744.17	215.94	282.65	321.23	581.12
	gcc	**72.63**	—	196.34	4381.86	23903.74	172.43	254.39	221.54	444.11
	w3c	**82.36**	—	319.40	300.29	20319.68	320.23	407.90	300.22	535.68
	sprot 50M	**34.07**	—	49.29	93.84	90.54	90.69	155.68	140.20	250.55
	rfc 50M	**32.78**	—	45.45	74.81	72.74	90.18	135.11	138.43	250.55
	reuters 50M	**38.24**	—	100.62	161.57	204.78	118.06	229.42	145.37	253.11
	linux 50M	**32.39**	—	41.79	65.77	137.00	98.67	120.04	131.94	255.65
	jdk 50M	**36.84**	—	85.31	145.99	344.27	123.18	192.99	137.53	247.80
	etext 50M	**34.69**	—	58.97	85.06	435.26	90.86	122.60	149.76	276.09
	gcc 50M	**46.12**	—	171.32	5008.70	26562.70	99.46	146.05	117.05	248.22
	w3c 50M	**38.77**	—	149.01	115.07	216.10	115.91	188.41	137.37	248.47
Artificial	random	**10.74**	—	12.16	17.28	13.85	20.46	39.00	52.75	72.61
	period 500 000	**18.42**	—	2796.74	349881.25	—	146.26	88.93	47.07	97.97
	period 1000	**22.06**	—	3942.28	115637.82	—	158.79	166.09	34.59	87.90
	period 20	24.23	—	197122.96	—	—	182.44	148.67	**22.62**	87.36
	Fibonacci	58.15	—	2889.33	—	—	154.03	161.92	**41.77**	79.07

Table A.6: Executed instructions on the *Large Scale x86* computer (*gcc* compiler version 4.0.3).

		Executed instructions per input character								
Sequence type	Sequence	*bpr*	*msufsort*	*deep shallow*	*cache*	*copy*	*qsufsort*	*difference cover*	*odd even*	*skew*
DNA sequence	*E. coli* genome	116	291	236	640	617	266	599	389	430
	A. thaliana chr. 4	140	333	241	875	955	288	644	392	440
	H. sapiens chr. 22	143	298	255	822	786	291	735	399	443
	C. elegans chr. 1	136	1918	284	1932	2101	362	783	402	439
	6 *Streptococci*	141	553	359	1215	1039	387	733	394	433
	4 *Chlamydophila*	144	2865	746	6613	6471	522	764	392	437
	3 *E. coli*	154	99033	985	64335	220244	684	804	394	442
Text	*bible*	150	231	249	584	588	316	631	429	409
	world	152	238	250	571	666	309	724	429	408
	sprot	163	287	448	1698	2334	381	961	450	432
	rfc	157	272	394	1100	1450	404	868	469	427
	howto	159	273	328	693	1885	368	696	440	447
	reuters	169	317	1031	3680	7078	417	1093	479	431
	linux	141	271	382	2248	4280	395	851	452	443
	jdk	169	326	1060	3236	13112	423	1195	480	427
	etext	164	280	438	1059	9165	401	694	451	448
	gcc	253	277	1239	—	—	397	918	461	444
	w3c	155	403	1591	2393	—	550	1129	468	436
	sprot 50M	159	279	447	1448	2421	368	938	445	426
	rfc 50M	155	264	362	972	1650	385	837	463	428
	reuters 50M	165	307	921	2797	6443	405	1052	472	432
	linux 50M	142	269	369	940	4121	421	826	446	444
	jdk 50M	167	319	929	2591	12248	413	1154	475	428
	etext 50M	160	277	426	902	11988	394	676	445	449
	gcc 50M	321	277	2575	—	—	404	956	461	442
	w3c 50M	170	307	1649	1918	7334	414	1135	466	430
Artificial	random	142	207	271	422	421	226	491	351	312
	period 500 000	198	8031	67512	—	—	661	1452	348	421
	period 1000	159	282	105229	—	—	644	1529	361	420
	period 20	177	207	—	—	—	703	1692	408	409
	Fibonacci	291	—	55949	—	—	698	1702	504	411

Bibliography

[1] Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, March 2004.

[2] Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. *Handbook on Computational Molecular Biology (Chapman & All/Crc Computer and Information Science Series)*, chapter Enhanced Suffix Arrays and Applications, pages (7–1)–(7–27). Chapman & Hall/CRC Press, December 2005.

[3] Rudolf Ahlswede, Bernhard Balkenhol, Christian Deppe, and Martin Fröhlich. A fast suffix-sorting algorithm. In *General Theory of Information Transfer and Combinatorics (GTIT-C)*, volume 4123 of *Lecture Notes in Computer Science*, pages 719–734. Springer Verlag, 2006.

