

Bioinformatics Algorithms

Enno Ohlebusch

Bioinformatics Algorithms

Sequence Analysis, Genome Rearrangements,
and Phylogenetic Reconstruction

Enno Ohlebusch
Institute of Theoretical
Computer Science
Faculty of Engineering
and Computer Science
University of Ulm
89069 Ulm
Germany

ISBN 978-3-00-041316-2

© Enno Ohlebusch 2013

All rights reserved. Except as otherwise expressly permitted under German Copyright Law, no part of this work may be reproduced in any form or by any means or used to make any derivative (such as translation, transformation or adaptation) without prior written permission of the copyright owner.

Cover design: Volker Haese
Printed in Germany

To my family, especially to my parents.

Contents

Preface	xiii
1 Molecular Biology in a Nutshell	1
1.1 Nucleic acids and proteins	1
1.2 Evolution	7
2 Exact String Matching	9
2.1 Basic string definitions	9
2.2 The naive algorithm	11
2.3 The Boyer-Moore-Horspool algorithm	12
2.4 The Knuth-Morris-Pratt algorithm	15
2.5 The Aho-Corasick algorithm for a set of patterns	24
3 Answering Range Minimum Queries in Constant Time	33
3.1 Basic definitions	33
3.2 Range minimum vs. lowest common ancestor	34
3.3 Range minimum queries	42
3.3.1 The sparse table algorithm	42
3.3.2 An optimal algorithm	43
3.4 Completing the proof of correctness	53
4 Enhanced Suffix Arrays	59
4.1 Suffix arrays	59
4.1.1 Linear-time construction	61
4.1.2 Induced sorting	68
4.2 The LCP-array	79
4.2.1 Linear-time construction	79
4.2.2 Longest common prefix	84
4.3 The lcp-interval tree	85
4.3.1 Finding child and parent intervals	88
4.3.2 Bottom-up traversal	93
4.3.3 Top-down traversal	98

4.3.4	Finding child intervals without RMQs	105
4.4	Suffix trees	110
4.4.1	Linear-time construction	113
5	Applications of Enhanced Suffix Arrays	115
5.1	Exact string matching	116
5.1.1	Forward search on suffix trees	116
5.1.2	Forward search on suffix arrays	117
5.1.3	Binary search	120
5.2	Lempel-Ziv factorization	125
5.2.1	Longest previous substring	126
5.2.2	Ultra-fast factorization	134
5.3	Finding repeats	138
5.3.1	Longest repeats	140
5.3.2	Supermaximal repeats	144
5.3.3	Maximal repeats	148
5.3.4	Maximal repeated pairs	149
5.3.5	Non-overlapping repeats	155
5.3.6	Maximal periodicities	157
5.4	Comparing two strings	173
5.4.1	Generalized suffix array	173
5.4.2	Longest common substring	181
5.4.3	Finding exact matches	183
5.5	Traversals with suffix links	185
5.5.1	Suffix links in the suffix tree	185
5.5.2	Suffix links in the lcp-interval tree	186
5.5.3	Computing suffix links space efficiently	187
5.5.4	Matching statistics	194
5.5.5	Merging two suffix arrays in linear time	203
5.6	Comparing multiple strings	206
5.6.1	Generalized suffix array	206
5.6.2	Longest common substring	208
5.6.3	Document frequency	216
5.6.4	Document retrieval	221
5.6.5	Shortest unique substrings	224
5.6.6	A distance measure for genomes	228
5.6.7	All-pairs suffix-prefix matching	231
5.7	String kernels	237
5.7.1	Machine learning	237
5.7.2	Calculating a string kernel	238
5.7.3	Calculating the kernel matrix	243
5.7.4	Classification	243
5.7.5	The TF-IDF weighting scheme	244

5.8	String mining	247
5.8.1	Extraction phase	248
5.8.2	Intersection phase	250
6	Making the Components of Enhanced Suffix Arrays Smaller	257
6.1	Constant time <i>rank</i> and <i>select</i> queries	257
6.2	Compressed suffix and LCP-arrays	262
6.2.1	Compressed suffix array	262
6.2.2	Compressed LCP-array	264
6.3	The balanced parentheses sequence of the LCP-array	265
6.3.1	Finding the parent interval	270
6.3.2	Finding child intervals	272
6.3.3	Computing $getInterval([i..j], c)$	274
6.3.4	Answering RMQs in constant time	275
6.3.5	Computing suffix link intervals	276
6.3.6	Attaching additional information	276
7	Compressed Full-Text Indexes	281
7.1	The components of a compressed full-text index	281
7.2	The Burrows-Wheeler transform	282
7.2.1	Encoding	282
7.2.2	Decoding	284
7.2.3	Data compression	287
7.2.4	Direct construction of the BWT	291
7.3	Backward search	299
7.3.1	A simple FM-index	299
7.3.2	The search algorithm	300
7.4	Wavelet trees	303
7.4.1	Answering <i>rank</i> and <i>select</i> queries	304
7.4.2	Retrieval of $SA[i]$ and the string starting at $SA[i]$	306
7.4.3	Implementation: If σ is a power of 2	307
7.4.4	Implementation: If σ is not a power of 2	310
7.4.5	Other types of wavelet trees	315
7.5	Analyzing a string space efficiently	315
7.5.1	Construction of the LCP-array from the BWT	315
7.5.2	Bottom-up traversal of the lcp-interval tree	321
7.5.3	Shortest unique substrings	322
7.5.4	Top-down traversal of the lcp-interval tree	323
7.5.5	Finding repeats	328
7.5.6	Lempel-Ziv factorization	332
7.6	Space-efficient comparison of two strings	336
7.6.1	Matching statistics	336
7.6.2	Maximal exact matches	340
7.6.3	Merging Burrows-Wheeler transformed strings	342

7.7	Space-efficient comparison of multiple strings	345
7.7.1	Document array, LCP-array, and correction terms	345
7.7.2	Document retrieval with wavelet trees	348
7.7.3	All-pairs suffix-prefix matching	352
7.8	Bidirectional search	357
7.8.1	Burrows-Wheeler transform of the reverse string	358
7.8.2	The suffix array of the reverse string	364
7.8.3	The lcp-array of the reverse string	366
7.8.4	The bidirectional search algorithm	369
7.9	Approximate string matching	374
7.9.1	Using backward search	375
7.9.2	Using bidirectional search	380
8	Sequence Alignment	385
8.1	Pairwise alignment	386
8.1.1	Distance methods	387
8.1.2	Computing an optimal alignment in linear space	393
8.1.3	Edit distance	398
8.1.4	Similarity methods	399
8.1.5	Distance vs. similarity	401
8.1.6	General similarity functions and gap penalties	402
8.2	Multiple alignment	406
8.2.1	Pruning the search space	409
8.2.2	A 2-approximation algorithm	411
8.2.3	Progressive alignment	415
8.3	Whole genome alignment	417
8.3.1	Basic definitions and concepts	418
8.3.2	A global chaining algorithm	420
8.3.3	Alternative data structures	423
8.3.4	Longest/heaviest increasing subsequence	424
9	Sorting by Reversals	429
9.1	Introduction	429
9.2	Basic definitions	437
9.3	The reality-desire diagram	440
9.4	Components	445
9.4.1	Elementary intervals	445
9.4.2	Finding cycles and components	448
9.5	Sorting a permutation without bad components	451
9.6	Dealing with bad components	455
9.6.1	Hurdles	458
9.6.2	A fortress	461
9.7	Sorting by reversals in quadratic time	467
9.7.1	Finding a happy clique	468

9.7.2	Searching the happy clique	473
10	Phylogenetic Reconstruction	477
10.1	Introduction	477
10.1.1	Methods of phylogenetic inference	479
10.1.2	Molecular anthropology	481
10.2	Basic definitions	489
10.3	Ultrametric distance matrices and trees	492
10.3.1	Characterization of ultrametric matrices	494
10.3.2	Construction algorithm	497
10.3.3	The UPGMA-algorithm	501
10.3.4	Fast UPGMA implementation based on quadrees	509
10.4	Additive distance matrices and trees	514
10.4.1	Ultrametric trees revisited	515
10.4.2	Reduction of the additive tree problem	516
10.4.3	Characterization of additive matrices	521
10.4.4	Construction algorithm	524
10.4.5	Splits and quartets	526
10.4.6	Uniqueness of the additive tree	528
10.5	Neighbor-joining algorithms	531
10.5.1	Farris' neighbor-joining algorithm	534
10.5.2	Saitou and Nei's neighbor-joining algorithm	540
10.5.3	Fast neighbor-joining	546
10.6	Non-additive dissimilarity matrices	548
10.6.1	Nearly additive matrices and quartet-consistency	549
10.6.2	Estimating edge weights	561
10.6.3	Bootstrapping	569
	Bibliography	571
	Index	599

Preface

The origins of this book go back to the 1990s, when members of the “Technische Fakultät” (joint Department of Computer Science and Biotechnology) at the University of Bielefeld, Germany, developed curricula for the diploma (equivalent to M.Sc.) in “Naturwissenschaftliche Informatik” (Computer Science in the Natural Sciences). Computer science students could choose either Biology, Chemistry or Physics as their second subject. In Bielefeld, my former supervisor Robert Giegerich was one of the driving forces behind the development of bioinformatics, the interdisciplinary field that combines molecular biology with computer science. At that time, I was a postdoc in his research group working on term rewriting systems. I inevitably became acquainted with bioinformatics, and it proved to be a stroke of luck. Robert prepared lecture notes for a course “Algorithms on Sequences,” and his work was later extended by my former colleague Stefan Kurtz. Parts of Chapter 2 (exact string matching) and Section 8.1 (pairwise alignment) can be traced back to Stefan’s manuscript [194]. It also contained a chapter on suffix trees and their bioinformatics applications. However, the linear-time construction of suffix trees is difficult to understand and teach. In 2003, it was independently shown by several authors that a direct linear-time construction of suffix arrays is possible. One of the algorithms used a rather simple and clever divide and conquer strategy, and this opened up new possibilities. Not only does this simplify teaching—because suffix trees can easily be built in linear time from suffix arrays—but, more importantly, suffix trees can be *completely replaced* by suffix arrays. Algorithms on suffix arrays are not only more space efficient than their counterparts on suffix trees, but they are also faster and easier to implement. I know from discussions with colleagues that a textbook on suffix arrays and their applications would be appreciated, so I wrote one.

One of the problems I encountered in writing this book was the vast literature on the subject. This is particularly true of phylogenetic reconstruction methods. Felsenstein [97] estimates that there are about 3,000 papers on methods for inferring phylogenies, and I have only read a small

fraction of those. Therefore, I apologize in advance to my colleagues if their important work has not been cited.

What the book is about

The primary reason for writing this book was to provide an overview of the state-of-the-art in string algorithms based on index structures. Dan Gusfield published his highly recommended textbook [139] over 15 years ago using the suffix tree as a central data structure. However, suffix trees cannot be used in large-scale applications. As already mentioned, it is possible to replace this index data structure by enhanced suffix arrays. There is an extensive literature, but no textbook, on suffix arrays. This book fills that gap. Additionally, it not only describes classic topics in the field of string algorithms from a new perspective, but also introduces important advanced techniques. For example, recent research focuses on compressed index structures that are based on the Burrows-Wheeler transform. Chapter 7 discusses new approaches in that field by focusing on wavelet trees and backward search, including the latest methods and algorithms. The book focuses on exact string problems (finding exact matches, exact repeats, etc.), rather than on the biologically more relevant inexact string problems (finding approximate matches, degenerate repeats, etc.). However, efficient methods that are robust under errors most often rely on methods that solve the corresponding exact string problem. In other words, one should study the latter in order to understand the former.

Apart from string algorithms, the book presents several other important topics in computer science and bioinformatics within a unified framework. It covers classic topics, such as exact string matching, alignments, and phylogenetic reconstruction as well as newer topics such as constant time range minimum queries and sorting by reversals. The reader should have a solid background in algorithms and data structures in order to fully understand the material covered here.

Emphasis is placed on concepts and methods used to resolve problems, but in contrast to many other texts, this book also puts emphasis on efficient implementations. To give an example, the vast majority of texts on phylogenetic reconstruction explain the UPGMA-algorithm and then state that it runs in quadratic time (if they are interested in the time complexity at all). The experienced reader will be able to find an $O(n^2 \log n)$ time solution, but usually not much more than that. This book provides enough detail to allow construction of an efficient implementation of the algorithm.

Given my background in theoretical computer science, all topics are thoroughly discussed, including proofs of correctness as well as worst-

case time and space complexity analyses. Many examples and figures make the material easier to access. The majority of the algorithms are presented in pseudo-code, so they should be easy to implement. There are exceptions though: To completely implement the space-efficient algorithms described in Chapters 6 and 7 is not an easy task. Fortunately, gifted programmers like my former Ph.D. student Simon Gog and others have developed libraries that can be used for this purpose.¹

How to use this book

The chapters in this book can be read independently, except for Chapters 4–7, and one can use each of the independent chapters as course material. All of the chapters can be used in computer science courses, some for advanced undergraduate students and some for graduate students. (Most parts of the book have been used in courses taught at the University of Ulm, predominantly at the graduate level.) Chapters 8–10 are intended to serve as a text for computer science oriented courses on bioinformatics.

A course on string algorithms and their applications to bioinformatics can be taught by starting with the material in Chapter 4, and then picking topics from Chapter 5. These chapters use range minimum queries, but the reader who is not interested in the details of how to answer such queries in constant time can skip Chapter 3.

Chapter 6 demonstrates that it is possible to implement the algorithms of Chapters 4 and 5 space efficiently, but the material is quite advanced and only suitable for graduate students. It is worth noting that Chapter 6 does not depend on Chapter 5.

The most “modern” part of this book is Chapter 7. Only parts of it are dependent upon material found in Chapters 4 to 6, so most experienced students can immediately begin with Chapter 7. However, it is recommended to read Chapter 4 first. Thus, a course on string algorithms that focuses on large-scale bioinformatics applications can be taught in this way.

Acknowledgments

I am indebted to Stefan Kurtz for sharing his lecture notes [194] with me. I am also much obliged to Johannes Fischer who used the material of Section 5.3.6 in a lecture held at the University of Tübingen in 2008 and gave me feedback on it.

I would like to thank my present and former students Mohamed Abouelhoda, Michael Arnold, Martin Bader, Timo Beller, Katharina Berger, Axel

¹Simon’s library can be found at <https://github.com/simongog/sdsl>.

Fürstberger, Simon Gog, Adrian Kügel, Christoph Mehre, Julian Rüth, Thomas Schnattinger, Florian Schüle, Benjamin Weggenmann, and Maike Zwerger for their assistance in preparing figures, providing implementations, and other important work. It has been a pleasure to work with so many excellent students; many of them are co-authors on scientific publications.

Thanks to my brother-in-law Volker Haese for the cover-design and to my friends Roland Ehle, Michael Schmeisser, and Silvio Weißenborn in helping to bring this book before the public.

Molecular Biology in a Nutshell

1.1 Nucleic acids and proteins

The three major macromolecules that are essential for all known forms of life are deoxyribonucleic acid (DNA), ribonucleic acid (RNA), and proteins.

DNA

Deoxyribonucleic acid is the carrier of genetic information of all known living organisms and many viruses. Most DNA molecules are double-stranded helices, consisting of two long polymers of simple units called nucleotides; see Figure 1.1. A nucleotide is composed of a phosphate group, a five-carbon sugar (2-deoxyribose), and a nucleobase attached to the sugar. There are four nucleotides in DNA that can be distinguished by their bases: adenine (A), cytosine (C), guanine (G), and thymine (T). A single strand of the DNA molecule has a backbone made of alternating sugars and phosphate groups: a phosphate group is linked to the 5'-carbon of the sugar of its nucleotide and the 3'-carbon of the sugar of another nucleotide (the numbers 1' to 5' refer to the carbon atom locations in the sugar structure). Figure 1.2 shows a schematic view of a single-stranded DNA.



Figure 1.1: Schematic view of the DNA double helix.

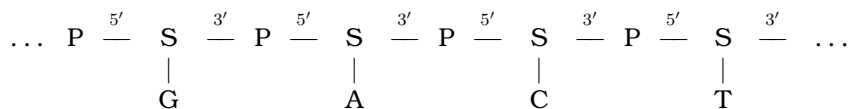


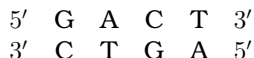
Figure 1.2: A schematic view of a DNA strand. P stands for a phosphate group and S for a sugar 2-deoxyribose.

A single strand of a DNA molecule—a chain of nucleotides—has a direction, conventionally noted as 5' to 3'. If we omit the sugar-phosphate backbone in Figure 1.2, we obtain



We often omit the 5' and 3' markers because the beginning of a DNA sequence is the 5' end unless otherwise stated.

A double-stranded DNA molecule consists of two complementary DNA sequences held together by base pairs. For example, the DNA sequence GACT and its complementary strand can be schematically viewed as



The Watson-Crick base pairs G-C and A-T (guanine-cytosine and adenine-thymine) are formed by hydrogen bonds. G and C are called complementary bases, and so are A and T. The reverse complement of a DNA sequence is obtained by writing its complement with the 5' end on the left and the 3' end on the right. For example, the reverse complement of GACT is AGTC.

The complementary nature of the based-paired structure of a double-stranded DNA molecule provides the basis for replication. DNA replication begins at specific locations called origins. The unwinding of double-stranded DNA into two single strands and the simultaneous synthesis of new strands forms a replication fork; see Figure 1.3. Each of the two single strands serves as a template for the production of its complementary strand: an enzyme called DNA polymerase synthesizes the new DNA by adding nucleotides matched to the template strand. Synthesis always occurs in the 5' to 3' direction. So one strand, the leading strand, can be synthesized continuously while the other, the lagging strand, must be synthesized discontinuously in short fragments, which are later joined. To sum up, after replication there are two identical copies of the original double-stranded DNA molecule. Although replication errors may occur, cellular proofreading ensures that the error rate is kept very low.

DNA molecules may be circular or linear, and they can have an enormous length (the length is measured by the number of base pairs). A DNA

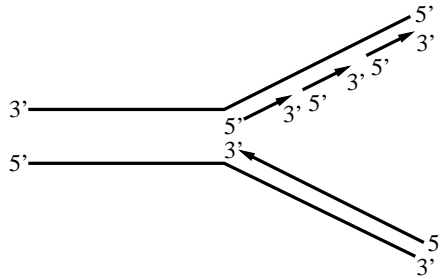


Figure 1.3: Schematic view of the replication fork.

molecule is packaged up tightly into a structure called chromosome. Usually, eukaryotic cells (cells with nuclei, like human cells) have large linear chromosomes, while prokaryotic cells (cells without nuclei, like bacterial cells) have smaller circular chromosomes. The number of chromosomes varies from species to species, but it is constant and characteristic to a given species. The complete set of genetic information is called genome. For example, the human genome is composed of 46 linear chromosomes (23 chromosome pairs) within the cell nucleus (the “nuclear genome”) and a small circular chromosome within the mitochondrion (the “mitochondrial genome”).

RNA and gene expression

Roughly speaking, a gene is a region of a genomic DNA sequence that encodes a protein,¹ and gene expression is the process by which information from a gene is used in the synthesis of a protein. As we shall see, several types of RNAs are involved in gene expression. Like DNA, RNA is made up of a long chain of nucleotides but T (thymine) is replaced with U (uracil). Unlike DNA, however, RNA is almost always a single-stranded molecule and intramolecular base pairings allow it to fold into structures like the cloverleaf structure in Figure 1.4.

Figure 1.5 provides a schematic view of the expression of a prokaryotic gene.² In our example, the coding sequence of the gene is on the upper DNA strand and it starts with ATG. The DNA strand on which the coding sequence is found is said to be the coding strand of the gene and the other DNA strand is said to be the template strand. In a first step, the

¹There are also non-protein coding genes whose products are functional RNAs.

²The expression of a eukaryotic gene is much more difficult because in a eukaryotic gene coding regions (exons) are separated from each other by non-coding regions (introns).

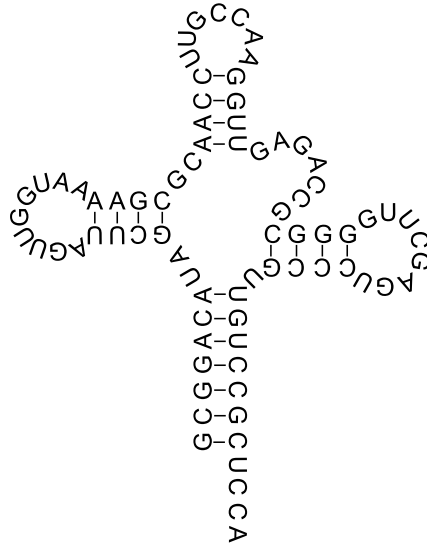


Figure 1.4: Schematic view of a tRNA. Its anticodon GCC (in 5' to 3' direction) appears at the very top.

gene is transcribed into an intermediate RNA molecule called messenger RNA (mRNA) because it now carries the genetic “message.” Transcription is carried out by an enzyme called RNA polymerase, which moves along the DNA in the 5' to 3' direction. It reads the bases from the template strand (and so is reading in the 3' to 5' direction from the point-of-view of the template strand), and synthesizes the mRNA by adding nucleotides matched to the template strand. This means that the final mRNA is a copy of the coding sequence of the gene, in which T is replaced with U.

In a second step,³ the mRNA is translated into a sequence of amino acids based on the genetic code shown in Figure 1.6. The genetic code consists of 64 triplets of nucleotides, called codons. With three exceptions (the STOP codons), each codon encodes for one of the 20 amino acids used in the synthesis of proteins. The codon wheel from Figure 1.6 can be used to decode a codon as follows. Start from the inside of the wheel: find the first nucleotide of the codon in the center of the wheel and work outwards, through the second and third ring (with the next nucleotides) to find the corresponding amino acid. Following this procedure, we find, for example, that AUG codes for the amino acid methionine (AUG is called a start

³In the expression of a eukaryotic gene, there is an intermediate step in which the transcribed pre-mRNA is processed into mature mRNA. Among other things, the introns are removed in a process called splicing.

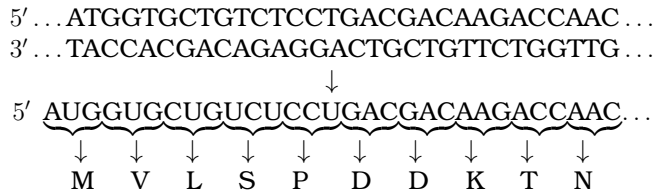


Figure 1.5: A schematic view of gene expression: DNA is transcribed into mRNA, which in turn is translated into a sequence of amino acids.

codon because it is usually the first codon of an mRNA transcript translated by a ribosome). Figure 1.6 also shows the three-letter and one-letter codes of the amino acids; for example, Met and M stand for methionine. Because there are only 20 different amino acids but 64 possible codons, most amino acids are encoded by more than one codon. In other words, the genetic code is redundant (this redundancy minimizes the harmful effects that incorrectly placed nucleotides can have on protein synthesis).

Biological decoding and protein synthesis is accomplished by the ribosome, a large and complex molecular machine made of ribosomal RNA (rRNA) molecules and a variety of proteins. A ribosome can bind to an mRNA molecule and “reads” it in 5′ to 3′ direction. Each of the codons of the mRNA is recognized by a transfer RNA (tRNA) molecule, whose so-called anticodon is the reverse complement of the codon. If, for example, the codon GGC is read by the ribosome, a tRNA with the anticodon GCC (in 5′ to 3′ direction; see Figure 1.4) enters one part of the ribosome and the bases on the codon and anticodon form hydrogen bonds and link:



The 3′ end of the tRNA is covalently linked to a specific amino acid. In our example, this is glycine; cf. Figure 1.6. The attached amino acid is then linked to the growing amino acid chain by another part of the ribosome. Translation stops when a STOP codon (UAA, UAG, or UGA) is reached, and the ribosome dissociates from the mRNA. To sum up, the genetic code is implemented by tRNA molecules. The ribosome facilitates decoding by inducing the binding of tRNAs with complementary anticodon sequences to that of the mRNA, and the amino acids attached to the tRNAs are then linked together. In this way, the mRNA is used as a template for synthesizing the corresponding amino acid sequence. This sequence “folds” into a specific three-dimensional structure, which determines the function of the produced protein.

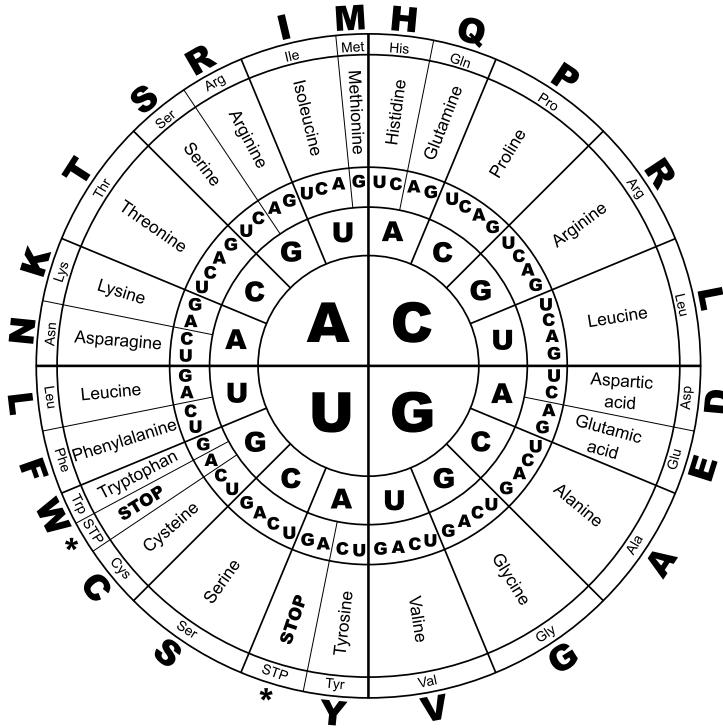


Figure 1.6: The genetic code.

Figure 1.7 summarizes the normal flow of information in molecular biology: DNA can be copied to DNA (DNA replication), DNA information can be copied into mRNA (transcription), and proteins can be synthesized using the information in mRNA as a template (translation).

Proteins

Proteins perform the majority of functions within each cell. The chief characteristic of proteins that allows their diverse set of functions is their ability to bind other molecules specifically and tightly. The region of the protein responsible for binding another molecule is known as the binding site and is often a depression or “pocket” on the molecular surface. This binding ability is mediated by the tertiary structure of the protein, which defines the binding site pocket, and by the chemical properties of the surrounding amino acids’ side chains. Proteins differ from one another primarily in their sequence of amino acids, which is dictated by the DNA

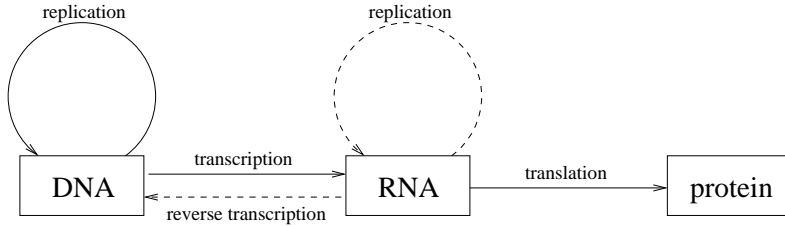


Figure 1.7: Solid arrows illustrate the classic flow of genetic information in molecular biology. Dotted arrows indicate special cases: reverse transcription is the transfer of information from RNA to DNA (it is used e.g. by retroviruses such as HIV), and RNA replication is the copying of one RNA to another (the replication method used by many viruses). In principle, direct translation from DNA to protein is also possible. In [62], Francis Crick writes about the central dogma of molecular biology: “It states that such information cannot be transferred back from protein to either protein or nucleic acid.”

sequence of their genes, and which usually results in folding of the protein into a specific three-dimensional structure that determines its activity.

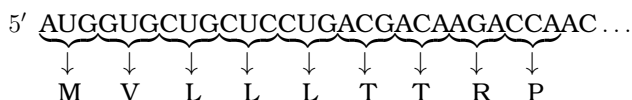
1.2 Evolution

Earth formed approximately 4.5 billion years ago and the planet’s environment has changed with time. There were periods when it changed drastically and there were periods when changes were slow and subtle. Life on earth appeared approximately 3.8 billion years ago and it also changed much with time: When environmental changes are drastic, species become extinct or evolve into new species. This process is called evolution. Its driving forces are mutation and natural selection.

A mutation is a permanent change in the genome (DNA molecules) of an individual. We distinguish between small-scale mutations that affect one or a few nucleotides and large-scale mutations that change the chromosomal structure. Large-scale mutations will be discussed in Chapter 9. Small-scale mutations include:

- Point mutations (often caused by copying errors during DNA replication) exchange a single nucleotide for another.
- Insertions add one or more extra nucleotides into the DNA.
- Deletions remove one or more nucleotides from the DNA.

It is believed that the overwhelming majority of such mutations are neutral mutations that have no significant effect on an organism's fitness, mostly because these mutations occur outside genes.⁴ If a mutation occurs within a gene, it can also be neutral (for instance, if the third nucleotide in the codon GGC is replaced with another nucleotide, the resulting codon will still encode the same amino acid glycine), but in many cases it will be harmful because it alters the product of a gene and may prevent it from functioning properly or completely. For example, cystic fibrosis (one of the most widespread life-shortening genetic diseases) is caused by a mutation in the gene cystic fibrosis transmembrane conductance regulator (CFTR). The most common mutation, $\Delta F508$, is a deletion (hence the Δ) of three nucleotides that results in a loss of the amino acid phenylalanine (F) at the 508-th position in the amino acid sequence of the produced protein. Patients with cystic fibrosis suffer from chronic lung infections, bacterial colonization, pancreatic problems, and reproductive difficulties. So $\Delta F508$ is definitely a harmful mutation. We would like to stress that the insertion or deletion of a number of nucleotides that is not evenly divisible by three can have even more dramatic effects. Because mRNA is read in codons (nucleotide triplets) during translation, such an insertion or deletion changes the reading frame (the grouping of the codons), resulting in a completely different amino acid chain. For example, if we delete the tenth nucleotide from the mRNA shown in Figure 1.5, then we obtain



instead of MVLSPDDKTN. Such a frameshift mutation will also alter the first stop codon encountered in the sequence. The amino acid chain being created could be abnormally short or abnormally long, and will most likely not fold into a functional protein.

So most mutations are either neutral or harmful, but some can be beneficial. This is because mutations can help an organism to adapt to its environment. For example, antibiotic resistance arises by spontaneous gene mutation. If a population of bacteria is exposed to an antibiotic, then any non-resistant bacteria will be killed or inhibited. Antibiotic-resistant bacteria, however, will survive and multiply, passing on their resistant genes from generation to generation. In effect, natural selection is occurring. Natural selection (the term was introduced by Darwin in his influential 1859 book “On the Origin of Species”) is the process by which organisms that best fit to their environment will survive (“survival of the fittest”) and pass their traits to their offspring.

⁴And outside regulatory regions (regions in the DNA that regulate gene expression).

Exact String Matching

Finding all occurrences of a pattern in a text is a problem that arises in many different contexts like text editing, information retrieval, and biological sequence analysis. For example, one of the typical functions of a text editor is to search for the occurrences of a particular string (the pattern) in a document (the text) and to replace them with another string, where both strings are supplied by the user. Another typical application is to search the World Wide Web for information using a web search engine. The user enters a query in form of one or several strings (the patterns) and the search engine returns a list of documents and web pages in which the patterns occur (so in this case, the text is in fact a collection of web pages). In bioinformatics, exact string matching is used to search for particular patterns in DNA sequences.

In this chapter, we present some well-known exact string matching algorithms that do *not* preprocess the text. In large-scale applications like searching for all occurrences of a particular pattern in a genome sequence, however, one usually computes an index of the text to accelerate the search. We will address this issue later.

2.1 Basic string definitions

Definition 2.1.1 An *alphabet* Σ is a finite set, whose elements are called *characters* (or letters). The size of Σ is denoted by $\sigma = |\Sigma|$.

For example, the basic modern Latin alphabet of lowercase letters is

$$\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

and $\sigma = 26$. In most parts of this book, we assume a total order on the alphabet Σ . For example, the standard order of the Latin alphabet is:

$$a < b < c < \cdots < x < y < z$$

In bioinformatics, the DNA alphabet

$$\Sigma = \{A, C, G, T\}$$

plays a central role.

Definition 2.1.2 A *string* S is a finite sequence of characters (where the characters are drawn from some alphabet Σ). The length of a string S , denoted by $|S|$, is the number of characters composing the string. The empty string, denoted by ε , has length 0. Given a string S , its *reverse string* is obtained by reversing the order of the characters in S .

For example, TATAAACG is a string of length 8 on the DNA alphabet. If the alphabet is not specified, then we tacitly assume that it consists of all characters appearing in S .

The concatenation of two strings is the string formed by writing the first, followed by the second, with no intervening space. For example, the concatenation of TATA and AACG is TATAAACG. The juxtaposition is used as the concatenation operator. That is, if u and v are strings, then uv is the concatenation of these two strings. The empty string is the identity for the concatenation operator, i.e., $\varepsilon\omega = \omega = \omega\varepsilon$ for each string ω .

In the biological literature, the word “sequence” is often used instead of “string.” For example, a string on the four-letter alphabet $\{A, C, G, T\}$ is called a DNA sequence, and a string on the twenty-letter amino acid alphabet is called an amino acid sequence (or protein sequence if it represents the primary structure of a protein). Throughout this book, we use the word “sequence” instead of “string” solely in those cases in which the alphabet is either the DNA alphabet or the amino acid alphabet.

For a given alphabet Σ , Σ^n denotes the set of all strings on Σ having length n . Formally, $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^n = \{a\omega \mid a \in \Sigma, \omega \in \Sigma^{n-1}\}$ for $n > 0$. The set of all strings on the alphabet Σ is

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$$

where \mathbb{N} is the set of natural numbers (including zero). The set of all non-empty strings on the alphabet Σ is $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Definition 2.1.3 Given a string S of length n , $S[i..j]$ denotes the *substring*¹ of S that begins at position i and ends at position j (by definition, $S[i..j]$ is

¹The words “string” and “sequence” can be used synonymously but “substring” and “subsequence” can not. In mathematics, a subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. By contrast, a substring is always a consecutive part of a string. This means that a substring of a string is a contiguous subsequence of the string, but a subsequence of a string is not necessarily a substring of the string.

the empty string if $i > j$). In particular, $S[1..n]$ is the whole string S , $S[1..j]$ is the *prefix* of S that ends at position j , and $S[i..n]$ is the *suffix* of S that starts at position i . A substring, prefix, or suffix of S is *proper* if it is not equal to S itself. Furthermore, $S[i]$ denotes the i -th character of S .

For example, AAA is a substring, TAT is a prefix, and CG is a suffix of the string $S = \text{TATAACG}$. Furthermore, $S[2]$ is the character A.

Note that the empty string is a substring, a prefix, and a suffix of any other string.

2.2 The naive algorithm

The formal definition of the exact string matching (or exact string searching) problem reads as follows.

Definition 2.2.1 Let $P \in \Sigma^+$ and $T \in \Sigma^+$ be strings of lengths m and n , respectively. We say that P *occurs* in T *beginning at position* $j + 1$ if $T[j + 1..j + m] = P[1..m]$. The *exact string matching problem* is the problem of finding all positions $j + 1$ in T at which an occurrence of P begins. As customary, we call the string P *pattern* and the string T *text*.

The exact string matching algorithm that immediately comes to mind is the naive algorithm that successively tests the condition $T[j + 1..j + m] = P[1..m]$ for each possible value of j (i.e., $0 \leq j \leq n - m$) by comparing the pattern P character by character (e.g. from left to right) with the substring $T[j + 1..j + m]$ of T . Algorithm 2.1 shows a pseudo-code implementation of this naive algorithm, and Figure 2.1 exemplifies it. In the worst case, Algorithm 2.1 needs $O(m \cdot n)$ time. For example, if $P = a^{m-1}c$ and $T = a^n$, then $m \cdot (n - m + 1)$ character comparisons are necessary.

Algorithm 2.1 The naive exact string matching algorithm: the pattern P is compared from left to right with any substring $T[j + 1..j + m]$ of T .

```

for  $j \leftarrow 0$  to  $n - m$  do
     $i \leftarrow 1$ 
    while  $i \leq m$  and  $T[j + i] = P[i]$  do
         $i \leftarrow i + 1$ 
    if  $i = m + 1$  then
        report match at position  $j + 1$ 

```

The naive algorithm is inefficient because it does not use information gained about the text for one value of j for other values of j . In Section 2.4, we will present an algorithm that keeps track of information about previous character comparisons to speed up the computation.

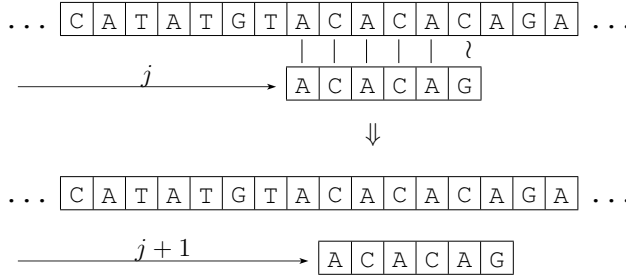


Figure 2.1: The naive string matching algorithm finds that the first five characters of the pattern P match the substring $T[j + 1..j + 5]$, but $T[j + 6] = C \neq G = P[6]$. Thus, the pattern is shifted by one position to the right. Subsequently, the pattern is compared from left to right with the substring $T[j + 2..j + 7]$.

2.3 The Boyer-Moore-Horspool algorithm

Boyer and Moore [41] devised an exact string matching algorithm that works very well in practice, especially for large alphabets. In contrast to the naive algorithm, the Boyer-Moore algorithm compares an m -length substring $T[j + 1..j + m]$ from *right to left* with the pattern $P[1..m]$. It uses two heuristics, the *bad-character heuristic* and the *good suffix heuristic* to avoid much of the work performed by the naive algorithm. These heuristics are so effective that the algorithm typically examines fewer than $m + n$ characters, yielding an expected sublinear-time method. With a certain extension, the Boyer-Moore algorithm has a worst-case running time of $O(m + n)$; we refer the reader to [139] for a thorough investigation of the subject.

Horspool [156] showed that in the normal usage of the algorithm, the *good suffix heuristic* does not make much improvement to the overall speed. He modified Boyer and Moore's *bad-character heuristic* so that it always proposes a shift to the right. Here, we content ourselves with this variant of the Boyer-Moore algorithm, which we call Boyer-Moore-Horspool algorithm. Horspool's *bad-character heuristic* works as follows. If a character mismatch occurs in the right-to-left scan, the pattern is shifted right relative to T so that the last character $c = T[j + m]$ of the current substring $T[j + 1..j + m]$ of T is aligned with the *rightmost* occurrence of c in $P[1..m - 1]$. If c does not occur in $P[1..m - 1]$, then $T[j + 1..j + m]$

can be skipped entirely. The precise shift value depends on the following function λ .

Definition 2.3.1 The function $\lambda : \Sigma \rightarrow \{1, \dots, m-1\}$ is defined by

$$\lambda(c) = \begin{cases} \max\{i \mid 1 \leq i \leq m-1, P[i] = c\} & \text{if } c \text{ occurs in } P[1..m-1] \\ 0 & \text{otherwise} \end{cases}$$

As an example consider the pattern $P = \text{ACACAG}$ on the alphabet $\Sigma = \{\text{A, C, G, T}\}$. Clearly, $\lambda(\text{A}) = 5$, $\lambda(\text{C}) = 4$, $\lambda(\text{G}) = 0$, and $\lambda(\text{T}) = 0$.

The function λ can easily be computed in $O(m)$ time by Algorithm 2.2 (the alphabet size is constant). The reader is invited to apply the algorithm to the pattern $P = \text{ACACAG}$.

Algorithm 2.2 Precomputation of the function λ .

for each $c \in \Sigma$ **do**

$\lambda(c) \leftarrow 0$

for $i \leftarrow 1$ **to** $m-1$ **do**

$\lambda(P[i]) \leftarrow i$

Horspool's heuristic proposes to shift the pattern $m - \lambda(T[j+m])$ positions to the right whenever a character mismatch occurs in the comparison of $P[1..m]$ with $T[j+1..j+m]$. It is readily verified that such a shift is safe, where a shift is called *safe* if it does not miss an occurrence of P in T . We would like to stress that the last character of P is excluded in the computation of the function λ in Algorithm 2.2. This ensures that every shift value is strictly positive.

Algorithm 2.3 Boyer-Moore-Horspool algorithm.

$j \leftarrow 0$

while $j \leq n - m$ **do**

$i \leftarrow m-1$

while $i \geq 0$ and $P[i+1] = T[j+i+1]$ **do**

$i \leftarrow i-1$

if $i < 0$ **then**

report match at position $j+1$

$j \leftarrow j + (m - \lambda(T[j+m]))$

The Boyer-Moore-Horspool algorithm is illustrated in Figure 2.2 and a pseudo-code implementation is shown in Algorithm 2.3. The worst case of Algorithm 2.3 occurs for the pattern $P = ca^{m-1}$ and the text $T = a^n$. This is because every test of the condition $T[j+1..j+m] = P[1..m]$ requires m character comparisons (recall that characters are scanned from right to left) and the pattern is always shifted one position to the right ($m - \lambda(a) = 1$). Consequently, the Boyer-Moore-Horspool algorithm has a worst-case time complexity of $O(m \cdot n)$.

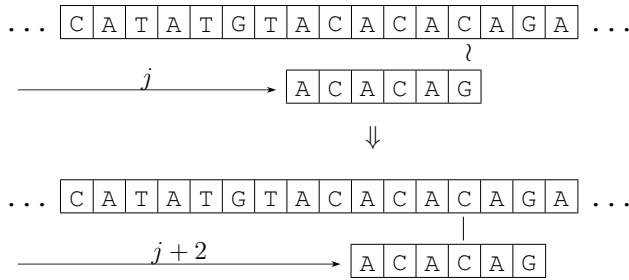


Figure 2.2: The Boyer-Moore-Horspool algorithm compares the pattern $P[1..6]$ from right to left with the substring $T[j+1..j+6]$. The first character comparison of $P[6] = \text{G}$ with $T[j+6] = \text{C}$ yields a mismatch. Therefore, P is shifted $m - \lambda(T[j+m]) = 6 - \lambda(\text{C}) = 2$ positions to the right so that the character $\text{C} = T[j+6]$ is aligned with the rightmost occurrence of C in $P[1..5]$. In the subsequent comparison of P with the substring $T[j+3..j+8]$, the first character comparison (from right to left) of $T[j+8]$ with $P[6]$ yields a match.

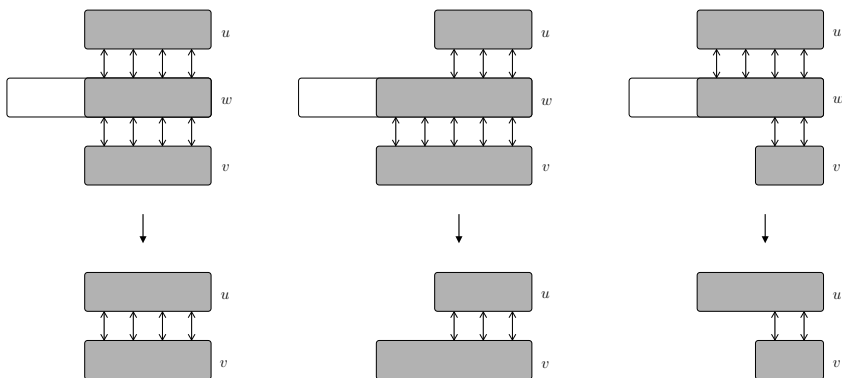


Figure 2.3: A graphical proof of Lemma 2.4.1. Both u and v are suffixes of w . Double-headed arrows connect matching regions (shown shaded). Left: If $|u| = |v|$, then $u = v$. Middle: If $|u| < |v|$, then u is a proper suffix of v . Right: If $|u| > |v|$, then v is a proper suffix of u .

2.4 The Knuth-Morris-Pratt algorithm

Knuth and Pratt invented the first $O(m + n)$ time exact string matching algorithm. Independently, Morris discovered virtually the same algorithm. They published their algorithm jointly [183], and ever since the algorithm is known as the Knuth-Morris-Pratt algorithm. The following lemma is a basic prerequisite to understand the algorithm.

Lemma 2.4.1 *Let the strings u and v both be suffixes of another string w .*

1. *If $|u| = |v|$, then $u = v$.*
2. *If $|u| < |v|$, then u is a proper suffix of v .*
3. *If $|u| > |v|$, then v is a proper suffix of u .*

Proof See Figure 2.3 for a graphical proof. □

The Knuth-Morris-Pratt algorithm compares certain length m substrings of T from left to right with the pattern P . Suppose that in the comparison of $T[j + 1..j + m]$ with P the first q characters match, i.e., $T[j + 1..j + q] = P[1..q]$, but then a mismatch occurs; see Figure 2.4. In this situation, the naive algorithm would shift the pattern one position to the right. Clearly, this shift is safe, i.e., it does not miss an occurrence of P in T . In order to characterize other safe shifts, suppose that there is an occurrence of P in T starting at a position $j' + 1$ with $j + 1 < j' + 1 \leq j + q$. So this

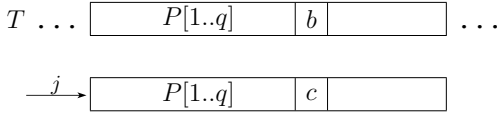


Figure 2.4: The pattern $P[1..m]$ is compared from left to right with the substring $T[j+1..j+m]$ of T . The first q characters match, i.e., $P[1..q] = T[j+1..j+q]$, but $P[q+1] = b \neq c = T[j+q+1]$.

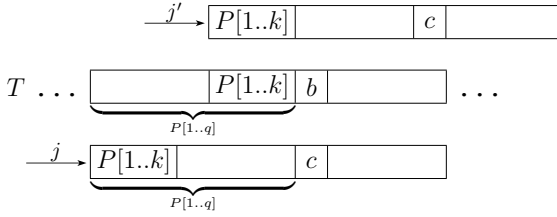


Figure 2.5: If there is a match starting in T at a position $j'+1$ with $j+1 < j'+1 \leq j+q$, then $T[j'+1..j+q]$ must be a prefix of P , say of length k (so $j' = j+q-k$). It follows that the prefix $P[1..k]$ of P must be a proper suffix of $P[1..q]$ because $P[1..q] = T[j+1..j+q]$.

occurrence overlaps with the substring $T[j+1..j+q] = P[1..q]$. It follows as a consequence that the overlapping part must be a prefix $P[1..k]$ of P and a proper suffix of $P[1..q]$; see Figure 2.5.

If we shift the pattern to the right so that the suffix $P[1..k] = T[j+q-k+1..j+q]$ of $T[j+1..j+q]$ is aligned with the prefix $P[1..k]$ of P , then this shift is *safe*, provided that $P[1..k]$ is the *longest* prefix of P that is a proper suffix of $P[1..q]$. (If there was an occurrence of P in T starting at a position $j''+1$ with $j+1 < j''+1 < j'+1$, then the overlapping part would, of course, also be a prefix of P and a proper suffix of $P[1..q]$, and it would be longer than $P[1..k]$. This, however, is impossible.) Moreover, in sharp contrast to the naive algorithm, the first k characters of P are not compared with $T[j+q-k+1..j+q]$, because we know already that they match. Thus, the subsequent comparison starts with $P[k+1]$ and $T[j+q+1]$. Figure 2.6 illustrates the idea of the Knuth-Morris-Pratt algorithm, and in the rest of this section we elaborate on this idea.

In a preprocessing step, the Knuth-Morris-Pratt algorithm computes the so-called *prefix function* π , defined as follows.

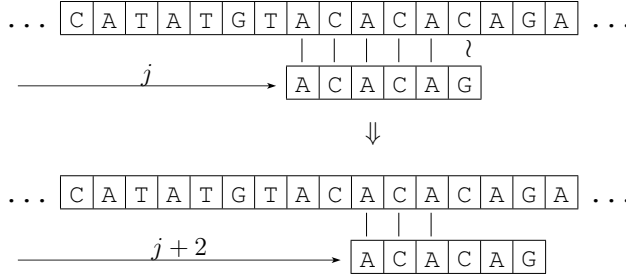


Figure 2.6: In the comparison of the length $m = 6$ substring $T[j + 1..j + 6]$ of T with P , the first $q = 5$ characters match, i.e., $T[j + 1..j + 5] = P[1..5]$, but then a mismatch occurs. $P[1..3] = \text{ACA}$ is the longest prefix of P that is also a proper suffix of $P[1..5] = \text{ACACA}$. The pattern is shifted to the right so that the length 3 suffix $T[j + 3..j + 5] = \text{ACA}$ of $T[j + 1..j + 5]$ is aligned with $P[1..3] = \text{ACA}$. The subsequent comparison starts with the characters $T[j + 6]$ and $P[4]$.

q	1	2	3	4	5	6
$P[1..q]$	A	AC	ACA	ACAC	ACACA	ACACAG
$\pi(P[1..q])$	ε	ε	A	AC	ACA	ε
$\text{prefixtab}[q]$	0	0	1	2	3	0

Figure 2.7: The prefix function π for the pattern $P = \text{ACACAG}$. Because both the parameter and the return value of π are prefixes of P , one can use q —the length of the prefix—instead of $P[1..q]$. This gives the array prefixtab with $\text{prefixtab}[q] = |\pi(P[1..q])|$.

Definition 2.4.2 For every $q \in \{1, \dots, m\}$, $\pi(P[1..q])$ returns the *longest* prefix of P that is a proper suffix of $P[1..q]$.

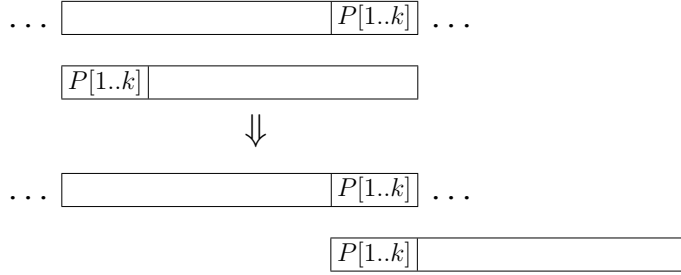
Figure 2.7 shows the prefix function π for the pattern $P = \text{ACACAG}$. We postpone the computation of π until the search phase of the algorithm has been fully developed.

If the pattern $P[1..m]$ is compared with the substring of T starting at position $j + 1$, and $P[1..q]$ is a prefix of $T[j + 1..j + m]$, then the Knuth-Morris-Pratt algorithm proceeds as follows:

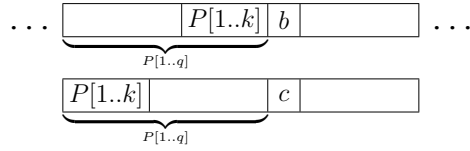
1. If $q = m$, then a match is reported.
 - $\pi(P[1..m])$ returns the longest prefix $P[1..k]$ of P that is also a proper suffix of P ; see case (1) in Figure 2.8.
 - The pattern P is shifted to the right so that the suffix $P[1..k]$ of $T[j + 1..j + m]$ is aligned with the prefix $P[1..k]$ of P ; see case (1) in Figure 2.8.
 - If $P[1..k] = \varepsilon$ (i.e., $k = 0$), this means that P is shifted m positions to the right.
2. If $q < m$, there are the following subcases:
 - a) $b \neq c$ and $q \neq 0$:
 - $\pi(P[1..m])$ returns the longest prefix $P[1..k]$ of P that is a proper suffix of $P[1..q]$; see case (2) in Figure 2.8.
 - The pattern P is shifted to the right so that the suffix $P[1..k]$ of $T[j + 1..j + q]$ is aligned with the prefix $P[1..k]$ of P ; see case (2a) in Figure 2.8.
 - b) $b \neq c$ and $q = 0$:
 - The pattern P is shifted one position to the right; see case (2b) in Figure 2.8.
 - c) $b = c$:
 - The algorithm proceeds with $q + 1$; see case (2c) in Figure 2.8.

In each of the four cases described above, the Knuth-Morris-Pratt algorithm either shifts the pattern at least one position to the right (cases (1), (2a), and (2b)) or it reads the next character of the text (cases (2b) and (2c)). Altogether, the four cases can occur at most $2n$ times. Thus, the Knuth-Morris-Pratt algorithm runs in $O(n)$ time provided that the prefix function π is available. Since π solely depends on the pattern P , it can be precomputed and it will be shown below how to do this in $O(m)$ time. In Algorithm 2.4, the pseudo-code implementation of the Knuth-Morris-Pratt algorithm, the prefix function π is represented by an array *prefixtab* such that $\text{prefixtab}[q] = |\pi(P[1..q])|$; see Figure 2.7.

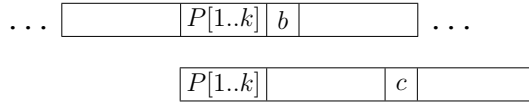
Case (1), $q = m$:



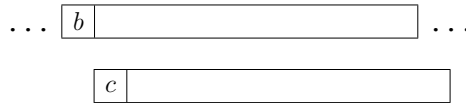
Case (2), $q < m$:



Case (2a), $b \neq c$ and $q \neq 0$:



Case (2b), $b \neq c$ and $q = 0$:



Case (2c), $b = c$:

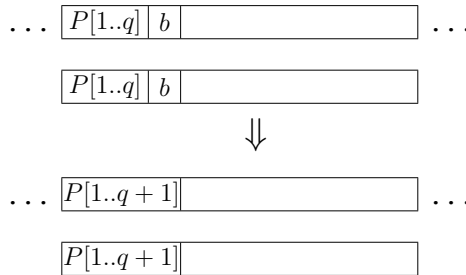


Figure 2.8: The case differentiation in the Knuth-Morris-Pratt algorithm.

Algorithm 2.4 The Knuth-Morris-Pratt algorithm.

```

j ← 0
q ← 0
while j + q < n do
  if q = m /* case (1) */
    report match at position j + 1
    j ← j + q - prefixtab[q]
    q ← prefixtab[q]
  else
    b ← T[j + q + 1]
    c ← P[q + 1]
    if b ≠ c then
      if q > 0 then /* case (2a) */
        j ← j + q - prefixtab[q]
        q ← prefixtab[q]
      else /* case (2b) */
        j ← j + 1
    else /* case (2c) */
      q ← q + 1

```

Precomputation of the prefix function π

Suppose that ω is a non-empty prefix of P . By definition, $\pi(\omega)$ is the longest proper suffix of ω that is also a prefix of P . For $k \geq 1$ we define the iterative application π^k of π as follows:

$$\pi^k(\omega) = \begin{cases} \pi(\omega) & \text{if } k = 1 \\ \pi^{k-1}(\pi(\omega)) & \text{if } k > 1 \text{ and } \pi(\omega) \neq \varepsilon \\ \perp & \text{otherwise} \end{cases}$$

In the example of Figure 2.7, we have $\pi^3(\text{ACACA}) = \pi^2(\text{ACA}) = \pi(\text{A}) = \varepsilon$. Throughout the book, \perp stands for the undefined value.

Lemma 2.4.3 is the key to a linear-time precomputation of the prefix function π , or equivalently, the array *prefixtab*.

Lemma 2.4.3 *Let ω be a non-empty prefix of P and $v \in \Sigma^*$. String v is a proper suffix of ω and a prefix of P if and only if $v = \pi^k(\omega)$ for some $k \geq 1$.*

Proof We show by induction on k that $\pi^k(\omega)$ is a proper suffix of ω and a prefix of P for all k with $\pi^k(\omega) \neq \perp$. For $k = 1$, we have $\pi^k(\omega) = \pi(\omega)$ and the claim follows from the definition of π . The inductive hypothesis states that $v' = \pi^{k-1}(\omega)$ is a proper suffix of ω and a prefix of P . In the inductive step, we must prove the claim for $v = \pi^k(\omega)$. Because v' is a proper suffix of ω and $\pi(v')$ is a proper suffix of v' , it follows that $\pi(v')$ is a proper suffix of ω ; see Figure 2.9 for a graphical proof. Moreover, $\pi(v')$ is a prefix of P .

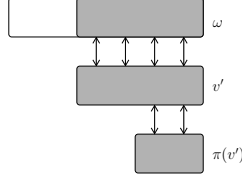


Figure 2.9: If v' is a proper suffix of ω and $\pi(v')$ is a proper suffix of v' , then $\pi(v')$ is also a proper suffix of ω .

Thus, $v = \pi^k(\omega) = \pi(\pi^{k-1}(\omega)) = \pi(v')$ is a proper suffix of ω and a prefix of P . This already proves the if-direction. Conversely, let v be a proper suffix of ω and a prefix of P . We give a proof of contradiction for the fact that $v = \pi^k(\omega)$ for some $k \geq 1$. So suppose that $v \neq \pi^k(\omega)$ for all $k \geq 1$. Because v and all $\pi^k(\omega) \neq \perp$ are proper suffixes of ω , there must be some $q \geq 1$ so that $|\pi^{q+1}(\omega)| < |v| < |\pi^q(\omega)|$. According to Lemma 2.4.1, v is a proper suffix of $\pi^q(\omega)$. By definition, $\pi^{q+1}(\omega) = \pi(\pi^q(\omega))$ is the longest proper suffix of $\pi^q(\omega)$ that is a prefix of P . However, v is a proper suffix of $\pi^q(\omega)$, a prefix of P , and $|\pi^{q+1}(\omega)| < |v|$. This contradiction proves the lemma. \square

We compute π iteratively for all prefixes ω of P , from the shortest to the longest prefix. The shortest non-empty prefix of P consists solely of the first character of P , i.e., $|\omega| = 1$. In this case the empty string ε is the only proper suffix of ω , and ε is a prefix of P . Hence $\pi(\omega) = \varepsilon$, or equivalently, $\text{prefixtab}[1] = 0$. Now let $|\omega| \geq 2$. Then $\omega = uc$ for some non-empty string u and some character c . As we compute π iteratively from the shortest to the longest prefix, the values $\pi(u), \pi^2(u), \dots, \pi^\ell(u)$, where $\pi^\ell(u) = \varepsilon$, are already known and stored. Assume for a moment that $\pi(\omega) \neq \varepsilon$. Because c is the last character of ω and $\pi(\omega)$ is a suffix of ω , c is also the last character of $\pi(\omega)$. Thus, $\pi(\omega) = vc$ for some string v . Clearly, v must be a proper suffix of u because $\pi(\omega) = vc$ is a suffix of $\omega = uc$. Furthermore, since $\pi(\omega)$ is a prefix of P , so is v . Thus, Lemma 2.4.3 is applicable and we infer that $v = \pi^k(u)$ for some $k \geq 1$.

These considerations lead to the following algorithmic idea: If $\omega = uc$, then we iteratively look-up the strings $\pi(u), \pi^2(u), \dots, \pi^\ell(u)$ from the longest to the shortest. All these strings are prefixes of P and we search for the first one, say v , with the property that vc still is a prefix of P . If we find such a string v , then $\pi(\omega) = vc$. If such a v does not exist, then $\pi(\omega) = \varepsilon$. The pseudo-code in Algorithm 2.5 implements this approach.

Since v is a prefix of P in Algorithm 2.5, we can test whether vc is a prefix of P by testing whether $P[|v|+1] = c$ holds true. With this observation, it is easy to compute the array *prefixtab* instead of π ; see Algorithm 2.6. Figure 2.10 exemplifies how Algorithm 2.6 works.

Algorithm 2.5 Linear-time computation the prefix function π .

```

 $\pi(P[1..1]) \leftarrow \varepsilon$ 
for  $q \leftarrow 2$  to  $m$  do
     $c \leftarrow P[q]$ 
     $v \leftarrow \pi(P[1..q-1])$ 
    while  $vc$  is not a prefix of  $P$  and  $v \neq \varepsilon$  do
         $v \leftarrow \pi(v)$ 
    if  $vc$  is a prefix of  $P$  then
         $\pi(P[1..q]) \leftarrow vc$ 
    else
         $\pi(P[1..q]) \leftarrow \varepsilon$ 

```

Algorithm 2.6 Linear-time computation of the array *prefixtab*, version 1.

```

prefixtab[1]  $\leftarrow 0$ 
for  $q \leftarrow 2$  to  $m$  do
     $c \leftarrow P[q]$ 
     $k \leftarrow \text{prefixtab}[q-1]$ 
    while  $P[k+1] \neq c$  and  $k > 0$  do
         $k \leftarrow \text{prefixtab}[k]$ 
    if  $P[k+1] = c$  then
         $\text{prefixtab}[q] \leftarrow k+1$ 
    else
         $\text{prefixtab}[q] \leftarrow 0$ 

```

Algorithm 2.7 Linear-time computation of the array *prefixtab*, version 2.

```

prefixtab[1]  $\leftarrow 0$ 
 $k \leftarrow 0$ 
for  $q \leftarrow 2$  to  $m$  do
     $c \leftarrow P[q]$ 
    while  $P[k+1] \neq c$  and  $k > 0$  do
         $k \leftarrow \text{prefixtab}[k]$ 
    if  $P[k+1] = c$  then
         $k \leftarrow k+1$ 
     $\text{prefixtab}[q] \leftarrow k$ 

```

q	1	2	3	4	5	6
$P[q]$	A	C	A	C	A	G
$prefixtab[q]$	0	0	1	2		

Figure 2.10: Suppose that Algorithm 2.6 computed the values $prefixtab[q]$ for all q with $1 \leq q \leq 4$. To compute $prefixtab[5]$, the condition $P[k+1] \neq c$ of the while-loop in Algorithm 2.6 is evaluated for $k = prefixtab[5-1] = 2$ and $c = A$. Because $P[3] = A$, the body of the while-loop is not executed and $prefixtab[5]$ is set to 3. In the computation of $prefixtab[6]$, the condition $P[k+1] \neq c$ and $k > 0$ of the while-loop is true for $k = prefixtab[6-1] = 3$ and $c = G$ because $P[4] = C$. Thus, in the first execution of the body of the while-loop, k is set to $prefixtab[3] = 1$. The condition $P[k+1] \neq c$ and $k > 0$ of the while-loop again evaluates to true for $k = 1$ and $c = G$, so that k is set to $prefixtab[1] = 0$ in the second execution of the body of the while-loop. Then, since $k = 0$, the while-loop is left. In the final if-then-else statement $prefixtab[6]$ gets the value 0 because $P[1] = A \neq G$.

In order to clearly see that Algorithm 2.6 has a worst-case time complexity of $O(m)$, we transform the program a little. The while-loop is left either because $P[k+1] = c$ or because $k = 0$. Thus, the last four lines in Algorithm 2.6 can be replaced with the last three lines in Algorithm 2.7. Now, the fourth statement of Algorithm 2.6 is superfluous, provided that k is initialized with 0. These considerations yield Algorithm 2.7. We first observe that the correctness of Algorithm 2.7 immediately follows from that of Algorithm 2.6 because it maintains the following invariant: at the start of each iteration of the for-loop we have $k = prefixtab[q-1]$. Obviously, this is true when the loop is first entered and the assignment $prefixtab[q] \leftarrow k$ ensures that it remains true in each successive iteration. Clearly, the worst-case time complexity of Algorithm 2.7 crucially depends on the number of iterations of the while-loop (all other computations take constant time). In each iteration of the while-loop, k is decremented because $k > prefixtab[k]$. How many times that can be is the key question. In each execution of the for-loop, the value of k either increases by one or remains unchanged (at zero). So the total increase of k over the entire algorithm is at most $m-1$. Thus, the total number of decrements (and of executions of the while-loop) is bounded by m . All in all, Algorithm 2.7 runs in $O(m)$ time.

In summary, the Knuth-Morris-Pratt algorithm has a worst-case time complexity of $O(m+n)$. The algorithm can be generalized to search simultaneously for several patterns as we shall see in Section 2.5.

Exercise 2.4.4 Demonstrate an analogy between the precomputation of the prefix function and the search phase of the Knuth-Morris-Pratt algorithm.

2.5 The Aho-Corasick algorithm for a set of patterns

In the previous sections, we were searching for all positions in the text at which an occurrence of the pattern begins. Here, it is convenient to search for all positions at which a pattern ends. We say that a pattern P occurs in text T ending at position j if $T[j - m + 1..j] = P[1..m]$. Clearly, P occurs in T ending at position j if and only if it occurs in T beginning at position $j - m + 1$. So searching for all end positions of such occurrences entails no loss of generality.

In fact, we study a more general problem, namely the problem of finding all occurrences of a set $\mathcal{P} = \{P^1, \dots, P^k\}$ of patterns in a text T . Let $|P^i| = m_i$, $\sum_{i=1}^k m_i = m$, and $|T| = n$. If one uses the Knuth-Morris-Pratt algorithm k times to search for all occurrences of each pattern P^i in T individually, then one can find all occurrences of P^1, \dots, P^k in T in $O(k \cdot n + m)$ time. By contrast, the Aho-Corasick algorithm [8, 139] requires only $O(n + m + z)$ time, where z denotes the overall number of occurrences. In other words, its run time does not only depend on the size of the input, but also on the size of the output. An algorithm with this property is said to be *output-sensitive*. The Aho-Corasick algorithm is based on the following data structure, which is exemplified in Figure 2.11.

Definition 2.5.1 The *trie*² (or *keyword tree*) \mathcal{K} of a set $\mathcal{P} = \{P^1, \dots, P^k\}$ of strings (or keywords) is a rooted tree satisfying the following conditions:

1. Each edge is labeled with exactly one character.
2. Any two edges out of the same node have distinct labels.
3. For any node v in \mathcal{K} , the string \hat{v} must be a prefix of some $P^i \in \mathcal{P}$, where \hat{v} denotes the string obtained by a concatenation of the edge-labels on the path from the root to v .
4. For each string $P^i \in \mathcal{P}$, there is exactly one node v in \mathcal{K} with $\hat{v} = P^i$; this node is labeled with the number i .

Assuming a fixed-size alphabet, the trie of a set $\mathcal{P} = \{P^1, \dots, P^k\}$ of strings can be computed incrementally in $O(m)$ time as follows: Let \mathcal{K}_i denote the trie of the set $\mathcal{P} = \{P^1, \dots, P^i\}$, where $1 \leq i < k$. \mathcal{K}_1 just consists of a single path of $|P^1|$ edges out of the root. Each edge on this path is

²The name comes from retrieval and is pronounced *try*; see [115].

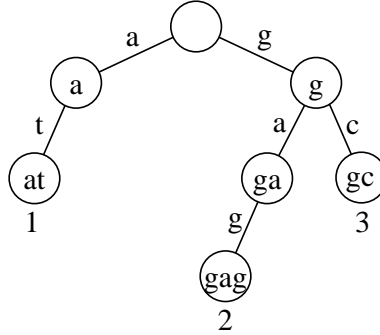


Figure 2.11: Trie of the patterns $P^1 = at$, $P^2 = gag$, and $P^3 = gc$. The labels within the nodes are only for illustrative purposes; they are not part of the trie.

labeled with a character of P^1 and when read from the root, these characters spell out P^1 . To create \mathcal{K}_{i+1} from \mathcal{K}_i , start at the root of \mathcal{K}_i and follow, as far as possible, the unique path in \mathcal{K}_i that matches the characters in P^{i+1} in order. This path is unique because at any branching node v of \mathcal{K}_i the characters on the edges out of v are distinct. If pattern P^{i+1} is exhausted (fully matched), then number the node at which the match ends with the number $i + 1$. If a node v is reached where no further match is possible, but P^{i+1} is not fully matched, then create a new path out of v labeled with the remaining unmatched part of P^{i+1} and number the endpoint of that path with the number $i + 1$; see Figure 2.12 for an example. Obviously, to create \mathcal{K}_{i+1} from \mathcal{K}_i takes $O(m_{i+1})$ time and thus the overall time complexity is $O(m)$.

With the help of the trie of \mathcal{P} , it is possible to use the same ideas as in the Knuth-Morris-Pratt algorithm to efficiently search for all occurrences in T of patterns in \mathcal{P} . To this end, we introduce the following generalization of the prefix function of Definition 2.4.2.

Definition 2.5.2 For any prefix u of a pattern in \mathcal{P} , $\pi(u)$ denotes the longest proper suffix of u that is a prefix of a pattern in \mathcal{P} .

Because every node v in the trie \mathcal{K} of \mathcal{P} represents a prefix of at least one pattern in \mathcal{P} , $\pi(\hat{v})$ is the longest proper suffix of \hat{v} that is also a prefix of a pattern in \mathcal{P} . Moreover, there is a node w in \mathcal{K} so that $\hat{w} = \pi(\hat{v})$. (If $\pi(\hat{v}) = \varepsilon$, then $w = \text{root}$.) Due to the properties of the trie \mathcal{K} , this node w is uniquely determined. These considerations lead to the following definition.

Definition 2.5.3 For each node v in \mathcal{K} , let w be the node in \mathcal{K} so that $\hat{w} = \pi(\hat{v})$. A pointer $\pi_f(v)$ from v to w is called *failure link* of v .

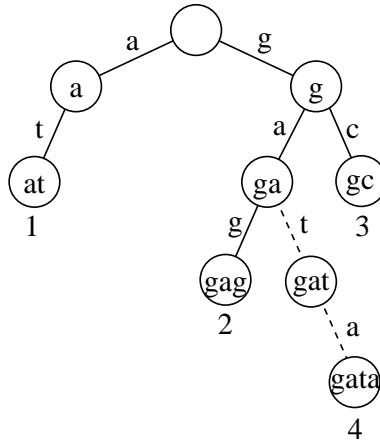


Figure 2.12: The pattern $P^4 = gata$ is inserted into the trie of the patterns $P^1 = at$, $P^2 = gag$, and $P^3 = gc$ by following the path ga to the node v (with the illustrative label ga) and creating the new path ta out of v .

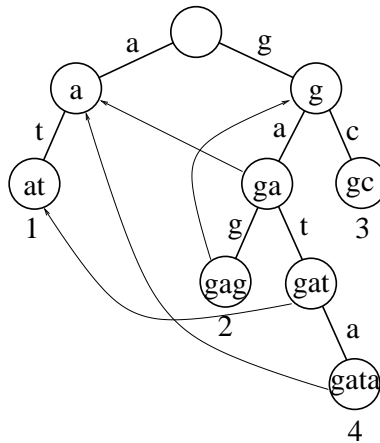


Figure 2.13: Trie of the pattern $P^1 = at$, $P^2 = gag$, $P^3 = gc$, and $P^4 = gata$ with failure links. (For better readability, failure links pointing to the root node are omitted.)

Algorithm 2.8 This version of the Aho-Corasick Algorithm works correctly if no pattern in \mathcal{P} is a proper substring of any other pattern in \mathcal{P} .

```

 $j \leftarrow 1$ 
 $v \leftarrow \text{root}$ 
repeat
    while there is an edge  $(v, v')$  labeled  $T[j]$  do
        if  $v'$  is labeled with number  $i$  then
            report “Pattern  $P^i$  occurs in  $T$  ending at position  $j$ .”
             $v \leftarrow v'$ 
             $j \leftarrow j + 1$ 
        if  $v = \text{root}$  then
             $j \leftarrow j + 1$ 
        else
             $v \leftarrow \pi_f(v) \text{ } / \star$  follow the failure link  $\star /$ 
until  $j > n$ 

```

Figure 2.13 depicts the failure links of our example.

Suppose we know the failure link $\pi_f(v)$ for each node v in \mathcal{K} . We use j to indicate the “current character” $T[j]$ of T to be compared with a character on \mathcal{K} . To understand the use of the failure links in Algorithm 2.8, suppose we have traversed the tree to node v but cannot continue (i.e., character $T[j]$ does not occur on any edge out of v). We know that the string \hat{v} occurs in T starting at position $j - |\hat{v}|$ and ending at position $j - 1$. By the definition of the failure link $\pi_f(v)$, it is guaranteed that string $\pi(\hat{v})$ matches string $T[j - |\pi(\hat{v})|..j - 1]$. That is, the algorithm could traverse \mathcal{K} from the root to node $\pi_f(v)$ and be sure to match all the characters on this path with the characters in T starting from position $j - |\pi(\hat{v})|$. So there is no need to actually make the comparisons on the path from the root to node $\pi_f(v)$. Instead, the comparisons should begin at node $\pi_f(v)$, comparing character $T[j]$ against the characters on the edges out of $\pi_f(v)$. If $|\pi(\hat{v})| = 0$, then the comparisons begin at the root of \mathcal{K} . The only case remaining is when the mismatch occurs at the root. In this case, j must be incremented by 1 and comparisons again begin at the root. A pseudo-code implementation of this method can be found in Algorithm 2.8.

As a matter of fact, Algorithm 2.8 already solves the problem of finding all occurrences of the patterns P^1, \dots, P^k in T provided that no pattern is a proper substring of any other pattern. The proof of this fact is left to the reader (the argument is similar to the one in the Knuth-Morris-Pratt algorithm).

Exercise 2.5.4 Given the trie of the pattern set \mathcal{P} with all failure links, show that Algorithm 2.8 has a worst-case time complexity of $O(n)$. (The argument is also similar to the one used to analyze the search time of the Knuth-Morris-Pratt algorithm.)

However, if a pattern in \mathcal{P} is a proper substring of another pattern in \mathcal{P} , then Algorithm 2.8 may miss occurrences of a pattern in the text. To see this, observe that in our running example, pattern $P^1 = at$ is a proper substring of $P^4 = gata$, and let $T = gataca$. Starting at the root of the trie in Figure 2.13, Algorithm 2.8 follows the matching path $gata$, reports that pattern P^4 occurs in T ending at position 4, and then follows the failure link from node $gata$ to node a . Now, the search phase continues with that node. Obviously, this is not correct because the occurrence of pattern P^1 ending at position 3 is not reported. We overcome this problem with the help of the output function and Lemma 2.5.7.

Definition 2.5.5 The output function *output* assigns to each node v in \mathcal{K} a subset of $\{1, \dots, k\}$ as follows. Let $v, \pi_f(v), \pi_f^2(v), \dots, \pi_f^q(v)$ be the path of failure links from v to the node $\pi_f^q(v) = \text{root}$, and let i_1, \dots, i_ℓ be the set of pattern numbers encountered on that path. Then, $\text{output}(v) = \{i_1, \dots, i_\ell\}$. The set $\text{output}(v)$ is called *output set* of v .

For the proof of Lemma 2.5.7 we need the following auxiliary result, which is a generalization of Lemma 2.4.3.

Lemma 2.5.6 Let v be a node in \mathcal{K} with $v \neq \text{root}$ and $u \in \Sigma^*$. u is a proper suffix of \hat{v} and a prefix of a pattern in \mathcal{P} if and only if $u = \pi^q(\hat{v})$ for some $q \geq 1$.

Proof Similar to the proof of Lemma 2.4.3. □

Lemma 2.5.7 Suppose node v is reached in the comparison of the current character $T[j]$ with a character on \mathcal{K} . Then pattern P^i occurs in T ending at position j if and only if $i \in \text{output}(v)$.

Proof “only-if-direction”: Suppose P^i occurs in T ending at position j . Then P^i must be a suffix of \hat{v} . If $P^i = \hat{v}$, then v is labeled with number i ; hence $i \in \text{output}(v)$. Otherwise, according to Lemma 2.5.6, there is a $q \geq 1$ so that $\pi^q(\hat{v}) = P^i$. Then $v, \pi_f(v), \pi_f^2(v), \dots, \pi_f^q(v)$ is a path of failure links from v to the node $w = \pi_f^q(v)$ and $\hat{w} = P^i$. That is, node $w = \pi_f^q(v)$ is numbered i . Therefore, $i \in \text{output}(v)$.

“if-direction”: Let $i \in \text{output}(v)$ and let w be the node labeled with number i . If $v = w$, then P^i occurs in T ending at position j because $\hat{v} = P^i$. Otherwise, if $v \neq w$, then—by definition of $\text{output}(v)$ —there is a path of failure links $v, \pi_f(v), \pi_f^2(v), \dots, \pi_f^q(v)$ from v to the node $w = \pi_f^q(v)$, $q \geq 1$. This means that $P^i = \hat{w} = \pi^q(\hat{v})$. It follows from Lemma 2.5.6 that P^i is a proper suffix of \hat{v} . Thus, P^i occurs in T ending a position j . □

The full Aho-Corasick search algorithm uses Lemma 2.5.7 and a pseudo-code implementation is given in Algorithm 2.9. Apart from the output, Algorithm 2.9 is precisely identical to Algorithm 2.8. Hence the search time is also $O(n)$, and its total running time is $O(n + z)$, where z is the overall number of occurrences of patterns from \mathcal{P} in the text T .

Algorithm 2.9 Aho-Corasick algorithm.

```

 $j \leftarrow 1$ 
 $v \leftarrow \text{root}$ 
repeat
    while there is an edge  $(v, v')$  labeled  $T[j]$  do
        for each  $i \in \text{output}(v')$  do
            report "Pattern  $i$  occurs in  $T$  ending at position  $j$ ."
             $v \leftarrow v'$ 
             $j \leftarrow j + 1$ 
        if  $v = \text{root}$  then
             $j \leftarrow j + 1$ 
        else
             $v \leftarrow \pi_f(v)$  /* follow the failure link */
until  $j > n$ 

```

Precomputation of the failure links and the output sets

Let us now turn to the computation of the failure links and the output function. Recall that for any node v in \mathcal{K} , $w = \pi_f(v)$ is the unique node in \mathcal{K} so that \hat{w} is the longest proper suffix of \hat{v} that is a prefix of a pattern in \mathcal{P} . Clearly, if $\text{depth}(v) = 0$ (i.e., v is the root) or $\text{depth}(v) = 1$ (i.e., v is a child of the root), then $\pi_f(v) = \text{root}$. Suppose, for some d , $\pi_f(v)$ has been computed for every node v with $\text{depth}(v) \leq d$. The task now is to compute $\pi_f(v)$ for a node v with $\text{depth}(v) = d + 1$. Let v' be the parent of v in \mathcal{K} and let c be the character on the edge (v', v) . We are looking for the node u to which the failure link $\pi_f(v)$ of v points, and we know node $w' = \pi_f(v')$ because $\text{depth}(v') = d$. Just as in the explanation of the Knuth-Morris-Pratt algorithm, the string \hat{u} must be a (not necessarily proper) suffix of \hat{w}' followed by character c . So the first thing to check is whether there is an edge (w', w) out of node $w' = \pi_f(v')$ labeled with character c . If that edge does exist, then $\pi_f(v)$ points to the node w and we are done. Otherwise, \hat{u} is a *proper* suffix of \hat{w}' followed by character c . So we next examine $\pi_f^2(v')$ to see whether there is an edge out of it labeled with character c . Continuing in this way, we arrive at Algorithm 2.10 for computing $\pi_f(v)$.

The output function can also be computed during the computation of the failure links. Clearly, $\text{output}(\text{root}) = \emptyset$. Suppose, for some d , $\text{output}(v)$ has been computed for every node v with $\text{depth}(v) \leq d$. The task now is to compute $\text{output}(v)$ for a node v with $\text{depth}(v) = d + 1$. Note that $\text{depth}(\pi_f(v)) \leq d$. Clearly, $\text{output}(v)$ coincides with $\text{output}(\pi_f(v))$ except for the case in which v is numbered i . In this case, i must also be added to $\text{output}(v)$. To summarize, the output function can be computed with the

Algorithm 2.10 Computing the failure link and the output set for node v .

```

let  $v'$  be the parent of  $v$  in  $\mathcal{K}$ 
let  $c$  be the character on the edge  $(v', v)$ 
 $w' \leftarrow \pi_f(v')$ 
while there is no edge out of  $w'$  labeled  $c$  and  $w' \neq \text{root}$  do
     $w' \leftarrow \pi_f(w')$ 
if there is an edge  $(w', w)$  out of  $w'$  labeled  $c$  then
     $\pi_f(v) \leftarrow w$ 
else
     $\pi_f(v) \leftarrow \text{root}$ 
if  $v$  is numbered  $i$  then
     $\text{output}(v) \leftarrow \{i\} \cup \text{output}(\pi_f(v))$ 
else
     $\text{output}(v) \leftarrow \text{output}(\pi_f(v))$ 

```

help of the failure link $\pi_f(v)$ as follows:

$$\text{output}(v) = \begin{cases} \emptyset & \text{if } v = \text{root} \\ \{i\} \cup \text{output}(\pi_f(v)) & \text{else if } v \text{ is labeled with number } i \\ \text{output}(\pi_f(v)) & \text{otherwise} \end{cases}$$

Algorithm 2.10 first computes the failure link $\pi_f(v)$ of v and then its output set $\text{output}(v)$ with the aid of $\pi_f(v)$.

As an example, consider Figure 2.13 (page 26) and suppose that the failure links and output sets have already been computed for all nodes of depth ≤ 2 . We will compute the failure links and output sets for the nodes of depth 3. First, let v be the node with illustrative label *gag*. Algorithm 2.10 follows the failure link of the parent node *ga* to node *a*. Since there is no edge out of *a* labeled *g*, it further follows the failure link of node *a* to the root node. (At this point, the while-loop is left.) Because there is an edge out of the root node with label *g*, the failure link of node v points to node *g*. Furthermore, as v is labeled with the pattern number 2, it follows that $\text{output}(v) = \{2\} \cup \text{output}(\pi_f(v)) = \{2\}$. Second, let v be the node with illustrative label *gat*. Again, Algorithm 2.10 follows the failure link of the parent node *ga* to node *a*. Because there is an edge out of node *a* labeled *t*, the failure link of node v points to node *at*. It follows from the fact that v is not numbered that $\text{output}(v) = \text{output}(\pi_f(v)) = \{1\}$.

To compute all failure links and output sets, we repeatedly apply Algorithm 2.10 to the nodes in \mathcal{K} in a breadth-first manner, starting at the root. The worst-case time complexity of this method crucially depends on the number of iterations of the while-loop over the entire algorithm because all other computations take constant time (using linked lists to represent the output set of a node v , the statement $\text{output}(v) \leftarrow \{i\} \cup \text{output}(\pi_f(v))$ can indeed be executed in constant time). We will show

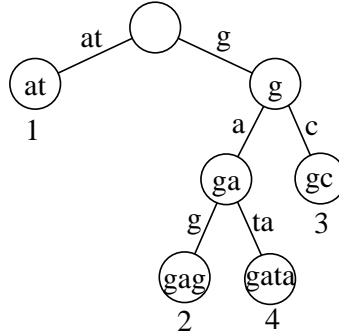


Figure 2.14: PATRICIA tree of the pattern $P^1 = at$, $P^2 = gag$, $P^3 = gc$, and $P^4 = gata$.

that for one pattern, say P^i , the algorithm follows at most $m_i = |P^i|$ failure links. (The argument is very similar to that given in the Knuth-Morris-Pratt algorithm; it is instructive to revisit the precomputation of the prefix function π .) When Algorithm 2.10 is applied to a node v , then we obviously have $\text{depth}(\pi_f(v)) \leq \text{depth}(\pi_f(v')) + 1$, where v' is the parent node of v . So the total increase of depth over all executions of Algorithm 2.10 for nodes on the path for pattern P^i is bounded by m_i . On the other hand, in each iteration of the while-loop, the depth of the current node decreases by at least one because the depth of the node to which its failure link points is strictly smaller than the depth of the node itself. It follows that the while-loop is executed at most m_i times for pattern P_i . Consequently, there are at most m executions of the while-loop over the entire algorithm. Hence, the worst-case time complexity is $O(m)$.

We would like to conclude this section with the following hint to PATRICIA trees [226]. A PATRICIA tree is a *compact trie*, i.e., a representation of a trie where all nodes with one child are merged with their parents. Figure 2.14 shows the PATRICIA tree of our running example.

Answering Range Minimum Queries in Constant Time

This chapter deals with the problem of answering minimum range queries and lowest common ancestor queries in constant time, under the constraint that only linear time is spent in a preprocessing phase. In subsequent chapters we will make extensive use of this, but readers who are not interested in the algorithms that solve this problem may safely skip this chapter. However, they should acquaint themselves with the basic definitions.

3.1 Basic definitions

Definition 3.1.1 Given an array $A[1..n]$ of integers¹ and two indices i and j with $1 \leq i \leq j \leq n$, a *range minimum query* on the interval $[i..j]$ returns an index k so that $A[k] = \min\{A[l] \mid i \leq l \leq j\}$. Such a query will henceforth be denoted by $\text{RMQ}_A(i, j)$.

We stress that our array indexing starts at 1.

By definition, an interval $[i..j]$ with $i > j$ is empty and $\text{RMQ}_A(i, j)$ is undefined in this case. Moreover, if the minimum element in $A[i..j]$ occurs more than once in $A[i..j]$, then the index of the minimum element is not unique. In this case, we assume that $\text{RMQ}_A(i, j)$ returns the first index of the minimum element in $A[i..j]$, unless stated otherwise.

As an example, consider the array $A = [8, 4, 6, 2, 12, 10, 14, 6]$. The range minimum query $\text{RMQ}_A(5, 8)$, returns the index 8 because $A[8] = 6$ is the minimum element in $A[5..8] = [12, 10, 14, 6]$.

¹Here, the array elements are integers, i.e., elements of the totally ordered set \mathbb{Z} . However, one can use any other totally ordered set instead of the integers.

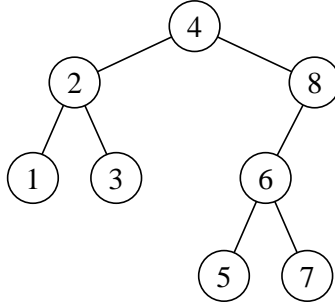


Figure 3.1: In this tree, $\text{LCA}_T(5, 7)$ returns the node 6, while $\text{LCA}_T(1, 7)$ returns the root node 4.

Definition 3.1.2 Given a rooted tree T and two nodes u and v in T , a *lowest common ancestor query* on u and v , denoted by $\text{LCA}_T(u, v)$, returns the node farthest from the root that is an ancestor of both u and v .

Recall that in a rooted tree the ancestors of a node u are the nodes on the direct path from u to the root of the tree. Therefore, the lowest common ancestor of u and v is the node at which the path from u to the root meets the path from v to the root. As an example, consider the tree T in Figure 3.1. The lowest common ancestor query $\text{LCA}_T(5, 7)$ returns the node 6, while $\text{LCA}_T(1, 7)$ returns the root node 4.

Early papers on finding lowest common ancestors in trees are [9] and [306]. Harel and Tarjan [146] showed that a fixed tree can be preprocessed in linear time and space so that LCA queries can be answered in constant time. Their algorithm was simplified by Schieber and Vishkin [283], but it cannot be called “simple” algorithm. Gabow et al. [117] showed that the problems of answering range minimum queries and lowest common ancestor queries are linearly equivalent. Berkman and Vishkin [39] observed that the reduction from LCA to RMQ via the Euler-Tour (see Section 3.2) is in fact a reduction to a special RMQ-problem, in which consecutive array elements differ by one. They solved this restricted RMQ-problem and obtained a new algorithm for finding lowest common ancestors in constant time. Bender and Farach-Colton [35] later provided a simplified presentation of their algorithm.

3.2 Range minimum vs. lowest common ancestor

In this section, it will be shown that range minimum queries can be answered in constant time (with linear time and space preprocessing) if and

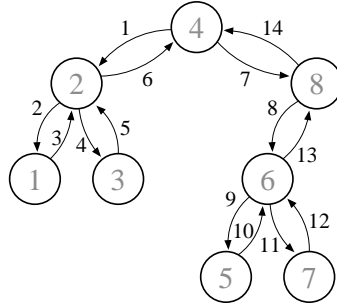


Figure 3.2: The Euler Tour 4, 2, 1, 2, 3, 2, 4, 8, 6, 5, 6, 7, 6, 8, 4 of the tree from Figure 3.1 is obtained by following the consecutively numbered arrows.

only if lowest common ancestor queries can be answered in constant time (with linear-time preprocessing). We follow the approach of Gabow et al. [117] and Bender & Farach-Colton [35].

Definition 3.2.1 Let T be a rooted tree with n nodes, numbered from 1 to n . We perform the following operations recursively at each node, starting with the root node of T :

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

Suppose that during this so-called *preorder traversal*² the label of the new node encountered is written down every time an edge is traversed. This sequence of nodes is called the *Euler tour* of T . Because the traversal visits each of the $n - 1$ edges twice (once in each direction), the Euler tour of T has length $2n - 1$. We define the *representative* of a node in an Euler tour to be the index of the first occurrence of that node in the Euler tour.

The preceding definition is exemplified in Figure 3.2.

We say that an algorithm with preprocessing time $p(n)$ and query time $q(n)$ has time complexity $\langle p(n), q(n) \rangle$.

Theorem 3.2.2 *If there exists an $\langle O(n), O(1) \rangle$ time algorithm for answering range minimum queries, then there is also an $\langle O(n), O(1) \rangle$ time algorithm for finding lowest common ancestors.*

²A preorder traversal is also called depth-first traversal.

Proof Given a rooted tree T with n nodes (numbered from 1 to n), proceed as follows:

1. Compute the Euler tour of T and store it in array $E[1..2n-1]$.
(In our example, $E[1..15] = [4, 2, 1, 2, 3, 2, 4, 8, 6, 5, 6, 7, 6, 8, 4]$.)
2. Compute the depths³ of all nodes in the Euler tour $E[1..2n-1]$ and store them in array $D[1..2n-1]$. That is, $D[i]$ is the length of the path from node $E[i]$ to the root of T .
(In our example, $D[1..15] = [0, 1, 2, 1, 2, 1, 0, 1, 2, 3, 2, 3, 2, 1, 0]$.)
3. Compute the representative of each node and store it in array $R[1..n]$.
(In our example, $R[1..8] = [3, 2, 5, 1, 10, 9, 12, 8]$.)

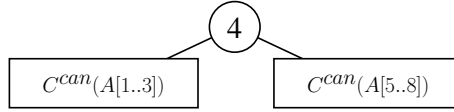
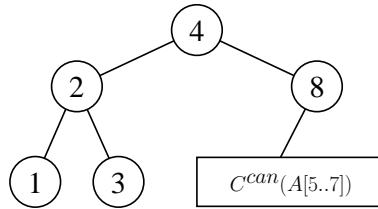
Each of the three computations takes $O(n)$ time. With the help of these arrays, $\text{LCA}_T(u, v)$ can be computed as follows: The nodes in the Euler tour $E[1..2n-1]$ between the first visits to u and v are $E[R[u]..R[v]]$ (or $E[R[v]..R[u]]$ if v is encountered first). The key observation is that the shallowest node encountered between the first visits to nodes u and v is the lowest common ancestor of these nodes. Because the depths of $E[R[u]..R[v]]$ are stored in $D[R[u]..R[v]]$, the range minimum query $\text{RMQ}_D(R[u], R[v])$ returns the position of the shallowest node in $E[R[u]..R[v]]$. Consequently, $\text{LCA}_T(u, v) = E[\text{RMQ}_D(R[u], R[v])]$. Clearly, $E[\text{RMQ}_D(R[u], R[v])]$ can be computed in constant time because each of the three array references takes $O(1)$ time and we assume that range minimum queries can be answered in $O(1)$ time. It is easily seen that the required preprocessing can be done in linear time. This is because array D can be preprocessed in $O(n)$ time so that range minimum queries can be answered in constant time, and the computation of the three arrays also takes $O(n)$ time. \square

In order to show the converse of Theorem 3.2.2, we introduce the Cartesian tree of an array as a key concept; this data structure was invented by Vuillemin [321].

Definition 3.2.3 Let $A[l..r]$ be an array of integers. A *Cartesian tree* $\mathcal{C}(A)$ of A is a labeled binary tree defined as follows:

- The root of $\mathcal{C}(A)$ corresponds to the minimum element of A , and the root is labeled with a position i of this minimum.
- The left child of the root is the Cartesian tree of $A[l..i-1]$ if $i > l$, otherwise it has no left child.
- The right child of the root is the Cartesian tree of $A[i+1..r]$ if $i < r$, otherwise it has no right child.

³In a rooted tree, the depth of a node u is the length of the path from u to the root. The root itself has depth 0.

Figure 3.3: Computing the Cartesian tree of $A = [8, 4, 6, 2, 12, 10, 14, 6]$.Figure 3.4: Computing the Cartesian tree of $A = [8, 4, 6, 2, 12, 10, 14, 6]$.

Because of the recursive nature of this definition, it subsumes Cartesian trees of arrays of size n that are not indexed 1 through n . However, we are not interested in such arrays. Henceforth, we will assume that an array of size n is indexed 1 through n , unless stated otherwise. Consequently, the n nodes of a Cartesian tree of an array of size n are labeled with numbers from 1 to n .

If the array A contains multiple occurrences of an element, then its Cartesian tree may not be unique. To avoid this, we define a strict order \prec by $(A[i], i) \prec (A[j], j)$ if and only if $A[i] < A[j]$, or $A[i] = A[j]$ and $i < j$. With respect to this order, the minimum element in A (and in each subarray of A) is unique. Consequently, the Cartesian tree of A w.r.t. \prec is unique. It is called *canonical* Cartesian tree of A and denoted by $C^{can}(A)$.

Again, we use the array $A = [8, 4, 6, 2, 12, 10, 14, 6]$ as an example. The minimum element 2 in $A[1..8]$ occurs at position 4, thus the root of $C^{can}(A)$ is labeled with 4 and we have to compute the canonical Cartesian trees of $A[1..3]$ and $A[5..8]$ recursively; see Figure 3.3. Let us further compute $A[5..8]$. Since the minimum element 6 in $A[5..8]$ occurs at position 8, there is no right child, and we merely have to compute the canonical Cartesian tree of $A[5..7]$; see Figure 3.4. All in all, $C^{can}(A)$ turns out to be the tree from Figure 3.1.

Definition 3.2.4 The *rightmost path* in a Cartesian tree (or more generally in a binary tree) is obtained by starting at the root of the tree and following right child pointers. The path ends at the first node that has no right child; see Figure 3.5.

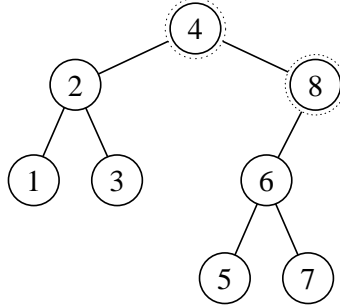


Figure 3.5: The rightmost path of this tree consists of the nodes 4 and 8.

We next show how $\mathcal{C}^{can}(A)$ can be built incrementally, i.e., for every i with $1 \leq i \leq n-1$ we build $\mathcal{C}^{can}(A[1..i+1])$ from $\mathcal{C}^{can}(A[1..i])$. Moreover, in each construction step from $\mathcal{C}^{can}(A[1..i])$ to $\mathcal{C}^{can}(A[1..i+1])$, we identify the number l_{i+1} of nodes that are on the rightmost path in $\mathcal{C}^{can}(A[1..i])$ but not on the rightmost path in $\mathcal{C}^{can}(A[1..i+1])$. Consequently, after the construction of $\mathcal{C}^{can}(A)$, we have a sequence l_1, l_2, \dots, l_n of these numbers, which characterizes $\mathcal{C}^{can}(A)$ as we shall see later.

Initially, the canonical Cartesian tree of $A[1..1]$ consists just of a root labeled with position 1 and $l_1 = 0$.

Let v_1, \dots, v_k be the nodes on the rightmost path in $\mathcal{C}^{can}(A[1..i])$ and let p_1, \dots, p_k be their labels. It follows from the definition of the (canonical) Cartesian tree that the node with label $i+1$ must be at the end of the rightmost path in $\mathcal{C}^{can}(A[1..i+1])$. Therefore, we climb up the rightmost path in $\mathcal{C}^{can}(A[1..i])$ until we find the position where $i+1$ belongs. More precisely, starting with $m = k$, we decrease m by 1 as long as $A[p_m] > A[i+1]$ holds. Then we proceed by case analysis.

1. If $m = k$, i.e., $A[p_k] \leq A[i+1]$, then a new node w with label $i+1$ becomes the right child of v_k and $l_{i+1} = 0$; see Figure 3.6.
2. If $m = 0$, i.e., $A[p_1] > A[i+1]$, then a new node w with label $i+1$ becomes the root of the tree and $\mathcal{C}^{can}(A[1..i])$ becomes its left child. In this case, $l_{i+1} = k$ nodes are removed from the rightmost path in $\mathcal{C}^{can}(A[1..i])$; see Figure 3.7.
3. If $1 \leq m \leq k-1$, then m is the index so that $A[p_m] \leq A[i+1]$ and $A[p_{m'}] > A[i+1]$ for all $m < m' \leq k$. In this case a new node w with label $i+1$ becomes the right child of v_m and the subtree rooted at v_{m+1} becomes the left child of w . Note that in this case $l_{i+1} = k - m$ nodes are removed from the rightmost path in $\mathcal{C}^{can}(A[1..i])$; see Figure 3.8.

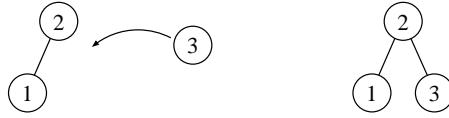


Figure 3.6: The step from $\mathcal{C}^{can}(A[1..2])$ to $\mathcal{C}^{can}(A[1..3])$ corresponds to case (1) in the algorithm that incrementally builds $\mathcal{C}^{can}(A)$.

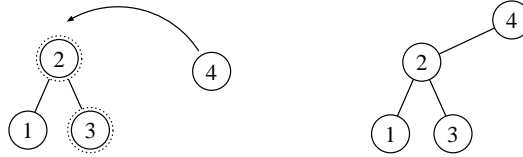


Figure 3.7: The step from $\mathcal{C}^{can}(A[1..3])$ to $\mathcal{C}^{can}(A[1..4])$ corresponds to case (2) in the algorithm that incrementally builds $\mathcal{C}^{can}(A)$. The nodes 2 and 3 occur on the rightmost path in $\mathcal{C}^{can}(A[1..3])$ but not on the rightmost path in $\mathcal{C}^{can}(A[1..4])$, hence $l_4 = 2$.

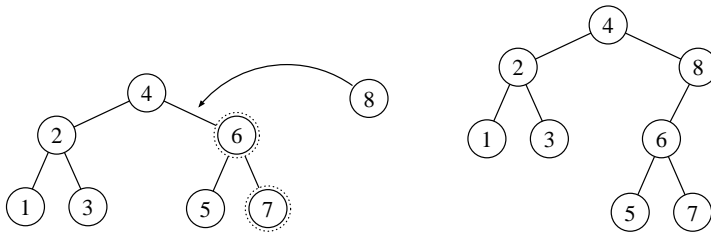


Figure 3.8: The step from $\mathcal{C}^{can}(A[1..7])$ to $\mathcal{C}^{can}(A[1..8])$ corresponds to case (3) in the algorithm that incrementally builds $\mathcal{C}^{can}(A)$. The nodes 6 and 7 occur on the rightmost path in $\mathcal{C}^{can}(A[1..7])$ but not on the rightmost path in $\mathcal{C}^{can}(A[1..8])$, hence $l_8 = 2$. The whole sequence l_1, \dots, l_8 is 0, 1, 0, 2, 0, 1, 0, 2.

The worst-case time complexity of the algorithm is obtained by the following amortized analysis. In each step exactly one node—the new node with label $i + 1$ —is added to the (rightmost path of the new) tree. Finding the position where $i + 1$ belongs requires walking up the rightmost path in $\mathcal{C}^{can}(A[1..i])$. Whenever the value of the variable m is decreased in this walk, a node (the node v_m) leaves the rightmost path, and it is not traversed again. Since the overall number of nodes on all rightmost paths during the construction is bounded by n , the overall construction time of $\mathcal{C}^{can}(A)$ is $O(n)$.

Using the same kind of reasoning as above, we can show that the sequence of natural numbers l_1, l_2, \dots, l_n satisfies

$$\sum_{k=1}^i l_k < i \text{ for all } 1 \leq i \leq n$$

As already mentioned, one can remove at most as many nodes from the rightmost path in $\mathcal{C}^{can}(A[1..i])$ as have been inserted. This, in combination with the fact that in each step exactly one node joins the rightmost path, implies $\sum_{k=1}^i l_k \leq i$. The inequality is strict because at least one node (the root node) remains on the rightmost path.

Definition 3.2.5 For every $n \in \mathbb{N}$, \mathcal{L}_n denotes the set of all sequences of natural numbers l_1, l_2, \dots, l_n satisfying $\sum_{k=1}^i l_k < i$ for all $1 \leq i \leq n$.

Exercise 3.2.6 Given array A of size n , the sequence l_1, l_2, \dots, l_n can be computed without actually constructing $\mathcal{C}^{can}(A)$. Explain why Algorithm 3.1 does this correctly.

Algorithm 3.1 Given array A of size n , compute the sequence l_1, l_2, \dots, l_n .

```

 $R[0] \leftarrow -\infty$       /* array  $R$  stores the (current) values on the rightmost path */
for  $i \leftarrow 1$  to  $n$  do
     $l[i] \leftarrow 0$       /* array  $l$  stores the sequence  $l_1, l_2, \dots, l_n$  */
     $q \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $n$  do
        while  $R[q] > A[i]$  do
             $l[i] \leftarrow l[i] + 1$ 
             $q \leftarrow q - 1$       /* remove  $R[q]$  from the rightmost path */
         $q \leftarrow q + 1$ 
         $R[q] \leftarrow A[i]$       /*  $A[i]$  is the new rightmost leaf */
    return array  $l$ 

```

The following characterization of the canonical Cartesian tree of an array A will turn out to be very useful.

Lemma 3.2.7 *For two arrays $A[1..n]$ and $B[1..n]$, we have $\mathcal{C}^{can}(A) = \mathcal{C}^{can}(B)$ if and only if $l_1^A, l_2^A, \dots, l_n^A = l_1^B, l_2^B, \dots, l_n^B$, where $l_1^A, l_2^A, \dots, l_n^A$ and $l_1^B, l_2^B, \dots, l_n^B$ are the sequences of natural numbers obtained in the construction of $\mathcal{C}^{can}(A)$ and $\mathcal{C}^{can}(B)$, respectively.*

Proof If $l_1^A, l_2^A, \dots, l_n^A = l_1^B, l_2^B, \dots, l_n^B$, then exactly the same steps were applied in the construction of $\mathcal{C}^{can}(A)$ and $\mathcal{C}^{can}(B)$. Thus, $\mathcal{C}^{can}(A) = \mathcal{C}^{can}(B)$.

Conversely, suppose that $l_1^A, l_2^A, \dots, l_n^A \neq l_1^B, l_2^B, \dots, l_n^B$. Let $i + 1$ be the first index at which the sequences differ, i.e., $l_1^A, l_2^A, \dots, l_i^A = l_1^B, l_2^B, \dots, l_i^B$ and $l_{i+1}^A \neq l_{i+1}^B$. Without loss of generality, we may assume that $l_{i+1}^A > l_{i+1}^B$, hence $l_{i+1}^A > 0$. Note that $l_1^A, l_2^A, \dots, l_i^A = l_1^B, l_2^B, \dots, l_i^B$ implies $\mathcal{C}^{can}(A[1..i]) = \mathcal{C}^{can}(B[1..i])$.

As $l_{i+1}^A > 0$, at least one of the nodes on the rightmost path in $\mathcal{C}^{can}(A[1..i])$ is removed in the construction step from $\mathcal{C}^{can}(A[1..i])$ to $\mathcal{C}^{can}(A[1..i+1])$. Let r be the root of the subtree that becomes the left child of the new node $i + 1$ in $\mathcal{C}^{can}(A[1..i+1])$. In the construction step from $\mathcal{C}^{can}(B[1..i])$ to $\mathcal{C}^{can}(B[1..i+1])$, the new node $i + 1$ is inserted below the node r because $l_{i+1}^A > l_{i+1}^B$. Consequently, in $\mathcal{C}^{can}(A[1..i+1])$ (and hence in $\mathcal{C}^{can}(A)$) node $n + 1$ is an ancestor of node r , while in $\mathcal{C}^{can}(B[1..i+1])$ (and hence in $\mathcal{C}^{can}(B)$) node r is an ancestor of node $n + 1$. Therefore, $\mathcal{C}^{can}(A) \neq \mathcal{C}^{can}(B)$. \square

Now we prove the converse of Theorem 3.2.2.

Theorem 3.2.8 *If there is an $\langle O(n), O(1) \rangle$ time algorithm for finding lowest common ancestors, then there is also an $\langle O(n), O(1) \rangle$ time algorithm for answering range minimum queries.*

Proof Given array $A[1..n]$, we build $\mathcal{C}^{can}(A)$ in linear time, and prepare it in linear time for constant time lowest common ancestor queries. For ease of presentation, let us identify each node in $\mathcal{C}^{can}(A)$ with its label. Then, for any two nodes i and j with $i < j$ in $\mathcal{C}^{can}(A)$, we have

$$\text{RMQ}_A(i, j) = \text{LCA}_{\mathcal{C}^{can}(A)}(i, j)$$

To see this, consider the node $k = \text{LCA}_{\mathcal{C}^{can}(A)}(i, j)$ (i.e., the lowest common ancestor of i and j in $\mathcal{C}^{can}(A)$ is labeled with some index k). According to the definition of the canonical Cartesian tree of A , we have $i \leq k \leq j$. We claim that $A[k]$ is the minimum element in the subarray $A[i..j]$. If there were an element $A[k'] < A[k]$ with $i \leq k' \leq j$, then the node with label k' would be an ancestor of node k and it would separate i and j (i.e., i is in its left child and j is in its right child). Consequently, this node k' would be the lowest common ancestor of i and j , a contradiction. Similarly, one can show that k is the first position in A at which the minimum element appears, i.e., every other index l with $i \leq l \leq j$ and $A[l] = A[k]$ must satisfy $l > k$. Hence, $\text{RMQ}_A(i, j) = k$. \square

3.3 Range minimum queries

It is the goal of this section to derive an $\langle O(n), O(1) \rangle$ time algorithm for answering range minimum queries, but we start with some elementary considerations.

A “brute-force” algorithm *without* preprocessing would scan the array A from index i to index j for every query $\text{RMQ}_A(i, j)$. In the worst case, it has time complexity $\langle O(1), O(n) \rangle$.

A naive algorithm *with* preprocessing would compute the answers to all queries in advance and store them in a look-up table RMQ . Given this $n \times n$ matrix RMQ , a range minimum query on the interval $[i..j]$ can then be answered in constant time by a table look-up $\text{RMQ}[i, j]$. In fact, we merely need a triangular matrix because we are solely interested in queries on intervals $[i..j]$ with $i \leq j$. The triangular matrix RMQ can be computed in $O(n^2)$ time by a dynamic programming algorithm using $\text{RMQ}[i, i] = i$ and the recurrence

$$\text{RMQ}[i, j] = \begin{cases} i & \text{if } A[i] \leq A[\text{RMQ}(i+1, j)] \\ \text{RMQ}(i+1, j) & \text{otherwise} \end{cases}$$

Exercise 3.3.1 Give pseudo-code for the dynamic programming algorithm that computes the look-up table (the triangular matrix).

The naive algorithm with preprocessing has time complexity $\langle O(n^2), O(1) \rangle$. Constant query time is exactly what we want, but the quadratic preprocessing time can be improved, as we shall see next.

3.3.1 The sparse table algorithm

The preprocessing time can be reduced to $O(n \log n)$ time by precomputing and storing only the answers to range minimum queries on intervals whose length is a power of two. That is, for every starting position i with $1 \leq i \leq n$ and every j with $0 \leq j \leq \lfloor \log n \rfloor$, we compute the first position of the minimum element in the “block” $A[i..i + 2^j - 1]$ of size 2^j and store it in $M[i, j]$. Formally, if $i + 2^j - 1 \leq n$, then $M[i, j]$ is the smallest k with $i \leq k \leq i + 2^j - 1$ and $A[k] = \min\{A[i..i + 2^j - 1]\}$. The value of $M[i, j]$ is undefined whenever $i + 2^j - 1 > n$.

Table M is an $n \times (\lfloor \log n \rfloor + 1)$ matrix that can be filled in $O(n \log n)$ time by a dynamic programming algorithm using $M[i, 0] = i$ and the fact that the position of the minimum element in a block of size 2^j starting at position i can be obtained by comparing the minima of its two constituent blocks $A[i..i + 2^{j-1} - 1]$ and $A[i + 2^{j-1}..i + 2^j - 1]$ of size 2^{j-1} . Formally, for $1 \leq i \leq n$ and $1 \leq j \leq \lfloor \log n \rfloor$ so that $i + 2^j - 1 \leq n$, we have

$$M[i, j] = \begin{cases} M[i, j-1] & \text{if } A[M[i, j-1]] \leq A[M[i + 2^{j-1}, j-1]] \\ M[i + 2^{j-1}, j-1] & \text{otherwise} \end{cases}$$

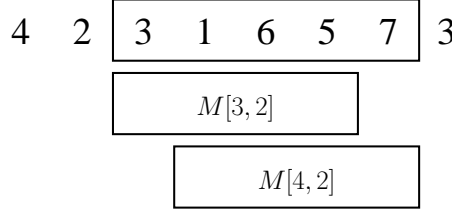


Figure 3.9: Two overlapping blocks that exactly cover the interval $[3..7]$.

As already mentioned, $M[i, j]$ is undefined whenever $i + 2^j - 1 > n$.

In order to answer a range minimum query on the interval $[i..j]$ in constant time with the help of the *sparse table* M , we select two overlapping blocks that exactly cover the interval $[i..j]$; this is illustrated in Figure 3.9.

Observe that the largest block that starts at position i and fits into the interval $[i..j]$ has size 2^l , where $l = \lfloor \log(j - i + 1) \rfloor$. The minimum in this block is

$$A[M[i, l]] = \min\{A[i..i + 2^{\lfloor \log(j-i+1) \rfloor} - 1]\}$$

Moreover, the minimum in the largest block that ends at position j and fits into the interval $[i..j]$ is

$$\begin{aligned} & A[M[j - 2^l + 1, l]] \\ = & \min\{A[j - 2^{\lfloor \log(j-i+1) \rfloor} + 1..j - 2^{\lfloor \log(j-i+1) \rfloor} + 1 + 2^{\lfloor \log(j-i+1) \rfloor} - 1]\} \\ = & \min\{A[j - 2^{\lfloor \log(j-i+1) \rfloor} + 1..j]\} \end{aligned}$$

Because the two blocks overlap, the minimum of the interval $[i..j]$ is

$$\min\{A[M[i, l]], A[M[j - 2^l + 1, l]]\} \text{ where } l = \lfloor \log(j - i + 1) \rfloor$$

To sum up, a range minimum query on the interval $[i..j]$ can be answered in constant time by computing

$$\text{RMQ}_A(i, j) = \begin{cases} M[i, l] & \text{if } A[M[i, l]] \leq A[M[j - 2^l + 1, l]] \\ M[j - 2^l + 1, l] & \text{otherwise} \end{cases} \quad (3.1)$$

This gives the *sparse table algorithm* with time complexity $\langle O(n \log n), O(1) \rangle$; see Bender and Farach-Colton [35].

3.3.2 An optimal algorithm

Based on the sparse table algorithm, we are now able to derive an optimal $\langle O(n), O(1) \rangle$ time algorithm. (Another fast algorithm for answering range minimum queries in constant time was developed by Alstrup et al. [12].)