[4] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Factor oracle: a new structure for pattern matching. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics (SOFSEM 1999)*, volume 1725 of *Lecture Notes in Computer Science*, pages 295–310. Springer Verlag, November 1999.

[5] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 258–264. ACM Press, August 2002.

[6] Arne Andersson, N. Jesper Larsson, and Kurt Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, March 1999.

[7] Antonitio, Patrick J. Ryan, William F. Smyth, Andrew Turpin, and Xiaoyang Yu. New suffix array algorithms — linear but not fast? In *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, pages 148–156, July 2004.

[8] Alberto Apostolico. *Combinatorial Algorithms on Words*, chapter The myriad virtues of subword trees, pages 85–96. Springer Verlag New York, 1985.

[9] Alberto Apostolico and Wojciech Szpankowski. Self-alignments in words and their applications. *Journal of Algorithms*, 13(3):446–467, September 1992.

[10] Ross Arnold and Timothy C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the Data Compression Conference (DCC 1997)*, pages 201–210. IEEE Computer Society, March 1997.

[11] Genome Bioinformatics Group at the University of California, Santa Cruz, USA. UCSC genome browser. `http://hgdownload.cse.ucsc.edu/downloads.html`. Last visited: April 1, 2007.

[12] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Efficient text searching of regular expressions. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP 1989)*, volume 372 of *Lecture Notes in Computer Science*, pages 46–62. Springer Verlag, July 1989.

[13] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, October 1997.

[14] Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003)*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer Verlag, August 2003.

[15] Dror Baron and Yoram Bresler. Anti-sequential suffix sorting for bwt-based data compression. *IEEE Transactions on Computers*, 54(4):385–397, April 2005.

[16] Michael Beckstette, Dirk Strothmann, Robert Homann, Robert Giegerich, and Stefan Kurtz. *PoSSuMsearch*: Fast and sensitive matching of position specific scoring matrices using enhanced suffix arrays. In *Proceedings of the German Conference on Bioinformatics (GCB 2004)*, Lecture Notes in Informatics, pages 53–64. Gesellschaft für Informatik, October 2004.

[17] Tim Bell, Matt Powell, Joffre Horlor, and Ross Arnold. The canterbury large corpus. `http://corpus.canterbury.ac.nz/resources/large.tar.gz`. Last visited: June 18, 2007.

[18] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN 2000)*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer Verlag, April 2000.

[19] Johan Bengtsson. memtime. `http://freshmeat.net/projects/memtime`. Last visited: June 18, 2007.

[20] Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, James Ostell, and David L. Wheeler. Genbank. *Nucleic Acids Research*, 31(1):23–27, 2003.

[21] Jon Bentley. *Programming pearls*. ACM Press, 1986.

[22] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Software: Practice and Experience*, 23(11):1249–1265, November 1993.

[23] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1997)*, pages 360–369. Society for Industrial and Applied Mathematics, January 1997.

[24] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22(2):221–242, April 1993.

[25] Jean Berstel. *The Book of L.*, chapter Fibonacci words–a survey, pages 11–26. Springer Verlag, 1986.

[26] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *Journal of Computer and Systems Sciences*, 7(4):448–461, August 1973.

[27] Anselm Blumer, Janet A. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[28] Miklos Bona. *Combinatorics of Permutations*, volume 29 of *Discrete Mathematics and Its Applications*. McGraw-Hill, June 2004.

[29] Miklos Bona. *Introduction to Enumerative Combinatorics*. Walter Rudin Student Series in Advanced Mathematics. McGraw-Hill, September 2005.

[30] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proceedings of the 3rd Annual International Conference on Research in Computational Molecular Biology (RECOMB 1999)*, pages 77–83. ACM Press, April 1999.

[31] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer Verlag, June 2003.

[32] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report – Research Report 124, Digital System Research Center, May 1994.

[33] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4–5):327–344, October 1994.

[34] Charalambos A. Charalambides. *Enumerative Combinatorics*, volume 19 of *Discrete Mathematics and Its Applications*. Chapman & Hall / CRC Press, 2002.

[35] Dog Genome Sequencing Consortium. Genome sequence, comparative analysis and haplotype structure of the domestic dog. *Nature*, 438(7069):803–819, December 2005.

[36] International Human Genome Sequencing Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431(7011):931–945, October 2004.

[37] Mouse Genome Sequencing Consortium. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420(6915):520–562, December 2002.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[39] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, January 2002.

[40] Maxime Crochemore, Jacques Désarménien, and Dominique Perrin. A note on the Burrows-Wheeler transformation. *Theoretical Computer Science*, 332(1–3):567–572, February 2005.

[41] Erik D. Demaine and Alejandro López-Ortiz. A linear lower bound on index size for text retrieval. *Journal of Algorithms*, 48(1):2–15, August 2003.

[42] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO 2005)*, pages 86–97. Society for Industrial and Applied Mathematics, January 2005.

[43] Dorit Dor and Uri Zwick. Selecting the median. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1995)*, pages 28–37. Society for Industrial and Applied Mathematics, January 1995.

[44] Jean-Pierre Duval and Arnaud Lefebvre. Words over an ordered alphabet and suffix permutations. *RAIRO – Theoretical Informatics and Applications*, 36(3):249–259, July–September 2002.

[45] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science (FOCS 1997)*, pages 137–143. IEEE Computer Society, October 1997.

[46] Martin Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming (ICALP 1996)*, volume 1099 of *Lecture Notes in Computer Science*, pages 550–561. Springer Verlag, July 1996.

[47] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, November 2000.

[48] Paolo Ferragina and Roberto Grossi. The String B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, March 1999.

[49] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398. IEEE Computer Society, November 2000.

[50] Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1–2):13–28, June 2001.

[51] Paolo Ferragina and Giovanni Manzini. Compression boosting in optimal linear time using the burrows-wheeler transform. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 655–663. Society for Industrial and Applied Mathematics, January 2004.

[52] Wolfgang Gerlach. Dynamic FM-index for a collection of texts with applications to space-efficient construction of the compressed suffix array. Diplomarbeit, Technische Fakultät, Universität Bielefeld, Germany, February 2007.

[53] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreigth and Weiner: A unifying view of linear-time suffix tree constructions. *Algorithmica*, 19(3):331–353, November 1997.

[54] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. *Software: Practice and Experience*, 33(11):1035–1049, September 2003.

[55] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Timsuff Snider. New indices for text: Pat trees and pat arrays. In W. B. Frakes and Ricardo A. Baeza-Yates, editors, *Information retrieval: data structures and algorithms*, pages 66–82. Prentice-Hall, 1992.

[56] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[57] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406. ACM Press, May 2000.

[58] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

[59] Charles A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[60] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. Efficient multiple genome alignment. In *Proceedings of the 10th International Conference on Intelligent Systems for Molecular Biology (ISMB 2002), Bioinformatics*, volume 18 (Supplement 1), pages S312–S320, August 2002.

[61] Christophe Hohlweg and Christophe Reutenauer. Lyndon words, permutations and trees. *Theoretical Computer Science*, 307(1):173–178, September 2003.

[62] Patrick Holthaus. String algorithms on enhanced suffix arrays. Bachelor thesis, Technische Fakultät, Universität Bielefeld, Germany, August 2006.

[63] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003)*, pages 251–260. IEEE Computer Society, October 2003.

[64] Peter Husemann. Kompressionsverstärkung für Textdaten unter Benutzung der Burrows-Wheeler-Transformation (in german). Diplomarbeit, Technische Fakultät, Universität Bielefeld, Germany, May 2006.

[65] Costas S. Iliopoulos, Dennis Moore, and William F. Smyth. A characterization of the squares in a Fibonacci string. *Theoretical Computer Science*, 172(1–2):281–291, February 1997.

[66] Shunsuke Inenaga and Masayuki Takeda. On-line linear-time construction of word suffix trees. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, volume 4009 of *Lecture Notes in Computer Science*, pages 60–71. Springer Verlag, July 2006.

[67] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of the 6th International Conference on String Processing and Information Retrieval and the 5th International Workshop on Groupware (SPIRE/CRIWG 1999)*, pages 81–88. IEEE Computer Society Press, September 1999.

[68] Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(2):129–133, May 1999.

[69] Juha Kärkkäinen. Home page. `http://www.cs.helsinki.fi/u/tpkarkka/`. Last visited: January 9, 2007.

[70] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 1995)*, volume 937 of *Lecture Notes in Computer Science*, pages 191–204. Springer Verlag, July 1995.

[71] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer Verlag, June 2003.

[72] Juha Kärkkäinen and Peter Sanders. Source code for the *skew* algorithm. `http://www.mpi-inf.mpg.de/~sanders/programs/suffix`, 2003. Last visited: January 9, 2007.

[73] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, November 2006.

[74] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proceeding of the 2nd Annual International Conference on Computing and Combinatorics (COCOON 1996)*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer Verlag, June 1996.

[75] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th ACM Symposium on Theory of Computing (STOC 1972)*, pages 125–136. ACM Press, May 1972.