A:	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">1 2 3 4</div>	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">4 2 3 1</div>	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">6 5 7 3</div>	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">8 7 6 5</div>
	B_1	B_2	B_3	B_4
A' :	1	1	3	5
B' :	1	4	4	4

Figure 3.10: A is an array of size 2^{16} , hence the block size is 4. The figure shows the first four blocks of A and the first four entries in the arrays A' and B' .

From now on, we follow the work of Fischer and Heun [104, 108]. To simplify the presentation, we henceforth assume that n is a power of two. The preprocessing step consists of the following phases:⁴

1. Divide the array A conceptually into non-overlapping consecutive blocks $B_1, \dots, B_{n/s}$, each of size $s = \frac{\log n}{4}$.
2. Compute two arrays A' and B' of size $\frac{n}{s}$, where $A'[i]$ stores the minimum in block B_i and $B'[i]$ stores its position in B_i .
3. Preprocess array A' as in the sparse table algorithm so that every range minimum query $\text{RMQ}_{A'}(i, j)$ can be answered in constant time (based on the corresponding sparse table M').

Figure 3.10 illustrates the preprocessing step. Obviously, the first two phases both require $O(n)$ time. Moreover, we have seen in Section 3.3.1 that the look-up table M' can be constructed in $O(\frac{n}{s} \log(\frac{n}{s}))$ time and space. Since $O(\frac{n}{s} \log(\frac{n}{s})) = O(\frac{4n}{\log n} \log(\frac{4n}{\log n})) = O(n)$, also the third stage takes only $O(n)$ time and space.

How can we utilize the arrays A' and B' and the look-up table M' to answer $\text{RMQ}_A(i, j)$? Let us consider the special case in which i is the first position of a block, say of block B_p , and j is the last position of a block, say of block B_q , where $p \leq q$. That is, $i = (p-1) \cdot s + 1$ and $j = q \cdot s$. Then the interval $[i..j]$ spans the blocks B_p, \dots, B_q . For example, the interval $[5..12]$ in Figure 3.10 spans the blocks B_2 and B_3 . The query $\text{RMQ}_A(i, j)$ can be answered as follows: First, a constant time query $\text{RMQ}_{A'}(p, q)$ returns the position k of the minimum element in $A'[p..q]$ (based on the sparse table M' and the array A' ; see Equation 3.1). This means that the minimum element in $A[i..j]$ can be found in block B_k . Second, $B'[k]$ provides the relative position of this minimum in B_k . Thus, the absolute position of this minimum in A is $(k-1) \cdot s + B'[k]$. In the example of Figure 3.11, $\text{RMQ}_{A'}(2, 3)$

⁴For a simpler presentation, we omit floors and ceilings.

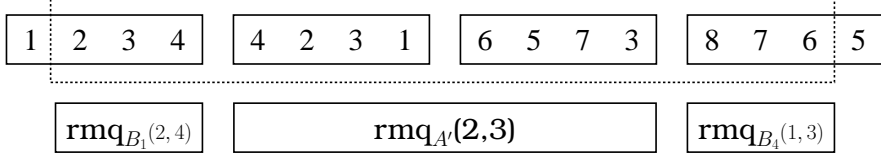


Figure 3.11: The answer to $\text{RMQ}_A(5, 12)$ can be determined with the help of the three queries $\text{RMQ}_{B_1}(2, 4)$, $\text{RMQ}_{A'}(2, 3)$, and $\text{RMQ}_{B_4}(1, 3)$.

returns 2, hence the answer to $\text{RMQ}_A(5, 12)$ is $(2 - 1) \cdot 4 + B'[2] = 4 + 4 = 8$. In this way, queries $\text{RMQ}_A(i, j)$ in which i is the first position of a block and j is the last position of a block can be answered in constant time.

In the general case, if i is not the first position of a block, j is not the last position of a block, and i and j do not occur in the same block, then we compute

- (a) the position of the minimum from i to the end of i 's block,
- (b) the position of the minimum of all blocks between i 's and j 's block,
- (c) the position of the minimum from the beginning of j 's block to j .

Clearly, the overall minimum is the smallest of the three minima, and $\text{RMQ}_A(i, j)$ returns its position. We exemplify this by the query $\text{RMQ}_A(2, 15)$ in Figure 3.11.

- (a) The relative position in B_1 of the minimum from position 2 to the end of block B_1 is obtained by $\text{RMQ}_{B_1}(2, 4) = 2$. Therefore, its absolute position in A is $(1 - 1) \cdot 4 + 2 = 2$.
- (b) The minimum of the blocks B_2 and B_3 can be found in block B_2 because $\text{RMQ}_{A'}(2, 3)$ returns 2. Its relative position in B_2 is thus $B'[2] = 4$ and its absolute position in A is $(2 - 1) \cdot 4 + 4 = 8$.
- (c) The relative position in B_4 of the minimum from the beginning of block B_4 to position 15 is obtained by $\text{RMQ}_{B_4}(1, 3) = 3$. Hence, its absolute position in A is $(4 - 1) \cdot 4 + 3 = 15$.

The overall minimum is $\min\{A[2], A[8], A[15]\} = \min\{2, 1, 6\} = 1$ and $\text{RMQ}_A(i, j)$ returns its position 8.

To obtain a complete constant time algorithm, we must be able to answer in-block queries (range minimum queries inside a block) in constant time. If we answer in-block queries by a linear scan of the block, then this takes $O(\log n)$ time because the block size is $s = \frac{\log n}{4}$. The resulting

algorithm is very useful in practice, but its worst-case time complexity of $\langle O(n), O(\log n) \rangle$ is not optimal.

The rest of this section is solely devoted to the details of answering in-block queries in constant time (with only linear-time preprocessing). The following lemma plays a central role.

Lemma 3.3.2 *For two arrays $A[1..n]$ and $B[1..n]$, we have $\text{RMQ}_A(i, j) = \text{RMQ}_B(i, j)$ for all $1 \leq i \leq j \leq n$ if and only if $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$.*

Proof We prove the lemma by induction on the size n . The base case $n = 1$ is trivial. Let $n > 1$ and suppose that the lemma is true for all arrays of size less than n .

“ \Rightarrow ” Assume that $\text{RMQ}_A(i, j) = \text{RMQ}_B(i, j)$ for all $1 \leq i \leq j \leq n$. Because $\text{RMQ}_A(1, n) = m = \text{RMQ}_B(1, n)$, the roots of $\mathcal{C}^{\text{can}}(A)$ and $\mathcal{C}^{\text{can}}(B)$ are both labeled with m . Since $\text{RMQ}_A(i, j) = \text{RMQ}_B(i, j)$ for all $1 \leq i \leq j \leq m - 1$ and for all $m + 1 \leq i \leq j \leq n$, it follows from the inductive hypothesis that $\mathcal{C}^{\text{can}}(A[1..m - 1]) = \mathcal{C}^{\text{can}}(B[1..m - 1])$ and $\mathcal{C}^{\text{can}}(A[m + 1..n]) = \mathcal{C}^{\text{can}}(B[m + 1..n])$. Hence the canonical Cartesian trees of A and B coincide.

“ \Leftarrow ” Let m be the label of the root of $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$. Then for every $1 \leq i \leq m \leq j \leq n$ we have $\text{RMQ}_A(i, j) = m = \text{RMQ}_B(i, j)$. Thus, suppose that the range minimum queries have parameters (i, j) so that $1 \leq i \leq j < m$ or $m < i \leq j \leq n$. Because $\mathcal{C}^{\text{can}}(A[1..m - 1]) = \mathcal{C}^{\text{can}}(B[1..m - 1])$ and $\mathcal{C}^{\text{can}}(A[m + 1..n]) = \mathcal{C}^{\text{can}}(B[m + 1..n])$, it is a consequence of the inductive hypothesis that $\text{RMQ}_A(i, j) = \text{RMQ}_B(i, j)$ for all $1 \leq i \leq j < m$ and for all $m < i \leq j \leq n$, and we are done. \square

So *every* range minimum query $\text{RMQ}(i, j)$ returns the *same* results on two *different* arrays A and B , provided that A and B are of the same *type*, i.e., $\mathcal{C}^{\text{can}}(A) = \mathcal{C}^{\text{can}}(B)$. Thus, to precompute all range minimum queries on all possible arrays boils down to precomputing them on all possible types of arrays. How many different canonical Cartesian trees are there? It will be shown in Corollary 3.4.6 that the number of different canonical Cartesian trees with n nodes coincides with the n -th Catalan number.

Definition 3.3.3 For every $n \in \mathbb{N}$, the n -th *Catalan number* C_n is defined by

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

For subsequent analyses, we remark that $C_n = O(\frac{4^n}{n^{3/2}})$. This is because Stirling's formula

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right)$$

implies that

$$C_n = \frac{4^n}{\sqrt{\pi n^{3/2}}} \left(1 + O\left(\frac{1}{n}\right)\right)$$

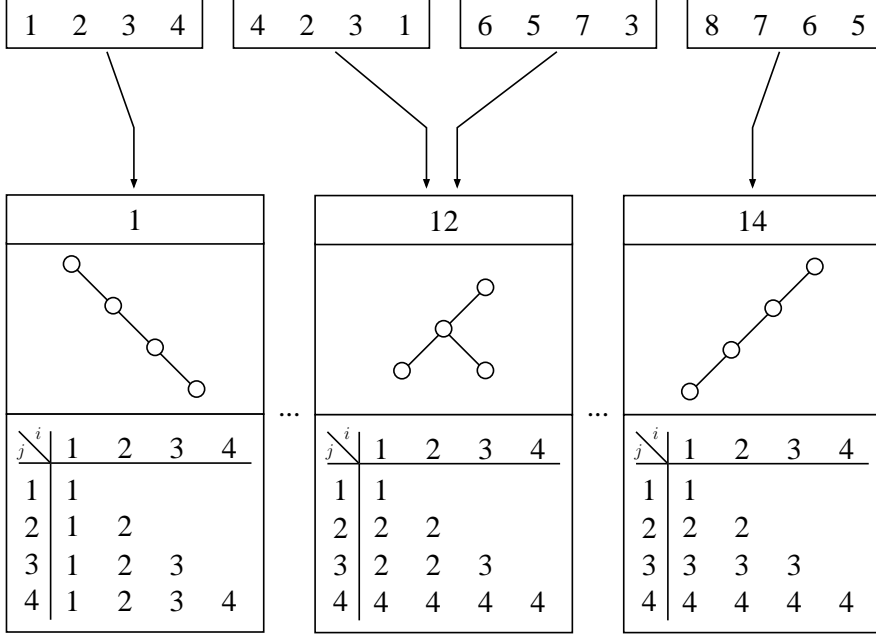


Figure 3.12: Four blocks and their types 1, 12, and 14, respectively. The entries of the look-up table P for these types are shown below the canonical Cartesian trees of the blocks. The second and the third block are of the same type 12.

Lemma 3.3.4 All possible answers to range minimum queries on all possible arrays of size s can be computed in $O(C_s \cdot s^2) = O(4^s \sqrt{s})$ time and stored in a three dimensional look-up table P of size $C_s \times s \times s = O(4^s \sqrt{s})$.

Proof For a fixed array of size s , there are $O(s^2)$ queries and the two-dimensional look-up table of size $s \times s$ for this particular array can be constructed in $O(s^2)$ time using the naive algorithm described at the beginning of this section. Because there are $C_s = O(\frac{4^s}{s^{3/2}})$ different types of arrays of size s , the three dimensional look-up table P has size $C_s \times s \times s$ and it can be computed in $O(C_s \cdot s^2) = O(\frac{4^s s^2}{s^{3/2}}) = O(4^s \sqrt{s})$ time. \square

Figure 3.12 shows some entries of the three dimensional look-up table P , namely those for the types 1, 12, and 14.

If we apply the preceding lemma to $s = \frac{\log n}{4}$, the computation takes $O(4^{(\log n/4)} (\frac{\log n}{4})^{1/2}) = O(\sqrt[2]{n \log n}) = O(n)$ time and space.

A side note on the method used above: The technique of precomputing all solutions (here: all answers to range minimum queries) to all possible

problems (here: all types of arrays) of a certain small size and storing them in a look-up table is called the *Four-Russians* technique [16].

Recall that we want to be able to answer in-block queries in constant time (with only linear-time preprocessing). If we knew the type $t(B_k)$ of a block B_k , then a range minimum query $\text{RMQ}_{B_k}(i, j)$ could be answered in constant time by a table look-up $P(t(B_k), i, j)$. In order to determine the type $t(B_k)$ in constant time, we further precompute a type array T of size n/s so that $T[k] = t(B_k)$. Then, $\text{RMQ}_{B_k}(i, j)$ can be answered in constant time by the table look-up $P(T[k], i, j)$.

Thus the remaining question is: How do we determine the type of each of the $\frac{4n}{\log n}$ blocks in $O(n)$ time? To define a function that does exactly this, we need the ballot numbers.

Definition 3.3.5 The *ballot numbers* $C_{(p,q)}$ are defined for all $p, q \in \mathbb{N}$ by $C_{(0,q)} = 1$ and

$$C_{(p,q)} = \begin{cases} C_{(p,q-1)} + C_{(p-1,q)} & \text{if } 1 \leq p \leq q \\ 0 & \text{if } p > q \end{cases}$$

It will be shown in Section 3.4 that the ballot number $C_{(s,s)}$ equals the s -th Catalan number C_s .

Now we have all the ingredients to define $f : \mathcal{L}_s \rightarrow \{0, 1, 2, \dots, C_s - 1\}$ as

$$f(l_1, l_2, \dots, l_s) = \sum_{i=1}^s \sum_{j=0}^{l_i-1} C_{(s-i, s-j-\sum_{k=1}^{i-1} l_k)}$$

It is clear that $f(l_1, l_2, \dots, l_s)$ can be computed in $O(s)$ time because there are at most $\sum_{i=1}^s l_i < s$ summands.

To motivate the definition, we define a directed graph G_s as follows. Its set of nodes consists of all pairs (p, q) of natural numbers with $0 \leq p \leq q \leq s$. Node (p, q) corresponds to the ballot number $C_{(p,q)}$. Furthermore, there are directed edges from node (p, q) to node $(p, q-1)$ whenever $p \leq q-1$ and from node (p, q) to node $(p-1, q)$ whenever $p > 0$. These edges model the dependencies among ballot numbers according to their definition $C_{(p,q)} = C_{(p,q-1)} + C_{(p-1,q)}$; see Figure 3.13.

A sequence $l_1, l_2, \dots, l_s \in \mathcal{L}_s$ gives rise to a path from node (s, s) to node $(0, 0)$ in G_s : In step i , the path first goes from node $(s-i+1, s-\sum_{k=1}^{i-1} l_k)$ l_i steps upwards to node $(s-i+1, s-\sum_{k=1}^i l_k)$ and then one step to the left to node $(s-i, s-\sum_{k=1}^i l_k)$. After the s -th step, the path is at the node $(0, s-\sum_{k=1}^s l_k)$, from which it further goes upwards to node $(0, 0)$.

For example, consider the sequence $0, 1, 0, 2$. It corresponds to the path $(4, 4) \rightarrow (3, 4) \rightarrow (3, 3) \rightarrow (2, 3) \rightarrow (1, 3) \rightarrow (1, 2) \rightarrow (1, 1) \rightarrow (0, 1) \rightarrow (0, 0)$ in the graph G_4 ; see Figure 3.14.

Conversely, every path in (s, s) to $(0, 0)$ in G_s corresponds to a sequence from \mathcal{L}_s . This is a consequence of the definition of G_s : at each node one

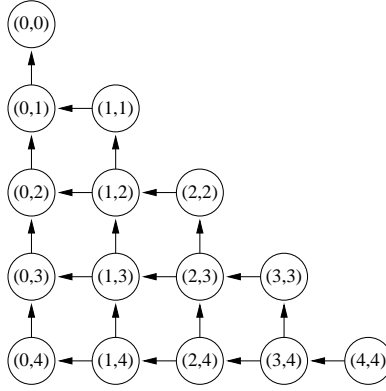


Figure 3.13: The graph G_4 , in which node (p, q) corresponds to the ballot number $C_{(p,q)}$.

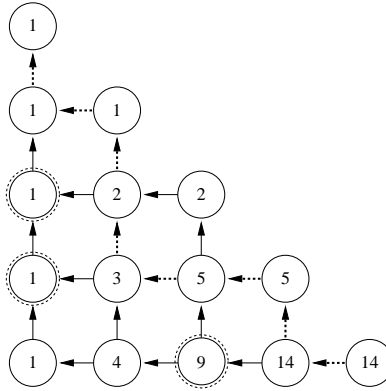


Figure 3.14: The graph G_4 , in which node (p, q) is labeled with the ballot number $C_{(p,q)}$. The path corresponding to the sequence 0, 1, 0, 2 is drawn with dotted arrows.

can move at most as many nodes upwards as have already been traversed by going left.

With each path corresponding to a sequence $l_1, l_2, \dots, l_s \in \mathcal{L}_s$, we associate a sum of ballot numbers as follows. In step i , if the path from node $(s - i + 1, s - \sum_{k=1}^{i-1} l_k)$ climbs l_i steps upwards to $(s - i + 1, s - \sum_{k=1}^{i-1} l_k - l_i)$, then the associated sum is $\sum_{j=0}^{l_i-1} C_{(s-i, s-j-\sum_{k=1}^{i-1} l_k)}$ (the sum is 0 if $l_i = 0$). The whole path is associated with the overall sum of the associated sums of each step. We stress that the mapping f applied to the sequence $l_1, l_2, \dots, l_s \in \mathcal{L}_s$ computes exactly this value $\sum_{i=1}^s \sum_{j=0}^{l_i-1} C_{(s-i, s-j-\sum_{k=1}^{i-1} l_k)}$.

Again, consider the sequence $0, 1, 0, 2$, which corresponds to the path $(4, 4) \rightarrow (3, 4) \rightarrow (3, 3) \rightarrow (2, 3) \rightarrow (1, 3) \rightarrow (1, 2) \rightarrow (1, 1) \rightarrow (0, 1) \rightarrow (0, 0)$ in the graph G_4 . There are three upwards moves $(p, q) \rightarrow (p, q - 1)$ in which $p > 0$, viz. $(3, 4) \rightarrow (3, 3)$, $(1, 3) \rightarrow (1, 2)$, and $(1, 2) \rightarrow (1, 1)$. Each of these upward moves $(p, q) \rightarrow (p, q - 1)$ contributes the value $C_{(p-1, q)}$ to the overall sum. Thus, in our example, we have $f(0, 1, 0, 2) = C_{(2, 4)} + C_{(0, 3)} + C_{(0, 2)} = 9 + 1 + 1 = 11$; cf. Figure 3.14.

Figure 3.15 illustrates the relationship between these concepts.

Let us come back to the problem of determining the type of each of the $\frac{4n}{\log n}$ blocks in $O(n)$ time. By definition, blocks B_p and B_q are of the same type if and only if $\mathcal{C}^{can}(B_p) = \mathcal{C}^{can}(B_q)$. Let $l_1^{B_p}, l_2^{B_p}, \dots, l_s^{B_p}$ and $l_1^{B_q}, l_2^{B_q}, \dots, l_s^{B_q}$ be the sequences of natural numbers obtained during the construction of the canonical Cartesian trees of B_p and B_q , respectively. By Lemma 3.2.7, $\mathcal{C}^{can}(B_p) = \mathcal{C}^{can}(B_q)$ if and only if $l_1^{B_p}, l_2^{B_p}, \dots, l_s^{B_p} = l_1^{B_q}, l_2^{B_q}, \dots, l_s^{B_q}$. Moreover, it will be proven in Theorem 3.4.5 that the mapping f (as defined above) is bijective. To sum up,

$$\begin{aligned} \mathcal{C}^{can}(B_p) &= \mathcal{C}^{can}(B_q) \\ \Leftrightarrow l_1^{B_p}, l_2^{B_p}, \dots, l_s^{B_p} &= l_1^{B_q}, l_2^{B_q}, \dots, l_s^{B_q} \\ \Leftrightarrow f(l_1^{B_p}, l_2^{B_p}, \dots, l_s^{B_p}) &= f(l_1^{B_q}, l_2^{B_q}, \dots, l_s^{B_q}) \end{aligned}$$

Consequently, the number $f(l_1^{B_k}, l_2^{B_k}, \dots, l_s^{B_k})$ is a unique representation of the type of a block B_k . Like the sequence $l_1^{B_k}, l_2^{B_k}, \dots, l_s^{B_k}$ (Exercise 3.2.6), $f(l_1^{B_k}, l_2^{B_k}, \dots, l_s^{B_k})$ can be computed in $O(s)$ time without actually constructing $\mathcal{C}^{can}(B_k)$; see Exercise 3.3.8.

Since our array indexing starts at 1 and not at 0, we define the type of a block B_k to be $t(B_k) = 1 + f(l_1^{B_k}, l_2^{B_k}, \dots, l_s^{B_k})$. It can be computed in $O(s)$ time because $f(l_1, l_2, \dots, l_s)$ can be computed in $O(s)$ time.

As already mentioned, for each block B_k , $1 \leq k \leq \frac{n}{s}$, we precompute its type $t(B_k)$ and store it in the type array T at position k . Because there are $O(\frac{n}{s})$ blocks, the overall time to compute the type array T is $O(n)$.

Figure 3.16 summarizes the whole process of answering queries.





























A	$C^{can}(A)$	path	(l_1, l_2, l_3, l_4)	sum	$f(l_1, l_2, l_3, l_4)$
1234			$(0, 0, 0, 0)$	0	0
1243			$(0, 0, 0, 1)$	1	1
1342			$(0, 0, 0, 2)$	1+1	2
2341			$(0, 0, 0, 3)$	1+1+1	3
1324			$(0, 0, 1, 0)$	4	4
1432			$(0, 0, 1, 1)$	4+1	5
2431			$(0, 0, 1, 2)$	4+1+1	6
2314			$(0, 0, 2, 0)$	4+3	7
3421			$(0, 0, 2, 1)$	4+3+1	8
2134			$(0, 1, 0, 0)$	9	9
2143			$(0, 1, 0, 1)$	9+1	10
3241			$(0, 1, 0, 2)$	9+1+1	11
3214			$(0, 1, 1, 0)$	9+3	12
4321			$(0, 1, 1, 1)$	9+3+1	13

Figure 3.15: The second column contains all possible canonical Cartesian trees of arrays with the four entries 1, 2, 3, 4 (the first column shows one of many possible arrays having that canonical Cartesian tree). The third column depicts the corresponding path in the graph G_4 , while the fourth column shows the corresponding sequence from \mathcal{L}_4 . The fifth column depicts the sum of ballot numbers corresponding to the path in the graph G_4 , and the last column contains $f(l_1, l_2, l_3, l_4)$.

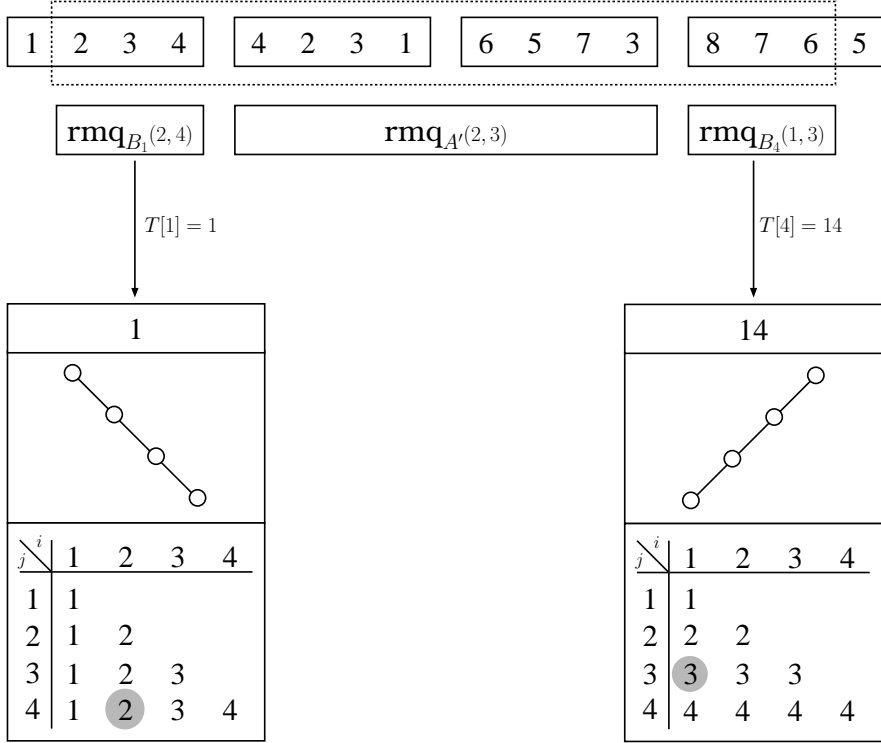


Figure 3.16: Summary: The range minimum query $\text{RMQ}_A(2, 15)$ is answered by (a) answering the left in-block query $\text{RMQ}_{B_1}(2, 4)$ (by first looking up the type of block B_1 in the type array T , yielding $T[1] = 1$, followed by the table look-up $P(1, 2, 4)$), the query $\text{RMQ}_{A'}(2, 3)$, and the right in-block query $\text{RMQ}_{B_4}(1, 3)$, (b) computing the absolute positions in A from the returned relative positions, and (c) comparing the minima at these absolute positions in A . Because all steps can be done in constant time, the range minimum query can be answered in constant time.

Theorem 3.3.6 Given array A , after a preprocessing step taking linear time and space, range minimum queries of the form $\text{RMQ}_A(i, j)$ can be answered in constant time.

Proof This is the bottom line of the considerations in this section. □

Corollary 3.3.7 Given tree T , after a preprocessing step taking linear time and space, lowest common ancestors in T can be found in constant time.

Proof Combine Theorem 3.2.2 with Theorem 3.3.6. □

Exercise 3.3.8 Use Exercise 3.2.6 to prove that Algorithm 3.2 computes the number $f(l_1^A, l_2^A, \dots, l_s^A)$ for an array A of size s , without actually constructing the sequence $l_1^A, l_2^A, \dots, l_s^A$.

Algorithm 3.2 Given array A of size s , compute $f(l_1^A, l_2^A, \dots, l_s^A)$.

```

 $R[0] \leftarrow -\infty$  /* array  $R$  stores the (current) values on the rightmost path */
 $f \leftarrow 0$ 
 $q \leftarrow 0$ 
for  $i \leftarrow 1$  to  $s$  do
  while  $R[q] > A[i]$  do
     $f \leftarrow f + C_{(s-i, s-i+q+1)}$ 
     $q \leftarrow q - 1$  /* remove  $R[q]$  from the rightmost path */
   $q \leftarrow q + 1$ 
   $R[q] \leftarrow A[i]$  /*  $A[i]$  is the new rightmost leaf */
return  $f$ 

```

3.4 Completing the proof of correctness

Our goal is to show that $f : \mathcal{L}_s \rightarrow \{0, 1, 2, \dots, C_s - 1\}$ defined by

$$f(l_1, l_2, \dots, l_s) = \sum_{i=1}^s \sum_{j=0}^{l_i-1} C_{(s-i, s-j-\sum_{k=1}^{i-1} l_k)}$$

is a bijection. To achieve this goal, we need a few prerequisites.

Lemma 3.4.1 For all $q \in \mathbb{N}$, we have $C_{(1,q)} = q$.

Proof Straightforward by induction on q . □

Lemma 3.4.2 For $1 \leq p \leq q$, the ballot number $C_{(p,q)}$ can be computed by

$$C_{(p,q)} = \binom{p+q}{p} - \binom{p+q}{p-1}$$

Proof The proof is by induction on the well-founded order \prec defined on $\mathbb{N}_{>0} \times \mathbb{N}_{>0}$ defined by $(p', q') \prec (p, q)$ if and only if $p' < p$ or $p' = p$ and $q' < q$. The base case $p = q = 1$ is readily verified. If $p = 1$ and $q \in \mathbb{N}_{>0}$, then $C_{(1,q)} = \binom{1+q}{1} - \binom{1+q}{1-1} = q$, which is the correct value by Lemma 3.4.1. Now consider $(p, q) \in \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ with $2 \leq p \leq q$. By the inductive hypothesis, we may assume that the lemma holds for all $(p', q') \in \mathbb{N}_{>0} \times \mathbb{N}_{>0}$ with $(p', q') \prec (p, q)$. In the inductive step, we first apply the definition $C_{(p,q)} = C_{(p,q-1)} + C_{(p-1,q)}$ of $C_{(p,q)}$, and then the inductive hypothesis to $C_{(p,q-1)}$ and $C_{(p-1,q)}$.

$$\begin{aligned} C_{(p,q)} &= C_{(p,q-1)} + C_{(p-1,q)} \\ &= \binom{p+q-1}{p} - \binom{p+q-1}{p-1} + \binom{p+q-1}{p-1} - \binom{p+q-1}{p-2} \\ &= \frac{(p+q-1)!}{(q-1)!p!} - \frac{(p+q-1)!}{(q+1)!(p-2)!} \\ &= \frac{q(p+q-1)!}{q!p!} - \frac{(p-1)(p+q-1)!}{(q+1)!(p-1)!} \\ &= \frac{(p+q-p)(p+q-1)!}{q!p!} - \frac{(p-1+q+1-(q+1))(p+q-1)!}{(q+1)!(p-1)!} \\ &= \frac{(p+q)!}{q!p!} - \frac{p(p+q-1)!}{q!p!} - \frac{(p+q)!}{(q+1)!(p-1)!} + \frac{(q+1)(p+q-1)!}{(q+1)!(p-1)!} \\ &= \frac{(p+q)!}{q!p!} - \frac{(p+q-1)!}{q!(p-1)!} - \frac{(p+q)!}{(q+1)!(p-1)!} + \frac{(p+q-1)!}{q!(p-1)!} \\ &= \binom{p+q}{p} - \binom{p+q}{p-1} \end{aligned}$$

□

It is easy to verify that

$$\binom{p+q}{p} - \binom{p+q}{p-1} = \frac{q-p+1}{q+1} \binom{p+q}{p}$$

Hence

$$C_{(s,s)} = \frac{1}{s+1} \binom{2s}{s} = C_s$$

Therefore, the ballot number $C_{(s,s)}$ equals the s -th Catalan number C_s .

In order to show that f is bijective, we need two more lemmata.

Lemma 3.4.3 For all $1 \leq p \leq q$, we have

$$C_{(p,q)} = \sum_{j=p}^q C_{(p-1,j)}$$

Proof We proceed by induction on q . The base case $q = 1$ is true because $C_{(1,1)} = 1$ and $\sum_{j=1}^1 C_{(0,j)} = C_{(0,1)} = 1$. By the definition of the ballot numbers, $C_{(p,q)} = C_{(p,q-1)} + C_{(p-1,q)}$. Hence, it follows from the inductive hypothesis that

$$C_{(p,q)} = C_{(p,q-1)} + C_{(p-1,q)} = \sum_{j=p}^{q-1} C_{(p-1,j)} + C_{(p-1,q)} = \sum_{j=p}^q C_{(p-1,j)}$$

□

Lemma 3.4.4 For all $p > 0$, we have

$$C_{(p-1,p)} = 1 + \sum_{j=0}^{p-2} C_{(p-j-2,p-j)}$$

Proof By induction on p . The base case $p = 1$ holds true because $C_{(0,1)} = 1 + \sum_{j=0}^{-1} C_{(1-j-2,1-j)} = 1$ (the sum is empty). The inductive hypothesis states that $C_{(p-2,p-1)} = 1 + \sum_{j=0}^{p-3} C_{(p-1-j-2,p-1-j)}$. The following derivation proves the lemma:

$$\begin{aligned} C_{(p-1,p)} &= C_{(p-1,p-1)} + C_{(p-2,p)} && \text{(def. ballot numbers)} \\ &= C_{(p-1,p-2)} + C_{(p-2,p-1)} + C_{(p-2,p)} && \text{(def. ballot numbers)} \\ &= C_{(p-2,p-1)} + C_{(p-2,p)} && \text{(because } C_{(p-1,p-2)} = 0) \\ &= 1 + \sum_{j=0}^{p-3} C_{(p-1-j-2,p-1-j)} + C_{(p-2,p)} && \text{(inductive hypothesis)} \\ &= 1 + \sum_{j=1}^{p-2} C_{(p-j-2,p-j)} + C_{(p-2,p)} && \text{(index shift)} \\ &= 1 + \sum_{j=0}^{p-2} C_{(p-j-2,p-j)} - C_{(p-2,p)} + C_{(p-2,p)} \\ &= 1 + \sum_{j=0}^{p-2} C_{(p-j-2,p-j)} \end{aligned}$$

□

Theorem 3.4.5 *The mapping $f : \mathcal{L}_s \rightarrow \{0, 1, 2, \dots, C_s - 1\}$ defined by*

$$f(l_1, l_2, \dots, l_s) = \sum_{i=1}^s \sum_{j=0}^{l_i-1} C_{(s-i, s-j-\sum_{k=1}^{i-1} l_k)}$$

is bijective.

Proof \mathcal{L}_s together with the lexicographic order $<_{lex}$ (which compares sequences from left to right) is a totally ordered set. Its smallest element is $0, 0, \dots, 0$ and its largest element is $0, 1, \dots, 1$. Note that $f(0, 0, \dots, 0) = 0$ and

$$\begin{aligned} f(0, 1, \dots, 1) &= \sum_{i=2}^s C_{(s-i, s-\sum_{k=1}^{i-1} l_k)} \\ &= \sum_{i=2}^s C_{(s-i, s-i+2)} \\ &= \sum_{i=0}^{s-2} C_{(s-i-2, s-i)} && \text{(index shift)} \\ &= C_{(s-1, s)} - 1 && \text{(by Lemma 3.4.4)} \\ &= C_{(s, s)} - 1 && \text{(def. ballot numbers)} \\ &= C_s - 1 \end{aligned}$$

We have to show that f is injective, i.e., $l_1, l_2, \dots, l_s \neq l'_1, l'_2, \dots, l'_s$ implies $f(l_1, l_2, \dots, l_s) \neq f(l'_1, l'_2, \dots, l'_s)$, and that f is surjective, i.e., for each $m \in \{0, 1, 2, \dots, C_s - 1\}$ there is a sequence $l_1, l_2, \dots, l_s \in \mathcal{L}_s$ with $f(l_1, l_2, \dots, l_s) = m$.

Let $0, 1, \dots, 1 \neq l_1, l_2, \dots, l_s \in \mathcal{L}_s$ and let $l'_1, l'_2, \dots, l'_s \in \mathcal{L}_s$ be its successor w.r.t. $<_{lex}$. It is readily verified that f is bijective if we can show that

$$f(l'_1, l'_2, \dots, l'_s) = 1 + f(l_1, l_2, \dots, l_s)$$

The successor $l'_1, l'_2, \dots, l'_s \in \mathcal{L}_s$ of a sequence $l_1, l_2, \dots, l_s \in \mathcal{L}_s$ can be obtained as follows. Determine the largest index m so that $\sum_{k=1}^m l_k < m - 1$. Note that this implies $\sum_{k=1}^i l_k = i - 1$ for all $i > m$, or equivalently, $l_i = i - 1 - \sum_{k=1}^{i-1} l_k$. In particular, for $i = m + 1$ we have $l_{m+1} = m - \sum_{k=1}^m l_k$. Thus, for $i = m + 2$, it follows that $l_{m+2} = m + 1 - \sum_{k=1}^{m+1} l_k = m + 1 - m = 1$. Analogously, we derive $l_{m+3} = \dots = l_s = 1$. The successor l'_1, l'_2, \dots, l'_s of the sequence l_1, l_2, \dots, l_s satisfies $l'_1 = l_1, l'_2 = l_2, \dots, l'_{m-1} = l_{m-1}, l'_m = l_m + 1$, and $l'_{m+1} = \dots = l'_s = 0$. With

$$S = \sum_{i=1}^m \sum_{j=0}^{l_i-1} C_{(s-i, s-j-\sum_{k=1}^{i-1} l_k)}$$

the images of the sequences w.r.t. f can be expressed as

$$\begin{aligned}
 f(l'_1, l'_2, \dots, l'_s) &= f(l_1, l_2, \dots, l_{m-1}, l_m + 1, 0, \dots, 0) \\
 &= S + C_{(s-m, s-l_m-\sum_{k=1}^{m-1} l_k)} \\
 &= S + C_{(s-m, s-\sum_{k=1}^m l_k)} \\
 &= S + C_{(s-m, s-m+l_{m+1})}
 \end{aligned}$$

where the last equality follows from $\sum_{k=1}^m l_k = m - l_{m+1}$, and

$$\begin{aligned}
 f(l_1, l_2, \dots, l_s) &= f(l_1, l_2, \dots, l_m, l_{m+1}, 1, \dots, 1) \\
 &= S + \sum_{j=0}^{l_{m+1}-1} C_{(s-m-1, s-j-\sum_{k=1}^m l_k)} + \sum_{i=m+2}^s C_{(s-i, s-\sum_{k=1}^{i-1} l_k)} \\
 &= S + \sum_{j=0}^{l_{m+1}-1} C_{(s-m-1, s-j-m+l_{m+1})} + \sum_{i=m+2}^s C_{(s-i, s-i+2)}
 \end{aligned}$$

In summary, we must show

$$C_{(s-m, s-m+l_{m+1})} = 1 + \sum_{j=0}^{l_{m+1}-1} C_{(s-m-1, s-m+l_{m+1}-j)} + \sum_{i=m+2}^s C_{(s-i, s-i+2)} \quad (3.2)$$

Setting $p = s - m$, it follows

$$\sum_{j=0}^{l_{m+1}-1} C_{(s-m-1, s-m+l_{m+1}-j)} = \sum_{j=0}^{l_{m+1}-1} C_{(p-1, p+l_{m+1}-j)} = \sum_{j=p+1}^{p+l_{m+1}} C_{(p-1, j)}$$

and

$$\sum_{i=m+2}^s C_{(s-i, s-i+2)} = \sum_{i=0}^{s-m-2} C_{(s-m-2-i, s-m-i)} = \sum_{i=0}^{p-2} C_{(p-i-2, p-i)}$$

If we further set $q = s - m + l_{m+1} = p + l_{m+1}$, then the preceding two equalities imply that Equation (3.2) is equivalent to

$$C_{(p, q)} = 1 + \sum_{j=p+1}^q C_{(p-1, j)} + \sum_{i=0}^{p-2} C_{(p-i-2, p-i)}$$

This equality, however, follows easily with the preceding lemmata.

$$\begin{aligned}
 C_{(p, q)} &= \sum_{j=p}^q C_{(p-1, j)} && \text{(by Lemma 3.4.3)} \\
 &= \sum_{j=p+1}^q C_{(p-1, j)} + C_{(p-1, p)} \\
 &= \sum_{j=p+1}^q C_{(p-1, j)} + 1 + \sum_{i=0}^{p-2} C_{(p-i-2, p-i)} && \text{(by Lemma 3.4.4)}
 \end{aligned}$$

□

Corollary 3.4.6 *The number of different canonical Cartesian trees with n nodes (numbered from 1 to n) is C_n .*

Proof According to Lemma 3.2.7, the canonical Cartesian trees $\mathcal{C}^{can}(A)$ and $\mathcal{C}^{can}(B)$ of two arrays $A[1..n]$ and $B[1..n]$ coincide if and only if $l_1^A, l_2^A, \dots, l_s^A = l_1^B, l_2^B, \dots, l_s^B$, where $l_1^A, l_2^A, \dots, l_s^A$ and $l_1^B, l_2^B, \dots, l_s^B$ are the sequences of natural numbers obtained in the construction of $\mathcal{C}^{can}(A)$ and $\mathcal{C}^{can}(B)$, respectively. Therefore, there are as many canonical Cartesian trees as there are sequences in \mathcal{L}_n , and $|\mathcal{L}_n| = C_n$ by Theorem 3.4.5. \square

Exercise 3.4.7 Prove that the number of different binary trees with n nodes is C_n .

Hint: Show that every binary tree corresponds to exactly one canonical Cartesian tree in which the nodes are numbered 1 to n .

Enhanced Suffix Arrays

The meteoric increase of DNA sequences produced by next-generation sequencers demands new approaches in computer science for storing, analyzing, and mining the accumulating data. While the databases are growing rapidly, the data representing DNA sequences of the human chromosomes do not change much over time. As a consequence, index data structures such as suffix arrays or suffix trees can be used to efficiently solve a myriad of sequence analysis problems.

In this chapter, we will introduce the suffix array and the LCP-array of a string, and present linear-time algorithms to construct them. The combination of these (and possibly other) arrays is called an *enhanced suffix array*. Furthermore, we define the lcp-interval tree and provide tree traversal algorithms that solely rely on the enhanced suffix array. At the end of the chapter, we shall see that the lcp-interval tree of a string S coincides with the suffix tree of S . The main advantage of using an lcp-interval tree is that it can be traversed without building the tree itself.

Memory is not an issue here. As we shall see in Chapter 6, the memory usage of enhanced suffix arrays can be reduced significantly.

4.1 Suffix arrays

To construct the suffix array of a string S boils down to sorting all suffixes of S in lexicographic order (also known as alphabetical order, dictionary order, or lexical order). This order is induced by an order on the alphabet Σ . In this book, Σ is an ordered alphabet of constant size σ . It is sometimes convenient to regard Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] < \Sigma[2] < \dots < \Sigma[\sigma]$. Conversely, each character in Σ is mapped to a number in $\{1, \dots, \sigma\}$. The smallest character is mapped to 1, the second smallest character is mapped to 2, and so on. In this way, we can use a character as an index for an array.

Definition 4.1.1 Let $<$ be a total order on the alphabet Σ . This order induces the *lexicographic order* on Σ^* (which we again denote by $<$) as follows: For $s, t \in \Sigma^*$, define $s < t$ if and only if either s is a proper prefix of t or there are strings $u, v, w \in \Sigma^*$ and characters $a, b \in \Sigma$ with $a < b$ so that $s = uav$ and $t = ubw$.

To determine the lexicographic order of two strings, their first characters are compared. If they differ, then the string whose first character comes earlier in the alphabet is the one which comes first in lexicographic order. If the first characters are the same, then the second characters are compared, and so on. If a position is reached where one string has no more characters to compare while the other does, then the shorter string comes first in lexicographic order.

In algorithms that need to determine the lexicographic order of two suffixes of the same string S , a cumbersome distinction between “has more characters” and “has no more characters” can be avoided by appending the special symbol $\$$ (called *sentinel character*) to S . In the following, we assume that $\$$ is smaller than all other elements of the alphabet Σ . If $\$$ occurs nowhere else in S and the lexicographic order of two suffixes of S is determined as described above, then it cannot happen that one suffix has no more characters to compare. As we shall see later, there are other situations in which it is convenient to append the special symbol $\$$ to a string.

Definition 4.1.2 Let S be a string of length n . For every i , $1 \leq i \leq n$, S_i denotes the i -th suffix $S[i..n]$ of S . The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic order of the n suffixes of the string S . That is, it satisfies $S_{SA[1]} < S_{SA[2]} < \dots < S_{SA[n]}$.

The *inverse suffix array* ISA is an array of size n so that for any k with $1 \leq k \leq n$ the equality $ISA[SA[k]] = k$ holds.

The inverse suffix array is sometimes also called *rank*-array because $ISA[i]$ specifies the rank of the i -th suffix among the lexicographically ordered suffixes. More precisely, if $j = ISA[i]$, then suffix S_i is the j -th lexicographically smallest suffix. Obviously, the inverse suffix array can be computed in linear time from the suffix array. Figure 4.1 shows the suffix array and the inverse suffix array of the string $S = ctaataatg$.

Suffixes of S that share a common prefix occur consecutively in the suffix array SA because SA contains the suffixes of S in lexicographically sorted order. In other words, they form an interval in the suffix array.

Definition 4.1.3 Let SA be the suffix array of the string S . For every substring ω of S , the ω -interval in SA is the interval $[i..j]$ so that

- ω is not a prefix of $S_{SA[i-1]}$,

i	SA	ISA	$S_{SA[i]}$
1	3	5	<i>aataatg</i>
2	6	7	<i>aatg</i>
3	4	1	<i>ataatg</i>
4	7	3	<i>atg</i>
5	1	8	<i>ctaataatg</i>
6	9	2	<i>g</i>
7	2	4	<i>taataatg</i>
8	5	9	<i>taatg</i>
9	8	6	<i>tg</i>

Figure 4.1: Suffix array and inverse suffix array of the string $S = ctaataatg$.

- ω is a prefix of $S_{SA[k]}$ for all $i \leq k \leq j$,
- ω is not a prefix of $S_{SA[j+1]}$.

For example, $[7..9]$ is the t -interval in the suffix array of Figure 4.1, and the taa -interval is $[7..8]$.

Exercise 4.1.4 Let u and v be substrings of S . Show that

- the u -interval $[i..j]$ and the v -interval $[p..q]$ do not overlap, i.e., neither $p < i \leq q < j$ nor $i < p \leq j < q$,
- the u -interval $[i..j]$ is a subinterval of the v -interval $[p..q]$ if and only if v is a prefix of u .

4.1.1 Linear-time construction

The suffix array was devised by Manber and Myers [214] and independently by Gonnet et al. [129] under the name PAT array. Ten years later, it was shown independently and contemporaneously by Kärkkäinen and Sanders [175], Kim et al. [180], Ko and Aluru [184], and Hon et al. [154] that a direct linear-time construction of the suffix array is possible. To date, over 20 different suffix array construction algorithms (SACAs) are known. It is beyond the scope of this book to review all of them; the reader is referred to the overview article of Puglisi et al. [262]. It is worth mentioning that the SACAs that perform best in practice have a non-linear worst-case time complexity. In this section, we explain the simplest of the three above-mentioned linear-time algorithms, namely the *skew algorithm* of Kärkkäinen and Sanders [175]. In Section 4.1.2, we will present

the induced sorting algorithm devised by Nong et al. [244].

In what follows, we shall assume that $n \bmod 3 = 0$, where $n = |S|$. In our running example $S = \text{ctaataatg}$ and thus $n = 9$. The skew algorithm follows the divide and conquer paradigm. It divides the sequence (or list) $[S_i \mid 1 \leq i \leq n] = [S_1, S_2, S_3, \dots, S_n]$ of suffixes of S into two sequences:

- The sequence $[S_i \mid 1 \leq i \leq n, i \bmod 3 \neq 1] = [S_2, S_3, S_5, S_6, \dots]$ of suffixes of S that start at a position i with $i \bmod 3 \neq 1$. This sequence has $\frac{2n}{3}$ elements.
- The sequence $[S_i \mid 1 \leq i \leq n, i \bmod 3 = 1] = [S_1, S_4, S_7, \dots]$ of suffixes of S that start at a position i with $i \bmod 3 = 1$. This sequence has $\frac{n}{3}$ elements.

Then the skew algorithm proceeds in three phases.

1. The sequence $[S_i \mid 1 \leq i \leq n, i \bmod 3 \neq 1]$ is sorted *recursively*.
2. The sequence $[S_i \mid 1 \leq i \leq n, i \bmod 3 = 1]$ is sorted *non-recursively* with the aid of the information gained in phase 1.
3. The two sorted sequences are *merged* as in the merge sort algorithm; see e.g. [61]. So phase 3 is the conquer-step of the algorithm.

Figure 4.2 illustrates the idea. Let us elaborate on the three phases.

Phase 1: The sequence of suffixes $[S_i \mid 1 \leq i \leq n, i \bmod 3 \neq 1]$ is sorted as follows: First, the sequence of triples $[S[i..i+2] \mid 1 \leq i \leq n, i \bmod 3 \neq 1]$ is stably sorted in linear time by a radix sort. (Details on the radix sort and how it can be implemented so that it is *stable*, i.e., elements with the same value appear in the output sequence in the same order as in the input sequence, can e.g. be found in [61].) To deal with boundary cases, we adopt the convention that $S[n-1..n+1] = S[n-1..n]\$$ and $S[n..n+2] = S[n]\$\$,$ where $\$$ is the sentinel character. If triple $S[i..i+2]$ is the k -th different triple occurring in the sorted sequence, then the *lexicographic name* $\bar{i} = k$ is associated with $S[i..i+2]$. Note that $\bar{i} \in [1..\frac{2n}{3}]$. Furthermore, $\bar{i} < \bar{j}$ if and only if $S[i..i+2] < S[j..j+2]$.

In our example, an application of radix sort to $[taa, aat, taa, aat, tg\$, g\$\$]$ yields the sequence $[aat, aat, g\$\$, taa, taa, tg\$]$. The lexicographic names associated with the triples are shown in Figure 4.3.

If the triples get pairwise different lexicographic names, then the suffixes are sorted lexicographically and phase 1 of the algorithm is completed. Otherwise, the skew algorithm *recursively* computes the suffix array \overline{SA} of the string \overline{S} , which is the concatenation of the strings of lexicographic names $[\bar{i} \mid 1 \leq i \leq n, i \bmod 3 = 2]$ and $[\bar{i} \mid 1 \leq i \leq n, i \bmod 3 = 0]$; see also Algorithm 4.1. In our example, $\overline{S} = 334112$. The suffix array \overline{SA} of the string 334112 can be found in Figure 4.4.

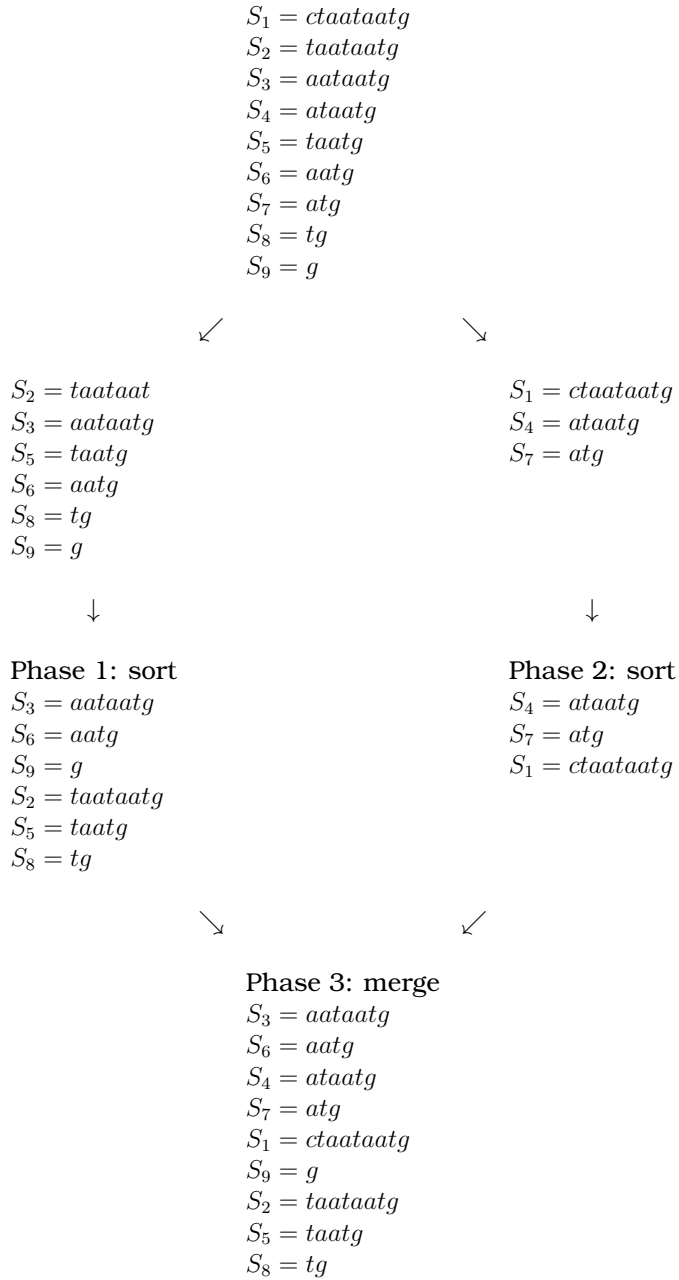


Figure 4.2: Overview of the skew algorithm.

i	2	3	5	6	8	9
\bar{i}	3	1	3	1	4	2
$S[i..i+2]$	<i>taa</i>	<i>aat</i>	<i>taa</i>	<i>aat</i>	<i>tg\$</i>	<i>g\$\$</i>

Figure 4.3: The lexicographic names associated with the triples.

k	\overline{SA}	\overline{ISA}	$\overline{S_{SA[k]}}$	i	S_i
1	4	4	112	3	<i>aataatg</i>
2	5	5	12	6	<i>aatg</i>
3	6	6	2	9	<i>g</i>
4	1	1	334112	2	<i>taataatg</i>
5	2	2	34112	5	<i>taatg</i>
6	3	3	4112	8	<i>tg</i>

Figure 4.4: Suffix array of the string $\overline{S} = 334112$. The suffix S_i of S that is represented by $\overline{S_{SA[k]}}$ is shown in the same row.

Algorithm 4.1 Given the lexicographic names \bar{i} , compute the string \overline{S} .

$\overline{S} \leftarrow \varepsilon$ /* \overline{S} is the empty string */

$i \leftarrow 2$

while $i \leq n$ **do**

 append \bar{i} to \overline{S}

$i \leftarrow i + 3$

$i \leftarrow 3$

while $i \leq n$ **do**

 append \bar{i} to \overline{S}

$i \leftarrow i + 3$

According to the lexicographic naming and the definition of \bar{S} , the suffix $\bar{S}_{\frac{i+1}{3}}$ of \bar{S} represents the suffix S_i of S if $i \bmod 3 = 2$. More precisely, the string $\bar{S}[\frac{i+1}{3}..n]$ represents the string $S[i..n]\$$. In our example, for $i = 2$, we have $\bar{S}[\frac{2+1}{3}..n] = \bar{S}[1..3] = 334112[1..3] = 334$, where 334 stands for *taataatg*\$. The string *taataatg* in turn is the second suffix S_2 of S . Analogously, for $i \bmod 3 = 0$, the suffix $\bar{S}_{\frac{n+1}{3}}$ of \bar{S} represents the suffix S_i of S . To be precise, the string $\bar{S}[\frac{n+1}{3}..n]$ represents the string $S[i..n]\$$. In our example, for $i = 3$, we have $\bar{S}[\frac{9+1}{3}..n] = \bar{S}[4..6] = 334112[4..6] = 112$, where 112 stands for *aataatg*\$. The string *aataatg* in turn is the third suffix S_3 of S .

In order to simplify the presentation, let us define the transformation function $\tau : \{i \mid 1 \leq i \leq n, i \bmod 3 \neq 1\} \rightarrow \{1, \dots, \frac{2n}{3}\}$ by

$$\tau(i) = \begin{cases} \frac{i+1}{3}, & \text{if } i \bmod 3 = 2 \\ \frac{n+i}{3}, & \text{if } i \bmod 3 = 0 \end{cases}$$

So, for $i \bmod 3 \neq 1$, the suffix S_i of S is represented by the suffix $\bar{S}_{\tau(i)}$ of \bar{S} .

The mapping $\tau^{-1} : \{1, \dots, \frac{2n}{3}\} \rightarrow \{i \mid 1 \leq i \leq n, i \bmod 3 \neq 1\}$ defined by

$$\tau^{-1}(j) = \begin{cases} 3j - 1, & \text{if } 1 \leq j \leq \frac{n}{3} \\ 3j - n, & \text{if } \frac{n}{3} < j \leq \frac{2n}{3} \end{cases}$$

is the inverse of τ (it is left as an exercise to the reader to verify this). Therefore, for $1 \leq j \leq \frac{2n}{3}$, the suffix \bar{S}_j of \bar{S} represents the suffix $S_{\tau^{-1}(j)}$ of S .