[76] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer Verlag, July 2001.

[77] Carsten Kemena. Algorithms on enhanced suffix arrays and their application in bioinformatics. Bachelor thesis, Technische Fakultät, Universität Bielefeld, Germany, August 2006.

[78] Dong K. Kim, Junha Jo, and Heejin Park. A fast algorithm for constructing suffix arrays for fixed-size alphabets. In *Proceedings of the 3rd International Workshop on Experimental and Efficient Algorithms (WEA 2004)*, volume 3059 of *Lecture Notes in Computer Science*, pages 301–314. Springer Verlag, May 2004.

[79] Dong K. Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer Verlag, June 2003.

[80] Dong K. Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2–4):126–142, June 2005.

[81] Krzysztof C. Kiwiel. Partitioning schemes for quicksort and quickselect. *Computing Research Repository (CoRR)*, cs.DS/0312054, December 2003. `http://arxiv.org/abs/cs.DS/0312054`.

[82] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison–Wesley, second edition, 1998.

[83] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[84] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer Verlag, June 2003.

[85] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2–4):143–156, June 2005.

[86] Fabian Kulla and Peter Sanders. Scalable parallel suffix array construction. In *Proceedings of the 13th European User's Group Meeting: Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI 2006)*, volume 4192 of *Lecture Notes in Computer Science*, pages 22–29. Springer Verlag, September 2006.

[87] Stefan Kurtz. The vmatch homepage. `http://www.vmatch.de`. Last visited: June 18, 2007.

[88] Stefan Kurtz. Reducing the space requirements of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, November 1999.

[89] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, November 2001.

[90] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, May 1999.

[91] Sunglim Lee and Kunsoo Park. Efficient implementations of suffix array construction algorithms. In *Proceedings of the 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, pages 64–72, July 2004.

[92] Ross A. Lippert, Clark M. Mobarry, and Brian P. Walenz. A space-efficient construction of the burrows-wheeler transform for genomic data. *Journal of Computational Biology*, 12(7):943–951, September 2005.

[93] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *17th Annual Symposium on Combinatorial Pattern Matching (CPM*

2006), number 4009 in Lecture Notes in Computer Science, pages 306–317. Springer Verlag, July 2006.

[94] Veli Mäkinen. Compact suffix array – a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, January 2003.

[95] Ketil Malde, Eivind Coward, and Inge Jonassen. Fast sequence clustering using a suffix array algorithm. *Bioinformatics*, 19(10):1221–1226, July 2003.

[96] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[97] Michael A. Maniscalco. `http://www.michael-maniscalco.com/msufsort.htm`. Last visited: January 9, 2007.

[98] Michael A. Maniscalco and Simon J. Puglisi. An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithms, to appear.*

[99] Michael A. Maniscalco and Simon J. Puglisi. Faster lightweight suffix array construction. In *Proceeding of the 17th Australasian Workshop on Combinatorial Algorithms (AWOCA 2006)*, pages 16–29, July 2006.

[100] Giovanni Manzini. A lightweight suffix array and bwt construction algorithm. `http://www.mfn.unipmn.it/~manzini/lightweight/`. Last visited: January 9, 2007.

[101] Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004)*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer Verlag, July 2004.

[102] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, June 2004.

[103] George E. Martin. *Counting: The Art of Enumerative Combinatorics*. Undergraduate Texts in Mathematics. Springer Verlag, 2001.

[104] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[105] M. Douglas McIlroy. `http://cm.bell-labs.com/cm/cs/who/doug/ssort.c`, 1997. Last visited: June 18, 2007.

[106] Peter M. McIlroy, Keith Bostic, and M. Douglas McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[107] Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 11–12. Society for Industrial and Applied Mathematics, January 2005.

[108] Krisztián Monostori, Arkady Zaslavsky, and Heinz Schmidt. Suffix vector: space- and time-efficient alternative to suffix trees. In *Proceedings of the 25th Australasian Conference on Computer Science (ACSC 2002)*, pages 157–165. Australian Computer Society, January 2002.

[109] Dennis Moore, William F. Smyth, and Dianne Miller. Counting distinct strings. *Algorithmica*, 23(1):1–13, April 1999.

[110] Maxim Mozgovoy, Kimmo Fredriksson, Daniel R. White, Mike Joy, and Erkki Sutinen. Fast plagiarism detection system. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE 2005)*, volume 3772 of *Lecture Notes in Computer Science*, pages 267–270. Springer Verlag, November 2005.

[111] Joong Chae Na. Linear-time construction of compressed suffix arrays using o(n log n)-bit working space for large alphabets. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM 2005)*, volume 3537 of *Lecture Notes in Computer Science*, pages 57–67. Springer Verlag, June 2005.