The crucial point is that for $i \neq j$ with $i \bmod 3 \neq 1$ and $j \bmod 3 \neq 1$, we have $S_i < S_j$ if and only if $\bar{S}_{\tau(i)} < \bar{S}_{\tau(j)}$; see Exercise 4.1.5. In other words, one can decide whether $S_i < S_j$ or $S_j < S_i$ by comparing the relative order of their representatives in $\bar{S}\bar{A}$. Let k and l be the indices so that $\bar{S}_{\bar{S}\bar{A}[k]}$ and $\bar{S}_{\bar{S}\bar{A}[l]}$ represent S_i and S_j , respectively. Then $S_i < S_j$ if and only if $k < l$. Therefore, we are faced with the following problem: Given i with $1 \leq i \leq n$ and $i \bmod 3 \neq 1$, find the index k such that $\bar{S}_{\bar{S}\bar{A}[k]}$ represents S_i . Because $\bar{S}_{\tau(i)}$ represents S_i , it follows that $\bar{S}\bar{A}[k] = \tau(i)$. That is, $k = \bar{S}\bar{A}[\tau(i)]$. To sum up, we have $S_i < S_j$ if and only if $\bar{S}\bar{A}[\tau(i)] < \bar{S}\bar{A}[\tau(j)]$.

Phase 2: The task of sorting $[S_i \mid 1 \leq i \leq n, i \bmod 3 = 1]$ is equivalent to sorting $[(S[i], S_{i+1}) \mid 1 \leq i \leq n, i \bmod 3 = 1]$. Clearly, $i \bmod 3 = 1$ implies that $(i+1) \bmod 3 = 2$. Moreover, the lexicographic order of all suffixes $[S_{i+1} \mid 1 \leq i \leq n-1, (i+1) \bmod 3 = 2]$ is implicitly contained in $\bar{S}\bar{A}$. Algorithm 4.2 shows how the lexicographically sorted list of these suffixes can be extracted from $\bar{S}\bar{A}$. Consequently, one pass of radix sort w.r.t. the first component of the pair $(S[i], S_{i+1})$ yields the desired result. It is rather obvious that this phase runs in linear time.

In our example, we have to sort $[S_1, S_4, S_7] = [ctaataatg, ataataatg, atg]$, and Algorithm 4.2 returns the list $[S_2, S_5, S_8]$. Then radix sort applied to the list $[(c, S_2), (a, S_5), (a, S_8)]$ sorts this list according to the first component. In our

Algorithm 4.2 Given the suffix array \overline{SA} , determine the lexicographically sorted *list* (sequence) of suffixes S_i with $i \bmod 3 = 2$.

```

list ← [ ]          /* list is the empty list */
for k ← 1 to  $\frac{2n}{3}$  do
  j ←  $\overline{SA}[k]$ 
  i ←  $\tau^{-1}(j)$ 
  if  $i \bmod 3 = 2$  then
    append i to list      /* append  $S_i$  to list */

```

example, this yields the list $[(a, S_5), (a, S_8), (c, S_2)]$. So phase 2 of the skew algorithm returns the sequence $[S_4, S_7, S_1] = [ataatg, atg, ctaataatg]$.

Phase 3: As in the merge sort algorithm (see e.g. [61]), one merges the sorted sequence of suffixes S_i with $i \bmod 3 = 1$ with the sorted sequence of suffixes S_j with $j \bmod 3 \neq 1$. A suffix S_i with $i \bmod 3 = 1$ is compared to a suffix S_j with $j \bmod 3 \neq 1$ according to the following criteria:

a) If $i \bmod 3 = 1$ and $j \bmod 3 = 2$, then

$$\begin{aligned}
 S_i < S_j &\Leftrightarrow (S[i], S_{i+1}) <_{lex} (S[j], S_{j+1}) \\
 &\Leftrightarrow (S[i], \overline{ISA}[\tau(i+1)]) <_{lex'} (S[j], \overline{ISA}[\tau(j+1)])
 \end{aligned}$$

because $(i+1) \bmod 3 = 2$ and $(j+1) \bmod 3 = 0$. Note that the order $<_{lex}$ is the lexicographic product of the orders on strings, while the order $<_{lex'}$ is the lexicographic product of the order on strings (here of length 1) and the natural order on the natural numbers. For the convenience of the reader, we recall the formal definition of the lexicographic product of two orders: Given two partial orders $<_1$ and $<_2$ on sets M_1 and M_2 , respectively, their *lexicographic product* $<_{lex}$ is defined by $(u_1, u_2) <_{lex} (v_1, v_2)$ if and only if either $u_1 <_1 v_1$ or $u_1 = v_1$ and $u_2 <_2 v_2$, where $u_1, v_1 \in M_1$ and $u_2, v_2 \in M_2$.

b) If $i \bmod 3 = 1$ and $j \bmod 3 = 0$, then¹

$$\begin{aligned}
 S_i < S_j &\Leftrightarrow (S[i..i+1], S_{i+2}) <_{lex} (S[j..j+1], S_{j+2}) \\
 &\Leftrightarrow (S[i..i+1], \overline{ISA}[\tau(i+2)]) <_{lex'} (S[j..j+1], \overline{ISA}[\tau(j+2)])
 \end{aligned}$$

because $(i+2) \bmod 3 = 0$ and $(j+2) \bmod 3 = 2$.

Consequently, a comparison of two suffixes takes only constant time. Because there are at most $n - 1$ comparisons, the merging phase also runs in linear time. It is illustrated in Figure 4.5.

¹In the boundary case $j = n$, we set $S[j+1] = \$$. Because $\$$ is smaller than any other character, the value of $\overline{ISA}[\tau(j+2)]$ does not matter in this case.

i (pos. in S)	4	7	1		3	6	9	2	5	8
1-2 comp.	$(a, 5)$	$(a, 6)$	$(c, 4)$					$(t, 1)$	$(t, 2)$	$(t, 3)$
1-0 comp.	$(at, 2)$	$(at, 3)$	$(ct, 1)$		$(aa, 5)$	$(aa, 6)$	$(g\$, \perp)$			

Figure 4.5: Merging phase: The sorted sequence of the suffixes S_i with $i \bmod 3 = 1$ (left to the three vertical lines) is merged with the sorted sequence of the suffixes S_i with $i \bmod 3 \neq 1$ (right to the three vertical lines). The row “1-2 comp.” contains the pairs that are used in comparisons where $i \bmod 3 = 1$ and $j \bmod 3 = 2$, while the row “1-0 comp.” contains the pairs that are used in comparisons where $i \bmod 3 = 1$ and $j \bmod 3 = 0$. The result of the merging phase is the suffix array $[3, 6, 4, 7, 1, 9, 2, 5, 8]$ of S ; see also Figure 4.1 (page 61).

Exercise 4.1.5 For $i \neq j$ with $i \bmod 3 \neq 1$ and $j \bmod 3 \neq 1$, prove that $S_i < S_j$ if and only if $\bar{S}_{\tau(i)} < \bar{S}_{\tau(j)}$.

Hint: Expand each lexicographic name in $\bar{S}_{\tau(i)}$ and $\bar{S}_{\tau(j)}$ by its corresponding triple and use a case distinction on $i \bmod 3$ and $j \bmod 3$. Use the fact that $\$$ is smaller than any other character in the alphabet Σ .

Theorem 4.1.6 Given a string S of length n , the skew algorithm constructs the suffix array of S in time $\Theta(n)$.

Proof The recurrence describing the worst-case running time of the skew algorithm is:

$$T(n) = \begin{cases} \Theta(1) & , \text{ if } n < 3 \\ T(2n/3) + \Theta(n), & \text{ if } n \geq 3 \end{cases}$$

According to the extended *master theorem* [11, 286], a recurrence of the form

$$T(n) = \sum_{i=1}^m T(\alpha_i \cdot n) + \Theta(n^k)$$

where $0 \leq \alpha_i \leq 1, m \geq 1, k \geq 0$, has the solution

$$T(n) = \begin{cases} \Theta(n^k), & \text{ if } \sum_{i=1}^m \alpha_i^k < 1 \\ \Theta(n^k \log n), & \text{ if } \sum_{i=1}^m \alpha_i^k = 1 \\ \Theta(n^c), & \text{ if } \sum_{i=1}^m \alpha_i^k > 1 \end{cases}$$

where c is the solution of the equation $\sum_{i=1}^m \alpha_i^c = 1$. Therefore, our recurrence has the solution $T(n) = \Theta(n)$. \square

The following observation is worth mentioning. If in phase 2, the sequence of suffixes $[S_i \mid 1 \leq i \leq n, i \bmod 3 = 1]$ would also be sorted recursively by the skew algorithm, then this would yield the recurrence

$$T(n) = T(n/3) + T(2n/3) + \Theta(n)$$

According to the master theorem, this recurrence has the solution $T(n) = \Theta(n \log n)$. In other words, the resulting algorithm would not be linear!

4.1.2 Induced sorting

In this section, we will explain another interesting linear-time SACA devised by Nong et al. [244]. It shares two key features with the skew algorithm: it is a recursive algorithm and it uses the method of induced sorting. We speak of “induced sorting” whenever a complete sort of a selected subset of suffixes can be used to “induce” a complete sort of other subsets of suffixes. In the skew algorithm, for example, a complete sort of the suffixes $\{S_i \mid 1 \leq i \leq n, i \bmod 3 \neq 1\}$ can be used to induce a complete sort of the suffixes $\{S_i \mid 1 \leq i \leq n, i \bmod 3 = 1\}$.

The induced sorting algorithm² of Nong et al. [244] heavily depends on the work of Ko and Aluru [184] (the reader can find the description of several precursor algorithms in [262]). Ko and Aluru classified suffixes into S-type and L-type suffixes and showed that a complete sort of the S-type suffixes can be used to induce a complete sort of the L-type suffixes (or vice versa). The contribution of Nong et al. is the insight that it is enough to sort the usually small set of LMS-substrings and to use it to induce the order of all suffixes. Moreover, the lexicographic order of the LMS-substrings is determined by the same principle, using recursion if needed.

In the rest of this section, let S be a string of length $n + 1$ that is terminated by the sentinel $\$$. The suffixes of S are classified into two types: Suffix S_i is S-type if $S_i < S_{i+1}$ and it is L-type if $S_i > S_{i+1}$. The last suffix, the sentinel $\$$, is S-type. We use a bit array $T[1..n + 1]$ to store the types of the suffixes: $T[i] = 0$ means that suffix S_i is L-type and $T[i] = 1$ means that it is S-type. For better readability, we write $T[i] = L$ instead of $T[i] = 0$ and $T[i] = S$ instead of $T[i] = 1$.

Lemma 4.1.7 *All suffixes can be classified in $O(n)$ time by a right-to-left scan of S .*

Proof It is readily verified that S_i is S-type if (a) $S[i] < S[i + 1]$ or (b) $S[i] = S[i + 1]$ and S_{i+1} is S-type. Analogously, S_i is L-type if (a) $S[i] > S[i + 1]$ or

²Nong et al. speak of “almost pure induced-sorting” but we will use the shorter term “induced sorting.”

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S	i	m	i	m	m	m	i	s	i	s	m	i	s	i	s	s	i	i	p	i	\$
type	S	L	S	L	L	L	S	L	S	L	L	S	L	S	L	L	S	S	L	L	S
LMS			*				*		*			*		*			*			*	

Figure 4.6: The type classification of the suffixes of the string S proceeds from right to left. It starts with the S-type suffix $S_{21} = \$$. Suffixes S_{19} and S_{20} are L-type because $p > i > \$$, whereas suffix S_{18} is S-type because $i < p$. Both suffixes S_{17} and S_{18} start with the same character i , so S_{17} is S-type because S_{18} is S-type. The same argument applies to the suffixes S_{15} and S_{16} : S_{15} is L-type because S_{16} is L-type. The LMS-positions of S are marked with an asterisk. For example, 3 is an LMS-position because suffix S_3 is S-type and the preceding suffix S_2 is L-type.

(b) $S[i] = S[i + 1]$ and S_{i+1} is L-type. Therefore, starting with the S-type sentinel $\$$, in a right-to-left scan of S , we can determine the types of all of its suffixes. \square

Figure 4.6 shows an example of type classification.

Lemma 4.1.8 *An S-type suffix is lexicographically greater than any L-type suffix starting with the same first character.*

Proof For an indirect proof, suppose that there is an S-type suffix S_i and an L-type suffix S_j so that $S_i[1] = S[i] = a = S[j] = S_j[1]$ and $S_i < S_j$. We can write $S_i = aubv$ and $S_j = aucw$, where $b \neq c$ are characters and u , v , and w are (possibly empty) strings.

1. Suppose that u consists solely of a 's. Because S_i is S-type, it follows that $a \leq b$. Similarly, since S_j is L-type, it follows that $a \geq c$. The combination of these two facts yields $c \leq b$. However, $S_i < S_j$ implies $b < c$, a contradiction.
2. Otherwise, u contains a character other than a . Let d be the leftmost character in u that is different from a . Because S_i is S-type, it follows that $a < d$. Similarly, since S_j is L-type, it follows that $d > a$. This contradiction proves the lemma.

\square

Corollary 4.1.9 *In the suffix array of S , among all suffixes that begin with the same character, the L-type suffixes appear before the S-type suffixes.*

Proof Direct consequence of Lemma 4.1.8 □

We employ an array C of size σ to divide the suffix array of S into buckets (without loss of generality, we assume that all characters from the ordered alphabet Σ appear in the string S). For every $c \in \Sigma$, we define $C[c] = \sum_{b \in \Sigma, b < c} \text{cnt}[b]$, where $\text{cnt}[b]$ is the number of occurrences of character b in S . In other words, if we consider all characters in Σ that are smaller than c , then $C[c]$ is the overall number of their occurrences in S . The c -interval $[i..j]$ can be determined by $i = C[c] + 1$ and $j = C[c + 1]$ (where $c + 1$ is the character that follows c in the alphabet Σ). In the following, we call the c -interval the c -bucket. Every c -bucket $[i..j]$ can further be divided into the interval $[i..k]$ containing all L-type suffixes starting with character c and the interval $[k + 1..j]$ containing all S-type suffixes starting with character c . The interval $[i..k]$ is called the L-type region and $[k + 1..j]$ is called the S-type region of the c -bucket ($k = i - 1$ or $k = j$ is possible, i.e., the S-type region or the L-type region of the bucket may be empty).

Definition 4.1.10 A position i , $1 < i \leq n + 1$, in S with $T[i - 1] = \text{L}$ and $T[i] = \text{S}$ (i.e., suffix S_{i-1} is L-type and suffix S_i is S-type) is called *LMS-position* (leftmost S-type position); see Figure 4.6.

Now we are in a position to formulate the induced sorting algorithm.

Phase 0: Compute the type array T by a right-to-left scan of S .

Phase I: Compute all LMS-positions in S and sort the corresponding suffixes in ascending lexicographic order (we will elaborate on this phase later).

Phase II:

1. Scan the sorted sequence of LMS-positions from right to left. For each position encountered in the scan,³ move it to the current end of its bucket in A (initially, the end of a c -bucket is the index $C[c + 1]$), and shift the current end of the bucket by one position to the left.
2. Scan the array A from left to right. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an L-type suffix, move its start position $A[i] - 1$ in S to the current front of its bucket in A (initially, the front of a c -bucket is the index $C[c] + 1$), and shift the current front of the bucket by one position to the right.
3. Scan the array A from right to left. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an S-type suffix, move its start position $A[i] - 1$ in

³Each undefined entry \perp is ignored because it does not correspond to a suffix.

S to the current end of its bucket in A (initially, the end of a c -bucket is the index $C[c + 1]$), and shift the current end of the bucket by one position to the left.

An illustration of phase II can be found in Figure 4.7. It should be stressed that in step 3, the suffixes starting at LMS-positions (these are of S-type) are already in the array A . This does no harm because each of these suffixes will be overwritten before it is reached in the right-to-left scan of A ; see Lemma 4.1.12.

Lemma 4.1.11 *Step 2 of phase II correctly sorts all L-type suffixes of S .*

Proof First, we show that every L-type suffix is placed at its correct position. We proceed by induction on the number q of placed L-type suffixes. Clearly, $A[1] = n + 1$ because $\$$ appears at position $n + 1$ in S , and $\$$ is the lexicographically smallest character. Furthermore, $S_{A[i]-1} = S_n = c\$$ is an L-type suffix because $S[n] = c > \$ = S[n + 1]$. Consequently, position n is moved to the front of the c -bucket. This is certainly correct because the suffix S_n is the lexicographically smallest suffix starting with character c . As an inductive hypothesis, suppose that q L-type suffixes have been placed correctly. We have to show that the $(q + 1)$ -th L-type suffix will be placed correctly. Suppose that in the left-to-right scan of the array A we are at index $i > 1$ with $A[i] \neq \perp$, and let $A[i] = j + 1$ for some $j \geq 1$. That is, S_{j+1} is either an S-type suffix starting at an LMS-position or an L-type suffix that has already been placed, and suffix S_j is the $(q + 1)$ -th L-type suffix that has to be placed. Let $c = S[j]$. For a proof by contradiction, suppose that when we move the start position j to the current front of the c -bucket in A , there is already an L-type suffix S_k in the c -bucket that is lexicographically greater than S_j . So in the c -bucket, k is left to j . This means that there must be an index $i' < i$ so that $A[i'] = k + 1$. In other words, $k + 1$ precedes $j + 1$ in the array A . Because both S_j and S_k are in the c -bucket, we have $S_j = cS_{j+1}$ and $S_k = cS_{k+1}$. In conjunction with $S_j < S_k$, this has $S_{j+1} < S_{k+1}$ as a consequence. Note that S_{j+1} is either an S-type suffix starting at an LMS-position or an L-type suffix, and the same is true for S_{k+1} . According to the inductive hypothesis, S-type suffixes starting at LMS-positions and the first q placed L-type suffixes are in the correct order. Thus, $j + 1$ must precede $k + 1$ in the array A . This, however, contradicts our previous conclusion that $k + 1$ precedes $j + 1$ in A . We conclude that the $(q + 1)$ -th L-type suffix S_j is placed correctly.

Second, we prove that every L-type suffix will actually be placed during the scan. For an inductive proof, assume that the q lexicographically smallest L-type suffixes have been placed. Let S_j be the $(q + 1)$ -th lexicographically smallest L-type suffix. We have to show that S_j will be placed during the scan. Clearly, $S_{j+1} < S_j$ because S_j is L-type (thus, if S_{j+1} appears in the array A , then it must appear left to S_j). Moreover, S_{j+1} is

either a suffix starting at an LMS-position or an L-type suffix. By the induction hypothesis, S_{j+1} has been placed. Consequently, when the scan reaches S_{j+1} , the L-type suffix S_j is placed. \square

Lemma 4.1.12 *Step 3 of phase II correctly sorts all S-type suffixes of S .*

Proof The proof is very similar to the proof of the preceding lemma. First, we show that every S-type suffix is placed at its correct position. We proceed by induction on the number q of placed S-type suffixes. For the base case, note that all suffixes starting with the largest character c_σ must be L-type. That is, the S-type region of the c_σ -bucket is empty, and hence $A[n+1]$ corresponds to an L-type suffix. In the right-to-left scan of A , let i be the first (rightmost) index so that $S_{A[i]-1} = cS_{A[i]}$ is an S-type suffix (clearly, $c < c_\sigma$ and it is not difficult to show that i belongs to the c_σ -bucket). The algorithm places the index $A[i] - 1$ at the very end of the c -bucket. This is correct because $S_{A[i]-1} = cS_{A[i]}$ is the lexicographically largest suffix that starts with the character c . As an inductive hypothesis, suppose that q S-type suffixes have been placed correctly. We have to show that the $(q+1)$ -th S-type suffix will be placed correctly. Suppose that in the right-to-left scan of the array A we are at index i , and let $A[i] = j+1$ for some $j \geq 1$. That is, S_{j+1} is either an L-type suffix or an S-type suffix that has already been placed, and suffix S_j is the $(q+1)$ -th S-type suffix that has to be placed. Let $c = S[j]$. For a proof by contradiction, suppose that when we move the start position j to the current end of the c -bucket in A , there is already an S-type suffix S_k in the c -bucket that is lexicographically smaller than S_j . So in the c -bucket, k is right to j . This means that there must be an index $i' > i$ so that $A[i'] = k+1$. In other words, in the right-to-left scan of A , $k+1$ is encountered before $j+1$. Because both S_j and S_k are in the c -bucket, we have $S_j = cS_{j+1}$ and $S_k = cS_{k+1}$. Moreover, $S_k < S_j$ has $S_{k+1} < S_{j+1}$ as a consequence. If S_{j+1} (S_{k+1}) is an L-type suffix, then it is at its correct position by Lemma 4.1.11. Otherwise, if S_{j+1} (S_{k+1}) is an S-type suffix, then it is at its correct position by the inductive hypothesis. Therefore, in the right-to-left scan of A , $j+1$ must be encountered before $k+1$, a contradiction to our assumption. We conclude that the $(q+1)$ -th S-type suffix S_j is placed correctly.

Second, we prove that every S-type suffix will actually be placed during the scan. For an inductive proof, assume that the q lexicographically largest S-type suffixes have been placed, and let S_j be the $(q+1)$ -th lexicographically largest S-type suffix. We have to show that S_j will be placed during the scan. Clearly, $S_j < S_{j+1}$ because S_j is S-type (thus, if S_{j+1} appears in the array A , then it must appear right to S_j). If S_{j+1} is an L-type suffix, then it was placed correctly in step 2. Otherwise, if S_{j+1} is an S-type suffix, then it has also been placed by the induction hypothesis. In both cases, S_{j+1} was encountered before and S_j is placed. \square

The following notions are used in the formulation of phase I of the induced sorting algorithm.

Definition 4.1.13 A substring of S that starts at an LMS-position and ends at the next LMS-position is called an *LMS-substring*. By definition, the sentinel is also an LMS-substring. Any suffix of an LMS-substring is called an *LMS-suffix*.

Now let us elaborate on phase I, which is illustrated in Figure 4.8. Note that steps 2 and 3 are verbatim the same as in phase II.

Phase I:

1. Scan the array T from left to right and place each LMS-position into its bucket and into an array P (so if there are m LMS-positions, then P has size m). To be precise, for each position j encountered in the scan, if j is an LMS-position, move it to the current end of its bucket in A (initially, the end of a c -bucket is the index $C[c+1]$), and shift the current end of the bucket by one position to the left. Furthermore, move j to the current front of array P (initially, the front of P is the index 1), and shift the current front of P by one position to the right.

In this step, each LMS-position j represents the last character $S[j]$ of the LMS-substring that ends at j (and not suffix S_j as in phase II).

2. Scan the array A from left to right. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an L-type suffix, move its start position $A[i]-1$ in S to the current front of its bucket in A (initially, the front of a c -bucket is the index $C[c]+1$), and shift the current front of the bucket by one position to the right.

In this step, a position j with $T[j] = L$ represents the LMS-suffix that starts at position j and ends at the next LMS-position (and not the L-type suffix S_j as in phase II).

3. Scan the array A from right to left. For each entry $A[i]$ encountered in the scan, if $S_{A[i]-1}$ is an S-type suffix, move its start position $A[i]-1$ in S to the current end of its bucket in A (initially, the end of a c -bucket is the index $C[c+1]$), and shift the current end of the bucket by one position to the left.

In this step, a position j with $T[j] = S$ represents the LMS-suffix that starts at position j and ends at the next LMS-position (and not the S-type suffix S_j as in phase II). Thus, if j is an LMS-position, then it represents the LMS-substring that starts at position j and ends at the next LMS-position.

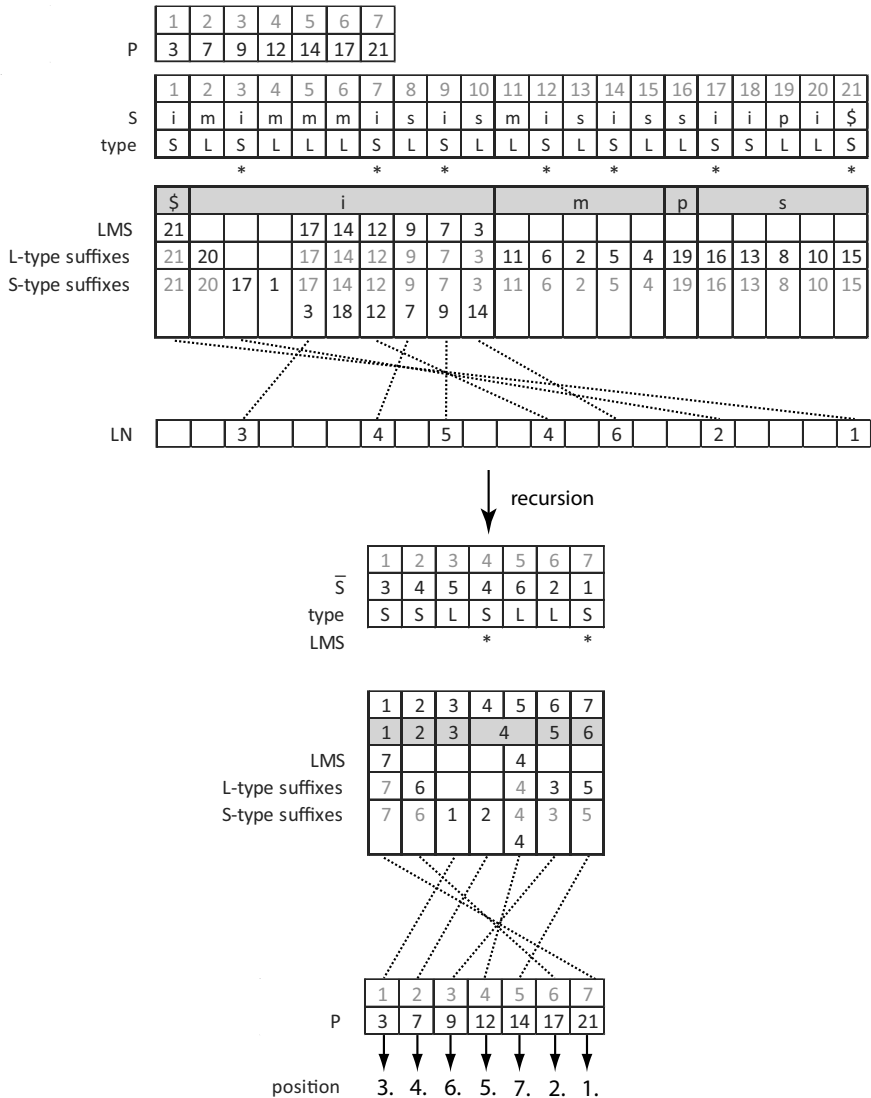


Figure 4.8: Phase I of the induced sorting algorithm.

4. Initialize an array $LN[1..n+1]$ that will contain the new lexicographic names of the LMS-substrings, and initialize the counter \bar{i} for new lexicographic names to 1. Because the sentinel is the lexicographically smallest LMS-substring, it gets the smallest new lexicographic name, i.e., $LN[n+1] = 1$. Furthermore, initialize $prev = n+1$. Now, starting at index $i = 2$, scan the array A from left to right. Whenever an entry $A[i] = j$ is encountered so that j is the start position of an LMS-substring do the following:
 - Compare the LMS-substring starting at position $prev$ in S with the LMS-substring starting at position j in S (character by character).
 - If they are different, increment \bar{i} by one.
 - Set $LN[j] = \bar{i}$ and $prev = j$.
5. Scan the array $P[1..m]$ from left to right and for each k with $1 \leq k \leq m$ set $\bar{S}[k] = LN[P[k]]$.
6. If $\bar{i} = m$, then the LMS-substrings are pairwise different. Compute the suffix array \bar{SA} of the string \bar{S} directly by $\bar{SA}[\bar{S}[k]] = k$.
7. Otherwise, *recursively* compute the suffix array \bar{SA} of the string \bar{S} .
8. The ascending lexicographic order of all the suffixes of S that start at an LMS-position is $P[\bar{SA}[1]], P[\bar{SA}[2]], \dots, P[\bar{SA}[m]]$.

Lemma 4.1.14 *After steps 1-3 of phase I, LMS-substrings appear in lexicographic order in A .*

Proof As already mentioned, in phase I an LMS-position j corresponds to the last character $S[j]$ of the LMS-substring ending at position j . By contrast, in phase II an LMS-position j corresponds to the suffix S_j . After step 1 of phase I, each LMS-position is in the S-type region of its bucket. That is, the length 1 suffixes (last characters) of all LMS-substrings are in the correct order in A . Now the proofs of Lemmata 4.1.11 and 4.1.12 apply with a grain of salt, and we conclude that after steps 1-3 all LMS-suffixes of length greater than 1 are in the correct order in A .⁴ Of course, the sentinel (more precisely, its position $n+1$) is still the first entry of A . Therefore, all LMS-substrings appear in lexicographic order in A (note that lexicographically adjacent LMS-substrings may be identical). \square

By Lemma 4.1.14, LMS-substrings (represented by their start positions) appear in lexicographic order in the array A . Thus, there are indices

⁴In step 1, an LMS-position j represents the length 1 LMS-suffix $S[j]$. In step 3, however, j represents the LMS-substring starting at position j . So length 1 LMS-suffixes are no longer represented.

$1 \leq i_1 < \dots < i_m \leq n+1$ and positions j_1, \dots, j_m so that $A[i_1] = j_1, \dots, A[i_m] = j_m$ (hence j_1, \dots, j_m is a permutation of $P[1], \dots, P[m]$). Step 4 renames all LMS-substrings according to their lexicographic order in the array A , where identical LMS-substrings get the same new name. To be precise, in step 4 we compare the current LMS-substring, say $S[j_k..j_{k+1}]$, with the previous LMS-substring $S[j_{k-1}..j_k]$. Suppose that the previous LMS-substring $S[j_{k-1}..j_k]$ has got the new lexicographic name \bar{i} . Now, if $S[j_k..j_{k+1}] = S[j_{k-1}..j_k]$, then $S[j_k..j_{k+1}]$ gets the same lexicographic name \bar{i} . Otherwise, $S[j_k..j_{k+1}]$ gets the new lexicographic name $\bar{i} + 1$. In both cases, the lexicographic name of $S[j_k..j_{k+1}]$ is stored at position j_k in array LN; see Figure 4.8. The new string \bar{S} is obtained by setting $\bar{S}[k] = \text{LN}[P[k]]$ in step 5. Now, there are two possibilities (step 6 or step 7). Either all LMS-substrings are pairwise different (which is equivalent to $\bar{i} = m$ after step 4) or there are at least two identical LMS-substrings. In the former case (step 6), the inverse suffix array ISA of the string \bar{S} coincides with \bar{S} (viewed as an array). In the latter case (step 7), we recursively apply the whole induced sorting algorithm to the string \bar{S} to get its suffix array $\overline{\text{SA}}$; see Figure 4.8. So after step 6 or step 7, we know the lexicographic order of all suffixes of \bar{S} . Lemma 4.1.15 proves that step 8 yields the lexicographic order of all the suffixes of S that start at an LMS-position (by means of the suffix array $\overline{\text{SA}}$ and the P array).

Lemma 4.1.15 *We have $\bar{S}_i < \bar{S}_j$ if and only if $S_{P[i]} < S_{P[j]}$.*

Proof Let u_i be the string obtained by replacing every lexicographic name in \bar{S}_i with the LMS-substring that is represented by this name. Furthermore, let v_i be the string obtained from $S_{P[i]}$ by doubling every character at an LMS-position. It is readily verified that $u_i = v_i$. Moreover, $u_i < u_j$ (where u_j is defined analogously) if and only if $S_{P[i]} < S_{P[j]}$. Hence the lemma follows. \square

All in all, the induced sorting algorithm correctly computes the suffix array. It remains to analyze its worst-case time-complexity. It is not difficult to see that each step in phases I and II takes at most $O(n)$ time. By definition 4.1.10, position 1 is not an LMS-position and there must be at least one position in between two consecutive LMS-positions. Hence $|\bar{S}| = m \leq \lfloor \frac{n+1}{2} \rfloor$. It follows from the master theorem that the whole induced sorting algorithm has a worst-case time complexity of $O(n)$.

Implementation details of the induced sorting algorithm:

1. It is not difficult to see that if two LMS-substrings are identical, then so are their type sequences. Consequently, in step 4 of phase I, if one compares characters and types of LMS-substrings simultaneously, then the comparison can be stopped when there is a character

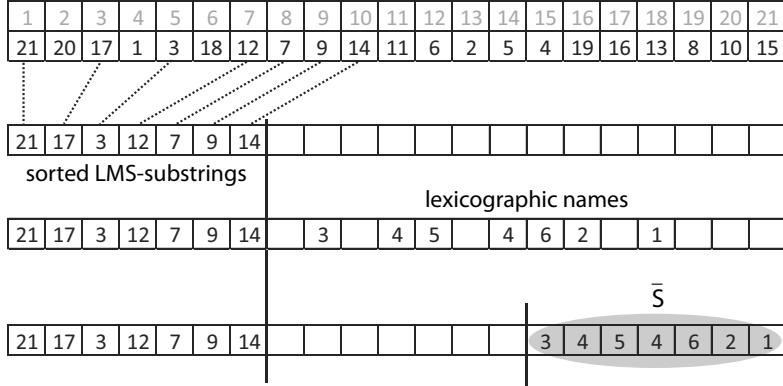


Figure 4.9: Apart from the string S , one array suffices in an implementation of the induced sorting algorithm.

mismatch or a type mismatch (this is done in the original algorithm by Nong et al. [244]).

2. Since $m \leq \lfloor \frac{n+1}{2} \rfloor$, the sorted LMS-substrings can be stored in the left half of the array A and the array LN (later the string \bar{S} and finally the array $\bar{S}A$) can be stored in the right half of A ; see Figure 4.9. As a matter of fact, it is possible to remove the type array T in the initial call of the induced sorting algorithm and to integrate the type arrays in recursive calls into the array A . Moreover, the pointers to the (current) front/end of the buckets are solely required in the initial call but not in recursive calls. In summary, the induced sorting algorithm can be implemented in such a way that it keeps only the string S (the input), the array A (the output), and the bucket pointers of the initial call (for a constant-size alphabet, these pointers take only constant space) in main memory; see [243] for details.

Exercise 4.1.16 Suppose that the word boundaries in a natural language text have already been determined by a parser. These word boundaries divide the text into tokens. As an example, consider the English sentence “This is a text.” and the corresponding sequence of tokens “This, is, a, text”. In this exercise, we assume that a text T is given as the concatenation of the tokens, in which tokens are separated by a special separator symbol $\#$. We assume that T consists of k tokens and n characters (including all occurrences of the separator symbol). In our example, $T = \text{This}\#\text{is}\#\text{a}\#\text{text}$ consists of 4 tokens and 14 characters. Let P be the set of positions at which a token starts, i.e., $P = \{1\} \cup \{i \mid 2 \leq i \leq n \text{ and } T[i-1] = \#\}$. The set of all suffixes of T starting with a full token is defined by $\{T_p \mid p \in P\}$, and the

word suffix array WSA is a permutation of P specifying the lexicographic order of the k suffixes from $\{T_p \mid p \in P\}$, i.e., it satisfies $T_{\text{WSA}[1]} < T_{\text{WSA}[2]} < \dots < T_{\text{WSA}[k]}$. Devise an $O(n)$ time and $O(k)$ space algorithm to construct the word suffix array of T .

Exercise 4.1.17 A *cyclic string* of length n is a string S in which the character at position n is considered to precede the character at position 1. The *cyclic string linearization problem* is the following: Choose a position to cut S so that the resulting linear string is the lexicographically smallest of all the n possible linear strings created by cutting S . Give an algorithm that solves this problem in $O(n)$ time.

4.2 The LCP-array

Throughout this book, $\text{lcp}(u, v)$ denotes the longest common prefix between two strings u and v , whereas $\text{lcs}(u, v)$ denotes the longest common suffix of u and v .

The suffix array is often augmented with the so-called LCP-array (or LCP-table), containing the lengths of the longest common prefixes between consecutive suffixes in SA. The formal definition reads as follows.

Definition 4.2.1 The *LCP-array* is an array of size $n + 1$ with boundary elements $\text{LCP}[1] = -1$ and $\text{LCP}[n + 1] = -1$, and for all i with $2 \leq i \leq n$ we have $\text{LCP}[i] = |\text{lcp}(S_{\text{SA}[i-1]}, S_{\text{SA}[i]})|$.

Figure 4.10 shows the LCP-array of the string $S = \text{ctaataatg}$. The LCP-array first appeared in the seminal paper of Manber and Myers [214] on suffix arrays (where it was called *Hgt* array).

A suffix array enhanced with the corresponding LCP-array will henceforth be called an *enhanced suffix array*. More generally, the generic name *enhanced suffix array* (and the acronym ESA) stands for data structures consisting of the suffix array enhanced with additional arrays.

4.2.1 Linear-time construction

It is possible to modify some SACAs so that they compute the LCP-array as a by-product of the suffix array construction. This has been shown in [175] for the skew algorithm presented in Section 4.1.1 and in [107] for the induced sorting algorithm presented in Section 4.1.2, but other SACAs could probably also be modified to produce the LCP-array.

Another approach is to construct the LCP-array from an already constructed suffix array. To date, several different LCP-array construction algorithms (LACAs) of this kind are known [33, 126, 174, 177, 216, 264], but to review all of them goes beyond the scope of this book. In this

i	SA	ISA	LCP	$S_{SA[i]}$
1	3	5	-1	aataatg
2	6	7	3	aatg
3	4	1	1	ataatg
4	7	3	2	atg
5	1	8	0	ctaataatg
6	9	2	0	g
7	2	4	0	taataatg
8	5	9	4	taatg
9	8	6	1	tg
10			-1	

Figure 4.10: Suffix array, inverse suffix array, and LCP-array of the string $S = ctaataatg$.

section, we present two of them: the first linear-time LACA owing to Kasai et al. [177] and the Φ -algorithm developed by Kärkkäinen et al. [174]. Another LACA can be found in Section 7.5.1.

We start with Kasai et al.'s algorithm, which computes the LCP-array from the string S and its suffix array SA; see Algorithm 4.3. In its first for-loop, the algorithm computes the inverse suffix array ISA. In the second for-loop, the algorithm computes lcp-values. It starts with the longest suffix $S_i = S_1$ of S (so $i = 1$), computes $j = ISA[i]$ and then $LCP[j]$ by a left-to-right comparison of the characters in $S_{SA[j-1]}$ and $S_{SA[j]} = S_i$. The same is done for the other suffixes S_i of S by incrementing i successively; see Figure 4.11 for an example. Algorithm 4.3 avoids many redundant character comparisons; this is justified by the following lemma.

Lemma 4.2.2 *For $2 \leq i \leq n$, we have $LCP[ISA[i]] \geq LCP[ISA[i-1]] - 1$.*

Proof Let $\ell = LCP[ISA[i-1]]$. Clearly, the lemma is true if $\ell \leq 1$, so suppose that $\ell \geq 2$. Let S_k be the suffix of S that immediately precedes S_{i-1} in the suffix array (that is, $k = SA[ISA[i-1] - 1]$). Since $LCP[ISA[i-1]] = \ell$, we have $|lcp(S_k, S_{i-1})| = \ell$. In other words, $S[k..k+\ell-1] = S[i-1..i+\ell-2]$ and $S[k+\ell] \neq S[i+\ell-1]$. By omitting the first character, we get $S[k+1..k+\ell-1] = S[i..i+\ell-2]$ and $S[k+\ell] \neq S[i+\ell-1]$, or equivalently, $|lcp(S_{k+1}, S_i)| = \ell - 1$. Moreover, S_{k+1} is lexicographically smaller than S_i because S_k is lexicographically smaller than S_{i-1} and $S[k] = S[i-1]$. Thus, S_{k+1} precedes S_i in the suffix array. Since $|lcp(S_{k+1}, S_i)| = \ell - 1$, all suffixes between S_{k+1} and S_i (if there is one at all) have a common prefix of length $\ell - 1$. Hence $LCP[ISA[i]] \geq \ell - 1$. \square

Algorithm 4.3 Computation of the LCP-array using S and its suffix array. In the pseudo-code, it is possible that $S[n+1]$ is accessed. To avoid a “cannot access beyond end of string” error, we assume that $S[n+1] = \$$.

```

LCP[1]  $\leftarrow$  -1
LCP[n+1]  $\leftarrow$  -1
for  $i \leftarrow 1$  to  $n$  do
    ISA[SA[i]]  $\leftarrow$   $i$ 
 $\ell \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$  do
     $j \leftarrow$  ISA[i]
    if  $j > 1$  then
         $k \leftarrow$  SA[j-1] /*  $S_k$  precedes  $S_i$  in SA */
        while  $S[k+\ell] = S[i+\ell]$  do
             $\ell \leftarrow \ell + 1$ 
        LCP[j]  $\leftarrow$   $\ell$ 
         $\ell \leftarrow \max\{\ell - 1, 0\}$ 

```

According to the preceding lemma, if $\ell = \text{LCP}[\text{ISA}[i-1]]$ is known, then one can skip at least $\ell - 1$ character comparisons in the computation of $\text{LCP}[\text{ISA}[i]]$; see Figure 4.11. Not only does this imply the correctness of Algorithm 4.3, but it is also the key to the linear run time of the algorithm.

Theorem 4.2.3 *Given a string S of length n and its suffix array, Algorithm 4.3 constructs the LCP-array in time $O(n)$.*

Proof We use an amortized analysis to show that the while-loop is executed at most $2n$ times. It is readily verified that this implies the theorem.

Each comparison in the while-loop ends with a mismatch, so there are $n-1$ mismatches (redundant character comparisons) in total. If a position p in S is involved in a match (i.e., in the while-loop $p = i + \ell$ and $S[k + \ell] = S[i + \ell]$), then this particular occurrence of character $S[p]$ will be skipped in further suffix comparisons. So there are at most n matches. \square

Kärkkäinen et al. [174] proposed a variant of Kasai et al.’s algorithm, which first computes a permuted LCP-array (PLCP-array) with the help of the so-called Φ -array and then derives the LCP-array from the PLCP-array. They called it “ Φ -algorithm” because it uses the Φ -array, which “is in some way symmetric to the ψ array.” (The ψ array will be introduced in Definition 5.5.4.)

Definition 4.2.4 Let S be a string of length n and let SA be its suffix array. We define the corresponding Φ -array and PLCP-array, respectively,

i	SA	ISA	LCP	$S_{SA[i]}$
1	3	5	-1	<i>aat aatg</i>
2	6	7	?	<i>aat g</i>
3	4	1	1	<i>ataatg</i>
4	7	3		<i>atg</i>
5	1	8	0	<i>ctaataatg</i>
6	9	2		<i>g</i>
7	2	4	0	<i>taat aatg</i>
8	5	9	4	<i>taat g</i>
9	8	6		<i>tg</i>
10			-1	

Figure 4.11: Algorithm 4.3 has computed the LCP-entries for the suffixes $S_1 = ctaataatg$, $S_2 = taataatg$, $S_3 = aataatg$, $S_4 = ataatg$, and $S_5 = taatg$. The suffix $S_5 = taatg$ occurs at index $i = 8$ and its longest common prefix with the suffix $S_2 = taataatg$ at index $i = 7$ has length 4. Hence $LCP[8] = 4$. Next, Algorithm 4.3 computes the LCP-entry for the suffix $S_6 = aatg$ at index $i = 2$. By Lemma 4.2.2, $S_6 = aatg$ has at least the first three characters in common with the suffix $S_3 = aataatg$ at index $i = 1$. Therefore, these three characters are skipped and the next character comparison of $S_6[4] = g$ with $S_3[4] = a$ yields a mismatch. So $LCP[2] = 3$.

i	1	2	3	4	5	6	7	8	9
$S[i]$	c	t	a	a	t	a	a	t	g
$\Phi[i]$	7	9	0	6	2	3	4	5	1
$\text{PLCP}[i]$	0	0	-1	1	4	3	2	1	0

Figure 4.12: Φ -array and PLCP-array of $S = ctaataatg$; cf. Figure 4.11.

as follows: For all i with $1 \leq i \leq n$ let

$$\Phi[i] = \begin{cases} \text{SA}[\text{ISA}[i] - 1] & \text{if } \text{ISA}[i] \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$\text{PLCP}[i] = \begin{cases} |\text{lcp}(S_i, S_{\Phi[i]})| & \text{if } \Phi[i] \neq 0 \\ -1 & \text{otherwise} \end{cases}$$

Figure 4.12 shows an example. So in the suffix array SA , the suffix S_j is immediately preceded by the suffix $S_{\Phi[j]}$ unless $\Phi[j] = 0$. For $j \neq 1$, we have

$$\text{PLCP}[\text{SA}[j]] = |\text{lcp}(S_{\text{SA}[j]}, S_{\Phi[\text{SA}[j]]})| = |\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[j-1]})| = \text{LCP}[j] \quad (4.1)$$

In case $j = 1$ this equation holds also true: we have $\text{LCP}[1] = -1$ and $\text{PLCP}[\text{SA}[1]] = -1$ because $\Phi[\text{SA}[1]] = 0$. Thus, the PLCP-array is a permutation of the LCP-array (apart from the artificial entry $\text{LCP}[n+1] = -1$). The difference between the two arrays is that the lcp-values occur in text order (position order) in the PLCP-array, whereas they occur in suffix array order (lexicographic order) in the LCP-array. As a consequence, we obtain the following corollary to Lemma 4.2.2.

Corollary 4.2.5 *For $2 \leq i \leq n$, we have $\text{PLCP}[i] \geq \text{PLCP}[i-1] - 1$.*

Proof For $j = \text{ISA}[i]$, Equation 4.1 has the form

$$\text{LCP}[\text{ISA}[i]] = \text{PLCP}[\text{SA}[\text{ISA}[i]]] = \text{PLCP}[i]$$

Furthermore, Lemma 4.2.2 states that $\text{LCP}[\text{ISA}[i]] \geq \text{LCP}[\text{ISA}[i-1]] - 1$ for all $i \neq 1$. Hence $\text{PLCP}[i] \geq \text{PLCP}[i-1] - 1$ for all i with $i \neq 1$. \square

Pseudo-code of the Φ -algorithm is shown in Algorithm 4.4. A comparative analysis of Algorithm 4.3 and the Φ -algorithm can be found in [126]. The bottom line is that the Φ -algorithm is faster because of better memory locality: it merely needs sequential access to the Φ -array and the PLCP-array in its second for-loop. However, in virtually all applications lcp-values are required to be in suffix array order, so that in a final step the PLCP-array must be converted into the LCP-array. Although this final step (the third for-loop of the Φ -algorithm) has poor memory locality because it needs random access to the PLCP-array, the overall algorithm is still faster than Kasai et al.'s.

Algorithm 4.4 The Φ -algorithm computes the LCP-array using S and its suffix array SA .

```

 $\Phi[SA[1]] \leftarrow 0$ 
for  $i \leftarrow 2$  to  $n$  do           /* compute the  $\Phi$ -array */
     $\Phi[SA[i]] \leftarrow SA[i - 1]$ 
 $\ell \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$  do
     $k \leftarrow \Phi[i]$ 
    if  $k \neq 0$  then
        while  $S[k + \ell] = S[i + \ell]$  do      /*  $S_k$  precedes  $S_i$  in  $SA$  */
             $\ell \leftarrow \ell + 1$ 
         $PLCP[i] \leftarrow \ell$ 
         $\ell \leftarrow \max\{\ell - 1, 0\}$ 
    else
         $PLCP[i] \leftarrow -1$ 
for  $i \leftarrow 1$  to  $n$  do
     $LCP[i] \leftarrow PLCP[SA[i]]$ 
 $LCP[n + 1] \leftarrow -1$ 

```

Exercise 4.2.6 Prove the correctness of the Φ -algorithm and analyze its worst-case time complexity.

4.2.2 Longest common prefix

Given the LCP-array, the length of the longest common prefix between two consecutive suffixes in the suffix array can be determined in constant time. An obvious question is: Is it possible to compute the length of the longest common prefix between two *arbitrary* suffixes in the suffix array? Based on the results from Chapter 3, the next lemmata give an answer in the affirmative.

Lemma 4.2.7 *Given a string S of length n and its LCP-array, we have*

$$|\text{lcp}(S_{SA[i]}, S_{SA[j]})| = \text{LCP}[\text{RMQ}_{\text{LCP}}(i + 1, j)] = \min_{i < k \leq j} \{\text{LCP}[k]\}$$

for all indices $1 \leq i < j \leq n$.

Proof The second equality follows from the definition of $\text{RMQ}_{\text{LCP}}(i + 1, j)$. Let us turn to the first equality. It is readily verified that a string ω is a common prefix of $S_{SA[i]}$ and $S_{SA[j]}$ if and only if it is a common prefix of all suffixes $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$. Let $m = \text{RMQ}_{\text{LCP}}(i + 1, j)$ and $\ell = \text{LCP}[m]$. Obviously, $S[i..i + \ell - 1]$ is a common prefix of all suffixes $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$, hence of $S_{SA[i]}$ and $S_{SA[j]}$. Moreover, $S[i..i + \ell]$ is not a common prefix of

$S_{SA[m-1]}$ and $S_{SA[m]}$ because $|\text{lcp}(S_{SA[m-1]}, S_{SA[m]})| = \text{LCP}[m] = \ell < \ell + 1 = |S[i..i + \ell]|$. Consequently, $S[i..i + \ell]$ is not a common prefix of $S_{SA[i]}$ and $S_{SA[j]}$. All in all, $|\text{lcp}(S_{SA[i]}, S_{SA[j]})| = \ell$. \square

As in Chapter 3, we say that an algorithm has time complexity $\langle p(n), q(n) \rangle$ if its preprocessing time is $p(n)$ and its query is time $q(n)$.

Lemma 4.2.8 *There is an $\langle O(n), O(1) \rangle$ -time algorithm for answering longest common prefix queries between two suffixes of a string S .*

Proof We must show that after a linear-time preprocessing, $|\text{lcp}(S_i, S_j)|$ can be computed in constant time for all positions $1 \leq i \leq j \leq n$. Given string S of length n , one can compute the corresponding arrays SA, ISA, and LCP in $O(n)$ time. Moreover, the LCP-array can be preprocessed in linear time so that range minimum queries can be answered in constant time; see Section 3.3. For $i = j$, we have $|\text{lcp}(S_i, S_j)| = |S_i|$. Otherwise, for $i \neq j$, we have

$$|\text{lcp}(S_i, S_j)| = \begin{cases} \text{LCP}[\text{RMQ}_{\text{LCP}}(\text{ISA}[i] + 1, \text{ISA}[j])], & \text{if } \text{ISA}[i] < \text{ISA}[j] \\ \text{LCP}[\text{RMQ}_{\text{LCP}}(\text{ISA}[j] + 1, \text{ISA}[i])], & \text{if } \text{ISA}[j] < \text{ISA}[i] \end{cases}$$

This is a direct consequence of Lemma 4.2.7 because the indices $i' = \text{ISA}[i]$ and $j' = \text{ISA}[j]$ satisfy $S_{SA[i']} = S_i$ and $S_{SA[j']} = S_j$. \square

Corollary 4.2.9 *There is an $\langle O(n), O(1) \rangle$ -time algorithm for answering longest common suffix queries between two prefixes of a string S .*

Proof Observe that $\text{lcs}(S[1..i], S[1..j]) = \text{lcp}(S_{n-i+1}^{\text{rev}}, S_{n-j+1}^{\text{rev}})$, where S^{rev} denotes the reverse string of S . This, in combination with the fact that S^{rev} can be obtained in linear time from S , implies that $\text{lcs}(S[1..i], S[1..j])$ can also be computed in constant time after a linear-time preprocessing. \square

4.3 The lcp-interval tree

Most concepts of this section originate from Abouelhoda et al. [1]. The idea to use RMQs in this context stems from Fischer and Heun [108].

To see the usefulness of lcp-intervals, let us have a second look at the enhanced suffix array of the string $S = \text{ctaataatg}$, which is replicated in Figure 4.13. By definition 4.1.3, the a -interval is the interval $[1..4]$, the aa -interval is $[1..2]$, and the aat -interval is also $[1..2]$. By contrast, there is no substring ω of S so that the interval $[1..3]$ is an ω -interval. The next definition allows us to identify such intervals solely by means of the LCP-array. The declarations $\text{LCP}[1] = -1$ and $\text{LCP}[n+1] = -1$ ensure that the definition also covers the interval $[1..n]$.

i	SA	ISA	LCP	$S_{SA[i]}$	lcp-intervals		
1	3	5	-1	<i>aataatg</i>	0	1	3
2	6	7	3	<i>aatg</i>			2
3	4	1	1	<i>ataatg</i>		1	4
4	7	3	2	<i>atg</i>			
5	1	8	0	<i>ctaataatg</i>			
6	9	2	0	<i>g</i>			
7	2	4	0	<i>taataatg</i>			
8	5	9	4	<i>taatg</i>			
9	8	6	1	<i>tg</i>			
10			-1				

Figure 4.13: Enhanced suffix array and lcp-intervals of $S = ctaataatg$.

Definition 4.3.1 An interval $[i..j]$, $1 \leq i < j \leq n$, in an LCP-array is called an *lcp-interval of lcp-value ℓ* if and only if

1. $LCP[i] < \ell$,
2. $LCP[k] \geq \ell$ for all k with $i + 1 \leq k \leq j$,
3. $LCP[k] = \ell$ for at least one k with $i + 1 \leq k \leq j$,
4. $LCP[j + 1] < \ell$.

We will also use the shorthand ℓ - $[i..j]$ for an lcp-interval $[i..j]$ of lcp-value ℓ , and $[i..j]$ will be called ℓ -interval. Every index k , $i + 1 \leq k \leq j$, with $LCP[k] = \ell$ is called ℓ -index (or lcp-index) of $[i..j]$. The set of all ℓ -indices of an ℓ -interval $[i..j]$ will be denoted by $\ell Indices(i, j)$. Furthermore, we will say that the lcp-interval ℓ - $[i..j]$ represents the string $\omega = S[SA[i]..SA[i] + \ell - 1]$, where ω is the longest common prefix of the suffixes $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$.

For ease of presentation, it is useful to ensure that the interval $[1..n]$ is always an lcp-interval of lcp-value 0. By Definition 4.3.1, this is the case if and only if there is at least one k with $2 \leq k \leq n$ so that $LCP[k] = 0$. This in turn is the case if and only if the string S contains at least two different characters. Thus, we tacitly assume that $\$$ is appended to strings containing only one character.

As an example, consider Figure 4.13. $[1..4]$ is a 1-interval because $LCP[1] = -1 < 1$, $LCP[4 + 1] = 0 < 1$, $LCP[k] \geq 1$ for all k with $2 \leq k \leq 4$, and $LCP[3] = 1$. Furthermore, the lcp-interval 1- $[1..4]$ represents the string a and $\ell Indices(1, 4) = \{3\}$. Similarly, the lcp-interval 3- $[1..2]$ represents

the string aat . By definition, the string aa is not represented by an lcp-interval. This is because each lcp-interval $[i..j]$ only represents the longest common prefix of the suffixes $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$. So the lcp-interval $[1..2]$ represents aat and not aa .

Lemma 4.3.2 *Two lcp-intervals ℓ - $[i..j] \neq m$ - $[p..q]$ cannot overlap, i.e., one of the following cases must hold:*

- $[i..j]$ is a subinterval of $[p..q]$, i.e., $p \leq i < j \leq q$.
- $[p..q]$ is a subinterval of $[i..j]$, i.e., $i \leq p < q \leq j$.
- $[i..j]$ and $[p..q]$ are disjoint, i.e., $j < p$ or $q < i$.