[112] Gonzalo Navarro, Joao Paulo Kitajima, Berthier A. Ribeiro-Neto, and Nivio Ziviani. Distributed generation of suffix arrays. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching (CPM 1997)*, pages 102–115. Springer Verlag, June 1997.

[113] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), April 2007.

[114] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), July 2003.

[115] National Center of Biotechnology Information (NCBI). Genbank genomes. `ftp://ftp.ncbi.nih.gov/genbank/genomes/`. Last visited: January 9, 2007.

[116] Daniel Paarmann. Oligo Designer – Berechnung von Oligonukleotiden (in german). Diplomarbeit, Technische Fakultät, Universität Bielefeld, Germany, 2005.

[117] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., second edition, 1996.

[118] Simon J. Puglisi. Exposition and analysis of a suffix sorting algorithm. Technical Report CAS-05-02-WS, Department of Computing and Software, McMaster University Hamilton, Ontario, Canada, May 2005.

[119] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. The performance of linear time suffix sorting algorithms. In *Proceedings of the Data Compression Conference (DCC 2005)*, pages 358–367. IEEE Computer Society, March 2005.

[120] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys, to appear*, 39(2), June 2007.

[121] Sven Rahmann. Rapid large-scale oligonucleotide selection for microarrays. In *Proceedings of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002)*, pages 54–63. IEEE Press, August 2002.

[122] Wojciech Rytter. The structure of subword graphs and suffix trees of Fibonacci words. *Theoretical Computer Science*, 363(2):211–223, October 2006.

[123] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of the 11th International Symposium on Algorithms and Computation (ISAAC 2000)*, volume 1969 of *Lecture Notes in Computer Science*, pages 410–421. Springer Verlag, December 2000.

[124] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 225–232. Society for Industrial and Applied Mathematics, January 2002.

[125] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems, in press*, 2007.

[126] Arnold Schönhage, Mike Paterson, and Nicholas Pippenger. Finding the median. *Journal of Computer and Systems Sciences*, 13(2):184–199, October 1976.

[127] Klaus-Bernd Schürmann. Bpr. `http://bibiserv.techfak.uni-bielefeld.de/bpr`. Last visited: June 18, 2007.

[128] Klaus-Bernd Schürmann and Jens Stoye. Counting suffix arrays and strings. *Theoretical Computer Science, to appear*.

[129] Klaus-Bernd Schürmann and Jens Stoye. Suffix tree construction and storage with limited main memory. Technical Report 2003-06, Technische Fakultät, Universität Bielefeld, Germany, August 2003.

[130] Klaus-Bernd Schürmann and Jens Stoye. Counting suffix arrays and strings. Technical Report 2005-04, Technische Fakultät, Universität Bielefeld, Germany, August 2005.

[131] Klaus-Bernd Schürmann and Jens Stoye. Counting suffix arrays and strings. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE 2005)*, volume 3772 of *Lecture Notes in Computer Science*, pages 55–66. Springer Verlag, November 2005.

[132] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. In *Proceedings of the 7th Workshop on Algorithm Engineering*

and Experiments and the 2nd Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO 2005), pages 77–85. Society for Industrial and Applied Mathematics, January 2005.

[133] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, March 2007.

[134] Julian Seward. The *bzip2* homepage. `http://www.bzip.org`. Last visited: June 18, 2007.

[135] Julian Seward. On the performance of BWT sorting algorithms. In *Proceedings of the Data Compression Conference (DCC 2000)*, pages 173–182. IEEE Computer Society, March 2000.

[136] Julian Seward, Nicholas Nethercote, Jeremy Fitzhardinge, and other people. Valgrind. `http://www.valgrind.org`. Last visited: June 18, 2007.

[137] Richard C. Singleton. ACM Algorithm 347: an efficient algorithm for sorting with minimal storage. *Communications of the ACM*, 12(3):185–187, March 1969.

[138] Richard P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, April 1997.

[139] Wojciech Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEE Transactions on Information Theory*, 39(5):1647–1659, September 1993.

[140] Sven Twardziok and Patrick Schwientek. Largescale oligo nucleotide design for microarrays (in german). Bachelor thesis, Technische Fakultät, Universität Bielefeld, Germany, September 2006.

[141] Esko Ukkonen. On-line construction of suffix-trees. Technical Report A-1993-1, Department of Computer Science, University of Helsinki, Finland, 1993.

[142] Esko Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, September 1995.

[143] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, October 1973.

[144] Mikio Yamamoto and Kenneth W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, March 2001.