Proof Suppose to the contrary that $[i..j]$ and $[p..q]$ overlap, i.e., $i < p \leq j < q$ (the case $p < i \leq q < j$ is symmetric). By Definition 4.3.1, we have

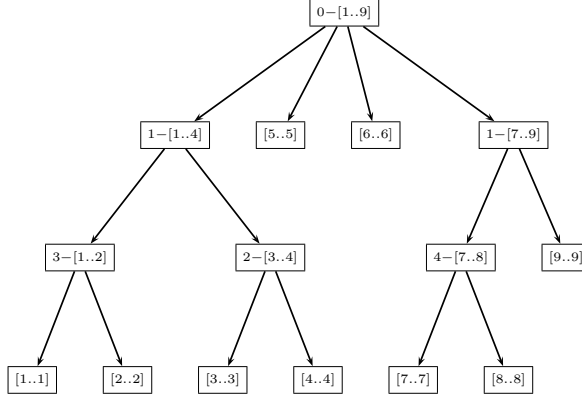
1. $LCP[i] < \ell$
2. $LCP[p] \geq \ell$
3. $LCP[j+1] < \ell$
4. $LCP[p] < m$
5. $LCP[j+1] \geq m$
6. $LCP[q+1] < m$

The combination of (2) and (4) yields $\ell \leq LCP[p] < m$, while the conjunction of (3) and (5) yields $m \leq LCP[j+1] < \ell$. In summary, we obtain $\ell < m < \ell$. This contradiction shows the lemma. \square

Definition 4.3.3 An m -interval $[p..q]$ is said to be *embedded* in an ℓ -interval $[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq p < q \leq j$) and $m > \ell$.⁵ The ℓ -interval $[i..j]$ is then called the interval *enclosing* $[p..q]$. If $[i..j]$ encloses $[p..q]$ and there is no interval embedded in $[i..j]$ that also encloses $[p..q]$, then $[p..q]$ is called a *child interval* of $[i..j]$ (vice versa, $[i..j]$ is the *parent interval* of $[p..q]$). This parent-child relationship constitutes a tree, which we call *lcp-interval tree*.

For instance, continuing the example of Figure 4.13, the child intervals of 1-[1..4] are 3-[1..2] and 2-[3..4]. The whole lcp-interval tree is shown in Figure 4.14. The root of an lcp-interval tree is always the 0-interval $[1..n]$. The lcp-interval tree of Figure 4.14 also contains singleton intervals, which are defined as follows.

⁵Note that we cannot have both $i = p$ and $j = q$ because $m > \ell$.

Figure 4.14: The lcp-interval tree for $S = ctaataatg$.

Definition 4.3.4 An interval $[k..k]$ is called *singleton interval*. The parent interval of such a singleton interval is the smallest lcp-interval $[i..j]$ that contains k .

How much space does an lcp-interval tree occupy? Clearly, there are exactly n singleton-intervals, hence n leaves. As each internal node of an lcp-interval tree is branching, there can be at most $n - 1$ internal nodes. Since the representation of a node needs at most three numbers, a node can be represented in constant space. It is readily seen that the number of edges is one less than the number of nodes. Consequently, there are at most $2n - 2$ edges because there are at most $2n - 1$ nodes in the lcp-interval tree. Since the edges are not labeled, we can surely represent each edge in constant space. To sum up, an lcp-interval tree requires only linear space. However, we will not construct this tree *explicitly*. As we shall see, it is possible to traverse this tree without constructing it.

4.3.1 Finding child and parent intervals

The next lemma shows how to determine child intervals.

Lemma 4.3.5 Let $[i..j]$ be an ℓ -interval. If $i_1 < i_2 < \dots < i_k$ are the ℓ -indices in ascending order, then the child intervals of $[i..j]$ are $[i..i_1 - 1]$, $[i_1..i_2 - 1]$, \dots , $[i_k..j]$ (note that some of them may be singleton intervals).

Proof Let $[p..q]$ be a non-singleton interval out of the intervals $[i..i_1 - 1]$, $[i_1..i_2 - 1]$, \dots , $[i_k..j]$ and let $m = \text{LCP}[\text{RMQ}_{\text{LCP}}(p + 1, q)]$. Since none of the indices $p + 1, \dots, q$ is an ℓ -index, it follows from Definition 4.3.1 that $\text{LCP}[k] > \ell$ for all k with $p + 1 \leq k \leq q$. Hence $m > \ell$. We claim that $[p..q]$ is an m -interval.

Note that $\text{LCP}[p] < \ell$ if $p = i$ and $\text{LCP}[p] = \ell$ if $p \neq i$. Analogously, $\text{LCP}[q+1] < \ell$ if $q = j$ and $\text{LCP}[q+1] = \ell$ if $q \neq j$. We have

1. $\text{LCP}[p] \leq \ell < m$,
2. $\text{LCP}[k] \geq m$ for all k with $p+1 \leq k \leq q$,
3. $\text{LCP}[k] = m$ for $k = \text{RMQ}_{\text{LCP}}(p+1, q)$
4. $\text{LCP}[q+1] \leq \ell < m$.

By Definition 4.3.1, $[p..q]$ is an m -interval. To show that m - $[p..q]$ is a child interval of ℓ - $[i..j]$, we must prove that there is no lcp-interval embedded in $[i..j]$ that encloses $[p..q]$. For a proof by contradiction, suppose that the lcp-interval r - $[lb..rb]$ is embedded in $[i..j]$ and encloses $[p..q]$. We have $m > r > \ell$, and at least one of the following cases must hold: (a) $lb < p < q \leq rb$ or (b) $lb \leq p < q < rb$. We prove the lemma for case (a); the other case follows similarly. By Definition 4.3.1, it follows that $\text{LCP}[k] \geq r > \ell$ for all k with $lb+1 \leq k \leq rb$. In particular, $\text{LCP}[p] > \ell$. This, however, contradicts the fact that $\text{LCP}[p] \leq \ell$. Consequently, m - $[p..q]$ is a child interval of ℓ - $[i..j]$.

Now suppose that $[p..q]$ is a singleton interval, i.e., $p = q$. Obviously, at least one of the indices p and $p+1$ must be an ℓ -index. That is, $\text{LCP}[p] = \ell$ or $\text{LCP}[p+1] = \ell$ (or both). One can show that there is no lcp-interval $[lb..rb]$ that is embedded in $[i..j]$ and encloses $[p..p]$ (the indirect proof is verbatim the same as above). Therefore, ℓ - $[i..j]$ is the smallest lcp-interval that contains $[p..p]$, that is, ℓ - $[i..j]$ is the parent interval of $[p..q]$. \square

As an example, we compute the child intervals of the lcp-interval 0 - $[1..9]$ of the LCP-array from Figure 4.13. The 0 -indices are (in ascending order) 5, 6, and 7. Thus, the child intervals of 0 - $[1..9]$ are $[1..4]$, $[5..5]$, $[6..6]$, and $[7..9]$.

Exercise 4.3.6 Implement a procedure that takes an lcp-interval as input and returns the list of its child intervals.

We employ two auxiliary arrays PSV_{LCP} and NSV_{LCP} to explain how the parent interval of an lcp-interval can be determined.

Definition 4.3.7 For any index $2 \leq i \leq n$, we define

$$\text{PSV}_{\text{LCP}}[i] = \max\{j \mid 1 \leq j < i \text{ and } \text{LCP}[j] < \text{LCP}[i]\}$$

and

$$\text{NSV}_{\text{LCP}}[i] = \min\{j \mid i < j \leq n+1 \text{ and } \text{LCP}[j] < \text{LCP}[i]\}$$

i	SA	LCP	$S_{SA[i]}$	PSV_{LCP}	NSV_{LCP}
1	3	-1	aaacatat		
2	4	2	aacatat	1	3
3	1	1	acaaacatat	1	7
4	5	3	acatat	3	5
5	9	1	at	1	7
6	7	2	atat	5	7
7	2	0	caaacatat	1	11
8	6	2	catat	7	9
9	10	0	t	1	11
10	8	1	tat	9	11
11		-1			

Figure 4.15: The enhanced suffix array of the string $S = acaaacatat$ with the arrays PSV_{LCP} and NSV_{LCP} .

PSV and NSV are acronyms for *previous smaller value* and *next smaller value*, respectively. Given the value $LCP[i]$ at index i , among all indices j so that j is smaller than i and $LCP[j]$ is smaller than $LCP[i]$, $PSV_{LCP}[i]$ is the largest index. Analogously, among all indices j so that j is larger than i and $LCP[j]$ is smaller than $LCP[i]$, $NSV_{LCP}[i] = j$ is the smallest index. Figure 4.15 shows the arrays PSV_{LCP} and NSV_{LCP} of the string $S = acaaacatat$.

In this section, we will omit the subscript LCP, i.e., we will write PSV instead of PSV_{LCP} and NSV instead of NSV_{LCP} .

Lemma 4.3.8 *Let $2 \leq k \leq n$ and $LCP[k] = \ell$. Then $[PSV[k]..NSV[k] - 1]$ is an lcp-interval of lcp-value ℓ .*

Proof We have

1. $LCP[PSV[k]] < \ell$ (by the definition of $PSV[k]$).
2. $LCP[m] \geq \ell$ for all m with $PSV[k] + 1 \leq m \leq NSV[k] - 1$.
3. $LCP[k] = \ell$ (note that $PSV[k] + 1 \leq k \leq NSV[k] - 1$).
4. $LCP[NSV[k]] < \ell$ (by the definition of $NSV[k]$).

Consequently, $[PSV[k]..NSV[k] - 1]$ is an ℓ -interval. \square

The following lemma explains how the parent interval $parent([i..j])$ of an lcp-interval $[i..j] \neq [1..n]$ can be determined with the help of the arrays LCP, PSV, and NSV.

Lemma 4.3.9 *Let $[i..j] \neq [1..n]$ be an lcp-interval ($[i..j]$ may be a singleton interval) with $\text{LCP}[i] = p$ and $\text{LCP}[j+1] = q$.*

- *If $p = q$, then*
 - *the parent interval of $[i..j]$ is the lcp-interval $[\text{PSV}[i]..\text{NSV}[i] - 1] = [\text{PSV}[j+1]..\text{NSV}[j+1] - 1]$,*
 - *the parent interval of $[i..j]$ has lcp-value $p = q$,*
 - *i and $j+1$ are consecutive p -indices of the parent interval of $[i..j]$.*
- *If $p > q$, then*
 - *the parent interval of $[i..j]$ is the lcp-interval $[\text{PSV}[i]..j]$,*
 - *the parent interval of $[i..j]$ has lcp-value p ,*
 - *i is the last p -index of the parent interval of $[i..j]$.*
- *If $p < q$, then*
 - *the parent interval of $[i..j]$ is the lcp-interval $[i..\text{NSV}[j+1] - 1]$.*
 - *the parent interval of $[i..j]$ has lcp-value q ,*
 - *$j+1$ is the first q -index of the parent interval of $[i..j]$.*

Proof We proceed by case analysis.

Case $p = q$: According to Lemma 4.3.8, $[\text{PSV}[i]..\text{NSV}[i] - 1]$ is an lcp-interval of lcp-value $p = q$. Clearly, i and $j+1$ are p -indices of that interval because $\text{LCP}[i] = p$ and $\text{LCP}[j+1] = p$. We claim that $\text{PSV}[i] = \text{PSV}[j+1]$ and $\text{NSV}[i] = \text{NSV}[j+1]$. This is certainly true if $[i..j]$ is a singleton interval. If $[i..j]$ is an lcp-interval of lcp-value ℓ , then $\text{LCP}[m] \geq \ell$ for all m with $i+1 \leq m \leq j$ and $\ell > p = q$ prove the claim. Let $i_1 < i_2 < \dots < i_k$ be the p -indices of the p -interval $[\text{PSV}[i]..\text{NSV}[i] - 1]$ in ascending order. Since i and $j+1$ are two consecutive p -indices, it follows that $i = i_r$ and $j+1 = i_{r+1}$ for some $1 \leq r < k$. By Lemma 4.3.5, $[i..j]$ is a child interval of the p -interval $[\text{PSV}[i]..\text{NSV}[i] - 1]$.

Case $p > q$: Again, by Lemma 4.3.8, $[\text{PSV}[i]..\text{NSV}[i] - 1]$ is an lcp-interval of lcp-value p . Obviously, i is a p -index of that interval, but $j+1$ is not. Because $q < p$, we have $\text{NSV}[i] = j+1$. Moreover, this implies that i is the last p -index of the p -interval $[\text{PSV}[i]..j]$. According to Lemma 4.3.5, $[i..j]$ is the last child interval of $[\text{PSV}[i]..j]$.

Case $p < q$: Similar to the previous case. □

As an example, consider Figure 4.15 and determine $\text{parent}([6..6])$. Since $p = \text{LCP}[6] = 2 > 0 = \text{LCP}[7] = q$, the second case of Lemma 4.3.9 applies, so that $\text{parent}([6..6]) = [\text{PSV}[6]..6] = [5..6]$. Furthermore, the lcp-interval $[5..6]$ has lcp-value 2, and 6 is the last (in fact, the only) 2-index of $[5..6]$. As another example, we search for parent interval of $[1..2]$. In this case $p =$

$\text{LCP}[1] = -1 < 1 = \text{LCP}[3] = q$, so that $\text{parent}([1..2]) = [1..\text{NSV}[2+1] - 1] = [1..6]$. Furthermore, the parent interval of $[1..2]$ has lcp-value 1 and 3 is its first 1-index.

Corollary 4.3.10 *Let $[i..j] \neq [1..n]$ be an lcp-interval ($[i..j]$ may be a singleton interval). The parent interval of $[i..j]$ has lcp-value $\max\{\text{LCP}[i], \text{LCP}[j+1]\}$.*

Proof This is a direct consequence of Lemma 4.3.9. \square

We have seen that child intervals can be determined with RMQs, while parent intervals can be determined with PSV and NSV values. As a matter of fact, it is also possible to determine the LCA of two lcp-intervals by means of RMQ, PSV, and NSV. However, this is left as an exercise for the reader because lowest common ancestors are not needed in the applications dealt with in this book.

Exercise 4.3.11 Give an algorithm in pseudo-code that takes two lcp-intervals $[i..j]$ and $[p..q]$ as input and returns their lowest common ancestor in the lcp-interval tree.

Hint: If $j < p$, then their LCA is the lcp-interval $[\text{PSV}[k]..\text{NSV}[k] - 1]$, where $k = \text{RMQ}(j+1, p)$.

In this chapter, we merely use the arrays PSV and NSV in proofs but not in algorithms. Nevertheless, we show here how to compute them in linear time. In the pseudo-code of Algorithm 4.5, the elements on the stack are pairs $\langle \text{idx}, \text{lcp} \rangle$, where $\text{lcp} = \text{LCP}[\text{idx}]$. The procedures *push* (pushes an element onto the stack) and *pop*() (pops an element from the stack and returns that element) are the usual stack operations, while *top*() provides a pointer to the topmost element of the stack. Moreover, $\text{top}().\text{idx}$ denotes the first component of the topmost element of the stack, while $\text{top}().\text{lcp}$ denotes the second component.

Initially, Algorithm 4.5 pushes the pair $\langle 1, -1 \rangle$ consisting of the first index and its lcp-value onto the stack, and sets $\text{PSV}[1]$ to \perp (so $\text{PSV}[1]$ does not exist). The following invariant is maintained in the for-loop of the algorithm: for every element e on the stack, $\text{PSV}[e.\text{idx}]$ is set correctly. In the while-loop, the algorithm tests whether the lcp-value of the current index k is strictly smaller than the lcp-value of the topmost element of the stack. If this is the case, then the next smaller lcp-value of the topmost element can be found at the current index k . Consequently, the assignment $\text{NSV}[\text{pop}().\text{idx}] \leftarrow k$ pops the topmost element from the stack and sets the corresponding NSV-entry to k . After the while-loop, one has $\text{LCP}[k] \geq \text{top}().\text{lcp}$. If the lcp-value of the topmost element of the stack is strictly smaller than that of the current index k , then the previous smaller lcp-value of the current index k is the index of the topmost element. Hence

Algorithm 4.5 Construction of the PSV and NSV arrays.

```

push( $\langle 1, -1 \rangle$ )      /* an element on the stack has the form  $\langle idx, lcp \rangle$  */
PSV[1]  $\leftarrow \perp$ 
for  $k \leftarrow 2$  to  $n + 1$  do
    while  $LCP[k] < top().lcp$  do
        NSV[pop().idx]  $\leftarrow k$ 
    if  $LCP[k] > top().lcp$  then
        PSV[k]  $\leftarrow top().idx$ 
    else
        PSV[k]  $\leftarrow PSV[top().idx]$ 
    push( $\langle k, LCP[k] \rangle$ )

```

the assignment $PSV[k] \leftarrow top().idx$ does the job. Otherwise, the equality $LCP[k] = top().lcp$ holds. In this case, the indices k and $top().idx$ have the same previous smaller lcp-value. By the loop-invariant, $PSV[top().idx]$ has been set correctly in a previous iteration of the for-loop. Therefore, $PSV[k] \leftarrow PSV[top().idx]$ assigns the correct value to $PSV[k]$. Finally, the pair $\langle k, LCP[k] \rangle$ is pushed onto the stack. Because $PSV[k]$ was set correctly, the loop-invariant also holds before the next execution of the for-loop.

4.3.2 Bottom-up traversal

In this section, we are going to describe a linear-time algorithm that traverses the lcp-interval tree in a bottom-up fashion with the help of a stack. We shall satisfy ourselves with the lcp-interval tree *without* singleton intervals. However, it is not difficult to modify the algorithm so that it also incorporates singleton intervals. To demonstrate the full capabilities of the method, we first show that the lcp-interval tree can be constructed in a bottom-up fashion. However, in applications we will not construct this tree *explicitly*. As we shall see, it is possible to traverse this tree without constructing it.

Pseudo-code for the bottom-up construction of the lcp-interval tree can be found in Algorithm 4.6. The elements on the stack are lcp-intervals represented by quadruples $\langle lcp, lb, rb, childList \rangle$, where lcp is the lcp-value of the interval, lb is its left boundary, rb is its right boundary, and $childList$ is a list of its child intervals. Furthermore, $add(list, c)$ appends the element c to the list $list$ and returns the result. Algorithm 4.6 traverses the lcp-interval tree by scanning the LCP-array from left to right (or, in many illustrations, from top to bottom). At each index k , the while-loop tests whether lcp-intervals on the stack end with the right boundary $k - 1$, and new lcp-intervals are detected in the penultimate if-statement.

Algorithm 4.6 Bottom-up traversal of the lcp-interval tree based on the LCP-array.

```

lastInterval  $\leftarrow \perp$ 
push( $\langle 0, 1, \perp, [ ] \rangle$ )
for  $k \leftarrow 2$  to  $n + 1$  do
     $lb \leftarrow k - 1$ 
    while  $LCP[k] < top().lcp$  do
         $top().rb \leftarrow k - 1$ 
         $lastInterval \leftarrow pop()$ 
        process( $lastInterval$ )
         $lb \leftarrow lastInterval.lb$ 
    if  $LCP[k] \leq top().lcp$  then
         $top().childList \leftarrow add(top().childList, lastInterval)$ 
         $lastInterval \leftarrow \perp$ 
    if  $LCP[k] > top().lcp$  then
        if  $lastInterval \neq \perp$  then
            push( $\langle LCP[k], lb, \perp, [lastInterval] \rangle$ )
             $lastInterval \leftarrow \perp$ 
        else push( $\langle LCP[k], lb, \perp, [ ] \rangle$ )

```

As an example, consider the execution of Algorithm 4.6 applied to the LCP-array of the string $S = ctaataatg$, shown in Figure 4.16. First, the 0-interval $[1..\perp]$ is pushed onto the stack. In the first iteration ($k = 2$) of the for-loop, the next lcp-interval $3-[1..\perp]$ is detected because $LCP[2] = 3 > 0 = top().lcp$. Consequently, it is pushed onto the stack; see Figure 4.17. In the next iteration ($k = 3$) the while-loop detects the end of this 3-interval because $LCP[3] = 1 < 3 = LCP[2]$. Thus, its right boundary $rb = k - 1 = 2$ is set, it is popped from the stack, and processed. Then, the if-statement inside the while-loop tests by $LCP[k] \leq top().lcp$ whether this 3-interval is a child of the lcp-interval $0-[1..\perp]$, which now lies on top of the stack. If so, it would be added to the child list of the topmost interval. Since $LCP[3] = 1 \not\leq 0 = top().lcp$, however, this is not the case. Thereafter, the while-loop is left and the lcp-interval $1-[1..\perp]$ is detected and pushed onto the stack. Because it is the parent interval of the “dangling” 3-interval, its child list must contain the interval $3-[1..2]$. The remaining part of the LCP-array is processed analogously.

i	1	2	3	4	5	6	7	8	9	10
$LCP[i]$	-1	3	1	2	0	0	0	4	1	-1

Figure 4.16: The LCP-array of the string $S = ctaataatg$; cf. Figure 4.13.

k	contents of the stack
	$\langle 0, 1, \perp, [] \rangle$
2	$\langle 3, 1, \perp, [] \rangle$ $\langle 0, 1, \perp, [] \rangle$
3	$\langle 1, 1, \perp, [\langle 3, 1, 2, [] \rangle] \rangle$ $\langle 0, 1, \perp, [] \rangle$
4	$\langle 2, 3, \perp, [] \rangle$ $\langle 1, 1, \perp, [\langle 3, 1, 2, [] \rangle] \rangle$ $\langle 0, 1, \perp, [] \rangle$
5	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [] \rangle, \langle 2, 3, 4, [] \rangle] \rangle] \rangle$
6	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [] \rangle, \langle 2, 3, 4, [] \rangle] \rangle] \rangle$
7	$\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [] \rangle, \langle 2, 3, 4, [] \rangle] \rangle] \rangle$
8	$\langle 4, 7, \perp, [] \rangle$ $\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [] \rangle, \langle 2, 3, 4, [] \rangle] \rangle] \rangle$
9	$\langle 1, 7, \perp, [\langle 4, 7, 8, [] \rangle] \rangle$ $\langle 0, 1, \perp, [\langle 1, 1, 4, [\langle 3, 1, 2, [] \rangle, \langle 2, 3, 4, [] \rangle] \rangle] \rangle$
10	

Figure 4.17: Contents of the stack during the run of Algorithm 4.6. $\langle 0, 1, 9, [\langle 1, 1, 4, [\langle 3, 1, 2, [] \rangle, \langle 2, 3, 4, [] \rangle] \rangle, \langle 1, 7, 9, [\langle 4, 7, 8, [] \rangle] \rangle] \rangle$ is the last interval that is processed (when $k = 10$). As a matter of fact, it is the whole lcp-interval tree corresponding to the LCP-array of Figure 4.16. The construction of the lcp-interval tree can be avoided by implementing the procedure *process* in Algorithm 4.6 accordingly: after *process* has processed *lastInterval* (the parameter of the procedure), the child list of *lastInterval* must be emptied of its contents by the assignment *lastInterval.childList* $\leftarrow []$; cf. Algorithm 4.7.

The correctness of Algorithm 4.6 is a direct consequence of Theorem 4.3.12.

Theorem 4.3.12 *Consider the for-loop of Algorithm 4.6 for some index k . Let top be the topmost interval on the stack and top_{-1} be the interval directly beneath it (note that $top_{-1}.lcp < top.lcp$). If $LCP[k] < top.lcp$, then before top will be popped from the stack in the while-loop, the following holds:*

1. *If $LCP[k] \leq top_{-1}.lcp$, then top is the child interval of top_{-1} .*
2. *If $LCP[k] > top_{-1}.lcp$, then top is the first child interval of the lcp-interval with lcp-value $LCP[k]$ that contains k . To be precise, top is the first child interval of $[top.lb..NSV[k] - 1]$.*

Proof (1) First, we show that top is embedded in top_{-1} . The following invariant is maintained in the for-loop of Algorithm 4.6: If $\langle \ell_1, lb_1, rb_1 \rangle, \dots, \langle \ell_m, lb_m, rb_m \rangle$ are the intervals on the stack, where $top = \langle \ell_m, lb_m, rb_m \rangle$, then $lb_i \leq lb_j$ and $\ell_i < \ell_j$ for all $1 \leq i < j \leq m$. Furthermore, because $\langle \ell_j, lb_j, rb_j \rangle$ will be popped from the stack before $\langle \ell_i, lb_i, rb_i \rangle$, it follows that $rb_j \leq rb_i$. Thus, the ℓ_j -interval $[lb_j..rb_j]$ is embedded in the ℓ_i -interval $[lb_i..rb_i]$. In particular, top is embedded in top_{-1} .

If top was not the child interval of top_{-1} , then there would be an lcp-interval $\langle lcp', lb', rb' \rangle$ so that top is embedded in $\langle lcp', lb', rb' \rangle$ and $\langle lcp', lb', rb' \rangle$ is embedded in top_{-1} . This, however, can only happen if $\langle lcp', lb', rb' \rangle$ is an interval on the stack that is above top_{-1} . This contradiction proves the claim.

(2) We have $LCP[top.lb] = top_{-1}.lcp < LCP[k] < top.lcp$ and $top.rb = k - 1$. By the third case of Lemma 4.3.9, it follows that (a) the parent interval of top is the lcp-interval $[top.lb..NSV[k] - 1]$, (b) the parent interval of top has lcp-value $\ell = LCP[k]$, and (c) k is the first ℓ -index of the parent interval of top . Thus, the lemma follows. \square

In Algorithm 4.6, the lcp-interval tree is traversed in a bottom-up fashion by a linear scan of the LCP-array, while storing information on a stack. Whenever an ℓ -interval is processed by the generic procedure *process*, only its child intervals have to be known. These are determined solely from the lcp-information, i.e., we do not need explicit parent-child pointers in our framework. It should be stressed that the algorithm exhibits strong locality of reference because of the sequential access to the LCP-array.

It is possible to solve several problems merely by specifying the procedure *process* in Algorithm 4.6; an example is given below. Other applications may require slight modifications of the algorithm; see Chapter 5.

Let us address the problem of finding all substrings of S having at least p and at most q occurrences in S , where $1 \leq p \leq q$. The goal is to give a linear-time algorithm that solves the problem. However, if $p = 1$ and

Algorithm 4.7 To find all substrings of S having at least p and at most q occurrences in S , where $2 \leq p \leq q$, plug this implementation of the procedure *process* in Algorithm 4.6.

```

process(lastInterval)
  for each  $\langle \ell, i, j, [] \rangle$  in lastInterval.childList do
    if  $p \leq (j - i + 1)$  and  $(j - i + 1) \leq q$  then
      output (lastInterval.lcp,  $\ell, [i..j]$ )
  lastInterval.childList  $\leftarrow []$  /* empty childList */

```

$q = n$, then the algorithm must output all substrings of S , and there are $O(n^2)$ substrings of S . In other words, a linear-time algorithm is impossible if every substring is output explicitly. For this reason, the algorithm must use an implicit representation of the output. Here, we will give a solution for the case $p \geq 2$. Exercise 4.3.15 asks you to solve the problem for the case $p = 1$. As in Algorithm 4.6, the lcp-interval tree of S is traversed in a bottom-up fashion. Suppose that the lcp-interval $m\text{-}[lb..rb]$ is going to be processed by the procedure *process*. At this point, all its child intervals are known. Let $\ell\text{-}[i..j]$ be one of those. Let the lcp-intervals $m\text{-}[lb..rb]$ and $\ell\text{-}[i..j]$ represent the strings u and ω , respectively; see Definition 4.3.1. Clearly, $\omega = uv$ for some string v of length $\ell - m$. The key observation is that every substring uv' , where v' is a non-empty prefix of v , occurs exactly $(j - i + 1)$ times in S . Thus, procedure *process* tests whether $p \leq (j - i + 1) \leq q$ is true. If so, it outputs $(m + 1, \ell, [i..j])$; meaning that every prefix of $\omega = S[\text{SA}[i]..\text{SA}[i] + \ell - 1]$ having a length in between $m + 1$ and ℓ occurs at least p times and at most q times in S , namely at the positions $\text{SA}[i], \dots, \text{SA}[j]$. Algorithm 4.7 implements this approach. Note that its last assignment $\text{lastInterval.childList} \leftarrow []$ empties the *childList* of *lastInterval*. This ensures that the lcp-interval tree is not constructed during the bottom-up traversal.

Exercise 4.3.13 Show that Algorithm 4.6 takes only linear time and space.

Exercise 4.3.14 Modify Algorithm 4.6 so that it also incorporates singleton intervals.

Exercise 4.3.15 Give a linear-time solution to the problem of finding all substrings of S having at most $q \geq 1$ occurrences in S .

Exercise 4.3.16 A string ω is called a *prefix tandem repeat* of string S if ω is a prefix of S and has the form uu for some string u . Give a linear-time algorithm to find the longest prefix tandem repeat of S .

Algorithm 4.8 *BuildTopDown*($[i..j]$) recursively constructs the subtree of the lcp-interval tree rooted at the lcp-interval $[i..j]$, using the LCP-array and RMQs thereon.

```

if  $i = j$  then return  $\langle \perp, i, i, [ ] \rangle$       /* singleton interval */
 $childList \leftarrow [ ]$ 
 $k \leftarrow i$ 
 $m \leftarrow \text{RMQ}(i + 1, j)$       /* first  $\ell$ -index of  $[i..j]$  */
 $\ell \leftarrow \text{LCP}[m]$ 
repeat
   $subtree \leftarrow \text{BuildTopDown}([k..m - 1])$ 
   $\text{add}(childList, subtree)$ 
   $k \leftarrow m$ 
if  $k = j$  then
  break
else
   $m \leftarrow \text{RMQ}(k + 1, j)$ 
until  $\text{LCP}[m] \neq \ell$ 
 $subtree \leftarrow \text{BuildTopDown}([k..j])$ 
 $\text{add}(childList, subtree)$ 
return  $\langle \ell, i, j, childList \rangle$ 

```

4.3.3 Top-down traversal

According to Lemma 4.3.5, determining the child intervals of an ℓ -interval $[i..j]$ boils down to finding the ℓ -indices of $[i..j]$ in ascending order. With range minimum queries (see Chapter 3) on the LCP-array this is easy: $\text{RMQ}(i + 1, j)$ yields the first ℓ -index i_1 , $\text{RMQ}(i_1 + 1, j)$ yields the second ℓ -index i_2 , etc.

We use this to construct the lcp-interval tree from the LCP-array in a top-down fashion. The pseudo-code of the procedure *BuildTopDown* can be found in Algorithm 4.8; it takes an lcp-interval $[i..j]$ as input and returns the subtree of the lcp-interval tree rooted at node $[i..j]$. Hence *BuildTopDown*($[1..n]$) yields the desired lcp-interval tree. As in Algorithm 4.6, nodes (i.e., lcp-intervals) in the lcp-interval tree are represented by quadruples $\langle lcp, lb, rb, childList \rangle$, where *lcp* is the lcp-value of the interval (this value is \perp in singleton intervals), *lb* is its left boundary, *rb* is its right boundary, and *childList* is the list of its child intervals.

Let us have a closer look at Algorithm 4.8. The first line contains the base case of the recursion: If $i = j$, then $[i..j]$ is a singleton interval, and the lcp-interval tree rooted at node $[i..j]$ consists solely of the node $\langle \perp, i, i, [] \rangle$. Otherwise, $i < j$ and the lcp-interval $[i..j]$ is not a singleton interval. Its child list is initialized to the empty list and k is set to the left boundary of the lcp-interval $[i..j]$. Furthermore, m is set to the first lcp-

index of the lcp-interval $[i..j]$ (note that every lcp-interval has at least one lcp-index and this can be obtained by the range minimum query $\text{RMQ}(i + 1, j)$) and ℓ is set to the lcp-value of $[i..j]$. When the repeat-until-loop is entered, the interval $[k..m - 1]$ is the first child interval of $[i..j]$ by Lemma 4.3.5. Consequently, *BuildTopDown* is called recursively with this child interval and it returns the subtree of the lcp-interval tree rooted at node $[k..m - 1]$. This subtree is added to the child list. Thereafter, the current ℓ -index is stored in variable k . The loop will be executed as long as $k < j$ and $\text{LCP}[\text{RMQ}(k + 1, j)] = \ell$, i.e., it will be executed as long as $[k..j]$ is not a singleton interval and there is another ℓ -index of the lcp-interval $[i..j]$, namely the index $m = \text{RMQ}(k + 1, j)$. In this case, *BuildTopDown* is called recursively with the child interval $[k..m - 1]$ (cf. Lemma 4.3.5), the returned subtree is added to the child list, and k is set to the current ℓ -index m .

After the loop is done, there are two possibilities.

- $k = j$: In this case, the recursive call *BuildTopDown* ($[k..j]$) yields the subtree consisting of one node, viz. the singleton interval $[k..j]$, and this subtree is added to the child list.
- $k < j$ and $\text{LCP}[\text{RMQ}(k + 1, j)] \neq \ell$: In this case, k is the last ℓ -index of the lcp-interval $[i..j]$ and, by Lemma 4.3.5, $[k..j]$ is the last child interval of the lcp-interval $[i..j]$. Thus, *BuildTopDown* is called recursively with this child interval and the returned subtree is added to the child list.

Finally, Algorithm 4.8 returns the lcp-interval tree rooted at the lcp-interval $[i..j]$ in form of the quadruple $\langle \ell, i, j, \text{childList} \rangle$.

Exercise 4.3.17 Show that Algorithm 4.8 takes only linear time and space. Modify the algorithm so that

- it returns the lcp-interval tree rooted at node $[i..j]$ without singleton intervals,
- it returns the list of all child intervals of $[i..j]$ instead of the lcp-interval tree rooted at node $[i..j]$.

We stress that in applications it is not necessary to actually construct the lcp-interval tree of a string. Slight modifications to Algorithm 4.8 suffice to obtain algorithms that traverse the lcp-interval tree in a top-down fashion without constructing it. Below, we provide two applications. The first one uses a *depth-first* traversal (similar to Algorithm 4.8), while the second one uses a *breadth-first* traversal of the lcp-interval tree.

In our first application, for each non-singleton lcp-interval ℓ - $[i..j]$ we wish to compute a value $\text{val}([i..j])$ defined as follows: For a non-empty string ω , let $\text{occ}_\omega(S)$ denote the number of occurrences of ω in S . Let u

i	SA	LCP	$S_{SA[i]}$	VAL
1	3	-1	<i>aaacatat</i>	
2	4	2	<i>aacatat</i>	8
3	1	1	<i>acaaacatat</i>	6
4	5	3	<i>acatat</i>	10
5	9	1	<i>at</i>	6
6	7	2	<i>atat</i>	8
7	2	0	<i>caaacatat</i>	0
8	6	2	<i>catat</i>	4
9	10	0	<i>t</i>	0
10	8	1	<i>tat</i>	2
11		-1		

Figure 4.18: The enhanced suffix array of $S = acaaacatat$ with VAL array.

be the string that is represented by the lcp-interval $\ell\text{-}[i..j]$ (i.e., u is the longest common prefix of the suffixes $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$), and define

$$val([i..j]) = \sum_{\omega \sqsubset u} occ_{\omega}(S)$$

where $\omega \sqsubset u$ means that ω is a non-empty prefix of u . In words, $val([i..j])$ is the number of all occurrences of all non-empty prefixes of u in S . In Section 5.7.2, the importance of these values will become clear. As an example, consider the lcp-interval $3\text{-}[3..4]$ in Figure 4.18. This lcp-interval represents the string $u = aca$. The prefixes a , ac , and aca of u have 6, 2, and 2 occurrences in S . Hence $val([3..4]) = 10$.

Our algorithm is based on the following lemma.

Lemma 4.3.18 *Let $q\text{-}[lb..rb]$ be a child interval of the lcp-interval $\ell\text{-}[i..j]$. Then*

$$val([lb..rb]) = val([i..j]) + (rb - lb + 1)(q - \ell).$$

Proof Let u be the string that is represented by $[i..j]$. This implies that $[lb..rb]$ represents a string uv , where $v \neq \varepsilon$. Let ω be a substring of S so that $\omega \sqsubset uv$ but $\omega \not\sqsubset u$. The key observation is that the ω -interval coincides with the uv -interval. In other words, ω occurs as often in S as uv does, namely

$(rb - lb + 1)$ times. Thus,

$$\begin{aligned}
val([lb..rb]) &= \sum_{\omega \sqsubseteq uv} occ_{\omega}(S) \\
&= \sum_{\omega \sqsubseteq u} occ_{\omega}(S) + \sum_{\omega \sqsubseteq uv, \omega \not\sqsubseteq u} occ_{\omega}(S) \\
&= val([i..j]) + \sum_{\omega \sqsubseteq uv, \omega \not\sqsubseteq u} (rb - lb + 1) \\
&= val([i..j]) + (rb - lb + 1) \sum_{\omega \sqsubseteq uv, \omega \not\sqsubseteq u} 1 \\
&= val([i..j]) + (rb - lb + 1) (q - \ell)
\end{aligned}$$

□

Again, consider the lcp-interval $3\text{--}[3..4]$ and its parent interval $1\text{--}[1..6]$; see Figure 4.18. We have $val([1..6]) = 6$ because a occurs six times in S . According to the previous lemma,

$$val([3..4]) = val([1..6]) + (4 - 3 + 1) (3 - 1) = 6 + 2 \cdot 2 = 10$$

To have constant-time access to the values, we store them in an additional array VAL. For an lcp-interval $\ell\text{--}[i..j]$, the value $val([i..j])$ can be stored at several locations. Among the options are

1. the first ℓ -index of $[i..j]$,
2. all ℓ -indices of $[i..j]$,
3. the home index of $[i..j]$, defined by

$$home([i..j]) = \begin{cases} i & \text{if } LCP[i] \geq LCP[j + 1] \\ j & \text{otherwise} \end{cases}$$

In what follows, we will use the second possibility; see Figure 4.18 for an example. The uniqueness of the alternative location $home([i..j])$ is due to Strothmann [302]; cf. Exercise 4.3.19.

The procedure $ValTopDown(\ell\text{--}[i..j], idx, val)$ of Algorithm 4.9 takes an lcp-interval $[i..j]$ of lcp-value ℓ , its first ℓ -index idx , and $val = val([i..j])$ as input and recursively computes the VAL array of the lcp-interval tree rooted at $[i..j]$. The lcp-value ℓ and the first lcp-index idx of the lcp-interval $[i..j]$ are supplied as parameters to the procedure because this avoids superfluous recomputations of these values. In order to get the whole VAL array, the procedure is called with the root interval $0\text{--}[1..n]$, its first 0-index $RMQ(2, n)$ and $val = 0$. In Algorithm 4.9, the value $val([lb..rb])$ of a child interval $q\text{--}[lb..rb]$ of $\ell\text{--}[i..j]$ is computed by a generic function *computeValue*. Here,

Algorithm 4.9 $ValTopDown(\ell-[i..j], idx, val)$ recursively computes the VAL array of the lcp-interval tree rooted at the lcp-interval $\ell-[i..j]$, where idx is the first ℓ -index of $[i..j]$ and $val = val([i..j])$. It uses the LCP-array and RMQs thereon.

```

 $k \leftarrow i$           /*  $k$  stores the left boundary of the current child interval */
 $m \leftarrow idx$       /*  $m$  stores the current  $\ell$ -index */
repeat
   $VAL[m] \leftarrow val$ 
  if  $k \neq m - 1$  then    /*  $[k..m - 1]$  is a non-singleton child of  $[i..j]$  */
     $childIdx \leftarrow RMQ(k + 1, m - 1)$  /* first lcp-index of  $[k..m - 1]$  */
     $q \leftarrow LCP[childIdx]$  /*  $q$  is the lcp-value of  $[k..m - 1]$  */
     $childVal \leftarrow computeValue(\ell, val, q, k, m - 1)$ 
     $ValTopDown(q-[k..m - 1], childIdx, childVal)$ 
     $k \leftarrow m$  /*  $k$  is left boundary of the next child interval */
  if  $k = j$  then
    return /* there is no more non-singleton child interval */
  else
     $m \leftarrow RMQ(k + 1, j)$  /*  $m$  is the next  $\ell$ -index unless  $LCP[m] \neq \ell$  */
until  $LCP[m] \neq \ell$ 
/*  $[k..j]$  is the last non-singleton child interval of  $[i..j]$  */
/* and  $m$  is the first lcp-index of  $[k..j]$  */
 $q \leftarrow LCP[m]$  /*  $q$  is the lcp-value of  $[k..j]$  */
 $childVal \leftarrow computeValue(\ell, val, q, k, j)$ 
 $ValTopDown(q-[k..j], m, childVal)$ 

```

$\text{computeValue}(\ell, val, q, lb, rb) = val + (rb - lb + 1)(q - \ell)$ by Lemma 4.3.18. We shall see in Exercise 4.3.20 and in Section 5.7.2 that it is sufficient to modify computeValue to solve related problems.

We will briefly explain Algorithm 4.9. As we have seen in Lemma 4.3.5, the child intervals of $[i..j]$ are $[i..i_1 - 1]$, $[i_1..i_2 - 1]$, \dots , $[i_k..j]$, where $i_1 < i_2 < \dots < i_k$ are the ℓ -indices of $[i..j]$. In Algorithm 4.9, the variables k and m store the left boundary of the current child interval and the current ℓ -index, respectively. Initially, k is set to the left boundary of the interval $[i..j]$ and m is set to the first ℓ -index idx of $[i..j]$. Hence the first child interval is $[k..m - 1]$. The body of the repeat-until-loop stores val in the VAL array at the current ℓ -index m and then deals with the current child interval $[k..m - 1]$ provided it is a non-singleton. In this case, the first lcp-index $childIdx$ of $[k..m - 1]$ is determined by the range minimum query $\text{RMQ}(k + 1, m - 1)$. Therefore, $q = \text{LCP}[childIdx]$ is the lcp-value of $[k..m - 1]$. According to Lemma 4.3.18, $childVal = val([k..m - 1])$ is computed by $\text{computeValue}(\ell, val, q, lb, rb) = val + (rb - lb + 1)(q - \ell)$. The computation proceeds recursively with the procedure call $\text{ValTopDown}(q - [k..m - 1], childIdx, childVal)$. Subsequently, k becomes the left boundary of the next child interval, which is m , and the next ℓ -index of $[i..j]$ must be determined. If $k = j$, then certainly there is no more ℓ -index and the last child interval of $[i..j]$ is the singleton interval $[j..j]$. In this case, the repeat-until-loop is left and the procedure terminates. Otherwise, m is set to $\text{RMQ}(k + 1, j)$. Now there are two possibilities: Either $\text{LCP}[m] = \ell$, in which case m is the next ℓ -index of $[i..j]$, or $\text{LCP}[m] \neq \ell$, in which case $[k..j]$ is the last (non-singleton) child interval of $[i..j]$ and m is the first lcp-index of $[k..j]$. In the first case, the loop is repeated, i.e., the next iteration of the loop sets $\text{VAL}[m]$ to val and deals with the next child interval $[k..m - 1]$. In the second case, Algorithm 4.9 deals with the last child interval $[k..j]$ of $[i..j]$ as with the previous child intervals. Figure 4.18 depicts the VAL array of our example.

Exercise 4.3.19 For an lcp-interval $[i..j]$, define

$$\text{home}([i..j]) = \begin{cases} i & \text{if } \text{LCP}[i] \geq \text{LCP}[j + 1] \\ j & \text{otherwise} \end{cases}$$

to be the *home index* of $[i..j]$. Prove that for any two lcp-intervals $[i..j]$ and $[p..q]$, the equality $\text{home}([i..j]) = \text{home}([p..q])$ implies $[i..j] = [p..q]$.

Exercise 4.3.20 Modify Lemma 4.3.18 and the function computeValue in such a way that Algorithm 4.9 computes

$$val([lb..rb]) = \sum_{\omega \sqsubseteq uv} |\omega| \cdot \text{occ}_\omega(S)$$

where uv is the string represented by the lcp-interval $[lb..rb]$.

As a second application of the top-down traversal, we will briefly describe how to find all shortest unique substrings. This is relevant in the design of primers for DNA sequences; for details see Section 5.6.5.

Definition 4.3.21 A substring $S[i..j]$ of S is *unique* if it occurs exactly once in S . The *shortest unique substring problem* is to find all shortest unique substrings of S .

For example, ca is the shortest unique substring of $acac$. If S consists solely of a 's, i.e., $S = a^n$, then the shortest unique substring is S itself. In the following, we assume that S contains at least two different characters.

It is readily verified that a substring u of S is unique if and only if it is prefix of exactly one suffix of S , say of $S_{SA[k]}$. In terms of lcp-intervals, this can be rephrased as: u is unique if and only if there is exactly one suffix $S_{SA[k]}$ so that u is prefix of $S_{SA[k]}$ but not of $S[SA[k]..SA[k] + \ell - 1]$, where $\ell - [i..j]$ is the parent interval of the singleton interval $[k..k]$. Moreover, the length of all the shortest unique substrings of S is $m + 1$, where m is the smallest lcp-value of all lcp-intervals having a singleton child interval. Using this observation, the shortest unique substring problem can be solved by a breadth-first traversal of the lcp-interval tree, using a queue. Initially, the queue contains only the root interval $0-[1..n]$. During the traversal, more lcp-intervals may be added to the queue. Besides the queue, the algorithm maintains a set M that contains all the shortest unique substrings detected so far (represented by their start position in S) and a variable min that stores their length. Initially, M is empty and $min = \infty$.

Suppose that $\ell - [i..j]$ is removed from the front of the queue, i.e., it is the lcp-interval that is processed next during the breadth-first traversal. The algorithm computes all its child intervals. If a singleton child interval $[k..k]$ of $[i..j]$ is detected, then the length $\ell + 1$ prefix of $S_{SA[k]}$ is a unique substring of S . Thus, if M is empty or $min > \ell + 1$, then M is set to $\{SA[k]\}$ and min is set to $\ell + 1$. If M is not empty and $min = \ell + 1$, then $SA[k]$ is added to M . Otherwise, M and min remain unchanged. If $\ell - [i..j]$ has no singleton child interval, then every child interval $q-[lb..rb]$ of $\ell - [i..j]$ satisfying $q + 1 \leq min$ is added to the back of the queue. Then, the algorithm proceeds with the next lcp-interval at the front of the queue, as described above, until the queue is empty. Finally, the algorithm outputs min and M . It is not difficult to see that the algorithm takes time proportional to the number of processed lcp-intervals. In the worst case, this is $O(n)$. However, in practice only a small number of lcp-intervals is processed.

Exercise 4.3.22 Given a string S on an alphabet Σ , a string $\omega \in \Sigma^+$ is called an *absent word* if it is not a substring of S . Develop an algorithm that takes a string S as input, and outputs all shortest absent words (suppose that the alphabet Σ consists of the characters appearing in S).

4.3.4 Finding child intervals without RMQs

In the top-down traversal of the lcp-interval tree we actually do not need constant-time range minimum queries. To see this, we first introduce Super-Cartesian trees, a slight variation of canonical Cartesian trees. We advise the reader to consult Section 3.2, where the definition and construction of canonical Cartesian trees is explained in detail.

Definition 4.3.23 Let $A[l..r]$ be an array of integers.⁶ The *Super-Cartesian tree* $\mathcal{C}^{sup}(A[l..r])$ of $A[l..r]$ is recursively constructed as follows:

- If $l > r$, then $\mathcal{C}^{sup}(A[l..r])$ is the empty tree.
- If $l \leq r$, then the minima of $A[l..r]$ appear at positions $p_1 < p_2 < \dots < p_k$ for some $k \geq 1$. In this case, create k nodes v_1, v_2, \dots, v_k and label each v_j with p_j . Node v_1 is the root of $\mathcal{C}^{sup}(A[l..r])$. For each j with $1 < j \leq k$, the node v_j is the right sibling of node v_{j-1} (in illustrations like Figure 4.20, node v_{j-1} is connected with v_j by a *horizontal edge*). Recursively construct $\mathcal{C}_1 = \mathcal{C}^{sup}(A[l..p_1-1])$, $\mathcal{C}_2 = \mathcal{C}^{sup}(A[p_1+1..p_2-1])$, \dots , $\mathcal{C}_{k+1} = \mathcal{C}^{sup}(A[p_k+1..r])$. For each j with $1 \leq j < k$, the left child of v_j is the root of \mathcal{C}_j . The left and right children of v_k are the roots of \mathcal{C}_k and \mathcal{C}_{k+1} , respectively.

We would like to emphasize that a node in a Super-Cartesian tree has either a right sibling or a right child but not both.

As an example, consider the LCP-array of the string $S = acaaacatat$ in Figure 4.19. The minima of the array $\text{LCP}[1..11]$ occur at positions $p_1 = 1$ and $p_2 = 11$. According to Definition 4.3.23, we must create two nodes v_1 and v_2 so that v_2 is the right sibling of v_1 . Furthermore, one must construct $\mathcal{C}_1 = \mathcal{C}^{sup}(\text{LCP}[1..0])$, $\mathcal{C}_2 = \mathcal{C}^{sup}(\text{LCP}[2..10])$, and $\mathcal{C}_3 = \mathcal{C}^{sup}(\text{LCP}[12..11])$ recursively. Because \mathcal{C}_1 and \mathcal{C}_3 are empty, it remains to construct \mathcal{C}_2 , the Super-Cartesian tree of $\text{LCP}[2..10]$. By Definition 4.3.23, \mathcal{C}_2 becomes the left child of node v_2 . Its construction is left to the reader (note that the minima of the array $\text{LCP}[2..10]$ occur at positions 7 and 9). The Super-Cartesian tree of the whole LCP-array is depicted in Figure 4.20.

Similar to the canonical Cartesian tree, the Super-Cartesian tree $\mathcal{C}^{sup}(A)$ of an array A can be build incrementally, i.e., for every i with $1 \leq i \leq n-1$, we build $\mathcal{C}^{sup}(A[1..i+1])$ from $\mathcal{C}^{sup}(A[1..i])$. The linear-time construction works by climbing up the rightmost path in $\mathcal{C}^{sup}(A[1..i])$. For the convenience of the reader, we describe the construction algorithm in full detail. However, we do not illustrate the algorithm with examples. For a better understanding, readers may wish to consult Section 3.2, where the incremental construction of a canonical Cartesian tree is explained in detail.

⁶The array elements are integers, i.e., elements of the totally ordered set \mathbb{Z} . However, one can use any other totally ordered set instead of the integers.

			CLD		
i	SA	LCP	L	R	$S_{SA[i]}$
1	3	-1		11	<i>aaacatat</i>
2	4	2			<i>aacatat</i>
3	1	1	2 ↗	5	<i>acaaacatat</i>
4	5	3		↗ 4	<i>acatat</i>
5	9	1	4 ↗	6	<i>at</i>
6	7	2		↗ 3	<i>atat</i>
7	2	0	3 ↗	9	<i>caaacatat</i>
8	6	2		↗ 8	<i>catat</i>
9	10	0	8 ↗	10	<i>t</i>
10	8	1		↗ 7	<i>tat</i>
11		-1	7 ↗		

Figure 4.19: The enhanced suffix array of the string $S = acaaacatat$.

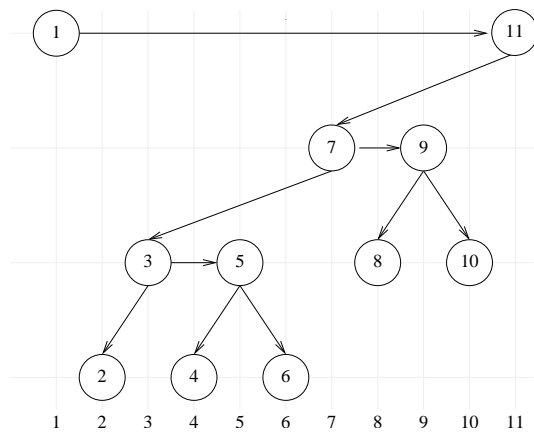


Figure 4.20: The Super-Cartesian tree of the LCP-array from Figure 4.19.

Definition 4.3.24 The *rightmost path* in a Super-Cartesian tree is obtained by starting at the root of the tree and following right sibling/right child pointers. The path ends at the first node that has no right sibling/right child.

Initially, the Super-Cartesian tree of $A[1..1]$ consists just of a root labeled with position 1. Let v_1, \dots, v_k be the nodes on the rightmost path in $\mathcal{C}^{sup}(A[1..i])$ and let p_1, \dots, p_k be their labels. It follows from the definition of the Super-Cartesian tree that the node with label $i + 1$ must be at the end of the rightmost path in $\mathcal{C}^{sup}(A[1..i + 1])$. Therefore, we climb up the rightmost path in $\mathcal{C}^{sup}(A[1..i])$ until we find the position where $i + 1$ belongs. More precisely, starting with $m = k$, we decrease m by 1 as long as $A[p_m] > A[i + 1]$ holds. Then we proceed by case analysis.

1. If $m = k$, i.e., $A[p_k] \leq A[i + 1]$, then:
 - If $A[p_k] = A[i + 1]$, then a new node w with label $i + 1$ becomes the right sibling of v_k .
 - If $A[p_k] < A[i + 1]$, then a new node w with label $i + 1$ becomes the right child of v_k .
2. If $m = 0$, i.e., $A[p_1] > A[i + 1]$, then a new node w with label $i + 1$ becomes the root of the tree and $\mathcal{C}^{sup}(A[1..i])$ becomes its left child.
3. If $1 \leq m \leq k - 1$, then m is the index so that $A[p_m] \leq A[i + 1]$ and $A[p_{m'}] > A[i + 1]$ for all $m < m' \leq k$.
 - If $A[p_m] = A[i + 1]$, then a new node w with label $i + 1$ becomes the right sibling of v_m and v_{m+1} becomes the left child of w .
 - If $A[p_m] < A[i + 1]$, then a new node w with label $i + 1$ becomes the right child of v_m and v_{m+1} becomes the left child of w .

The worst-case time complexity of the algorithm is obtained by the following amortized analysis. In each step exactly one node (the new node with label $i + 1$) is added to the rightmost path. Clearly, this implies that each node can leave the rightmost path only once. Moreover, whenever the value of the variable m is decreased, a node (the node v_m) leaves the rightmost path. Thus, the overall construction time of $\mathcal{C}^{sup}(A)$ is $O(n)$.

To enable a top-down traversal of the lcp-interval tree without range minimum queries, we store the Super-Cartesian tree in an additional table CLD, which we call *child table* because it can be used to determine child intervals in constant time. For didactic purposes, we will first store the child table CLD in two arrays CLD.L and CLD.R. We shall see in a moment, however, that one array suffices. By definition, a node in a Super-Cartesian tree has either a right sibling or a right child but not both. Therefore, for each node i , we store its left child in $\text{CLD}[i].L$ and its

right sibling/right child in $\text{CLD}[i].R$. For example, the child table CLD of the string $S = \text{acaaacatat}$ is depicted in Figure 4.19.

According to Lemma 4.3.5, finding all child intervals of an lcp-interval $\ell\text{-}[i..j]$ boils down to finding all ℓ -indices of that interval. The following theorem shows where the first ℓ -index of an lcp-interval $\ell\text{-}[i..j]$ can be found in the child table.

Theorem 4.3.25 *For every lcp-interval $\ell\text{-}[i..j]$ the following holds.*

1. *If $\text{LCP}[i] \leq \text{LCP}[j + 1]$, then $\text{CLD}[j + 1].L$ stores the first ℓ -index of the lcp-interval $[i..j]$.*
2. *If $\text{LCP}[i] > \text{LCP}[j + 1]$, then $\text{CLD}[i].R$ stores the first ℓ -index of the lcp-interval $[i..j]$.*

Proof (1a) If $\text{LCP}[i] = p = \text{LCP}[j + 1]$, then according to Lemma 4.3.9 the parent interval of $[i..j]$ is the p -interval $[\text{PSV}[i].. \text{NSV}[i] - 1]$, and i and $j + 1$ are consecutive p -indices of that interval. This implies that $j + 1$ is the right sibling of i in the Super-Cartesian tree of the LCP-array (hence $\text{CLD}[i].R = j + 1$). By definition, the left child of node $j + 1$ is the root of the Super-Cartesian tree of the array $\text{LCP}[i + 1..j]$. This root node (or equivalently $\text{CLD}[j + 1].L$) contains the first index of the minimum element in $\text{LCP}[i + 1..j]$. Hence it contains the first ℓ -index of $[i..j]$.

(1b) If $\text{LCP}[i] = p < q = \text{LCP}[j + 1]$, then Lemma 4.3.9 implies that the parent interval of $[i..j]$ is the q -interval $[i.. \text{NSV}[j + 1] - 1]$, and $j + 1$ is the first q -index of that interval. By definition, the left child of node $j + 1$ is the root of the Super-Cartesian tree of $\text{LCP}[i + 1..j]$. This root node (or equivalently $\text{CLD}[j + 1].L$) contains the first index of the minimum element in $\text{LCP}[i + 1..j]$, i.e., the first ℓ -index of $[i..j]$.

(2) If $\text{LCP}[i] = p > q = \text{LCP}[j + 1]$, then by Lemma 4.3.9 the parent interval of $[i..j]$ is the p -interval $[\text{PSV}[i]..j]$ and i is the last p -index of that interval. This implies that i has no right sibling in the Super-Cartesian tree of the LCP-array, and $\text{CLD}[i].R$ stores the right child of node i . According to the definition of the Super-Cartesian tree of the LCP-array, this right child of node i is the root of the Super-Cartesian tree of the array $\text{LCP}[i + 1..j]$, and this root node contains the first index of the minimum element in $\text{LCP}[i + 1..j]$. Thus, it contains the first ℓ -index of $[i..j]$. \square

Now we have all the ingredients to realize a top-down traversal of the lcp-interval tree without range minimum queries. Theorem 4.3.25 tells us where the first ℓ -index, say i_1 , of $[i..j]$ can be found. Using the child table, we find the second ℓ -index i_2 by $i_2 = \text{CLD}[i_1].R$, the third ℓ -index i_3 by $i_3 = \text{CLD}[i_2].R$, and so on. The index i_k is the last ℓ -index if $\text{LCP}[i_{k+1}] \neq \ell$. With this knowledge, the child intervals of $\ell\text{-}[i..j]$ can be determined straightforwardly according to Lemma 4.3.5. Algorithm 4.10 implements

Algorithm 4.10 Procedure *getChildIntervals*($[i..j]$) returns the list of all child intervals of the lcp-interval $[i..j]$. It is based on the LCP-array and the child table CLD.

```

list = [ ]
k ← i
if LCP[i] ≤ LCP[j + 1] then
    m ← CLD[j + 1].L
else m ← CLD[i].R
ℓ ← LCP[m]
repeat
    add(list, [k..m - 1])
    k ← m
    m ← CLD[m].R
until m = ⊥ or LCP[m] ≠ ℓ
add(list, [k..j])
return list

```

this approach. The procedure *getChildIntervals* applied to an lcp-interval $[i..j]$ returns the list of all child intervals of $[i..j]$.

Of course, the Super-Cartesian tree is only conceptual, i.e., we can construct the child table without it; see Algorithm 4.11.

Exercise 4.3.26 Apply Algorithm 4.11 to the example from Figure 4.19 and explain how it works.

Algorithm 4.11 Construction of the child table, based on the LCP-array.

```

CLD[1].R ← n + 1
push(⟨1, -1⟩)      /* an element on the stack has the form ⟨idx, lcp⟩ */
for k ← 2 to n + 1 do
    while LCP[k] < top().lcp do
        last ← pop()
        while top().lcp = last.lcp do
            CLD[top().idx].R ← last.idx
            last ← pop()
        if LCP[k] < top().lcp then
            CLD[top().idx].R ← last.idx
        else CLD[k].L ← last.idx
    push(⟨k, LCP[k]⟩)

```

To reduce the space requirement of the child table, only one array is used in practice. As a matter of fact, the memory cells of $\text{CLD}[i].R$ that

are unused can store the values of the $\text{CLD}.L$ array. To see this, note that $\text{CLD}[i+1].L \neq \perp$ if and only if $\text{LCP}[i] > \text{LCP}[i+1]$. In this case, however, we have $\text{CLD}[i].R = \perp$. In other words, $\text{CLD}[i].R$ is empty and can store the value $\text{CLD}[i+1].L$; see Figure 4.19. Finally, for a given index i , one can decide whether $\text{CLD}[i].R$ contains the value $\text{CLD}[i+1].L$ by testing whether $\text{LCP}[i] > \text{LCP}[i+1]$. To sum up, although the child table conceptually uses two arrays, only space for one array is actually required.

4.4 Suffix trees

Suffix trees were born long before suffix arrays entered the world. For constant-size alphabets, Weiner [330] devised the first linear-time suffix tree construction algorithm, and a few years later McCreight [218] gave a more space efficient algorithm to build suffix trees in linear time. These papers have a reputation for being notoriously complicated. Later Ukkonen [315] developed an *on-line* algorithm that processes the input text incrementally from left to right, and at each stage it has a suffix tree for the part of the text that has been processed so far. Although Ukkonen's algorithm allows a much simpler explanation, many students still have difficulties in fully understanding it. Giegerich and Kurtz [123] showed that the three algorithms are in fact more closely related than one would expect at first sight. For integer alphabets, Farach-Colton [93, 94] developed a (complex) divide and conquer suffix tree construction algorithm. This linear-time algorithm provided the basis of the linear-time SACA of Kim et al. [180], and it most likely also inspired the work of Kärkkäinen & Sanders [175] and Ko & Aluru [184].

Once constructed, the suffix tree can be used to efficiently solve a “myriad” of string processing problems [15], and Gusfield devotes about 70 pages of his book [139] to applications of suffix trees. It is no exaggeration to say that the suffix tree is one of the most important data structures in string processing.

While suffix trees play a prominent role in algorithmics, they are not as widespread in actual implementations of software tools as one would expect. There are two major reasons for this:

- (i) Although being asymptotically linear, the space consumption of typical implementations of suffix trees is quite large; see e.g. [193].
- (ii) In most applications, the suffix tree suffers from poor locality of memory reference, which causes a significant loss of efficiency on cached processor architectures.

As we shall see, the suffix tree of a string S coincides with the lcp-interval tree of S . The main difference is that the lcp-interval tree is

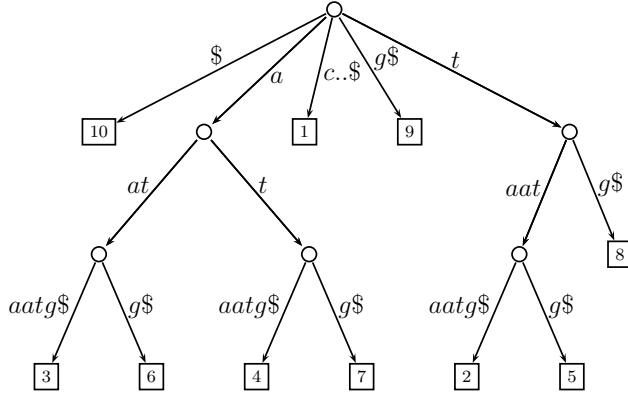


Figure 4.21: The suffix tree for $S\$ = ctaataatg\$$.

only conceptual (i.e., it is not really built) but it allows us to simulate all manner of suffix tree traversals very efficiently. Moreover, algorithms on lcp-interval trees are easier to implement because they are based on arrays and not on real tree structures. On the other hand, in some applications several arrays are required and thus the memory requirement for large strings (like genomes) is also high. Space issues will be discussed in Chapter 6.

Definition 4.4.1 Let S be a string of length n . A *suffix tree* for $S\$$ is a rooted tree ST with the following properties:

- (1) ST has exactly $n + 1$ leaves numbered 1 to $n + 1$.
- (2) Each internal node of ST is *branching*, i.e., it has at least two children.
- (3) Each edge of ST is labeled with a non-empty substring of $S\$$.
- (4) For each node α in ST and each $a \in \Sigma$, there is only one a -edge $\alpha \xrightarrow{av} \beta$ for some string v and some node β in ST . In other words, no two edges out of a node can have edge labels beginning with the same character.
- (5) The key feature of ST is that for any leaf i , the concatenation of the edge labels on the path from the root to leaf i exactly spells out the string $S[i..n]\$$, the i -th suffix of the string $S\$$.

Figure 4.21 shows the suffix tree for $S\$ = ctaataatg\$$.

The reader might wonder why we append the sentinel character $\$$ to S . If we did not do so, and a suffix s of S is a proper prefix of another suffix

of S , then s would be represented by an internal node and not by a leaf. By contrast, because the sentinel character does not occur in S , no suffix s of $S\$$ can be a proper prefix of another suffix of $S\$$. Hence every suffix of $S\$$ is represented by a leaf.

Let $\mathcal{L}(\alpha)$ denote the concatenation of the edge labels on the path from the root of ST to node α . Because of the property (4), node α can be uniquely identified with $\mathcal{L}(\alpha)$. Therefore, we denote α by \overline{w} if and only if $\mathcal{L}(\alpha) = w$ (the root node is denoted by $\overline{\epsilon}$). A string w occurs in ST if ST contains a node \overline{wu} for some (possibly empty) string u . In other words, w occurs in ST if it can be extended by some (possibly empty) string u so that \overline{wu} is a node in ST.

How much space does a suffix tree occupy? If S is a string of length n , then the suffix tree of $S\$$ has $n + 1$ leaves. Because each internal node of a suffix tree is branching, there cannot be more than n internal nodes. Clearly, the number of edges is one less than the number of nodes. Consequently, there are at most $2n$ edges because there are at most $2n + 1$ nodes in the suffix tree. Since the nodes are not labeled, we can surely represent each node in constant space. If we would represent edge labels explicitly, then the space consumption of the suffix tree would be $\Omega(n^2)$. By contrast, if we represent the label $S[i..j]$ of an edge by the pair of indices (i, j) , then each edge can also be stored in constant space. In this case, the space requirement of the suffix tree is $O(n)$.

Two sides of the same coin

Next, we briefly show that the suffix tree of a string $S\$$ is equivalent to the lcp-interval tree of S .

Consider an internal node \overline{u} in ST. By the definition of the suffix tree, the longest common prefix of all the suffixes represented by the leaves in the subtree rooted at \overline{u} is u . Let i and j be the leftmost and rightmost leaf, respectively, in the subtree rooted at node \overline{u} . The suffixes represented by the leaves in this subtree can be found in the interval $[\text{ISA}[i].. \text{ISA}[j]]$ of the suffix array of S . The smallest value in the interval $[\text{ISA}[i] + 1.. \text{ISA}[j]]$ of the LCP-array is $|u|$. Furthermore, $\text{LCP}[\text{ISA}[i]] < |u|$ and $\text{LCP}[\text{ISA}[j] + 1] < |u|$. Thus, $[\text{ISA}[i].. \text{ISA}[j]]$ is an lcp-interval of lcp-value $|u|$. Moreover, a leaf in ST with number $i \neq n + 1$ corresponds to the singleton interval $[\text{ISA}[i].. \text{ISA}[i]]$. (The leaf with number $n + 1$ represents the suffix $\$$ of $S\$$; this suffix is not represented in the lcp-interval tree of S .)

As an example, consider the node \overline{a} in the suffix tree of $S\$ = \text{ctaataatg}\$$ in Figure 4.21 and the enhanced suffix array of $S = \text{ctaataatg}$ in Figure 4.13 (page 86). The leaves in the subtree rooted at \overline{a} can be found in the interval $[\text{ISA}[3].. \text{ISA}[7]] = [1..4]$ of the enhanced suffix array of S . The smallest value in the interval $[2..4]$ of the LCP-array is 1, which equals $|a|$. Therefore, $[1..4]$ is an lcp-interval of lcp-value 1; cf. Figure 4.14 (page 88).

Conversely, it is not difficult to show that an lcp-interval tree of string S corresponds to the suffix tree of $S\$$.

4.4.1 Linear-time construction

It is also possible to directly construct the suffix tree of a string $S\$$ in linear time from the enhanced suffix array of S without making use of the lcp-interval tree. Crochemore and Rytter [72, Thm. 7.5] sketched a simple algorithm for this task, inspired by the work of Farach-Colton [93, 94]; see also [13]. In fact, their algorithm is very similar to the algorithm that incrementally builds the Cartesian tree of an array; see Section 3.2.

Here, we assume that S is not terminated by the sentinel character $\$$. If S is of length n , then its suffix array SA has size n , whereas the suffix tree of $S\$$ has $n + 1$ leaves. The algorithm creates intermediate trees T_0, T_1, \dots, T_n so that T_n is the suffix tree of $S\$$. The tree T_0 consists of the root, a leaf numbered $n + 1$, and an edge between them, which is labeled with the pair of indices $(n + 1, n + 1)$ representing the sentinel character $\$$. Then, the algorithm incrementally inserts the suffixes $S_{\text{SA}[1]}, S_{\text{SA}[2]}, \dots, S_{\text{SA}[n]}$ into the tree,⁷ hence it is called *suffix insertion algorithm*.

Given the compact trie T_{i-1} of the suffixes $\$, S_{\text{SA}[1]}, S_{\text{SA}[2]}, \dots, S_{\text{SA}[i-1]}$, the suffix $S_{\text{SA}[i]}$ must be inserted into T_{i-1} , yielding T_i . Since $S_{\text{SA}[i]}$ is lexicographically larger than all the suffixes in T_{i-1} and it shares the (longest) prefix of length $\text{LCP}[i]$ with $S_{\text{SA}[i-1]}$, we must add an edge on the rightmost path in T_{i-1} at the distance of $\text{LCP}[i]$ characters from the root. To find this location, we start at the rightmost leaf and climb up the rightmost path, edge by edge. If the location is an internal node v , then we add a new edge from v to a new leaf numbered i . The new edge gets the label $(\text{SA}[i] + \text{LCP}[i], n + 1)$ representing the string $S[\text{SA}[i] + \text{LCP}[i]..n]\$$. If the location is “inside” an edge, then this edge must be split and a new node, say w' , must be inserted at the location. Furthermore, a new edge from w' to a new leaf numbered i is added to the tree, and the new edge gets the label $(\text{SA}[i] + \text{LCP}[i], n + 1)$.

Algorithm 4.12 provides pseudo-code of the suffix insertion algorithm. We assume that every node v in the suffix tree has a pointer $v.\text{parent}$ to its parent and a field $v.d$ that stores its distance from the root in characters. In the pseudo-code, the label of an edge (v, w) in the suffix tree is denoted by $(v, w).\text{label}$. The procedure *makeNewLeaf*(j) creates a new leaf with number j . Furthermore, *splitEdge*(w, v, ℓ) takes an edge (w, v) and a natural number ℓ with $\ell < |(w, v).\text{label}|$ as input, creates a new internal node w' , and splits the edge (w, v) into two edges (w, w') and (w', v) , i.e., $w'.\text{parent} \leftarrow w$ and $v.\text{parent} \leftarrow w'$. The label (j, k) of the edge (w, v) is also

⁷Of course, $\$$ must be appended to the suffixes.

Algorithm 4.12 Computation of the suffix tree of a string $S\$$ from the suffix array and the LCP-array of S .

```

LCP[1]  $\leftarrow$  0          /* needed in this application */
 $v \leftarrow \text{root}$ 
 $v.\text{parent} = \text{root}$ 
 $v.d \leftarrow 0$ 
 $v' \leftarrow \text{makeNewLeaf}(n+1)$     /* $ is the  $(n+1)$ -th suffix of  $S\$$  */
 $v'.\text{parent} \leftarrow v$     /*  $v'$  is a child of the root node */
 $v'.d \leftarrow 1$ 
 $(v, v').\text{label} \leftarrow (n+1, n+1)$     /* the edge  $(\text{root}, v')$  is labeled with $ */
for  $i \leftarrow 1$  to  $n$  do    /* insert  $S[\text{SA}[i]..n]\$$  into the suffix tree */
     $v' \leftarrow \text{makeNewLeaf}(\text{SA}[i])$ 
     $v'.d \leftarrow n+2 - \text{SA}[i]$     /*  $v'.d \leftarrow |S[\text{SA}[i]..n]\$|$  */
    while  $v \neq \text{root}$  and  $(v.\text{parent}).d \geq \text{LCP}[i]$  do
         $v \leftarrow v.\text{parent}$ 
    if  $v.d = \text{LCP}[i]$  then    /*  $v'$  is a child of the internal node  $v$  */
         $v'.\text{parent} \leftarrow v$ 
         $(v, v').\text{label} \leftarrow (\text{SA}[i] + \text{LCP}[i], n+1)$     /* label  $S[\text{SA}[i] + \text{LCP}[i]..n]\$$  */
    else    /*  $v.d > \text{LCP}[i]$  */
         $w' \leftarrow \text{splitEdge}(v.\text{parent}, v, \text{LCP}[i] - v.\text{parent}.d)$ 
         $v'.\text{parent} \leftarrow w'$ 
         $(w', v').\text{label} \leftarrow (\text{SA}[i] + \text{LCP}[i], n+1)$     /* label  $S[\text{SA}[i] + \text{LCP}[i]..n]\$$  */
     $v \leftarrow v'$ 

```

split into two labels $(w, w').\text{label} \leftarrow (j, j + \ell - 1)$ and $(w', v).\text{label} \leftarrow (j + \ell, k)$. Because of the former, $w'.d \leftarrow w.d + \ell$ is the distance of w' from the root.

We show by an amortized analysis that Algorithm 4.12 runs in $O(n)$ time. Obviously, the algorithm creates $n+1$ leaf nodes and thus at most n internal nodes. Consequently, the number of edges in the final tree T_n (which is the suffix tree of $S\$$) is at most $2n$. Inserting the suffix $S_{\text{SA}[i]}$ into T_{i-1} requires walking up the rightmost path in T_{i-1} . Each edge that is traversed ceases to be on the rightmost path in T_i , and thus is not traversed again. When a new internal node w' is created “inside” an edge (w, v) in T_{i-1} , the edge is split into two edges (w, w') and (w', v) , a new edge from w' to the leaf with number i is inserted, and the rightmost path in T_i ends at the new leaf. Therefore, the edge (w', v) ceases to be on the rightmost path and is not traversed again. As each edge is charged once for traversing, it follows as a consequence that the total run-time of Algorithm 4.12 is $O(n)$.

Exercise 4.4.2 Use Algorithm 4.12 to construct the suffix tree of the string *acaaacatat* $\$$.

Applications of Enhanced Suffix Arrays

In this chapter, we discuss several applications of enhanced suffix arrays. The prime motivation for developing suffix trees and suffix arrays was fast exact string matching in situations in which the string S is kept fixed. An application of special importance in bioinformatics is sequence analysis, where S can be the DNA sequence of a chromosome. By concatenating the DNA sequences of all chromosomes (inserting a separator between each string), it is also possible to analyze a complete genome. This chapter is organized as follows. Section 5.1 discusses exact string searching algorithms. We shall see that the suffix array is an index data structure that significantly improves the speed of the search. Section 5.2 describes fast algorithms that compute the Lempel-Ziv factorization of a string. This factorization plays an important role in data compression and is also the basis for the detection of all tandem repeats. Section 5.3 presents several algorithms for finding various kinds of repeats. Repeat analysis is important because repetitive sequences are abundant in eukaryotic genomes (especially in mammalian and plant genomes). Then, we address the problem of comparing two or multiple strings. Sections 5.4 and 5.5 focus on the comparison of two strings. If one wishes to compare two eukaryotic genomes, then one usually starts with the computation of exact matches satisfying various criteria. A related problem is computing the matching statistics between two strings. Suffix-links in the (virtual) lcp-interval tree will prove to be the key in efficiently solving these and related problems. Section 5.6 deals with problems that involve multiple strings. Although it touches the field of document retrieval, the main emphasis is on the analysis of large collections of DNA sequences. Section 5.7 discusses string kernels. A kernel can be thought of as a similarity measure that is used to compare the data, and kernel methods are used e.g. for classification. Section 5.8 shows how to extract frequently occurring patterns from a set of string databases.

5.1 Exact string matching

In the following, P denotes a string of length m , called a *pattern*. It is assumed that m is small compared to the length n of the string S . We recall from Chapter 2 that the exact string matching problem (find all positions in S at which an occurrence of P begins) can be solved in $O(n + m)$ time.

In several applications, the string S is *static* (i.e., does not change) and many patterns P^1, \dots, P^k have to be matched against S . If all patterns are known in advance, then the Aho-Corasick algorithm (described in Section 2.5) finds all occurrences of P^1, \dots, P^k in S in $O(n + \sum_{i=1}^k |P^i| + z)$ time, where z denotes the overall number of occurrences. However, if yet another pattern P is input (which was not known in advance) then $O(n + m)$ time is further needed to find all positions in S at which an occurrence of P begins. In this situation, if the string S is static and the exact string matching problem has to be solved for an unknown number of patterns (which are successively input), it pays to build the enhanced suffix array of S in $O(n)$ time and to match each pattern against this index. We show how the enhanced suffix array of S allows us to answer

- *decision queries* of the type “Is P a substring of S ?” in optimal $O(m)$ time (for a constant-size alphabet),
- *counting queries* of the type “How often does P occur in S ?” in optimal $O(m)$ time (for a constant-size alphabet),
- *enumeration queries* of the type “Where are all z occurrences of P in S ?” in optimal $O(m + z)$ time (for a constant-size alphabet),

totally independent of the size of S .

5.1.1 Forward search on suffix trees

Since the suffix tree ST for $S\$$ contains all substrings of $S\$$, it is easy to verify whether some pattern P (of course, we assume that $\$$ does not occur in P) is a substring of S : just follow the path from the root directed by the characters of P . If at some point one cannot proceed with the next character in P , then P does not occur in the suffix tree and hence it is not a substring of S . Otherwise, if P occurs in the suffix tree, then it is a substring of S . This string matching algorithm answers a decision query in $O(m)$ time, provided that for each node α in ST and each $a \in \Sigma$ one can determine the a -edge outgoing from α (if it exists) in constant time. Constant time access is possible if the outgoing edges are stored in an array of size σ , i.e., the array has an entry for each character of the alphabet Σ . This, however, increases the space consumption of the suffix tree to $O(n\sigma)$, which is not tolerable in larger applications. Thus,

we assume that an internal node α stores its outgoing edges in an array whose size equals the number of children of α . With this implementation, the space usage of the suffix tree is in $O(n)$. A small disadvantage is that now a binary search on this array is required to determine an a -edge outgoing from α . Because each binary search takes $O(\log \sigma)$ time in the worst case, the string matching algorithm takes $O(m \log \sigma)$ time to answer a decision query.

For example, consider the suffix tree for $S\$ = \text{ctaataatg}\$$ from Figure 4.21 (page 111), and let $P = \text{aatc}$. Starting at the root, we first follow the outgoing a -edge. This leads to the branching node \bar{a} . Then we read the next two characters of P , follow the a -edge outgoing from node \bar{a} and go to the branching node \overline{aat} . However, there is no c -edge outgoing from \overline{aat} , and we cannot proceed further. In other words, P does not occur in the suffix tree and hence it is not a substring of S . Now suppose $P = \text{aat}$. Because P occurs in the suffix tree (it is represented by the node \overline{aat}), it follows that P is a substring of S . Note that the positions at which P starts in S are exactly the leaf numbers of the subtree rooted at node \overline{aat} .

Exercise 5.1.1 Devise a string matching algorithm on suffix trees that answers counting queries in optimal $O(m)$ time and enumeration queries in optimal $O(m + z)$ time (linear-time preprocessing is allowed).

5.1.2 Forward search on suffix arrays

The pattern matching algorithm described in the previous section can be simulated by a variant of Algorithm 4.8 (page 98), which traverses the (virtual) lcp-interval tree in a top-down fashion. The simulation is accomplished by the procedure *getInterval* presented in Algorithm 5.1. This procedure takes an lcp-interval $[i..j]$ representing some string ω and a character a as input, and it returns the ωa -interval (if ωa is not a substring of S , it returns \perp). Procedure *getInterval* generates the child intervals of $[i..j]$ one by one until the child interval is found that corresponds to the string ωa (to understand how child intervals are determined, it may be instructive to reread Sections 4.3.1 and 4.3.3). Note that Algorithm 5.1 runs in $O(\sigma)$ time, hence in constant time for a constant-size alphabet.

Pseudo-code of the string matching algorithm on the enhanced suffix array is shown in Algorithm 5.2. The algorithm calls *getInterval* ($[1..n], P[1]$) to determine the interval $[i..j]$ in which suffixes of S start with the prefix $P[1]$ of P . The following cases may occur:

- $[i..j] = \perp$. In this case, $P[1]$ is not a substring of S . Hence P does not occur in S .
- $[i..j]$ is a singleton interval. In this case, P occurs in S if and only if $S[\text{SA}[i]..\text{SA}[i] + m - 1] = P$.

Algorithm 5.1 Procedure $getInterval([i..j], a)$.

```

if  $i = j$  then
  if  $S[SA[i]] = a$  then
    return  $[i..i]$ 
  else
    return  $\perp$ 
 $k \leftarrow i$ 
 $m \leftarrow RMQ(i + 1, j)$       /* first  $\ell$ -index of  $[i..j]$  */
 $\ell \leftarrow LCP[m]$ 
repeat
  if  $S[SA[k] + \ell] = a$  then
    return  $[k..m - 1]$ 
   $k \leftarrow m$ 
  if  $k = j$  then
    break
  else
     $m \leftarrow RMQ(k + 1, j)$ 
until  $LCP[m] \neq \ell$ 
if  $S[SA[k] + \ell] = a$  then
  return  $[k..j]$ 
else
  return  $\perp$ 

```

- $[i..j]$ is an lcp-interval of lcp-value ℓ (note that $\ell = LCP[RMQ_{LCP}(i+1, j)]$). Let $\omega = S[SA[i]..SA[i] + \ell - 1]$ be the longest common prefix of the suffixes $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$. There are the following subcases:
 - $\ell \geq m$. Then, pattern P occurs in S if and only if $\omega[1..m] = P$.
 - $\ell < m$. Clearly, if $\omega \neq P[1..\ell]$, then P does not occur in S . Otherwise, the algorithm simply proceeds as it did before: it calls $getInterval([i..j], P[\ell + 1])$ to determine the interval in which suffixes of S start with the prefix $P[1..\ell + 1]$ of P , and so on.

Algorithm 5.2 runs in $O(m\sigma)$ time, hence in $O(m)$ time for a constant-size alphabet. The algorithm answers decision queries but it can also be used to answer counting queries: since the returned interval $[i..j]$ is the P -interval, there are $j - i + 1$ occurrences of P in S . Moreover, the start positions of the occurrences of P are $SA[i], \dots, SA[j]$. Thus, enumeration queries can be answered in optimal $O(m + z)$ time, where $z = j - i + 1$ is the number of occurrences of P in S .

Exercise 5.1.2 In the problem of *position-restricted string matching* [69, 213], one searches for all occurrences of a pattern P ($|P| = m$) in a substring $S[l..r]$ of the string S ($|S| = n$), under the assumption that an $O(n)$

Algorithm 5.2 Exact string matching: searching $P[1..m]$ in $SA[1..n]$.

```

 $k \leftarrow 1$ 
 $[i..j] \leftarrow [1..n]$ 
repeat
   $[i..j] \leftarrow \text{getInterval}([i..j], P[k])$ 
  if  $[i..j] = \perp$  then
    return  $\perp$ 
   $\ell \leftarrow m$ 
  if  $i < j$  then
     $\ell \leftarrow \min\{\ell, \text{LCP}[\text{RMQ}_{\text{LCP}}(i+1, j)]\}$ 
  if  $S[\text{SA}[i] + k - 1.. \text{SA}[i] + \ell - 1] \neq P[k..\ell]$  then
    return  $\perp$ 
   $k \leftarrow \ell + 1$ 
until  $k > m$ 
return  $[i..j]$ 

```

time and space preprocessing of S is allowed. As we have seen, this problem can be solved in $O(m + z)$ time on the enhanced suffix array of S , where z is the number of occurrences of P in S . However, this is not optimal because the number of occurrences of P in $S[l..r]$ may be (much) smaller than z . If the left boundary l equals one, then an optimal time algorithm is known [70]. This algorithm preprocesses the suffix array SA of S in $O(n)$ time so that range minimum queries can be answered in constant time. Then it takes the pattern P and the right boundary r of the interval $[1..r]$ as input, computes the P -interval $[i..j]$, and calls the procedure $\text{findOcc}(i, j, r, \emptyset)$ from Algorithm 5.3. This procedure returns the set occ of all occurrences of P in $S[l..r]$. The task of the exercise is to explain how the algorithm works and to show that it takes $O(m + |\text{occ}|)$ time.

Algorithm 5.3 Procedure $\text{findOcc}(i, j, r, \text{occ})$.

```

if  $i > j$  then return  $\text{occ}$ 
 $k \leftarrow \text{RMQ}_{\text{SA}}(i, j)$ 
if  $\text{SA}[k] < r$  then
   $\text{occ} \leftarrow \text{occ} \cup \{\text{SA}[k]\}$ 
   $\text{occ} \leftarrow \text{findOcc}(i, k - 1, r, \text{occ})$ 
   $\text{occ} \leftarrow \text{findOcc}(k + 1, j, r, \text{occ})$ 
return  $\text{occ}$ 

```

5.1.3 Binary search

To find all occurrences of a pattern P in the string S , we have to find the suffixes of S that have P as a prefix. Since the n suffixes of the string S appear in lexicographically increasing order in the suffix array, i.e., $S_{SA[1]} < S_{SA[2]} < \dots < S_{SA[n]}$, we can use binary search to find the P -interval in the suffix array. In the search, we have to compare P with suffixes of S , and these comparisons are solely based on the first m characters, where m is the length of P . The next definition formally defines the resulting order.

Definition 5.1.3 Let $<$ be the lexicographic order on the strings on the alphabet Σ . For each string u , we define

$$u|_m = \begin{cases} u & \text{if } |u| \leq m \\ u[1..m] & \text{otherwise} \end{cases}$$

The strict order $<_m$ on Σ^* is defined by: $u <_m v$ if and only if $u|_m < v|_m$. Furthermore, we define the order \leq_m as follows: $u \leq_m v$ if and only if $u|_m < v|_m$ or $u|_m = v|_m$.

In order to explain the practical method we are going to develop, we define

$$l_P = \min (\{i \mid P \leq_m S_{SA[i]}\} \cup \{n+1\})$$

and

$$r_P = \max (\{i \mid S_{SA[i]} \leq_m P\} \cup \{0\})$$

The next lemma states that it suffices to compute l_P and r_P to find all occurrences of P in S .

Lemma 5.1.4 *Pattern P occurs in S if and only if the interval $[l_P..r_P]$ is not empty. If $[l_P..r_P]$ is not empty, then the positions in S at which P occurs are $SA[l_P], SA[l_P + 1], \dots, SA[r_P]$.*

Proof The simple proof is left to the reader. □

Algorithm 5.4 gives pseudo-code for the procedures *findleftmost*(P) and *findrightmost*(P), which compute l_P and r_P by binary search. Both binary searches take $O(\log n)$ time, and each comparison in a binary search needs $O(m)$ character comparisons. Therefore, one can find the interval $[l_P..r_P]$ in time $O(m \log n)$.

However, the practical run-time behavior can be improved. To see this, let l and r be the current interval boundaries in a binary search. We keep track of the lengths h_l and h_r of the longest common prefix of P with $S_{SA[l]}$ and $S_{SA[r]}$, respectively. Because the suffixes in the suffix array are lexicographically ordered, it is clear that all suffixes between l and

Algorithm 5.4 Exact string matching using binary search.

procedure <i>findleftmost</i> (P) $l \leftarrow 1$ $r \leftarrow n$ if $P \leq_m S_{SA[1]}$ then return 1 if $P >_m S_{SA[n]}$ then return $n + 1$ while $r - l > 1$ do $mid \leftarrow \lfloor (l + r)/2 \rfloor$ if $P \leq_m S_{SA[mid]}$ then $r \leftarrow mid$ else $l \leftarrow mid$ return r	procedure <i>findrightmost</i> (P) $l \leftarrow 1$ $r \leftarrow n$ if $P <_m S_{SA[1]}$ then return 0 if $P \geq_m S_{SA[n]}$ then return n while $r - l > 1$ do $mid \leftarrow \lfloor (l + r)/2 \rfloor$ if $S_{SA[mid]} \leq_m P$ then $l \leftarrow mid$ else $r \leftarrow mid$ return l
---	--

r share the same prefix of length $\min\{h_l, h_r\}$. Hence $S_{SA[mid]}$ and P have a common prefix of length $\min\{h_l, h_r\}$, where $mid = \lfloor (l + r)/2 \rfloor$ is the midpoint between l and r . This common prefix can be skipped in the comparison of $S_{SA[mid]}$ and P . Pseudo-code of this technique is given in Algorithm 5.5; it originates from [2].

The procedure *sasearch*(P) in Algorithm 5.5 computes the boundaries of the interval $[l_P..r_P]$ and outputs all occurrences of P in S ; cf. Lemma 5.1.4. The procedures *findleftmost*(P) and *findrightmost*(P) have the same functionality as their relatives in Algorithm 5.4. The procedure *cmp* performs the comparison of a suffix of S with some string w . More precisely, let $1 \leq i \leq n$ and $v = S_{SA[i]}^\$$ and suppose that w and v have a common prefix of length q . Then *cmp*(w, i, q) delivers a pair (c, f_c) so that the following holds:

- c is the length of the longest common prefix of w and v , and $c \geq q$.
- If w is a prefix of v , i.e., $c = m$ holds, then $f_c = 0$.
- Otherwise, if w is not a prefix of v , then $w[c + 1] \neq v[c + 1]$ (because v ends with $\$$, it cannot be a prefix of w). There are two cases:
 - If $w[c + 1] < v[c + 1]$, then $f_c = -1$.
 - If $w[c + 1] > v[c + 1]$, then $f_c = 1$.

In summary, for $w = P$ we have

$$f_c = \begin{cases} -1 & \text{if } P < S_{SA[i]} \\ 0 & \text{if } P \text{ is a prefix of } S_{SA[i]} \\ +1 & \text{if } P > S_{SA[i]} \end{cases}$$

Algorithm 5.5 Exact string matching: A practical method.

procedure *findleftmost*(P) $l \leftarrow 1$ $r \leftarrow n$ $(h_l, f_l) \leftarrow \text{cmp}(P, l, 0)$ **if** $f_l \leq 0$ **then****return** 1 $(h_r, f_r) \leftarrow \text{cmp}(P, r, 0)$ **if** $f_r > 0$ **then****return** $n + 1$ **while** $r - l > 1$ **do** $mid \leftarrow \lfloor (l + r)/2 \rfloor$ $(c, f_c) \leftarrow \text{cmp}(P, mid, \min\{h_l, h_r\})$ **if** $f_c \leq 0$ **then** $(h_r, r) \leftarrow (c, mid)$ **else** $(h_l, l) \leftarrow (c, mid)$ **return** r **procedure** *cmp*(w, i, q) $v \leftarrow S_{SA[i]} \$$ $c \leftarrow q$ **while** $c < \min\{|w|, |v|\}$ **do****if** $w[c + 1] < v[c + 1]$ **then****return** $(c, -1)$ **else****if** $w[c + 1] > v[c + 1]$ **then****return** $(c, 1)$ **else** $c \leftarrow c + 1$ **return** $(c, 0)$ /* here $c = |w|$ */**procedure** *findrightmost*(P) $l \leftarrow 1$ $r \leftarrow n$ $(h_l, f_l) \leftarrow \text{cmp}(P, l, 0)$ **if** $f_l < 0$ **then****return** 0 $(h_r, f_r) \leftarrow \text{cmp}(P, r, 0)$ **if** $f_r \geq 0$ **then****return** n **while** $r - l > 1$ **do** $mid \leftarrow \lfloor (l + r)/2 \rfloor$ $(c, f_c) \leftarrow \text{cmp}(P, mid, \min\{h_l, h_r\})$ **if** $0 \leq f_c$ **then** $(h_l, l) \leftarrow (c, mid)$ **else** $(h_r, r) \leftarrow (c, mid)$ **return** l **procedure** *sasearch*(P) $l_P \leftarrow \text{findleftmost}(P)$ $r_P \leftarrow \text{findrightmost}(P)$ **if** $l_P \leq r_P$ **then****for** $j \leftarrow l_P$ **to** r_P **do**report "match at $SA[j]$ "**else**

report "no match found"

so that $f_c \leq 0$ if $P \leq_m S_{SA[l]}$ and $0 \leq f_c$ if $S_{SA[l]} \leq_m P$.

The string matching algorithm based on binary searches in the suffix array was first presented by Manber and Myers [214]. They also showed that the algorithm can be improved to a run time of $O(m + \log n)$, provided that in each binary search step the values $|\text{lcp}(S_{SA[l]}, S_{SA[mid]})|$ and $|\text{lcp}(S_{SA[mid]}, S_{SA[r]})|$ are known. With range minimum queries, these values can straightforwardly be computed by $|\text{lcp}(S_{SA[l]}, S_{SA[mid]})| = \text{LCP}[\text{RMQ}_{\text{LCP}}(l + 1, \text{mid})]$ and $|\text{lcp}(S_{SA[mid]}, S_{SA[r]})| = \text{LCP}[\text{RMQ}_{\text{LCP}}(\text{mid} + 1, r)]$. (Exercise 5.1.6 asks you to provide an implementation of the algorithm without range minimum queries.) The improved version of the procedure $\text{findleftmost}(P)$ can be found in Algorithm 5.6; procedure $\text{findrightmost}(P)$ can be improved likewise. Again, the algorithm keeps track of the length h_l and h_r of the longest common prefix of P with $S_{SA[l]}$ and $S_{SA[r]}$, respectively.

The correctness of Algorithm 5.6 relies on the following lemma, which shows that in some cases it is possible to decide whether a string is lexicographically smaller than another without further character comparisons.

Lemma 5.1.5 *Let u , v , and w be strings so that u is lexicographically smaller than v . If $|\text{lcp}(u, v)| < |\text{lcp}(u, w)|$, then w is also lexicographically smaller than v .*

Proof Let $k = |\text{lcp}(u, v)|$. Since $u < v$, it follows that the $(k + 1)$ -th character of u is smaller than the $(k + 1)$ -th character of v . Moreover, the inequality $|\text{lcp}(u, v)| < |\text{lcp}(u, w)|$ implies that the $(k + 1)$ -th character of u coincides with the $(k + 1)$ -th character of w . All in all, we have $u[1..k] = v[1..k] = w[1..k]$ and $u[k + 1] = w[k + 1] < v[k + 1]$. This shows the lemma. \square

To prove the correctness of Algorithm 5.6, consider an iteration of the while-loop and assume $h_l \geq h_r$ (the case $h_l < h_r$ can be shown similarly). There are the following cases.

- (a) $\text{lcp}_l = |\text{lcp}(S_{SA[l]}, S_{SA[mid]})| > |\text{lcp}(S_{SA[l]}, P)| = h_l$. Because $S_{SA[l]} < P$, an application of Lemma 5.1.5 with $u = S_{SA[l]}$, $v = S_{SA[mid]}$, and $w = P$ yields $S_{SA[mid]} < P$. Moreover, $|\text{lcp}(S_{SA[mid]}, P)| = |\text{lcp}(S_{SA[l]}, P)| = h_l$; a graphical proof of this fact can be found in Figure 5.1. Therefore, l is set to mid and h_l remains unchanged.
- (b) $\text{lcp}_l = |\text{lcp}(S_{SA[l]}, S_{SA[mid]})| = |\text{lcp}(S_{SA[l]}, P)| = h_l$. In this case, we know that the first h_l characters of P and $S_{SA[mid]}$ coincide, but the relationship between the remaining characters is unknown. Thus, the procedure call $\text{cmp}(P, \text{mid}, h_l)$ completes the comparison of P and $S_{SA[mid]}$.
- (c) $\text{lcp}_l = |\text{lcp}(S_{SA[l]}, S_{SA[mid]})| < |\text{lcp}(S_{SA[l]}, P)| = h_l$. Because $S_{SA[l]} < S_{SA[mid]}$, an application of Lemma 5.1.5 with $u = S_{SA[l]}$, $v = P$, and $w = S_{SA[mid]}$ yields $P < S_{SA[mid]}$ (further note that $P < S_{SA[mid]} < S_{SA[r]}$ implies $|\text{lcp}(S_{SA[mid]}, P)| \geq |\text{lcp}(S_{SA[mid]}, S_{SA[r]})| = h_r$). Moreover, it follows as in

Algorithm 5.6 This implementation of the procedure *findleftmost*(P) uses less character comparisons.

```

 $l \leftarrow 1$ 
 $r \leftarrow n$ 
 $(h_l, f_l) \leftarrow \text{cmp}(P, l, 0)$ 
if  $f_l \leq 0$  then
    return 1
 $(h_r, f_r) \leftarrow \text{cmp}(P, r, 0)$ 
if  $f_r > 0$  then
    return  $n + 1$ 
while  $r - l > 1$  do
     $mid \leftarrow \lfloor (l + r) / 2 \rfloor$ 
    if  $h_l \geq h_r$  then
         $lcp_l \leftarrow |\text{lcp}(S_{SA[l]}, S_{SA[mid]})|$ 
        if  $lcp_l > h_l$  then
             $(c, f_c) \leftarrow (h_l, 1)$ 
        else if  $lcp_l = h_l$  then
             $(c, f_c) \leftarrow \text{cmp}(P, mid, h_l)$ 
        else
             $(c, f_c) \leftarrow (lcp_l, -1)$ 
    else
         $lcp_r \leftarrow |\text{lcp}(S_{SA[mid]}, S_{SA[r]})|$ 
        if  $lcp_r > h_r$  then
             $(c, f_c) \leftarrow (h_r, -1)$ 
        else if  $lcp_r = h_r$  then
             $(c, f_c) \leftarrow \text{cmp}(P, mid, h_r)$ 
        else
             $(c, f_c) \leftarrow (lcp_r, 1)$ 
    if  $f_c \leq 0$  then
         $(h_r, r) \leftarrow (c, mid)$ 
    else
         $(h_l, l) \leftarrow (c, mid)$ 
return  $r$ 

```

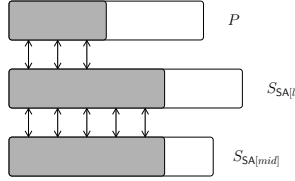


Figure 5.1: The equality $|\text{lcp}(S_{SA[mid]}, P)| = |\text{lcp}(S_{SA[l]}, P)|$ is a consequence of $|\text{lcp}(S_{SA[l]}, P)| < |\text{lcp}(S_{SA[l]}, S_{SA[mid]})|$.

case (a) that $|\text{lcp}(S_{SA[mid]}, P)| = |\text{lcp}(S_{SA[l]}, S_{SA[mid]})| = \text{lcp}_l$. Therefore, r is set to mid and h_r is set to lcp_l (note that the new value of h_r is greater than or equal to the old value of h_r).

It is a consequence of the preceding considerations that neither h_l nor h_r decreases in the binary search. Furthermore, if the procedure *cmp* detects further character matches, then either h_l or h_r is increased by the number of detected character matches. It follows that there are at most $m = |P|$ character matches in the binary search. Obviously, in each of the $O(\log n)$ binary search steps at most one character mismatch occurs. To sum up, the improved string matching algorithm has a worst-case time complexity of $O(m + \log n)$.

Exercise 5.1.6 Implement Algorithm 5.6 without range minimum queries. First, show that the values $|\text{lcp}(S_{SA[p]}, S_{SA[q]})|$ of at most $O(n)$ pairs (p, q) of indices are required. Second, show that these values can be computed incrementally (use the LCP-array for pairs of the form $(p - 1, p)$).

5.2 Lempel-Ziv factorization

For 30 years the Lempel-Ziv factorization [334] of a string has played an important role in data compression (e.g. it is used in *gzip*), and more recently it was used as the basis of linear-time algorithms for the detection of all maximal repetitions (runs) in a string [141, 185].

Definition 5.2.1 Let S be a string of length n on an alphabet Σ . The *Lempel-Ziv factorization* (or LZ-factorization for short) of S is a factorization $S = s_1 s_2 \cdots s_m$ so that each factor s_j , $1 \leq j \leq m$, is either

- (a) a letter $c \in \Sigma$ that does not occur in $s_1 s_2 \cdots s_{j-1}$, or
- (b) the longest substring of S that occurs at least twice in $s_1 s_2 \cdots s_j$.

Algorithm 5.7 Reconstruction of the string S , given its LZ-factorization $(\text{PrevOcc}_1, \text{LPS}_1), \dots, (\text{PrevOcc}_m, \text{LPS}_m)$.

```

 $i \leftarrow 1$ 
for  $j \leftarrow 1$  to  $m$  do
  if  $\text{LPS}_j = 0$  then
     $S[i] \leftarrow \text{PrevOcc}_j$ 
     $i \leftarrow i + 1$ 
  else
    for  $k \leftarrow 0$  to  $\text{LPS}_j - 1$  do
       $S[i] \leftarrow S[\text{PrevOcc}_j + k]$ 
       $i \leftarrow i + 1$ 

```

The Lempel-Ziv factorization can be represented by a sequence of pairs $(\text{PrevOcc}_1, \text{LPS}_1), \dots, (\text{PrevOcc}_m, \text{LPS}_m)$, where in case (a) $\text{PrevOcc}_j = c$ and $\text{LPS}_j = 0$, and in case (b) PrevOcc_j is a position in $s_1 s_2 \dots s_{j-1}$, at which an occurrence of s_j starts and $\text{LPS}_j = |s_j|$.

For example, the LZ-factorization of $S = \text{acaaacatat}$ is $s_1 = a$, $s_2 = c$, $s_3 = a$, $s_4 = aa$, $s_5 = ca$, $s_6 = t$, $s_7 = at$. This LZ-factorization can be represented by $(a, 0), (c, 0), (1, 1), (3, 2), (2, 2), (t, 0), (7, 2)$.

To appreciate the full value of this compression method, consider the string a^n that consists solely of a 's. Its LZ-factorization is $(a, 0), (1, n - 1)$.

In this section, we will develop efficient algorithms that compress a string by computing its LZ-factorization. The corresponding decompression algorithm is very simple, it is given in Algorithm 5.7.

5.2.1 Longest previous substring

Because it is difficult to predict in advance at which position a factor of the LZ-factorization starts, several algorithms compute the factors starting at every position i in S and select the right ones afterwards.

Definition 5.2.2 For a string S of length n , the *longest previous substring* (LPS) array of size n is defined by $\text{LPS}[1] = 0$ and

$$\text{LPS}[k] = \max\{\ell : 0 \leq \ell \leq n - k + 1; S[k..k + \ell - 1] \text{ is a substring of } S[1..k + \ell - 2]\}$$

for every k with $2 \leq k \leq n$. That is, $\text{LPS}[k]$ is the length of the longest prefix of S_k that has another occurrence in S starting strictly before position k .

If there is no $\ell > 0$ so that $S[k..k + \ell - 1]$ is a substring of $S[1..k + \ell - 2]$, then $\text{LPS}[k] = 0$ because for $\ell = 0$ we have $S[k..k + \ell - 1] = \varepsilon$ and the empty string ε is a substring of any other string. Clearly, we are not

$S[i]$	a	c	a	a	a	c	a	t	a	t
i	1	2	3	4	5	6	7	8	9	10
$LPS[i]$	0	0	1	2	3	2	1	0	2	1
$PrevOcc[i]$	a	c	1	3	1	2	5	t	7	8

Figure 5.2: The LPS and PrevOcc arrays of the string $S = acaaacatat\$$.

Algorithm 5.8 Computation of the LZ-factorization based on LPS and PrevOcc.

```

 $i \leftarrow 1$ 
while  $i < n$  do
  if  $LPS[i] = 0$  then
     $PrevOcc[i] \leftarrow S[i]$ 
  output  $(PrevOcc[i], LPS[i])$ 
   $i \leftarrow i + \max\{1, LPS[i]\}$ 

```

only interested in $LPS[k]$ but also in a position $j < k$ at which the longest previous substring occurred. Such a position j will be stored in the array PrevOcc, i.e., $PrevOcc[k] = j$. If there is no such position, i.e., if $LPS[k] = 0$, we set $PrevOcc[k] = S[k]$. As an example, the arrays LPS and PrevOcc of $S = acaaacatat$ are depicted in Figure 5.2.

Algorithm 5.8 shows that the Lempel-Ziv factorization can easily be computed from the arrays LPS and PrevOcc.

The following lemma characterizes $LPS[k]$; it can be viewed as a first step towards the computation of $LPS[k]$.

Lemma 5.2.3 *For every k with $2 \leq k \leq n$, the following equality holds true:*

$$LPS[k] = \max\{|\text{lcp}(S_p, S_k)| : 1 \leq p < k\}$$

Proof The equalities

$$\begin{aligned}
& LPS[k] \\
&= \max\{\ell \in \mathbb{N} : S[k..k+\ell-1] \text{ is a substring of } S[1..k+\ell-2]\} \\
&= \max\{\ell \in \mathbb{N} : S[k..k+\ell-1] \text{ is a prefix of some } S[p..k+\ell-2], 1 \leq p < k\} \\
&= \max\{|\text{lcp}(S_p, S_k)| : 1 \leq p < k\}
\end{aligned}$$

prove the lemma. □

We would like to enhance the suffix array of string S with the array LPS. Thus, we need the values of the array LPS in the order $SA[1], \dots, SA[n]$.

Observe that with $k = \text{SA}[i]$ and $p = \text{SA}[j]$, Lemma 5.2.3 can be rephrased as

$$\text{LPS}[\text{SA}[i]] = \max\{|\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})| : \text{SA}[j] < \text{SA}[i]\} \quad (5.1)$$

To deal with boundary cases, we introduce the following artificial entries in the suffix array: $\text{SA}[0] = 0$ and $\text{SA}[n+1] = 0$. Furthermore, we define $S_0 = \varepsilon$, so that $S_{\text{SA}[0]}$ and $S_{\text{SA}[n+1]}$ are the empty string. This implies that $\text{LCP}[1]$ and $\text{LCP}[n+1]$ must be 0 (and not -1 as in the preceding sections) because

$$\text{LCP}[1] = |\text{lcp}(S_{\text{SA}[0]}, S_{\text{SA}[1]})| = |\text{lcp}(\varepsilon, S_{\text{SA}[1]})| = |\varepsilon| = 0$$

and analogously $\text{LCP}[n+1] = |\text{lcp}(S_{\text{SA}[n]}, S_{\text{SA}[n+1]})| = 0$. So in contrast to previous and subsequent sections, *in the rest of* Section 5.2 we assume that $\text{LCP}[1] = 0 = \text{LCP}[n+1]$.

In the computation of $\text{LPS}[\text{SA}[i]]$, we will employ two auxiliary arrays PSV_{SA} and NSV_{SA} defined as follows.

Definition 5.2.4 For any index i with $1 \leq i \leq n$, we define

$$\text{PSV}_{\text{SA}}[i] = \max\{j : 0 \leq j < i \text{ and } \text{SA}[j] < \text{SA}[i]\}$$

and

$$\text{NSV}_{\text{SA}}[i] = \min\{j : i < j \leq n+1 \text{ and } \text{SA}[j] < \text{SA}[i]\}$$

In the following, we will omit the subscript SA, i.e., we will write PSV instead of PSV_{SA} and NSV instead of NSV_{SA} . The reader should be aware of the fact that the PSV and NSV arrays used in Section 4.3.1 are different arrays because they were defined w.r.t. the LCP-array and not w.r.t. the suffix array.

According to Definition 5.2.4, if there is no index j with $1 \leq j < i$ and $\text{SA}[j] < \text{SA}[i]$, we have $\text{PSV}[i] = 0$ because $0 < i$ and $0 = \text{SA}[0] < \text{SA}[i]$. Analogously, if there is no index j with $i < j \leq n$ and $\text{SA}[j] < \text{SA}[i]$, we have $\text{NSV}[i] = n+1$ because $i < n+1$ and $0 = \text{SA}[n+1] < \text{SA}[i]$. Figure 5.3 provides an example of the auxiliary arrays PSV and NSV.

In essence, the next lemma improves Equation 5.1. That is, in order to compute $\text{LPS}[\text{SA}[i]]$ for a given index i , it suffices to consider the closest index j with $\text{SA}[j] < \text{SA}[i]$ preceding i (viz. $\text{PSV}[i]$) and the closest index j with $\text{SA}[j] < \text{SA}[i]$ succeeding i (viz. $\text{NSV}[i]$) in the suffix array of S .

Lemma 5.2.5 For every i with $1 \leq i \leq n$, the following equality holds

$$\text{LPS}[\text{SA}[i]] = \max\{|\text{lcp}(S_{\text{SA}[\text{PSV}[i]]}, S_{\text{SA}[i]})|, |\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[\text{NSV}[i]])}|\}$$

where S_0 is the empty string ε .

i	SA	LCP	$S_{SA[i]}$	PSV $[i]$	NSV $[i]$	LPS[SA $[i]$]	PrevOcc[SA $[i]$]
0	0		ε				
1	3	0	<i>aaacatat</i>	0	3	1	1
2	4	2	<i>aacatat</i>	1	3	2	3
3	1	1	<i>acaaacatat</i>	0	11	0	\perp
4	5	3	<i>acatat</i>	3	7	3	1
5	9	1	<i>at</i>	4	6	2	7
6	7	2	<i>atat</i>	4	7	1	5
7	2	0	<i>caaacatat</i>	3	11	0	\perp
8	6	2	<i>catat</i>	7	11	2	2
9	10	0	<i>t</i>	8	10	1	8
10	8	1	<i>tat</i>	8	11	0	\perp
11	0	0	ε				

Figure 5.3: The enhanced suffix array of the string $S = acaaacatat$.

Proof For every i with $1 \leq i \leq n$, we have

$$\text{LPS}[\text{SA}[i]] = \max\{|\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})| : \text{SA}[j] < \text{SA}[i]\}$$

by Equation 5.1 and the extension of the suffix array with $\text{SA}[0] = 0$ and $\text{SA}[n+1] = 0$. (Note that $\text{LPS}[1] = \max\{|\text{lcp}(S_{\text{SA}[j]}, S_1)| : \text{SA}[j] < 1\} = |\text{lcp}(S_0, S_1)| = |\text{lcp}(\varepsilon, S_1)| = 0$.) Clearly,

$$\begin{aligned} \text{LPS}[\text{SA}[i]] &= \max\{|\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})| : 0 \leq j \leq n+1 \text{ and } \text{SA}[j] < \text{SA}[i]\} \\ &= \max(\{|\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})| : 0 \leq j < i \text{ and } \text{SA}[j] < \text{SA}[i]\} \\ &\quad \cup \{|\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[j]})| : i < j \leq n+1 \text{ and } \text{SA}[j] < \text{SA}[i]\}) \end{aligned}$$

We will show

$$|\text{lcp}(S_{\text{SA}[\text{PSV}[i]]}, S_{\text{SA}[i]})| = \max\{|\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})| : 0 \leq j < i \text{ and } \text{SA}[j] < \text{SA}[i]\}$$

and

$$|\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[\text{NSV}[i]]})| = \max\{|\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[j]})| : i < j \leq n+1 \text{ and } \text{SA}[j] < \text{SA}[i]\}$$

To prove the first equality, consider j with $j < \text{PSV}[i]$ and $\text{SA}[j] < \text{SA}[i]$. Since $S_{\text{SA}[j]}$ is lexicographically smaller than $S_{\text{SA}[\text{PSV}[i]]}$, which in turn is lexicographically smaller than $S_{\text{SA}[i]}$, it follows that $\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})$ is also a prefix of $\text{lcp}(S_{\text{SA}[\text{PSV}[i]]}, S_{\text{SA}[i]})$. Thus, $|\text{lcp}(S_{\text{SA}[j]}, S_{\text{SA}[i]})| \leq |\text{lcp}(S_{\text{SA}[\text{PSV}[i]]}, S_{\text{SA}[i]})|$. Now the first equality follows from the fact that $\text{SA}[j'] > \text{SA}[i]$ for all indices j' with $\text{PSV}[i] < j' < i$. The second equality can be shown similarly. \square

We are not only interested in $\text{LPS}[\text{SA}[i]]$ but also in a position $\text{SA}[j] < \text{SA}[i]$ at which the longest previous substring occurred. Such a position $\text{SA}[j]$

Algorithm 5.9 Computing LPS and PrevOcc using SA, PSV, NSV, and LCP.

Compute the arrays PSV and NSV.

Prepare the LCP-array for constant time range minimum queries.

for $i \leftarrow 1$ **to** n **do**

$l \leftarrow \text{RMQ}_{\text{LCP}}[\text{PSV}[i] + 1..i]$

$r \leftarrow \text{RMQ}_{\text{LCP}}[i + 1.. \text{NSV}[i]]$

$\text{LPS}[\text{SA}[i]] \leftarrow \max\{\text{LCP}[l], \text{LCP}[r]\}$

if $\text{LPS}[\text{SA}[i]] = 0$ **then** $\text{PrevOcc}[\text{SA}[i]] \leftarrow \perp$

else if $\text{LPS}[\text{SA}[i]] > \text{LCP}[r]$ **then** $\text{PrevOcc}[\text{SA}[i]] \leftarrow \text{SA}[\text{PSV}[i]]$

else $\text{PrevOcc}[\text{SA}[i]] \leftarrow \text{SA}[\text{NSV}[i]]$

will be stored in the array PrevOcc, i.e., $\text{PrevOcc}[\text{SA}[i]] = \text{SA}[j]$; see Figure 5.3. Note that $\text{PrevOcc}[\text{SA}[i]]$ is not necessarily the leftmost position at which the longest previous substring occurred.

A simple case analysis yields $\text{PrevOcc}[\text{SA}[i]]$:

1. If $\text{LPS}[\text{SA}[i]] = 0$, then there is no previous occurrence. Consequently, $\text{PrevOcc}[\text{SA}[i]] = \perp$, i.e., $\text{PrevOcc}[\text{SA}[i]]$ is undefined.
2. If $\text{LPS}[\text{SA}[i]] > |\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[\text{NSV}[i]]})|$, then $\text{LPS}[\text{SA}[i]] = |\text{lcp}(S_{\text{SA}[\text{PSV}[i]]}, S_{\text{SA}[i]})| > 0$ and $\text{PrevOcc}[\text{SA}[i]] = \text{SA}[\text{PSV}[i]]$.
3. Otherwise, $\text{LPS}[\text{SA}[i]] = |\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[\text{NSV}[i]]})| > 0$. Therefore, we set $\text{PrevOcc}[\text{SA}[i]] = \text{SA}[\text{NSV}[i]]$.

To obtain a linear-time algorithm, we compute $\text{LPS}[\text{SA}[i]]$ by constant time range minimum queries on the LCP-array. More precisely, for any index i with $1 \leq i \leq n$, we use the equalities (see Lemma 4.2.8)

$$|\text{lcp}(S_{\text{SA}[\text{PSV}[i]]}, S_{\text{SA}[i]})| = \text{LCP}[\text{RMQ}_{\text{LCP}}[\text{PSV}[i] + 1..i]]$$

and

$$|\text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[\text{NSV}[i]]})| = \text{LCP}[\text{RMQ}_{\text{LCP}}[i + 1.. \text{NSV}[i]]]$$

These considerations immediately yield the correctness of Algorithm 5.9.

The worst-case time complexity of Algorithm 5.9 is indeed $O(n)$. This is because the arrays PSV and NSV can be computed in linear time (Algorithm 4.5 on page 93), the preprocessing of the LCP-array for constant time range minimum queries can be done in linear time (Chapter 3), and each execution of the for-loop takes constant time.

However, Algorithm 5.9 is not very practicable because it uses too much space. In fact, we neither need the arrays PSV and NSV nor range minimum queries. To see this, we depict the values of the suffix array and the LCP-array in a (conceptual) graph introduced by Crochemore and Ilie [65]. The graph has $n + 2$ nodes each of which is labeled with $(i, \text{SA}[i])$; to deal with

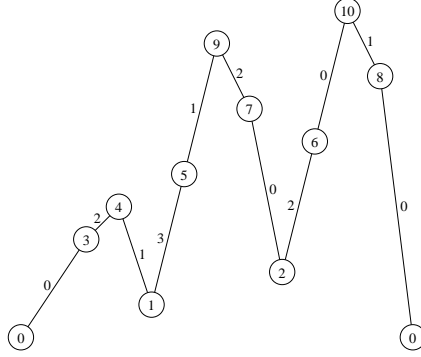


Figure 5.4: The graph for the string $S = acaaacatat$. Each node $(i, SA[i])$ is drawn as a circle with label $SA[i]$ at the point $(i, SA[i])$ in the plane \mathbb{R}^2 ; for better readability the x and y-axes are omitted. Consecutive nodes $(i, SA[i])$ and $(i + 1, SA[i + 1])$ are connected by an edge labeled with the value $LCP[i + 1]$.

boundary cases, the graph also contains the nodes labeled with $(0, SA[0])$ and $(n + 1, SA[n + 1])$. It is instructive to view each node $(i, SA[i])$ as a point in the plane \mathbb{R}^2 . Moreover, consecutive nodes $(i, SA[i])$ and $(i + 1, SA[i + 1])$ are connected by an edge labeled with the value $LCP[i + 1]$; see Figure 5.4 for an example.

As Algorithm 5.9, the algorithm we are going to develop is based on Lemma 5.2.5. In the graph, $|\text{lcp}(S_{SA[\text{PSV}[i]]}, S_{SA[i]})|$ is the minimum of the edge labels on the path from node $(\text{PSV}[i], SA[\text{PSV}[i]])$ to $(i, SA[i])$. Analogously, $|\text{lcp}(S_{SA[i]}, S_{SA[\text{NSV}[i]]})|$ is the minimum of the edge labels on the path from $(i, SA[i])$ to node $(\text{NSV}[i], SA[\text{NSV}[i]])$. Now consider a *peak* in the graph, i.e., a node $(i, SA[i])$ so that $SA[i - 1] < SA[i]$ and $SA[i + 1] < SA[i]$. In this case, $(\text{PSV}[i], SA[\text{PSV}[i]]) = (i - 1, SA[i - 1])$ and $(\text{NSV}[i], SA[\text{NSV}[i]]) = (i + 1, SA[i + 1])$. Thus, we have $\text{LPS}[SA[i]] = \max\{LCP[i], LCP[i + 1]\}$ by Lemma 5.2.5. That is, the correct value of $\text{LPS}[SA[i]]$ can be computed as the maximum of the labels of the edges with end point $(i, SA[i])$. The algorithm relies on the key observation that a deletion of the peak node $(i, SA[i])$ does not change the values $\text{PSV}[j]$, $\text{NSV}[j]$, $SA[\text{PSV}[j]]$, and $SA[\text{NSV}[j]]$ for any index $j \neq i$ with $1 \leq j \leq n$. So we delete node $(i, SA[i])$ and its edges from the graph, and add an edge labeled with $\min(LCP[i], LCP[i + 1])$ between nodes $(i - 1, SA[i - 1])$ and $(i + 1, SA[i + 1])$. In the transformed graph, for any index $j \neq i$ with $1 \leq j \leq n$, $\text{LPS}[SA[j]]$ can again be computed as the maximum of (a) the minimum of the edge labels on the paths from node $(\text{PSV}[j], SA[\text{PSV}[j]])$ to $(j, SA[j])$ and (b) the minimum of the edge labels from node $(j, SA[j])$ to $(\text{NSV}[j], SA[\text{NSV}[j]])$.

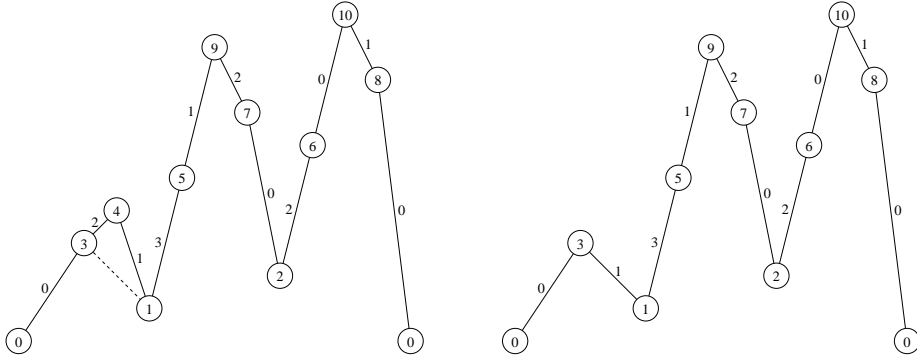


Figure 5.5: Removal of the first peak in the graph.

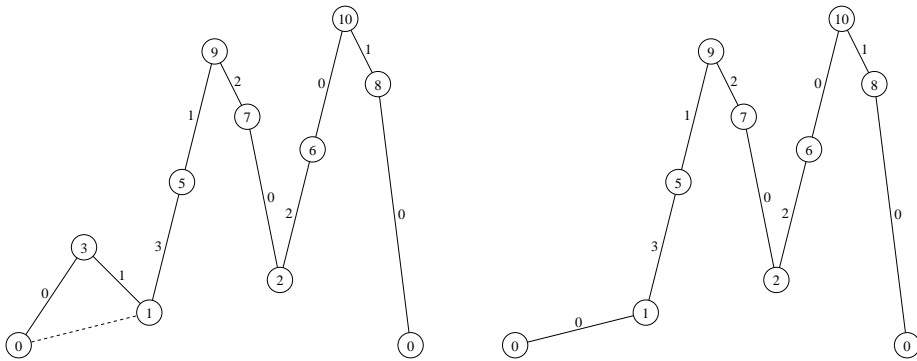


Figure 5.6: Removal of the second peak in the graph.

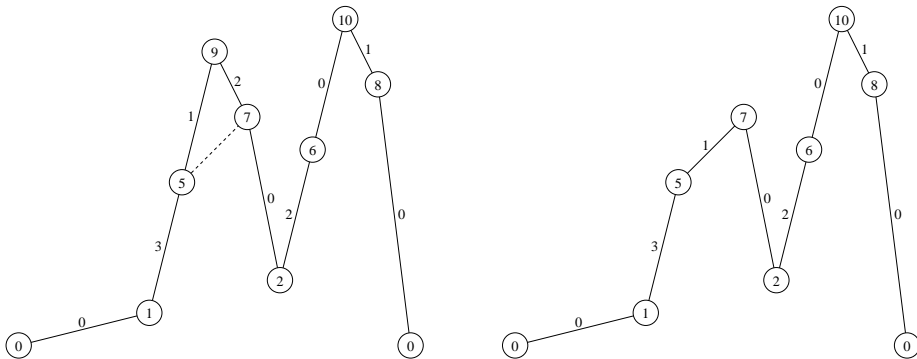


Figure 5.7: Removal of the third peak in the graph.

Algorithm 5.10 Computing LPS and PrevOcc using SA and LCP.

```

SA[n + 1] ← 0
LCP[n + 1] ← 0
push(⟨0, ⊥⟩)
for  $i \leftarrow 1$  to  $n + 1$  do
     $lcp \leftarrow \text{LCP}[i]$ 
    while  $\text{SA}[i] < \text{top.sa}$ 
         $\text{last} \leftarrow \text{pop}$ 
         $\text{LPS}[\text{last.sa}] \leftarrow \max(\text{last.lcp}, lcp)$ 
         $lcp \leftarrow \min(\text{last.lcp}, lcp)$ 
        if  $\text{LPS}[\text{last.sa}] = 0$  then  $\text{PrevOcc}[\text{last.sa}] \leftarrow \perp$ 
        else if  $\text{last.lcp} > lcp$  then  $\text{PrevOcc}[\text{last.sa}] \leftarrow \text{top.sa}$ 
        else  $\text{PrevOcc}[\text{last.sa}] \leftarrow \text{SA}[i]$ 
    push(⟨SA[i], lcp⟩)

```

This is because the label $\min(\text{LCP}[i], \text{LCP}[i + 1])$ of the new edge is the length of the longest common prefix of the suffixes $S_{\text{SA}[i-1]}$ and $S_{\text{SA}[i+1]}$ of S . Hence it is safe to eliminate peak by peak from the graph. Obviously, the order in which peaks are considered is arbitrary. Figures 5.5 to 5.7 illustrate the removal of the first three peaks (from left to right) from the graph of Figure 5.4.

Algorithm 5.10 processes nodes and peaks from left to right. Nodes that are not yet peaks are stored on a stack. As a matter of fact, we do not store the whole node $(i, \text{SA}[i])$ but only its second component $\text{SA}[i]$. The stack contains pairs $\langle sa, lcp \rangle$, where sa corresponds to the second component of a node in the graph and lcp corresponds to the longest common prefix between two suffixes of S . More precisely, Algorithm 5.10 maintains the following invariant: if $\langle sa, lcp \rangle$ appears directly above $\langle sa', lcp' \rangle$ in the stack, then $sa \geq sa'$ (equality can hold solely for $sa = \text{SA}[n + 1] = 0$ and $sa' = \text{SA}[0] = 0$) and lcp is the length of the longest common prefix between S_{sa} and $S_{sa'}$. Initially, $\langle \text{SA}[0], \text{LCP}[0] \rangle = \langle 0, \perp \rangle$ is pushed onto the stack. In the i -th iteration, there are two cases that have to be distinguished by the algorithm. If $\text{SA}[i] \geq \text{top.sa}$, then the node corresponding to the top element of the stack is not a peak yet. As a consequence, the pair $\langle \text{SA}[i], \text{LCP}[i] \rangle$ is pushed onto the stack. If otherwise $\text{SA}[i] < \text{top.sa}$, then the node corresponding to the top element of the stack is a peak. In this case, it is popped from the stack, its LCP-value is set, and its previous occurrence determined as in Algorithm 5.9. Upon termination of Algorithm 5.10, the arrays LPS and PrevOcc contain correct values for all i with $1 \leq i \leq n$ and the stack contains the two pairs $\langle 0, 0 \rangle$ and $\langle 0, 0 \rangle$ corresponding to the boundary nodes $(0, \text{SA}[0])$ and $(n + 1, \text{SA}[n + 1])$.

Corollary 5.2.6 *The LPS array is a permutation of the array $\text{LCP}[1..n]$.*

Proof This is a direct consequence of the preceding discussion. At the beginning, the graph contains the values $LCP[1], \dots, LCP[n+1]$ as labels of its edges. When a peak is removed from the graph, the maximum of the two labels of the adjacent edges becomes an LPS-value, whereas the minimum of the two labels becomes the label of the new edge. At the end, the graph contains only the boundary nodes $(0, SA[0])$ and $(n+1, SA[n+1])$, and an edge between them with label $LCP[n+1] = 0$. That is, during the transformation of the graph, the values $LCP[1], \dots, LCP[n]$ have become LPS-values. \square

Exercise 5.2.7 Algorithm 5.10 is based on the fact that every peak, i.e., every node $(i, SA[i])$ with $SA[i-1] < SA[i] > SA[i+1]$, can safely be removed from the graph. Improve the algorithm by further considering the case $SA[i-1] < SA[i] < SA[i+1]$ and $LCP[i] \geq LCP[i+1]$. Argue that in this case $LPS[SA[i]] = LCP[i]$ and modify the pseudo-code from Algorithm 5.10 accordingly.

Exercise 5.2.8 Compute the longest previous substring array by a bottom-up traversal of the lcp-interval tree (Algorithm 4.6 on page 94).

5.2.2 Ultra-fast factorization

According to Corollary 5.2.6, if one computes the LZ-factorization via the arrays LPS and PrevOcc, the computation of lcp-values is necessary. If one is solely interested in the factorization, however, it is disadvantageous to *precompute* the LCP-array. Algorithm 5.11 intermingles the computation of lcp-values with the computation of the arrays LPS and PrevOcc: it computes these arrays by (virtually) building the above-mentioned graph in *text order* and peak elimination.

More precisely, in the main procedure it computes lcp-values as in the Φ -algorithm (Algorithm 4.4 on page 84). Suppose that the algorithm has just computed the length ℓ of the longest common prefix of some suffix S_i of S and the suffix S_j that precedes S_i in the suffix array (so among all suffixes of S that are lexicographically smaller than S_i , S_j is the largest). In terms of the graph, this means that the algorithm has just detected an edge with label ℓ between nodes i and j ; but of course it does not build the graph. Instead it stores this edge in the arrays LPS and PrevOcc. To be precise, if $i > j$, it stores ℓ in $LPS[i]$ and j in $PrevOcc[i]$. Otherwise, if $i < j$, it stores ℓ in $LPS[j]$ and i in $PrevOcc[j]$. However, collisions may occur. If, for example $i > j$, it may be the case that the memory cells $LPS[i]$ and $PrevOcc[i]$ are already occupied; say $m = LPS[i]$ and $k = PrevOcc[i]$. In terms of the graph, this means that a peak has been detected because both k and j are smaller than i . Consequently, the peak is eliminated by a case distinction: either (a) $m < \ell$ or (b) $m \geq \ell$. Let us consider case

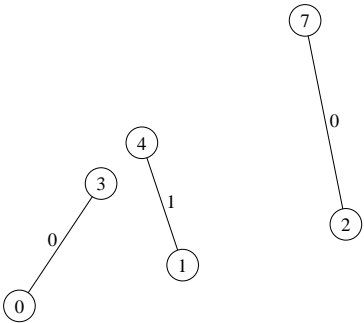
Algorithm 5.11 Computation of the arrays LPS and PrevOcc.

main procedure for $i \leftarrow 1$ to n do $\Phi[\text{SA}[i]] \leftarrow \text{SA}[i - 1]$ $\ell \leftarrow 0$ for $i \leftarrow 1$ to n do $j \leftarrow \Phi[i]$ while $S[i + \ell] = S[j + \ell]$ do $\ell \leftarrow \ell + 1$ if $i > j$ then $\text{sop}(i, \ell, j)$ else $\text{sop}(j, \ell, i)$ $\ell \leftarrow \max(\ell - 1, 0)$	procedure $\text{sop}(i, \ell, j)$ if $\text{LPS}[i] = \perp$ then $\text{LPS}[i] \leftarrow \ell$ $\text{PrevOcc}[i] \leftarrow j$ else if $\text{LPS}[i] < \ell$ then if $\text{PrevOcc}[i] > j$ then $\text{sop}(\text{PrevOcc}[i], \text{LPS}[i], j)$ else $\text{sop}(j, \text{LPS}[i], \text{PrevOcc}[i])$ $\text{LPS}[i] \leftarrow \ell$ $\text{PrevOcc}[i] \leftarrow j$ else $\quad \quad \quad /* \text{LPS}[i] \geq \ell */$ if $\text{PrevOcc}[i] > j$ then $\text{sop}(\text{PrevOcc}[i], \ell, j)$ else $\text{sop}(j, \ell, \text{PrevOcc}[i])$
---	--

(a); case (b) is similar. Since $m < \ell$, we set $\text{LPS}[i] \leftarrow \ell$ and $\text{PrevOcc}[i] \leftarrow j$ in accordance with the peak elimination procedure described above. (It should be pointed out that the entries $\text{LPS}[i]$ and $\text{PrevOcc}[i]$ are fixed from this point on.) Moreover, we have to insert a new edge between nodes j and k with label m . As we always store an edge at the index that is the maximum of its node labels, we further distinguish between the cases (i) $k < j$ and (ii) $k > j$. Again, we only consider the first case because the second case can be treated similarly. If the memory cells $\text{LPS}[j]$ and $\text{PrevOcc}[j]$ are still empty (i.e., $\text{LPS}[j] = \perp$ and $\text{PrevOcc}[j] = \perp$), then we store m in $\text{LPS}[j]$ and k in $\text{PrevOcc}[j]$. Otherwise, we have just detected another peak and we eliminate it in the same way as described above.

As an example, consider the execution of the main procedure of Algorithm 5.11 for $S = \text{acaaacatat}$ and $i = 4$. The upper part of Figure 5.8 shows the point of departure. Since $j = \Phi[4] = 3$ (cf. Figure 5.3 on page 129), S_4 is compared with S_3 and the length of the longest common prefix of S_4 and S_3 is 2. This results in the procedure call $\text{sop}(4, 2, 3)$ because $i = 4 > 3 = j$. Since $\text{LPS}[4] = 1 < 2 = \ell$ and $\text{PrevOcc}[4] = 1 < 3 = j$, there is another procedure call $\text{sop}(3, 1, 1)$. The lower part of Figure 5.8 shows the situation without the effect of $\text{sop}(3, 1, 1)$: peak node 4 is eliminated, i.e., $\text{LPS}[4]$ is set to $\ell = 2$ and $\text{PrevOcc}[4]$ is set to $j = 3$. Let us turn to the effect of the procedure call $\text{sop}(3, 1, 1)$: $\text{LPS}[3] = 0 < 1$ and $\text{PrevOcc}[3] = 0 < 1$ result in another procedure call $\text{sop}(1, 0, 0)$. The upper part of Figure 5.9 depicts the situation without the effect of $\text{sop}(1, 0, 0)$ (peak node 3 is eliminated),

$S[i]$	a	c	a	a	a	c	a	t	a	t
i	1	2	3	4	5	6	7	8	9	10
$LPS[i]$			0	1			0			
$PrevOcc[i]$			0	1			2			



$S[i]$	a	c	a	a	a	c	a	t	a	t
i	1	2	3	4	5	6	7	8	9	10
$LPS[i]$			0	2			0			
$PrevOcc[i]$			0	3			2			

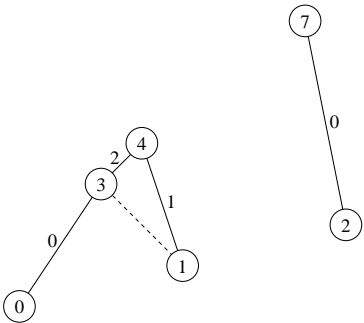
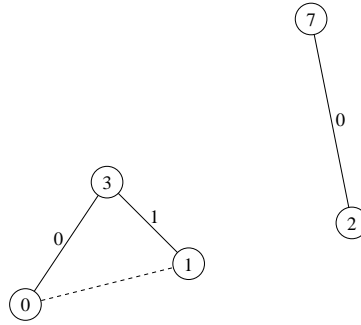


Figure 5.8: $i = 4$: Detection and removal of the first peak in the graph.

$S[i]$	a	c	a	a	a	c	a	t	a	t
i	1	2	3	4	5	6	7	8	9	10
$LPS[i]$			1	2			0			
$PrevOcc[i]$			1	3			2			



$S[i]$	a	c	a	a	a	c	a	t	a	t
i	1	2	3	4	5	6	7	8	9	10
$LPS[i]$	0		1	2			0			
$PrevOcc[i]$	0		1	3			2			

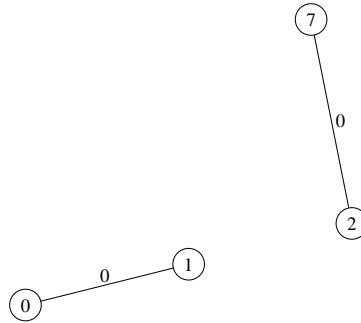


Figure 5.9: $i = 4$: Detection and removal of the second peak in the graph.

while the lower part shows the final situation (after the execution of the main procedure for $i = 4$).

Algorithm 5.11 runs in linear time. This is because the main procedure without the calls to the procedure *sop* takes $O(n)$ time (cf. Section 4.2.1) and an amortized analysis shows that there are at most $2n$ calls to the procedure *sop*.

Exercise 5.2.9 Setting $j = \text{SA}[i]$ in Lemma 5.2.5 yields the equality

$$\text{LPS}[j] = \max\{|\text{lcp}(S_{\text{SA}[\text{PSV}[\text{ISA}[j]]]}, S_j)|, |\text{lcp}(S_j, S_{\text{SA}[\text{NSV}[\text{ISA}[j]]]})|\}$$

- Modify Algorithm 4.5 (page 93) in such a way that it computes arrays PREV and NEXT defined by $\text{PREV}[j] = \text{SA}[\text{PSV}[\text{ISA}[j]]]$ and $\text{NEXT}[j] = \text{SA}[\text{NSV}[\text{ISA}[j]]]$.
- Give an algorithm that computes the array LPS using the arrays PREV and NEXT and the equality above. The algorithm shall determine $|\text{lcp}(S_{\text{PREV}[j]}, S_j)|$ and $|\text{lcp}(S_j, S_{\text{NEXT}[j]})|$ by comparing the substrings, character by character. Note that the arrays PREV and NEXT are accessed sequentially, so the algorithm has a good locality of reference.
- Modify the previous algorithm so that it directly computes the LZ-factorization.

It goes beyond the scope of this book to review the history of compression algorithms based on the ideas of Abraham Lempel and Jacob Ziv. Suffice it to say that Ziv and Lempel published two seminal papers on lossless data compression algorithms in 1977 and 1978 [334, 335], commonly known as LZ77 and LZ78. The LZ-factorization presented above can be viewed as a variant of the LZ77 method, but it is not the same as the method proposed originally in [334]. LZ-factorization algorithms that rely on the suffix array and the LCP-array include [1, 56, 65–67]. The presentation of Section 5.2.1 follows the approach in [65], while Section 5.2.2 originates from [250]. The reader can find more material on the topic in Section 7.5.6.

5.3 Finding repeats

In the analysis of a genome, a basic task is to locate and characterize the repetitive sequences (repeats). While bacterial genomes usually do not contain large amounts of repetitive sequences, a considerable portion of the genomes of higher organisms is composed of repeats. For example, more than half of the 3 billion basepairs of the (haploid) human genome is composed of repeats [161]. The repetitive elements of the human genome can be classified into two large groups: dispersed repetitive DNA and

tandemly repeated DNA. Dispersed repetitions vary in size and content and fall into two basic categories: transposable elements and segmental duplications [161]. Transposable elements belong to one of the following four classes: SINEs (short interspersed nuclear elements), LINEs (long interspersed nuclear elements), LTRs (long terminal repeats), and transposons. Segmental duplications, which might contain complete genes, have been divided into two classes: those that are chromosome-specific and those that have trans-chromosome duplications [255].

Tandem repeats (those that are found directly adjacent to one another) can also be classified into two categories: simple sequence repetitions (relatively short strings such as microsatellites) and larger ones, which are called blocks of tandemly repeated segments.

Clearly, one needs extensive algorithmic support for a systematic study of repetitive DNA on a genomic scale. References to software tools for finding repeats in genome sequences can be found in [142]. In this section, we present basic algorithms for this task. We start with the definitions of the different types of repeats.

Definition 5.3.1 A non-empty substring ω of S is a *repeat* if it occurs at least twice in S .

As an example, consider the string $S = ctaataatg$. The substring $taat$ is a repeat of length 4, aat and taa are repeats of length 3, etc.

Definition 5.3.2 A repeat ω of S is a *maximal repeat* if any extension of ω occurs fewer times in S than ω .

In our example, the substring $taat$ is a maximal repeat of the string $S = ctaataatg$ but the substrings aat and taa are not maximal repeats. For instance, if we extend the substring aat by the character t to the left, then the resulting substring $taat$ occurs as often in S as aat .

Definition 5.3.3 A repeat ω of S is a *supermaximal repeat* if ω is not a proper substring of another repeat.

As an example, consider the string $S = aacaaccac$. The supermaximal repeats of length ≥ 2 are aac and ca . Note that ac is not a supermaximal repeat because it is a proper substring of the repeat aac . Exercise 5.3.4 asks you to prove that every supermaximal repeat is also a maximal repeat.

Exercise 5.3.4 Show that a supermaximal repeat ω must necessarily be a maximal repeat.

Articles on finding the above-mentioned kinds of repeats include [1, 31, 114, 139, 235, 260, 263, 265]. A *tandem repeat* consists of two occurrences of the same substring that are directly adjacent to each other. For example, $aacaac$ is a tandem repeat of $S = aacaaccac$. We will treat tandemly repeated substrings in Section 5.3.6.

i	SA	LCP	$S[SA[i] - 1]$	$S_{SA[i]}$	lcp-intervals		
1	3	-1	t	$aataatg$	0	1	3
2	6	3	t	$aatg$			
3	4	1	a	$ataatg$			2
4	7	2	a	atg			
5	1	0	$\$$	$ctaataatg$		1	4
6	9	0	t	g			
7	2	0	c	$taataatg$			
8	5	4	a	$taatg$			
9	8	1	a	tg			
10		-1					

Figure 5.10: Enhanced suffix array and lcp-intervals of $S = ctaataatg$.

5.3.1 Longest repeats

Lemma 5.3.5 *Let m be the maximum value in the LCP-array of S . A non-empty substring ω of S is a longest repeat if and only if the ω -interval $[i..j]$ is an lcp-interval of lcp-value $|\omega| = m$.*

Proof “if”: If the ω -interval is a (non-singleton) lcp-interval of lcp-value m , then ω is a repeat of length m . It is a longest repeat because there cannot be a repeat of length greater than m (otherwise m would not be the maximum value in the LCP-array).

“only if”: Suppose that ω is a longest repeat. If $|\omega|$ were not the maximum value in the LCP-array, then there would be a longer repeat. So we must have $|\omega| = m$. Because ω is a repeat, the ω -interval $[i..j]$ satisfies $i < j$ and $LCP[k] \geq m$ for all k with $i < k \leq j$. Now the lemma follows from the fact that $LCP[k] > m$ is impossible. \square

Lemma 5.3.5 can be rephrased as follows: ω is a longest repeat if and only if $|\omega| = m$ and the ω -interval $[i..j]$ satisfies $i < j$ and $LCP[k] = m$ for all k with $i < k \leq j$.

For example, the maximum value in the LCP-array of $S = ctaataatg$ is 4; see Figure 5.10. Consequently, all longest repeats in S must have length 4. There is only one lcp-interval of lcp-value 4, namely the interval $[7..8]$. Hence there is only one longest repeat in S . Because the lcp-interval 4 - $[7..8]$ represents $taat$, the string $taat$ is the longest repeat in S .

The simple computation of all longest repeats can be found in Algorithm 5.12. Note that the maximum entry m in the LCP-array of S can be obtained as a by-product of the construction of the LCP-array or by a linear scan of the LCP-array. It follows as a consequence that Algorithm 5.12

Algorithm 5.12 Computation of all longest repeats.

```

 $m \leftarrow \max\{\text{LCP}[i] \mid 1 \leq i \leq n\}$ 
for  $i \leftarrow 1$  to  $n$  do
  if  $\text{LCP}[i] = m$  and  $\text{LCP}[i+1] \neq m$  then /* avoids redundant output */
    output  $S[\text{SA}[i]..\text{SA}[i] + m - 1]$ 

```

runs in $O(n)$ time. To prove the correctness of the algorithm, consider an index i with $\text{LCP}[i] = m$ and $\text{LCP}[i+1] \neq m$. Since m is the maximum entry in the LCP-array, it follows that $\text{LCP}[i+1] < m$. In other words, i is the right boundary of an m -interval. By Lemma 5.3.5, the string $S[\text{SA}[i]..\text{SA}[i] + m - 1]$ is a longest repeat. Because Algorithm 5.12 considers all m -intervals, it outputs all longest repeats.

At first glance, the worst-case time complexity of Algorithm 5.12 seems to be $O(n)$. Quite surprisingly, however, this is not true because the total length of all longest repeats of a string may be proportional to $n \log n$. To see this, we need a de Bruijn sequence. In combinatorial mathematics, a *de Bruijn sequence* (named after the Dutch mathematician Nicolaas Govert de Bruijn) of order k on the alphabet Σ is a cyclic string that contains each string from Σ^k exactly once as a substring. For example, the cyclic string 0000100110101111 is a de Bruijn sequence of order 4 on the binary alphabet $\{0, 1\}$. Figure 5.11 shows how cyclic and non-cyclic de Bruijn sequences can be constructed. Several other algorithms that construct de Bruijn sequences can be found in [182].

Lemma 5.3.6 *The total length of all longest repeats in a non-cyclic binary de Bruijn sequence S of order k is $\Omega(n \log n)$, where $n = 2^k + k - 1$ is the length of S .*

Proof A de Bruijn sequence of order k on the alphabet $\Sigma = \{0, 1\}$ contains each string from Σ^k exactly once as a substring. In other words, all longest repeats must have a length smaller than k . Let $\omega \in \Sigma^{k-1}$. Since both $\omega 0$ and $\omega 1$ are substrings of S , ω is a repeat. In fact, it is a longest repeat because it has length $k - 1$ and all longest repeats have a length $< k$. This shows that all 2^{k-1} binary strings of length $k - 1$ are longest repeats; see Figure 5.12. It follows as a consequence that the total length of all longest repeats in S equals $2^{k-1}(k - 1)$. Since $n = 2^k + k - 1$, or equivalently, $2^{k-1} = \frac{n-k+1}{2}$, we derive

$$2^{k-1}(k - 1) = \frac{n - k + 1}{2} \log_2 \left(\frac{n - k + 1}{2} \right)$$

In conjunction with $k \leq \frac{n}{2}$, this proves the lemma. \square

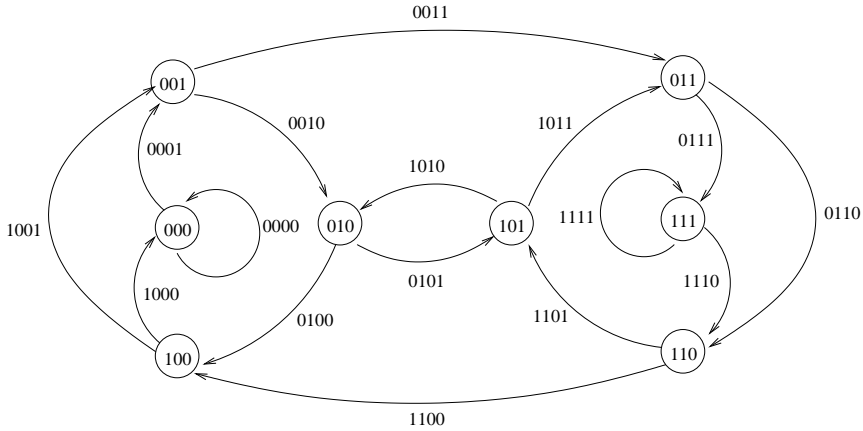


Figure 5.11: In the de Bruijn graph for k and the alphabet Σ (here $k = 4$ and $\Sigma = \{0, 1\}$), there is a node for each string from Σ^{k-1} . Furthermore, the graph has an edge (u, v) if there is a string $w \in \Sigma^k$ so that u is a prefix and v is a suffix of w . In this case, the edge (u, v) is labeled with w . The de Bruijn graph has an Eulerian cycle (a cycle that traverses each edge of the graph exactly once) because each node has indegree and out-degree $|\Sigma|$; see Exercise 5.3.7. In the graph above, the path 0000, 0001, 0010, 0100, 1001, 0011, 0110, 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000 is an Eulerian cycle. The concatenation of the first characters of the edge labels spells out the cyclic string 0000100110101111. By construction, the cyclic string ω obtained from an Eulerian cycle has length $|\Sigma|^k$, and it contains each length k string on Σ exactly once as a substring. Hence it is a de Bruijn sequence of order k on the alphabet Σ . Note that a non-cyclic de Bruijn sequence of length $|\Sigma|^k + k - 1$ can be obtained from a cyclic de Bruijn sequence ω by appending the length $k - 1$ prefix $\omega[1..k-1]$ to it. In our example, the non-cyclic de Bruijn sequence 0000100110101111000 is obtained from the cyclic de Bruijn sequence 0000100110101111 by appending 000 to it.

i	SA	LCP	$S[SA[i-1]]$	$S_{SA[i]}$
1	19	-1	0	0
2	18	1	0	00
3	17	2	1	000
4	1	3	\$	0000100110101111000
5	2	3	0	0001001101011111000
6	3	2	0	00100110101111000
7	6	3	1	001101011111000
8	4	1	0	01001101011111000
9	10	3	1	0101111000
10	7	2	0	01101011111000
11	12	3	1	01111000
12	16	0	1	1000
13	5	3	0	100110101111000
14	9	2	1	10101111000
15	11	3	0	101111000
16	15	1	1	11000
17	8	3	0	110101111000
18	14	2	1	111000
19	13	3	0	1111000
20		-1		

Figure 5.12: Suffix array and LCP-array of the non-cyclic binary de Bruijn sequence $S = 0000100110101111000$ of order 4.

Let us resume the analysis of the worst-case time complexity of Algorithm 5.12. Its run time is $O(n + z)$, where z is the size of the output, but z is not bounded by $O(n)$. In other words, the run time of Algorithm 5.12 depends not only on the size of the input but also on the size of the output. An algorithm with this property is called *output-sensitive* algorithm. If the output size varies widely, for example from linear in the size of the input to quadratic in the size of the input, then analyses that take the output size explicitly into account can produce better run time bounds that differentiate algorithms that would otherwise have identical asymptotic complexity. For example, an algorithm with worst-case time complexity $O(n + z)$, where z can (but must not) be proportional to n^2 , is better than an algorithm with a run time $\Theta(n^2)$. The latter always runs in quadratic time, whereas the former runs in linear time whenever the output size is linear.

Of course, Algorithm 5.12 can be turned into a linear-time algorithm. If it just reports that a longest repeat of length m starts at position $SA[i]$ (so instead of the string $S[SA[i]..SA[i] + m - 1]$, an implicit representation

thereof is output), then its worst-case time complexity is $O(n)$ because at most one longest repeat can start at any (fixed) position in S .

Exercise 5.3.7 An Eulerian cycle of a connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

- Show that G has an Eulerian cycle if and only if $\text{indegree}(v) = \text{outdegree}(v)$ for each node $v \in V$.
- Describe an $O(|E|)$ time algorithm to find an Eulerian cycle of G if one exists.

Exercise 5.3.8 Draw the de Bruijn graph for $k = 2$ and the alphabet $\Sigma = \{a, c, g, t\}$. Find an Eulerian cycle in the graph and write down the corresponding de Bruijn sequence.

Exercise 5.3.9 Show that a longest repeat ω must necessarily be a supermaximal repeat (hence a maximal repeat).

5.3.2 Supermaximal repeats

Lemma 5.3.11 provides a characterization of supermaximal repeats that directly leads to a linear-time algorithm to find all of them.

Definition 5.3.10 An lcp-interval $\ell\text{-}[i..j]$ is called a *local maximum* in the LCP-array if $\text{LCP}[k] = \ell$ for all k with $i + 1 \leq k \leq j$.

It is rather obvious that the local maxima in the LCP-array are the leaves of the lcp-interval tree *without* singleton intervals. For instance, in the LCP-array of Figure 5.10 (page 140), the local maxima are the intervals $[1..2]$, $[3..4]$, and $[7..8]$.

Lemma 5.3.11 A non-empty substring ω of S is a supermaximal repeat if and only if

- the ω -interval $[i..j]$ is an lcp-interval of lcp-value $\ell = |\omega|$ that is a local maximum in the LCP-array, and
- the characters $S[\text{SA}[i] - 1], S[\text{SA}[i + 1] - 1], \dots, S[\text{SA}[j] - 1]$ are pairwise distinct.¹

¹To cope with the case $\text{SA}[i] = 1$, we tacitly assume that $S[0] = \$$, where $\$$ is a character that does not occur in S .

Proof “if”: Since ω is a common prefix of the suffixes $S_{SA[i]}, \dots, S_{SA[j]}$ and $i < j$, it is a repeat of length ℓ . The characters $S[SA[i] + \ell], S[SA[i + 1] + \ell], \dots, S[SA[j] + \ell]$ are pairwise distinct because $[i..j]$ is a local maximum in the LCP-array. By the second condition, the characters $S[SA[i] - 1], S[SA[i + 1] - 1], \dots, S[SA[j] - 1]$ are also pairwise distinct. It follows that there is no repeat in S that contains ω . In other words, ω is a supermaximal repeat.

“only if”: Suppose ω is a supermaximal repeat of length $|\omega| = \ell$. Let $[i..j]$ be the ω -interval, i.e., $SA[i], SA[i + 1], \dots, SA[j]$, $1 \leq i < j \leq n$, are the consecutive entries in SA so that ω is a common prefix of $S_{SA[i]}, S_{SA[i+1]}, \dots, S_{SA[j]}$, but neither of $S_{SA[i-1]}$ nor of $S_{SA[j+1]}$. Because ω is supermaximal, the characters $S[SA[i] + \ell], S[SA[i + 1] + \ell], \dots, S[SA[j] + \ell]$ are pairwise distinct. Hence $LCP[k] = \ell$ for all k with $i + 1 \leq k \leq j$. Furthermore, $LCP[i] < \ell$ and $LCP[j + 1] < \ell$ hold because $[i..j]$ is the ω -interval (otherwise ω would also be a prefix of $S_{SA[i-1]}$ or $S_{SA[j+1]}$). All in all, $[i..j]$ is a local maximum of the array LCP . Finally, the characters $S[SA[i] - 1], S[SA[i + 1] - 1], \dots, S[SA[j] - 1]$ are pairwise distinct because ω is supermaximal. \square

In view of Lemma 5.3.11, we say that a local maximum $\ell\text{-}[i..j]$ in the LCP-array *induces* a supermaximal repeat if the string $\omega = S[SA[i]..SA[i] + \ell - 1]$ is a supermaximal repeat. According to Lemma 5.3.11, we can compute all supermaximal repeats of a string S as follows:

- (a) Find all local maxima in the LCP-array of S .
- (b) For every local maximum $\ell\text{-}[i..j]$ check whether $S[SA[i] - 1], S[SA[i + 1] - 1], \dots, S[SA[j] - 1]$ are pairwise distinct characters. If so, report that $\ell\text{-}[i..j]$ induces a supermaximal repeat.

Note that the output size of the algorithm is $O(n)$ because the number of local maxima (hence the number of supermaximal repeats) is smaller than n . By contrast, if the algorithm would output all supermaximal repeats explicitly, then the output size would *not* be linear (combine Lemma 5.3.6 with Exercise 5.3.9).

We claim that there is a procedure call $superMax(\langle \ell, lb, k - 1 \rangle)$ in Algorithm 5.13 only if $\ell\text{-}[lb..k - 1]$ is a local maximum. Exercise 5.3.12 asks you to prove that Algorithm 5.13 does not miss a local maximum (i.e., if $\ell\text{-}[i..j]$ is a local maximum, then the procedure $superMax(\langle \ell, i, j \rangle)$ will be called). Altogether, this implies the correctness of Algorithm 5.13.

To prove our claim, we show that the for-loop of Algorithm 5.13 keeps the invariant $locMax = true$ if and only if the sequence $LCP[lb], LCP[lb + 1], \dots, LCP[k]$ satisfies $LCP[lb] < LCP[lb + 1] = \dots = LCP[k]$, where k is the loop-variable. For each k , $2 \leq k \leq n + 1$, the algorithm compares the lcp-value $m = LCP[k]$ with the previous lcp-value $\ell = LCP[k - 1]$ and distinguishes three cases (a) $m > \ell$, (b) $m < \ell$. and (c) $m = \ell$.

(a) If $m > \ell$, then lb is set to $k - 1$ because a potential local maximum starts

Algorithm 5.13 Computation of local maxima.

```

 $\ell \leftarrow -1$       /*  $LCP[1] = -1$  */
 $locMax \leftarrow false$ 
for  $k \leftarrow 2$  to  $n + 1$  do
     $m \leftarrow LCP[k]$ 
    if  $m > \ell$  then
         $lb \leftarrow k - 1$ 
         $locMax \leftarrow true$ 
    else if  $m < \ell$  and  $locMax = true$  then
        /*  $\ell[lb..k - 1]$  is a local maximum */
         $superMax(\langle \ell, lb, k - 1 \rangle)$ 
         $locMax \leftarrow false$ 
     $\ell \leftarrow m$ 

```

at this index and the Boolean variable $locMax$ is set to *true*. Note that the invariant holds because $locMax = true$ and $LCP[lb] = LCP[k - 1] < LCP[k]$. (In particular, after the first iteration of the for-loop, we have $locMax = true$ and $LCP[lb] = LCP[1] = -1 < LCP[2] = LCP[k]$.)

(b) If $m < \ell$, then there are two subcases. Either (i) $locMax = true$ or (ii) $locMax = false$. In case (i), we have $LCP[lb] < LCP[lb + 1] = \dots = LCP[k - 1]$ by the loop-invariant. In combination with $LCP[k - 1] = \ell > m = LCP[k]$, this implies that $\ell[lb..k - 1]$ is a local maximum. Thus Algorithm 5.13 tests whether this local maximum induces a supermaximal repeat. Furthermore, it sets $locMax$ to *false*. Note that $LCP[lb] < LCP[lb + 1] = \dots = LCP[k - 1] = LCP[k]$ does *not* hold because $LCP[k - 1] > LCP[k]$. Therefore, the invariant is satisfied. In case (ii), we have $locMax = false$ and nothing is done. As in case (i), we conclude that the invariant holds.

(c) If $m = \ell$, then nothing is done. It is readily verified that the invariant holds in this case, too.

Exercise 5.3.12 Let $\ell[i..j]$ be an arbitrary but fixed local maximum. Show that Algorithm 5.13 calls the procedure $superMax(\langle \ell, i, j \rangle)$.

For each local maximum $\ell[i..j]$, the procedure $superMax$ must test if it induces a supermaximal repeat. In principle, this rather simple problem can be solved as follows:

- Initialize a bit array B of size σ containing a series of zeros.
- Scan through the sequence $S[SA[i] - 1], S[SA[i + 1] - 1], \dots, S[SA[j] - 1]$ of characters, and for each character c encountered do:
 - if $B[c] = 0$ (c has not been seen before), set $B[c]$ to 1,
 - if $B[c] = 1$ (c has been seen before), stop the scan.

Algorithm 5.14 Procedure *superMax*($\langle \ell, i, j \rangle$) tests whether the lcp-interval $\ell[i..j]$ induces a supermaximal repeat. It uses two global variables: a bit array B of size σ initially containing a series of zeros and an initially empty *list*.

```

pd  $\leftarrow$  true
k  $\leftarrow$  i
while k  $\leq$  j and pd = true do
    c  $\leftarrow$   $S[SA[k] - 1]$ 
    if  $B[c] = 0$  then
         $B[c] \leftarrow 1$ 
        add(list, c)
        k  $\leftarrow$  k + 1
    else
        pd  $\leftarrow$  false
for each c in list do          /* reset B and list */
     $B[c] \leftarrow 0$ 
    remove(list, c)
if pd = true then
    report that  $\ell[i..j]$  induces a supermaximal repeat

```

- If the scan was not stopped, report that $\ell[i..j]$ induces a supermaximal repeat.

There is a caveat, though. The run time of this algorithmic solution depends on the alphabet size σ . The bit array B must be initialized for each local maximum, and the time to do this is proportional to σ . Because the number of local maxima can be proportional to n (see Exercise 5.3.13), the time to compute all supermaximal repeats can be proportional to $n\sigma$. However, it is not difficult to obtain an alphabet independent run-time of $O(n)$. To this end, Algorithm 5.14 uses two global variables: the bit array B and an extra list that is used to record which characters have appeared during the scan of the interval $[i..j]$. Note that the size of the list cannot exceed the size of $[i..j]$ (nor σ). After the scan, the list is used to reset B in such a way that it again contains a series of zeros.

Exercise 5.3.13 Show that the number of local maxima in the LCP-array of a non-cyclic binary de Bruijn sequence S of order k is $\Omega(n)$, where $n = 2^k + k - 1$ is the length of S .

Exercise 5.3.14 Prove that the combination of Algorithms 5.13 and 5.14 applied to a string of length n reports (implicit representations of) all supermaximal repeats in a worst-case time complexity of $O(n)$.

5.3.3 Maximal repeats

We start with a characterization of maximal repeats that will help us to develop a linear-time algorithm to find them all.

Lemma 5.3.15 *A non-empty substring ω of S is a maximal repeat if and only if*

- *the ω -interval $[i..j]$ is an lcp-interval of lcp-value $\ell = |\omega|$, and*
- *the characters $S[\text{SA}[i] - 1], S[\text{SA}[i + 1] - 1], \dots, S[\text{SA}[j] - 1]$ are not all the same.*²

Proof “if”: Since ω is a common prefix of the suffixes $S_{\text{SA}[i]}, \dots, S_{\text{SA}[j]}$ and $i < j$, it is a repeat of length ℓ . The characters $S[\text{SA}[i] + \ell], S[\text{SA}[i + 1] + \ell], \dots, S[\text{SA}[j] + \ell]$ are not all the same because $[i..j]$ is an lcp-interval. Thus, if we extend ω by one character to the right, the resulting string occurs less often in S than ω . Analogously, if we extend ω by one character to the left, then the resulting string occurs less often in S than ω because the characters $S[\text{SA}[i] - 1], S[\text{SA}[i + 1] - 1], \dots, S[\text{SA}[j] - 1]$ are not all the same. Consequently, ω is a maximal repeat.

“only if”: Suppose ω is a maximal repeat. Let $[i..j]$ be the ω -interval, i.e., $\text{SA}[i], \text{SA}[i + 1], \dots, \text{SA}[j]$ are the consecutive entries in SA so that ω is a common prefix of $S_{\text{SA}[i]}, S_{\text{SA}[i+1]}, \dots, S_{\text{SA}[j]}$, but neither of $S_{\text{SA}[i-1]}$ nor of $S_{\text{SA}[j+1]}$. First, $[i..j]$ is a non-singleton interval because ω is a repeat. Second, we have $\text{LCP}[k] \geq \ell$ for all k with $i + 1 \leq k \leq j$ and $\text{LCP}[i] < \ell$ as well as $\text{LCP}[j + 1] < \ell$. Moreover, because ω is maximal, the characters $S[\text{SA}[i] + \ell], S[\text{SA}[i + 1] + \ell], \dots, S[\text{SA}[j] + \ell]$ are not all the same. Hence $\text{LCP}[k] = \ell$ for a k with $i + 1 \leq k \leq j$. All in all, $[i..j]$ is a non-singleton lcp-interval of lcp-value $\ell = |\omega|$. Finally, the characters $S[\text{SA}[i] - 1], S[\text{SA}[i + 1] - 1], \dots, S[\text{SA}[j] - 1]$ are not all the same because ω is maximal. \square

In view of Lemma 5.3.15, we say that an lcp-interval ℓ - $[i..j]$ induces a maximal repeat if the string $\omega = S[\text{SA}[i].. \text{SA}[i] + \ell - 1]$ is a maximal repeat.

According to Lemma 5.3.15, maximal repeats can be computed by a bottom-up traversal of the lcp-interval tree (Algorithm 4.6 on page 94). If an lcp-interval $\text{lastInterval} = [lb..rb]$ of lcp-value ℓ is popped from the stack in the while-loop of Algorithm 4.6, then we have $rb = k - 1$, where k is the current value of the loop variable. The procedure *process* must report that lastInterval induces a maximal repeat (namely $S[\text{SA}[lb].. \text{SA}[lb] + \ell - 1]$) if and only if the characters $S[\text{SA}[lb] - 1], S[\text{SA}[lb + 1] - 1], \dots, S[\text{SA}[rb] - 1]$ are not all the same. This test can be done by keeping track of the largest index $\text{lastdiff} < k$ at which $S[\text{SA}[\text{lastdiff} - 1] - 1]$ and $S[\text{SA}[\text{lastdiff}] - 1]$ differ (initially

²To cope with the case $\text{SA}[i] = 1$, we tacitly assume that $S[0] = \$$, where $\$$ is a character that does not occur in S .

Algorithm 5.15 Computation of maximal repeats by a bottom-up traversal of the lcp-interval tree.

```

push( $\langle 0, 1 \rangle$ )
lastdiff  $\leftarrow 1$ 
for  $k \leftarrow 2$  to  $n + 1$  do
     $lb \leftarrow k - 1$ 
    while  $LCP[k] < top().lcp$  do
         $lastInterval \leftarrow pop()$ 
         $lb \leftarrow lastInterval.lb$ 
         $\ell \leftarrow lastInterval.lcp$ 
        if  $lastdiff > lb$  then
            report that  $\ell[lb..k - 1]$  induces a maximal repeat
        if  $LCP[k] > top().lcp$  then
            push( $\langle LCP[k], lb \rangle$ )
        if  $S[SA[k - 1] - 1] \neq S[SA[k] - 1]$  then
             $lastdiff \leftarrow k$ 

```

$lastdiff = 1$). Since $lastdiff \leq rb$, the characters $S[SA[lb] - 1], S[SA[lb + 1] - 1], \dots, S[SA[rb] - 1]$ are not all the same if and only if $lastdiff > lb$.

Pseudo-code for the computation of maximal repeats by this approach can be found in Algorithm 5.15. In contrast to Algorithm 4.6, the elements on the stack are just pairs $\langle lcp, lb \rangle$, where lcp is the lcp-value of the interval and lb is its left boundary. This is because the algorithm does not need child information and the right boundary of an interval that is processed is $rb = k - 1$. Algorithm 5.15 runs in linear time because there are at most $n - 1$ lcp-intervals. If the algorithm would output all maximal repeats explicitly, then its worst-case time complexity would be $O(n + z)$, where the output size z can be quadratic (this happens e.g. for the input $S = a^n$, which consists of n copies of the character a).

5.3.4 Maximal repeated pairs

In this section, we address the problem of finding dispersed repeats in a string S . The algorithm we are going to present was first discovered and published by Baker [26]; see also [1, 139].

Definition 5.3.16 Let us represent the occurrence of the substring $S[i..j]$ of S by the pair (i, j) . A *repeated pair* $\langle (i, j), (i', j') \rangle$ is a pair of two different occurrences (i, j) and (i', j') of the same substring $S[i..j] = S[i'..j']$ of S , where $i < i'$. (i, j) is called the *left instance* of the repeated pair and (i', j') is called the *right instance*.



Figure 5.13: If $S[i..j] = S[i'..j']$ and $i < i'$, then $\langle (i, j), (i', j') \rangle$ is a repeated pair. It is maximal if and only if $S[i-1] = a \neq c = S[i'-1]$ and $S[j+1] = b \neq d = S[j'+1]$.

$\langle (3, 6), (8, 11) \rangle$	<i>gcgc</i>	<i>maximal</i>
$\langle (3, 5), (8, 10) \rangle$	<i>gcg</i>	
$\langle (4, 6), (9, 11) \rangle$	<i>cgc</i>	
$\langle (3, 4), (8, 9) \rangle$	<i>gc</i>	
$\langle (4, 5), (9, 10) \rangle$	<i>cg</i>	
$\langle (5, 6), (10, 11) \rangle$	<i>gc</i>	
$\langle (3, 4), (5, 6) \rangle$	<i>gc</i>	<i>maximal</i>
$\langle (3, 4), (10, 11) \rangle$	<i>gc</i>	<i>maximal</i>
$\langle (5, 6), (8, 9) \rangle$	<i>gc</i>	<i>maximal</i>
$\langle (8, 9), (10, 11) \rangle$	<i>gc</i>	<i>maximal</i>

Figure 5.14: All repeated pairs of the string $S = aggcgctgcgcc$ that have at least two characters.

Note that the left instance and the right instance of a repeated pair may overlap. Although this definition seems reasonable at first glance, a closer look reveals that it may lead to a huge number of repeated pairs: consider for example the string $S = a^n$. To avoid redundant repeated pairs, we confine ourselves to *maximal repeated pairs*.

Definition 5.3.17 A repeated pair $\langle (i, j), (i', j') \rangle$ is *left maximal* if $i = 1$ or $S[i-1] \neq S[i'-1]$, and it is *right maximal* if $j' = n$ or $S[j+1] \neq S[j'+1]$. A repeated pair is *maximal* if it is left maximal and right maximal.

Figure 5.13 illustrates the notions, while Figure 5.14 lists all repeated pairs of the string $S = aggcgctgcgcc$ that have a length ≥ 2 . The maximal repeated pairs are marked in the last column. Note that non-maximal repeated pairs can easily be obtained from the maximal repeated pairs. For example, the five non-maximal repeated pairs in Figure 5.14 can be derived from the first maximal repeated pair by the deletion of one or two characters.

We now develop the algorithm that computes all maximal repeated pairs and demonstrate how it operates on the string $S = aggcgctgcgcc$. Let $[i..j]$

be an ℓ -interval and $\omega = S[SA[i]..SA[i] + \ell - 1]$. Define

$$\mathcal{P}_{[i..j]} = \{SA[k] \mid i \leq k \leq j\}$$

i.e., $\mathcal{P}_{[i..j]}$ is the set of all positions p in S so that ω is a prefix of S_p . We divide $\mathcal{P}_{[i..j]}$ into disjoint and possibly empty sets according to the characters to the left of each position: For any $a \in \Sigma$ define

$$\mathcal{P}_{[i..j]}(a) = \{p \mid p \in \mathcal{P}_{[i..j]} \text{ and } S[p-1] = a\}$$

The algorithm computes these sets of positions by a bottom-up traversal of the lcp-interval tree. This means that the lcp-interval $[i..j]$ is processed after all its child intervals have been processed already.

Suppose that $[i..j]$ is a singleton interval, i.e., $i = j$. In this case, the preceding definitions imply $\mathcal{P}_{[i..i]} = \{p\}$, where $p = SA[i]$, and

$$\mathcal{P}_{[i..i]}(a) = \begin{cases} \{p\} & \text{if } S[p-1] = a \\ \emptyset & \text{otherwise} \end{cases}$$

The lcp-interval tree of the string $S = aggcgctgcgcc$ is shown in Figure 5.15. It is annotated with the non-empty sets of positions at its singleton intervals.

Now suppose that $i < j$. For each $a \in \Sigma$, $\mathcal{P}_{[i..j]}(a)$ is computed step by step while processing the child intervals of $[i..j]$. These are processed from left to right. Suppose that they are numbered, and that we have already processed q child intervals of $[i..j]$. By $\mathcal{P}_{[i..j]}^q(a)$ we denote the subset of $\mathcal{P}_{[i..j]}(a)$ obtained after processing the q -th child interval of $[i..j]$. Let $[i'..j']$ be the $(q+1)$ -th child interval of $[i..j]$. Due to the bottom-up strategy, $[i'..j']$ has been processed and hence the set of positions $\mathcal{P}_{[i'..j']}(b)$ is available for any $b \in \Sigma$.

The interval $[i'..j']$ is processed in the following way: First, repeated pairs are output by combining the sets of positions $\mathcal{P}_{[i..j]}^q(a)$ and $\mathcal{P}_{[i'..j']}(b)$. To be more precise, for all $p \in \mathcal{P}_{[i..j]}^q(a)$ and $p' \in \mathcal{P}_{[i'..j']}(b)$ with $a \neq b$, we output $\langle(p, p + \ell - 1), (p', p' + \ell - 1)\rangle$ if $p < p'$ and $\langle(p', p' + \ell - 1), (p, p + \ell - 1)\rangle$ if $p' < p$.

It is clear that $\langle(p, p + \ell - 1), (p', p' + \ell - 1)\rangle$ is indeed a repeated pair because $S[p..p + \ell - 1] = \omega = S[p'..p' + \ell - 1]$. By construction, only those positions p and p' are combined for which the characters immediately to the left, i.e., at positions $p - 1$ and $p' - 1$, are different. This guarantees left-maximality of the reported repeated pairs. Moreover, the set of positions $\mathcal{P}_{[i..j]}^q(a)$ was inherited from child intervals of $[i..j]$ that are different from $[i'..j']$. Hence the characters immediately to the right of ω at positions $p + \ell$ and $p' + \ell$ (if they exist) are different. As a consequence, the reported repeated pairs are also right-maximal (hence maximal).

Once the maximal repeated pairs for the current child interval $[i'..j']$ have been output, the union $\mathcal{P}_{[i..j]}^{q+1}(c) = \mathcal{P}_{[i..j]}^q(c) \cup \mathcal{P}_{[i'..j']}(c)$ is computed for all $c \in \Sigma$. That is, $[i..j]$ inherits the sets of positions from $[i'..j']$.

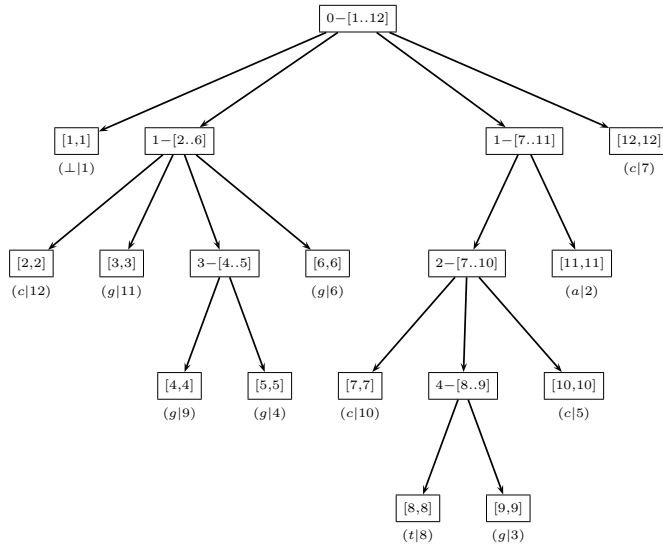


Figure 5.15: The lcp-interval tree of the string $S = aggcgctgcgcc$ with position sets at singleton intervals. Only non-empty sets of positions are shown, i.e., if $\mathcal{P}_{[i..i]}(a) = \{SA[i]\}$, then this is displayed as $(a|SA[i])$.

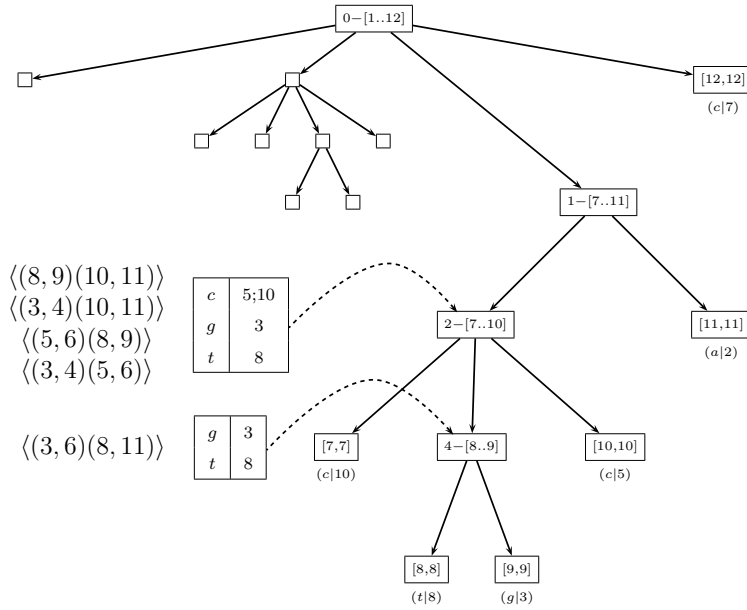


Figure 5.16: The annotation for the right part of the lcp-interval tree of Figure 5.15 and some repeated pairs.

Figure 5.16 exemplifies the computation of maximal repeated pairs by a bottom-up traversal of the lcp-interval tree of the string *aggcgctgcgcc*. Suppose that the traversal has reached node $2-[7..10]$, i.e., the child intervals of $2-[7..10]$ have been processed already. The algorithm starts with the first child interval $[7..7]$ of $2-[7..10]$. Because all sets of positions of $2-[7..10]$ are initially empty, no repeated pair is output and node $2-[7..10]$ inherits the set of positions $\mathcal{P}_{[7..7]}(c) = \{10\}$ from $[7..7]$. Then the second child interval $[8..9]$ is processed. This means that $\mathcal{P}_{[8..9]}(g) = \{3\}$ and $\mathcal{P}_{[8..9]}(t) = \{8\}$ are combined with $\mathcal{P}_{[7..10]}(c) = \{10\}$. The maximal repeated pairs $\langle(3, 4), (10, 11)\rangle$ and $\langle(8, 9), (10, 11)\rangle$ are output and node $2-[7..10]$ inherits $\mathcal{P}_{[8..9]}(g) = \{3\}$ and $\mathcal{P}_{[8..9]}(t) = \{8\}$ from $[8..9]$. Finally, the third child interval $[10..10]$ is processed. Its set of positions $\mathcal{P}_{[10..10]}(c) = \{5\}$ can only be combined with $\mathcal{P}_{[7..10]}(g) = \{3\}$ and $\mathcal{P}_{[7..10]}(t) = \{8\}$ but not with $\mathcal{P}_{[7..10]}(c) = \{10\}$. Therefore, the maximal repeated pairs $\langle(3, 4), (5, 6)\rangle$ and $\langle(5, 6), (8, 9)\rangle$ are output and $2-[7..10]$ inherits the set of positions $\mathcal{P}_{[10..10]}(c) = \{5\}$ from $[10..10]$, so that $\mathcal{P}_{[7..10]}(c) = \{5, 10\}$, $\mathcal{P}_{[7..10]}(g) = \{3\}$, and $\mathcal{P}_{[7..10]}(t) = \{8\}$.

The algorithm described above can be implemented by Algorithm 4.6 (page 94), which traverses the lcp-interval tree in linear time in a bottom-up fashion. The procedure *process* in Algorithm 4.6 must be implemented in such a way that it outputs maximal repeated pairs and maintains sets of positions on the stack (which are added as a fifth component to the quadruples).

We now have a closer look at the two operations that are performed when processing an lcp-interval $[i..j]$. To combine two sets of positions boils down to computing their Cartesian product, and each element of the Cartesian product is a maximal repeated pair. Thus, the combinations can be computed in $O(z)$ time, where z is the number of maximal repeated pairs. The union of two sets of positions can be implemented in constant time if one uses linked lists. For each lcp-interval, there are $O(\sigma)$ union operations. Since $O(n)$ lcp-intervals have to be processed, the algorithm runs in $O(n\sigma + z)$ time.

Let us analyze the space consumption of the algorithm. A set of positions $\mathcal{P}_{[i..j]}(a)$ is the union of sets of positions of the child intervals of $[i..j]$. If the child intervals of $[i..j]$ have been processed, the corresponding sets of positions are obsolete. Hence it is not required to copy sets of positions. Moreover, we only have to store the sets of positions for those lcp-intervals that are on the stack used for the bottom-up traversal of the lcp-interval tree. So it is natural to store references to the sets of positions on the stack together with other information about the lcp-interval. Thus the space required for the sets of positions is determined by the maximal size of the stack. Since this is $O(n)$, the space requirement is $O(n\sigma)$. In practice, however, the stack size is much smaller.

Exercise 5.3.18 Prove that ω is a maximal repeat if and only if there is a maximal repeated pair $\langle (i, j), (i', j') \rangle$ so that $\omega = S[i..j]$.

Exercise 5.3.19 Show that the number of right maximal repeated pairs is bounded by $O(n^2)$. Prove that the number of maximal repeated pairs of a non-cyclic binary de Bruijn sequence S of order k is $\Omega(n^2)$, where $n = 2^k + k - 1$ is the length of S .

5.3.5 Non-overlapping repeats

In the preceding sections, we did not care whether the repeats were overlapping or not. Next, we show how to find non-overlapping repeats.

Definition 5.3.20 A repeated pair $\langle (i, j), (i', j') \rangle$ is said to be *non-overlapping* if $j < i'$, i.e., its left instance $S[i..j]$ does not overlap with its right instance $S[i'..j']$. Otherwise, it is called *overlapping*. A repeat ω is *non-overlapping* if there is a non-overlapping repeated pair $\langle (i, j), (i', j') \rangle$ so that $\omega = S[i..j]$. Otherwise, it is called *overlapping*.

Again, consider the string $S = \text{ctaataatg}$. The longest non-overlapping repeats are *aat* and *taa* because the repeat *taat* is overlapping.

Lemma 5.3.21 Let $\ell[i..j]$ be an lcp-interval that represents the non-empty string ω . Define $p = \min\{\text{SA}[k] \mid i \leq k \leq j\}$, $q = \max\{\text{SA}[k] \mid i \leq k \leq j\}$, and $m = \min\{q - p, \ell\}$. If $m > 0$, then we have:

1. The prefix $u = S[\text{SA}[i].. \text{SA}[i] + m - 1]$ of ω is a non-overlapping repeat.
2. If $m > \ell_{\text{par}}$, where ℓ_{par} is the lcp-value of the parent interval of $\ell[i..j]$, then every prefix of ω that is longer than u (if there is one) is an overlapping repeat.

Proof (1) The pair $\langle (p, p+m-1), (q, q+m-1) \rangle$ is a repeated pair because $m \leq \ell$ and $S[p..p+\ell-1] = \omega = S[q..q+\ell-1]$ have $S[p..p+m-1] = u = S[q..q+m-1]$ as a consequence. Furthermore, it is non-overlapping because $m \leq q - p$ implies $p + m - 1 < q$. So u is a non-overlapping repeat.

(2) Let $m > \ell_{\text{par}}$. If $m = \ell$, then $u = \omega$ and there is no prefix of ω that is longer than u . So suppose $m < \ell$ and let v be a prefix of ω that is longer than u . The inequalities $\ell_{\text{par}} < m < \ell$ imply that $[i..j]$ is the v -interval, i.e.,

- v is not a prefix of $S_{\text{SA}[i-1]}$,
- v is a prefix of $S_{\text{SA}[k]}$ for all $i \leq k \leq j$,
- v is not a prefix of $S_{\text{SA}[j+1]}$.

Moreover, $m < \ell$ means that $m = q - p$. It follows that for all r and s with $i \leq r < s \leq j$ the repeated pair $\langle (SA[r], SA[r] + |v|), (SA[s], SA[s] + |v|) \rangle$ is overlapping because $|SA[r] - SA[s]| \leq q - p = m = |u| < |v|$. Hence v is an overlapping repeat. \square

Exercise 5.3.22 Use the string $S = aaaac$ to show the necessity of the condition $m > \ell_{par}$ in statement (2) of Lemma 5.3.21.

With the help of Lemma 5.3.21, all longest non-overlapping repeats can be computed by a bottom-up traversal of the lcp-interval tree; see Algorithm 4.6 (page 94). Our algorithm maintains the currently longest non-overlapping repeats in a list L_{cur} and their length in ℓ_{cur} . When the procedure *process* in Algorithm 4.6 is applied to the current lcp-interval $\ell-[i..j]$, all its child intervals, say $[i_1..j_1], [i_2..j_2], \dots, [i_k..j_k]$ are known (some of them may be singleton intervals), as well as their minimum SA-values p_1, \dots, p_k (where $p_r = \min\{SA[k] \mid i_r \leq k \leq j_r\}$ for $1 \leq r \leq k$) and their maximum SA-values q_1, \dots, q_k (where $q_r = \max\{SA[k] \mid i_r \leq k \leq j_r\}$ for $1 \leq r \leq k$). Then, procedure *process* computes $p = \min\{SA[k] \mid i \leq k \leq j\}$ and $q = \max\{SA[k] \mid i \leq k \leq j\}$ by $p = \min\{p_r \mid 1 \leq r \leq k\}$ and $q = \max\{q_r \mid 1 \leq r \leq k\}$, respectively. Furthermore, it computes $m = \min\{q - p, \ell\}$ and the lcp-value ℓ_{par} of the parent interval of $\ell-[i..j]$ (recall that $\ell_{par} = \max\{LCP[i], LCP[j + 1]\}$ by Corollary 4.3.10). Now the algorithm proceeds by case analysis:

- If $m \leq \ell_{par}$, there is nothing to do because in this case the repeat $u = S[SA[i]..SA[i] + m - 1]$ will be considered when the parent interval of $\ell-[i..j]$ is processed.
- If $m > \ell_{par}$, then by Lemma 5.3.21, $u = S[SA[i]..SA[i] + m - 1]$ is the longest prefix of ω that is a non-overlapping repeat. In this case there are further case distinctions:
 - If $m > \ell_{cur}$, then u is longer than the currently longest non-overlapping repeat. Hence we update ℓ_{cur} and L_{cur} by the assignments $\ell_{cur} \leftarrow m$ and $L_{cur} \leftarrow [u]$.
 - If $m = \ell_{cur}$, then u is added to the list L_{cur} .
 - If $m < \ell_{cur}$, there is nothing to do.

Upon termination of the algorithm, the list L_{cur} contains all longest non-overlapping repeats of S and ℓ_{cur} is their length.

Exercise 5.3.23 Analyze the worst-case time complexity of the algorithm described above.

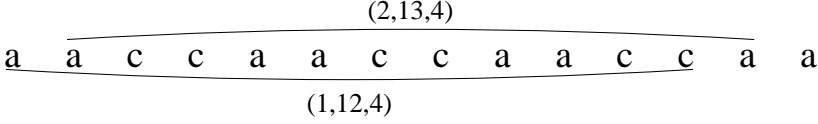


Figure 5.17: The tandem arrays (1, 12, 4) and (2, 13, 4) are overlapping.

5.3.6 Maximal periodicities

Next, we tackle the problem of finding tandemly repeated substrings in a string S of length n . Before presenting an optimal time algorithm, we develop a simpler $O(n \log n)$ -time algorithm to find all maximal periodicities.

In this section, it is convenient to use the following notation:

- $\text{lcp}(i, j) = \text{lcp}(S_i, S_j)$
- $\text{lcs}(i, j) = \text{lcs}(S[1..i], S[1..j])$

Definition 5.3.24 If a non-empty string ω can be written as $\omega = u^k$ for some $k \geq 2$, then it is called a *tandem array* (or *integer repetition*) with period-length $|u|$; otherwise ω is called *primitive*. A *tandem repeat* (or *square*) is a tandem array $\omega = u^k$ with $k = 2$.

Let b and e be positions in S (i.e., $1 \leq b \leq e \leq n$) and ℓ be a positive integer. The triple (b, e, ℓ) is a tandem array in S if $S[b..e]$ is a tandem array with period-length ℓ , i.e., $S[b..e] = u^k$ for some $k \geq 2$, where u consists of the first ℓ characters of $S[b..e]$ (note that this implies $1 \leq \ell \leq \frac{e-b+1}{2}$). Such a tandem array is called *right-maximal* if $(b, e + \ell, \ell)$ is not a tandem array in S , i.e., if there is no additional occurrence of u immediately after $S[b..e] = u^k$ in S ; it is called *left-maximal* if $(b - \ell + 1, e, \ell)$ is not a tandem array in S , i.e., if there is no additional occurrence of u immediately preceding $S[b..e] = u^k$ in S . It is called *maximal* if it is left- and right-maximal.

As an example consider the string $S = \text{aaccaaccaaccaaa}$. Clearly, every occurrence of aa and cc is a maximal tandem repeat in S with period-length 1. Besides those, there are the following maximal tandem arrays in S : (1, 12, 4), (2, 13, 4), and (3, 14, 4). Note that these three tandem arrays are overlapping; the overlap of (1, 12, 4) and (2, 13, 4) is visualized in Figure 5.17. To avoid overlapping tandem arrays with the same period-length, we introduce periodicities.

Definition 5.3.25 If a non-empty string ω can be written as $\omega = u^k v$, where $k \geq 2$ and v is a proper prefix of u , then it is called a *periodicity* (or *fractional repetition*) with *period-length* $|u|$. The triple (b, e, ℓ) is a periodicity in S if $S[b..e]$ is a periodicity with period-length ℓ (note that this implies

$1 \leq \ell \leq \frac{e-b+1}{2}$). Such a periodicity in S is called *left-maximal* if it cannot be extended to the left, i.e., if $(b-1, e, \ell)$ is not a periodicity in S ; it is called *right-maximal* if it cannot be extended to the right, i.e., if $(b, e+1, \ell)$ is not a periodicity in S . It is said to be a *maximal* if it is left- and right-maximal.

Lemma 5.3.26

1. A string ω of length m is a periodicity with period-length ℓ if and only if $\ell \leq \frac{m}{2}$ and $\omega[i] = \omega[i + \ell]$ for all i with $1 \leq i \leq m - \ell$.
2. (b, e, ℓ) is a periodicity in S if and only if $\ell \leq \frac{e-b+1}{2}$ and $S[i] = S[i + \ell]$ for all i with $b \leq i \leq e - \ell$ (or equivalently, $S[i - \ell] = S[i]$ for all i with $b + \ell \leq i \leq e$).
3. The periodicity (b, e, ℓ) in S is right-maximal if and only if $e = n$ or $S[e + 1 - \ell] \neq S[e + 1]$.
4. The periodicity (b, e, ℓ) in S is left-maximal if and only if $b = 1$ or $S[b - 1 + \ell] \neq S[b - 1]$.

Proof (1) If ω is a periodicity with period-length ℓ , then it can be written as $\omega = u^k v$, where $u = \omega[1..\ell]$, $k \geq 2$ and v is a proper prefix of u . This immediately implies $\ell \leq \frac{m}{2}$. For i with $1 \leq i \leq m - \ell$ there exist unique integers q and r so that $i = q\ell + r$ and $0 \leq r < \ell$. It follows that

$$\omega[i] = \omega[q\ell + r] = u[r] = \omega[(q + 1)\ell + r] = \omega[i + \ell]$$

Conversely, let $\ell \leq \frac{m}{2}$ and $\omega[i] = \omega[i + \ell]$ for all i with $1 \leq i \leq m - \ell$. Let q and r be the unique integers so that $m = q\ell + r$ and $0 \leq r < \ell$. Note that $q \geq 2$ because $\ell \leq \frac{m}{2}$. Define $u = \omega[1..\ell]$ and $v = \omega[1..r]$. Then $u = \omega[1..\ell] = \dots = \omega[(q - 1)\ell + 1..q\ell]$ and $v = \omega[1..r] = \omega[\ell + 1..r + \ell] = \dots = \omega[q\ell + 1..r + q\ell]$. Consequently, $\omega = u^q v$, where v is a proper prefix of u .

(2) This is a direct consequence of (1).

(3) “if”: If $e = n$, then the periodicity (b, e, ℓ) cannot be extended to the right, hence it is right-maximal. If $e \neq n$ and $S[e + 1 - \ell] \neq S[e + 1]$, then $(b, e + 1, \ell)$ is not a periodicity by (2). Consequently, (b, e, ℓ) is right-maximal.

“only if”: If $e \neq n$ and $S[e + 1 - \ell] = S[e + 1]$, then $(b, e + 1, \ell)$ is a periodicity by (2). This means that (b, e, ℓ) in S is not right-maximal.

(4) Analogous to (3). □

Figure 5.18 depicts all maximal periodicities in $S = aaccaaccaaccaa$. It is obvious that the periodicity $(2, 13, 4)$ is neither left- nor right-maximal in the string $S = aaccaaccaaccaa$; see Figure 5.19.

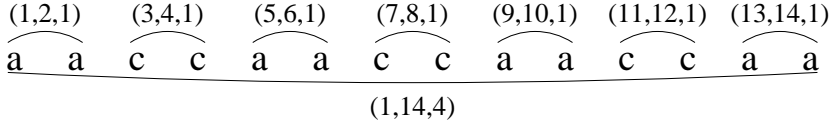


Figure 5.18: All maximal periodicities in the string $S = aaccaaccaacca$.

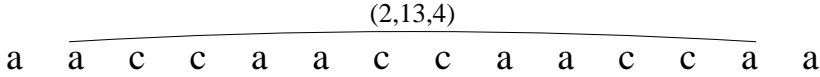


Figure 5.19: The periodicity $(2, 13, 4)$ is neither left- nor right-maximal.

An $O(n \log n)$ -time algorithm

Definition 5.3.27 We say that a periodicity (b, e, ℓ) has a

- *period to the left* of position i if $b + \ell \leq i \leq e + 1$, i.e., if it contains the ℓ -length substring $S[i - \ell..i - 1]$ of S ending immediately to the left of i (exclusive of i).
- *period starting* at position i if $b \leq i \leq e - \ell + 1$, i.e., if it contains the ℓ -length substring $S[i..i + \ell - 1]$ of S starting at position i .

Exercise 5.3.28 Let (b, e, ℓ) be a periodicity. Show that $b \leq i \leq e$ implies

- (b, e, ℓ) has a period to the left of i , or
- (b, e, ℓ) has a period starting at i (or both).

A first algorithm to find all maximal periodicities relies on the following fact: If there is a periodicity in S with period length ℓ that has a period to the left of position i in S , then the maximal extension of the periodicity

- to the left of $i - \ell$ (exclusive of $i - \ell$) consists of $L = |\text{lcs}(i - 1 - \ell, i - 1)|$ characters (see Figure 5.20),
- to the right of position i (inclusive of i) consists of $R = |\text{lcp}(i - \ell, i)|$ characters (see Figure 5.21), and
- $L + R \geq \ell$ (see Figure 5.22).

As a matter of fact, we prove a slightly more general statement in Lemma 5.3.29.

Lemma 5.3.29 For ℓ and i with $1 \leq \ell \leq \lfloor \frac{n}{2} \rfloor$ and $\ell + 1 \leq i \leq n + 1$, let $L = |\text{lcs}(i - 1 - \ell, i - 1)|$ and $R = |\text{lcp}(i - \ell, i)|$. Then the following statements are equivalent:

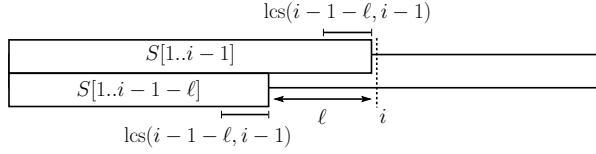


Figure 5.20: If there is a periodicity in S with period length ℓ having a period to the left of position i , then the maximal extension of the periodicity to the left of position $i - \ell$ (exclusive of $i - \ell$) consists of $|\text{lcs}(i - 1 - \ell, i - 1)|$ characters.

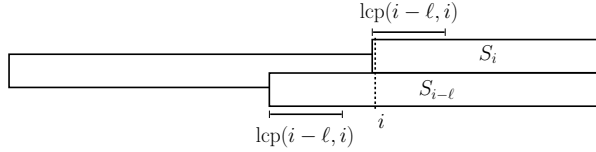


Figure 5.21: If there is a periodicity in S with period length ℓ having a period to the left of position i , then the maximal extension of the periodicity to the right of position i (inclusive of i) consists of $|\text{lcp}(i - \ell, i)|$ characters.

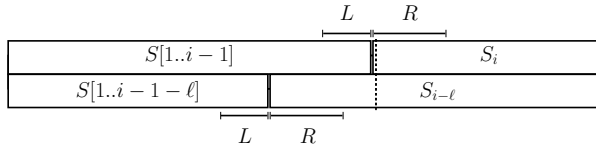


Figure 5.22: If there is a periodicity in S with period length ℓ having a period to the left of position i , then its maximal extension to the left consists of $L = |\text{lcs}(i - 1 - \ell, i - 1)|$ characters, its maximal extension to the right consists of $R = |\text{lcp}(i - \ell, i)|$ characters, and $L + R \geq \ell$.

1. There is a maximal periodicity (b, e, ℓ) in S that has a period starting at position $i - \ell$ in S (i.e., $b \leq i - \ell \leq e - \ell + 1$).
2. There is a maximal periodicity (b, e, ℓ) in S that has a period to the left of position i in S (i.e., $b + \ell \leq i \leq e + 1$).
3. $L + R \geq \ell$.
4. $(i - \ell - L, i - 1 + R, \ell)$ is a maximal periodicity in S .

Proof (1) \Rightarrow (2): Straightforward.

(2) \Rightarrow (3): $S[b..e]$ can be written as $S[b..e] = uvw$ with $u = S[b..i - \ell - 1]$, $v = S[i - \ell..i - 1]$, and $w = S[i..e]$. Observe that $u = \varepsilon$ (i.e., $b = i - \ell$) and $w = \varepsilon$ (i.e., $e = i - 1$) is possible. According to Lemma 5.3.26, we have $S[j] = S[j - \ell]$ for all j with $b + \ell \leq j \leq e$ because (b, e, ℓ) is a periodicity in S . This implies that $w = S[i..e]$ is a prefix of $S[i - \ell..n]$. Since (b, e, ℓ) is right-maximal, either $e = n$ or the inequality $S[e + 1] \neq S[e + 1 - \ell]$ holds. Therefore, w is the longest common prefix of $S[i..n]$ and $S[i - \ell..n]$, i.e., $w = \text{lcp}(i - \ell, i)$. A similar argumentation shows that $u = \text{lcs}(i - 1 - \ell, i - 1)$. It is clear that $e - b + 1 = |S[b..e]| = |uvw| = L + \ell + R$. Moreover, $2\ell \leq e - b + 1$; cf. Definition 5.3.25. Hence $L + R = e - b + 1 - \ell \geq \ell$.

(3) \Rightarrow (4): Define $u = \text{lcs}(i - 1 - \ell, i - 1)$, $v = S[i - \ell..i - 1]$, and $w = \text{lcp}(i - \ell, i)$. Note that $u = S[i - \ell - L..i - 1 - \ell]$ and $w = S[i..i - 1 + R]$. It follows that $uvw = S[i - \ell - L..i - 1 + R]$.

We claim that $(i - \ell - L, i - 1 + R, \ell)$ is a periodicity in S . It is a direct consequence of the definition of w that $S[j - \ell] = S[j]$ for all $i \leq j \leq i - 1 + R$, or equivalently, $S[j] = S[j + \ell]$ for all $i - \ell \leq j \leq i - 1 + R - \ell$. Analogously, it follows from the definition of u that $S[j] = S[j + \ell]$ for all $i - \ell - L \leq j \leq i - 1 - \ell$. In summary, we derive $S[j] = S[j + \ell]$ for all $i - \ell - L \leq j \leq i - 1 + R - \ell$. Moreover, the string $S[i - \ell - L..i - 1 + R]$ has length $(i - 1 + R) - (i - \ell - L) + 1 = R + \ell + L \geq 2\ell$, where the last inequality follows from the assumption $L + R \geq \ell$. According to Lemma 5.3.26, this proves the claim.

It is readily verified that $S[i - \ell + R] = S[i + R]$ would imply that w is not the *longest* common prefix of $S_{i-\ell}$ and S_i . Analogously, $S[i - 1 - \ell - L] = S[i - 1 - L]$ would contradict the fact that $u = \text{lcs}(i - 1 - \ell, i - 1)$. Therefore, $(i - \ell - L, i - 1 + R, \ell)$ is a maximal periodicity in S .

(4) \Rightarrow (1): This is obvious. □

Lemma 5.3.30 *Two maximal periodicities (b, e, ℓ) and (b', e', ℓ) with the same period length ℓ coincide (i.e., $b = b'$ and $e = e'$) if they overlap by at least ℓ characters.*

Proof Suppose that for some $\ell' \geq \ell$, the last ℓ' characters of $S[b..e]$ overlap the first ℓ' characters of $S[b'..e']$, i.e., $S[e - \ell' + 1..e] = S[b'..b' + \ell' - 1]$. By Lemma 5.3.26, we have $S[i] = S[i + \ell]$ for all i with $b \leq i \leq e - \ell$ and $S[j] = S[j + \ell]$ for all j with $b' \leq j \leq e' - \ell$. The combination of these facts

Algorithm 5.16 Finding all maximal periodicities in S .

```

for  $\ell \leftarrow 1$  to  $\lfloor \frac{n}{2} \rfloor$  do
   $i \leftarrow 2\ell + 1$ 
  while  $i \leq n + 1$  do
     $L \leftarrow |\text{lcs}(i - 1 - \ell, i - 1)|$ 
     $R \leftarrow |\text{lcp}(i - \ell, i)|$ 
    if  $L + R \geq \ell$  and  $R < \ell$  then
      output maximal periodicity  $(i - \ell - L, i - 1 + R, \ell)$ 
     $i \leftarrow i + \ell$ 

```

yields $S[i] = S[i + \ell]$ for all i with $\min\{b, b'\} \leq i \leq \max\{e - \ell, e' - \ell\}$. Since both periodicities are left-maximal, $b = b'$ follows. Similarly, we derive $e = e'$ from the right-maximality of the periodicities. \square

As a corollary to the preceding lemma, we conclude that if there is a maximal periodicity (b, e, ℓ) in S that has a period to the left of position i in S , then (b, e, ℓ) is the only maximal periodicity with this property. Moreover, $b = i - \ell - |\text{lcs}(i - 1 - \ell, i - 1)|$ and $e = i - 1 + |\text{lcp}(i - \ell, i)|$ by Lemma 5.3.29.

A naive algorithm for finding all maximal periodicities would use Lemma 5.3.29 for every position i in S and every possible period-length ℓ to test whether there is a maximal periodicity. Clearly, this naive algorithm has a worst-case running time of $O(n^2)$. The time complexity can be improved if one does not test at every position, but only at every ℓ -th position. The pseudo-code of this improved algorithm can be found in Algorithm 5.16. We would like to stress that the extra conditions $R < \ell$ in Algorithm 5.16 solely serves the purpose of avoiding that a maximal periodicity is output more than once.

Theorem 5.3.31 *Algorithm 5.16 outputs each maximal periodicity (b, e, ℓ) exactly once.*

Proof Consider a fixed period length ℓ , where $1 \leq \ell \leq \lfloor \frac{n}{2} \rfloor$. In Algorithm 5.16, i is initially set to $2\ell + 1$ and then incremented by ℓ at the end of each iteration of the while-loop. Thus, at the beginning of the $(k - 1)$ -th iteration of the while-loop, we have $i = k\ell + 1$, where $2 \leq k \leq \lfloor \frac{n}{\ell} \rfloor$. We show that in this iteration Algorithm 5.16 outputs every maximal periodicity (b, e, ℓ) with $k\ell \leq e < (k + 1)\ell$ exactly once. Note that such a maximal periodicity must have a period to the left of position $i = k\ell + 1$.

In the $(k - 1)$ -th iteration of the while-loop, Algorithm 5.16 tests whether $L + R \geq \ell$ holds. If so, there is a maximal periodicity $(i - \ell - L, i - 1 + R, \ell)$ having a period to the left of position i by Lemma 5.3.29 (hence $k\ell \leq e$). This maximal periodicity will be output only if $R < \ell$, which implies

$i - 1 + R = k\ell + R < (k + 1)\ell$. It should be emphasized that if the same maximal periodicity was detected in a previous iteration of the while-loop, it was not output there because of the condition $R < \ell$. Moreover, the same maximal periodicity will not be detected in the k -th iteration of the while-loop (nor in later iterations), because it does not have a period to the left of position $(k + 1)\ell + 1$. This is because it ends strictly before position $(k + 1)\ell$. \square

Let us analyze the worst-case complexity of Algorithm 5.16. It employs period-lengths from 1 to $\lfloor \frac{n}{2} \rfloor$ in the outer for-loop, and for every period-length ℓ , it tests $O(\frac{n}{\ell})$ positions in the inner while-loop. Therefore, the overall number of positions that are tested is $O(\sum_{\ell=1}^{\frac{n}{2}} \frac{n}{\ell})$. We have

$$\sum_{\ell=1}^{\frac{n}{2}} \frac{n}{\ell} = n \sum_{\ell=1}^{\frac{n}{2}} \frac{1}{\ell} \leq n (1 + \log_e \frac{n}{2})$$

because $1 + \log_e n$ is an upper bound for the n -th harmonic number $H_n = \sum_{\ell=1}^n \frac{1}{\ell}$; see e.g. [61]. Consequently, the algorithm tests $O(n \log n)$ positions. This coincides with its overall running time because one execution of the body of the while-loop requires only constant time; see Section 4.2.2.

An optimal time algorithm

Below we present an optimal time algorithm, which in its final form is due to Kolpakov and Kucherov [185]. (An alternative algorithm was developed contemporaneously by Gusfield and Stoye [141].) To deal with boundary cases, we assume from now on that S ends with the sentinel character $\$$. The algorithm is based on the Lempel-Ziv factorization $S = s_1 \dots s_m$ of S , which is explained in Section 5.2. For each factor s_j , let b_j and e_j be its start and end position in S . As in Section 5.2, we assume that the LZ-factorization of S is represented by a sequence of pairs $(\text{PrevOcc}_1, \text{LPS}_1), \dots, (\text{PrevOcc}_m, \text{LPS}_m)$:

- If $\text{LPS}_j = 0$, then $s_j = \text{PrevOcc}_j = c$ is the first occurrence of the character c in S .
- Otherwise, $\text{LPS}_j = |s_j| > 0$ and PrevOcc_j is a position in $s_1 s_2 \dots s_{j-1}$ at which a previous occurrence of s_j starts.

Definition 5.3.32 A periodicity (b, e, ℓ) is of type 2 if it is properly contained in some factor $s_j = S[b_j..e_j]$ of the Lempel-Ziv factorization of S , that is, $b_j < b < e < e_j$. A periodicity that is not of type 2 is of type 1.

The overall structure of the optimal time algorithm is:

1. Compute the Lempel-Ziv factorization of S in linear time.
2. Find all maximal periodicities of type 1.
3. Find all maximal periodicities of type 2.

We shall see that each of these three steps can be done in optimal time. As already mentioned, the algorithm was given by Kolpakov and Kucherov [185]. However, crucial parts should be attributed to Crochemore [64] (who first used the Lempel-Ziv factorization in this context) and to Main and Lorentz [210, 211] (who showed how to find all maximal periodicities of type 1).

Throughout this section, we use the string $S = aaccaaccaacaa\$$ as an example. Figure 5.23 shows its Lempel-Ziv factorization, Figure 5.24 depicts all maximal periodicities of type 1, and Figure 5.25 depicts all maximal periodicities of type 2.

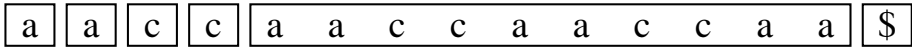


Figure 5.23: Lempel-Ziv factorization of the string $S = aaccaaccaacaa\$$.

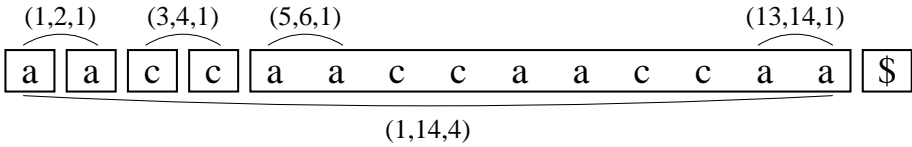


Figure 5.24: All maximal periodicities of type 1.

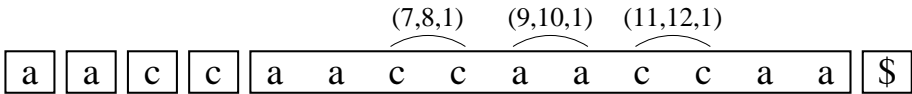


Figure 5.25: All maximal periodicities of type 2.

Computation of maximal periodicities of type 1

In the following, let $S = s_1 \dots s_m$ be the Lempel-Ziv factorization of S , where $s_j = S[b_j..e_j]$. Let (b, e, ℓ) be a maximal periodicity of type 1 that ends within some factor, say s_j . There are the following cases:

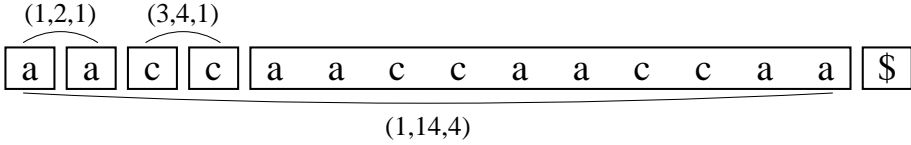


Figure 5.26: Maximal periodicities that cross the border between factors.

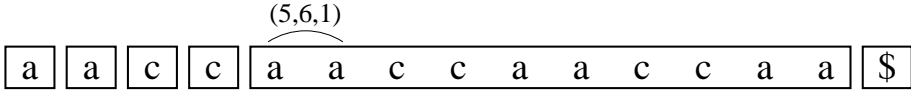


Figure 5.27: The maximal periodicity $(5, 6, 1)$ starts at $b_5 = 5$.

1. $b < b_j \leq e \leq e_j$, i.e., (b, e, ℓ) crosses the border to the neighboring factor s_{j-1} ; see Figure 5.26.

In this case, (b, e, ℓ) has a period to the left of b_j or a period starting at b_j (or both). To get mutually exclusive cases, we further divide this case into the following two subcases:

- a) (b, e, ℓ) has a period to the left of b_j .
- b) (b, e, ℓ) has a period starting at b_j , but no period to the left of b_j .

In this case, the period length ℓ must satisfy $\ell \leq |s_j|$ because the periodicity ends within s_j .

2. $b = b_j < e \leq e_j$, i.e., (b, e, ℓ) starts at position b_j and ends at or before position e_j ; see Figure 5.27.

As in the previous case, this means that (b, e, ℓ) has a period starting at b_j and $\ell \leq |s_j|$.

3. $b_j < b < e = e_j$, i.e., (b, e, ℓ) starts after position b_j and ends at position e_j : see Figure 5.28.

In this case, (b, e, ℓ) has a period to the left of b_{j+1} . Because it starts after b_j , the period length ℓ must satisfy $\ell \leq |s_j|$.

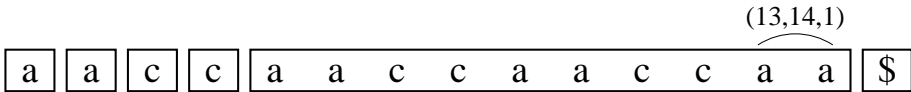


Figure 5.28: The maximal periodicity $(13, 14, 1)$ ends at $e_5 = 14$.

In other words, we can find all maximal periodicities of type 1 by testing at each position b_j whether there is a maximal periodicity having a period to the left of b_j or starting at b_j . Moreover, the test for a periodicity having a period starting at b_j can be restricted to period lengths ℓ that are smaller than or equal to $|s_j|$. (If $\ell > |s_j|$ were true, then the periodicity would not end within s_j .) The key observation that will yield a linear-time algorithm to find all maximal periodicities of type 1 is that the test for a periodicity having a period left of b_j can also be restricted to certain period lengths, as shown in the next lemma; cf. [210, 211].

Lemma 5.3.33 *Let (b, e, ℓ) be a periodicity that has a period to the left of position b_j , the beginning of factor s_j , $2 \leq j \leq m$. Then we have:*

1. $e \leq e_j$.
2. If $e_{j-1} < e$, then (b, e, ℓ) does not have a period to the left of b_{j-1} .
3. If $e_{j-1} < e$, then $|S[b..e]| < 2|s_{j-1}s_j|$ (hence $\ell < |s_{j-1}s_j|$).

Proof (1) Suppose to the contrary that $e > e_j$. Consider the suffix $\omega = S[b_j..e]$ of $S[b..e]$ and note that $|\omega| > |s_j| = |S[b_j..e_j]|$ because $e > e_j$. Since (b, e, ℓ) has a period to the left of position b_j and $e_j < e$, it follows $b + \ell \leq b_j \leq e_j < e$. Moreover, because $S[b..e]$ is a periodicity of period length ℓ , we have $S[i - \ell] = S[i]$ for all i with $b + \ell \leq i \leq e$ by Lemma 5.3.26. This implies $S[b_j - \ell..e - \ell] = S[b_j..e]$, and hence $S[b_j - \ell..e - \ell]$ is a previous occurrence of ω in S . This, however, contradicts the definition of the Lempel-Ziv factorization, according to which s_j is the longest prefix of $S[b_j..n]$ having a previous occurrence in S .

(2) If (b, e, ℓ) also has a period to the left of position b_{j-1} , then we infer from (1) that $e \leq e_{j-1}$. This, however, contradicts the assumption $e > e_{j-1}$.

(3) Suppose to the contrary that $|S[b..e]| \geq 2|s_{j-1}s_j|$, or equivalently, that $\frac{1}{2}|S[b..e]| \geq |s_{j-1}s_j|$. This implies that at least half of $S[b..e]$ occurs strictly before factor s_{j-1} . Since $|S[b..e]| \geq 2\ell$, at least ℓ characters of $S[b..e]$ occur strictly before the start position b_{j-1} of factor s_{j-1} . In other words, (b, e, ℓ) has a period to the left of position b_{j-1} . This, however, contradicts statement (2). \square

The pseudo-code for the computation of all maximal periodicities of type 1 can be found in Algorithm 5.17.

Theorem 5.3.34 *Algorithm 5.17 outputs every maximal periodicity of type 1 exactly once.*

Proof Let $s_1 \dots s_m$ be the Lempel-Ziv factorization of S , where $s_j = S[b_j..e_j]$. For a fixed j , $2 \leq j \leq m$, we show that Algorithm 5.17 outputs every maximal periodicity (b, e, ℓ) exactly once, which

Algorithm 5.17 Computation of all maximal periodicities of type 1.

```

Compute the Lempel-Ziv factorization  $s_1 \dots s_m$  of  $S$ , where  $s_j = S[b_j..e_j]$ .
for  $j \leftarrow 2$  to  $m$  do
  /* test for a maximal periodicity having a period to the left of  $b_j$  */
  for  $\ell \leftarrow 1$  to  $\min\{|s_{j-1}s_j| - 1, e_{j-1}\}$  do
     $L \leftarrow |\text{lcs}(e_{j-1} - \ell, e_{j-1})|$ 
     $R \leftarrow |\text{lcp}(b_j - \ell, b_j)|$ 
    if  $L + R \geq \ell$  and  $(R \geq 1 \text{ or } b_j - \ell - L > b_{j-1})$  then
      output maximal periodicity  $(b_j - \ell - L, e_{j-1} + R, \ell)$ 
  /* test for a maximal periodicity having a period starting at  $b_j$  */
  for  $\ell \leftarrow 1$  to  $|s_j|$  do
     $L' \leftarrow |\text{lcs}(e_{j-1}, e_{j-1} + \ell)|$ 
     $R' \leftarrow |\text{lcp}(b_j, b_j + \ell)|$ 
    if  $L' + R' \geq \ell$  and  $b_j + \ell - 1 + R' \leq e_j$  and  $L' < \ell$  then
      output maximal periodicity  $(b_j - L', e_{j-1} + \ell + R', \ell)$ 

```

1. ends within factor s_j and crosses the border to the neighboring factor s_{j-1} (i.e., $b < b_j \leq e \leq e_j$), where either
 - a) (b, e, ℓ) has a period to the left of b_j (hence $\ell < |s_{j-1}s_j|$ by Lemma 5.3.33), or
 - b) (b, e, ℓ) has a period starting at b_j , but no period to the left of b_j (hence $\ell \leq |s_j|$);
2. starts at b_j and ends at or before e_j (hence $\ell \leq |s_j|$);
3. starts after b_{j-1} and ends at e_{j-1} (hence $\ell \leq |s_{j-1}|$).

If (b, e, ℓ) has a period to the left of b_j , i.e., $b + \ell \leq b_j < e$, then we have $e \leq e_j$ by Lemma 5.3.33. Algorithm 5.17 detects the maximal periodicity (b, e, ℓ) because $L + R \geq \ell$ must hold by Lemma 5.3.29. It outputs (b, e, ℓ) only if (case 1a) it ends within s_j (this is the case if $R \geq 1$), or if (case 3) it ends at position e_{j-1} and starts after position b_{j-1} (this is the case if $R = 0$ and $b_j - \ell - L > b_{j-1}$).

If (b, e, ℓ) has a period starting at b_j and $\ell \leq |s_j|$ (cases 1b and 2), then Algorithm 5.17 detects the maximal periodicity (b, e, ℓ) because $L' + R' \geq \ell$ by Lemma 5.3.29. It outputs (b, e, ℓ) only if it ends within s_j (this is the case if $b_j + \ell - 1 + R' \leq e_j$) and if (b, e, ℓ) does not have a period to the left of b_j (this is the case if $L' < \ell$). \square

Let us analyze the worst-case complexity. It is clear that Algorithm 5.17 needs only linear space. Moreover, we have seen that the Lempel-Ziv factorization of S can be computed in linear time. Because the calculations within the for-loops can be done in constant time (see Section 4.2.2), it

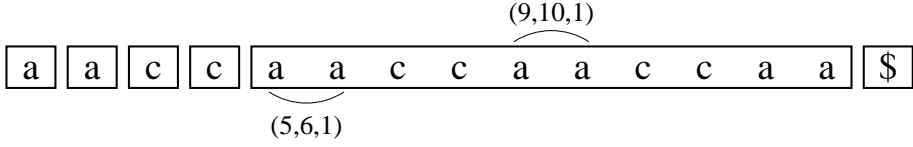


Figure 5.29: The maximal periodicity $(5, 6, 1)$ of type 1 is a previous occurrence of the maximal periodicity $(9, 10, 1)$ of type 2.

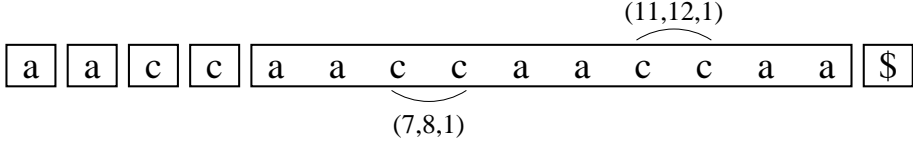


Figure 5.30: The maximal periodicity $(7, 8, 1)$ of type 2 is a previous occurrence of the maximal periodicity $(11, 12, 1)$ of type 2.

suffices to find out how often they are performed. It is quite obvious that this number is at most

$$\sum_{j=2}^m (|s_{j-1}s_j| - 1 + |s_j|) < 2n + n = 3n$$

Consequently, the algorithm runs in $O(n)$ time.

Computation of maximal periodicities of type 2

The final task is to find all maximal periodicities of type 2. To accomplish this, we make use of the fact that every maximal periodicity of this type has a previous occurrence to the left; see Figures 5.29 and 5.30.

Lemma 5.3.35 *Let b and e be positions in S so that $b_j < b < e < e_j$, where $s_j = S[b_j..e_j]$ is a factor of the Lempel-Ziv factorization of S . Then (b, e, ℓ) is a maximal periodicity if and only if $(b - \delta_j, e - \delta_j, \ell)$ is a maximal periodicity, where $\delta_j = b_j - \text{PrevOcc}_j$.*

Proof The condition $b_j < b < e < e_j$ implies $|s_j| \geq 4$. Thus, PrevOcc_j is the start position of a previous occurrence of s_j in S . This previous occurrence will be denoted by t_j . With $\delta_j = b_j - \text{PrevOcc}_j$, we have

$$t_j = S[b_j - \delta_j..e_j - \delta_j] = s_j$$

The statement of the lemma “ (b, e, ℓ) is a maximal periodicity if and only if $(b - \delta_j, e - \delta_j, \ell)$ is a maximal periodicity” is an obvious consequence of

Algorithm 5.18 Computation of all maximal periodicities of type 2.

Let $s_1 \dots s_m$ be the Lempel-Ziv factorization of S , where $s_j = S[b_j..e_j]$.
 For $1 \leq i \leq n$, let $L(i)$ be the list of all maximal periodicities of type 1
 with starting position i , sorted in increasing order of the end positions.
for $j \leftarrow 2$ **to** m **do**
 if $|s_j| \geq 4$ **then** / * a smaller factor has no periodicities of type 2 * /
 $\delta_j \leftarrow b_j - \text{PrevOcc}_j$
 for $i \leftarrow b_j + 1$ **to** $e_j - 1$ **do**
 for each $(i - \delta_j, e, \ell)$ in $L(i - \delta_j)$ **do**
 if $e + \delta_j \geq e_j$ **then break**
 output $(i, e + \delta_j, \ell)$ and prepend it to $L(i)$

$S[b..e] = S[b - \delta_j..e - \delta_j]$ and the fact that the characters to the left and to the right of the occurrences coincide. More precisely, $S[b - 1] = S[b - \delta_j - 1]$ and $S[e + 1] = S[e - \delta_j + 1]$ holds because $S[b..e]$ is properly contained in s_j (i.e., $b_j < b < e < e_j$) and $S[b - \delta_j..e - \delta_j]$ is properly contained in $t_j = s_j$ (i.e., $b_j - \delta_j < b - \delta_j < e - \delta_j < e_j - \delta_j$). \square

To retrieve all maximal periodicities of type 2, we set up an array $A[1..n]$ of initially empty lists $L(i)$. Moreover, we change Algorithm 5.17 so that each detected maximal periodicity (b, e, ℓ) is not output, but put instead into the list $L(e)$ corresponding to its end position. Then we process all lists in increasing order and sort the maximal periodicities by bucket sort into n lists according to their starting position. After this sort, maximal periodicities having the same starting position b occur inside list $L(b)$ in increasing order of their end positions. Obviously, this sorting procedure takes linear time because there are $O(n)$ maximal periodicities of type 1.

Now we use Lemma 5.3.35 to identify all maximal periodicities of type 2. The pseudo-code can be found in Algorithm 5.18. The algorithm assumes that all maximal periodicities of type 1 have been computed and those that start at position i are stored in the list $L(i)$ in increasing order of their end positions. Then, for each position i strictly within factor s_j , all maximal periodicities that start at i and end strictly within factor s_j can be determined from the list $L(i - \delta_j)$. This is because by Lemma 5.3.35 $(i - \delta_j, e, \ell)$ is a maximal periodicity if and only if $(i, e + \delta_j, \ell)$ is a maximal periodicity, provided that $b_j < i < e + \delta_j < e_j$. Thus, we scan the list $L(i - \delta_j)$ from left to right (in increasing order of the end positions) and an entry $(i - \delta_j, e, \ell)$ with $e < e_j - \delta_j$ correspond to a maximal periodicity $(i, e + \delta_j, \ell)$ of type 2 because it is properly contained in factor s_j . Once we find an entry $(i - \delta_j, e, \ell)$ with $e \geq e_j - \delta_j$ in the list $L(i - \delta_j)$, we can proceed with the next position $i + 1$, because neither $(i - \delta_j, e, \ell)$ nor the remaining members of the list $L(i - \delta_j)$ are maximal periodicities of type 2.

We would like to stress that Algorithm 5.18 maintains the following invariant: after each execution of the inner for-loop, the entries in list $L(i)$ appear in increasing order of their end positions. This is because the new entries have their end positions inside s_j , whereas the old entries of type 1 have their end positions outside s_j . Since Algorithm 5.18 proceeds from left to right, it finds all maximal periodicities of type 2. Upon termination of the algorithm, all maximal periodicities (of types 1 and 2) have been computed and are stored in the lists $L(i)$, where $1 \leq i \leq n$.

Let us analyze the worst-case time complexity of the algorithm. Whenever the body of the inner for-loop is executed, a maximal periodicity is output. Therefore, the time spent by Algorithm 5.18 is proportional to the number of all maximal periodicities of type 2. Consequently, Algorithm 5.18 computes maximal periodicities in $O(n+z)$ time, where z is the number of all maximal periodicities. This is optimal because n is the size of the input and z is the size of the output.

Exercise 5.3.36 The sequence of *Fibonacci strings* is defined by $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1}f_{i-2}$. Thus, the sequence continues with $f_2 = 10$, $f_3 = 101$, $f_4 = 10110$, and $f_5 = 10110101$. Determine all maximal periodicities in $f_7 = 101101011011010110101$.

Exercise 5.3.37 Given all maximal periodicities in a string S , develop an algorithm that computes all

- maximal tandem arrays in S ,
- tandem arrays in S ,
- tandem repeats in S .

Apply the algorithm to the Fibonacci string $f_7 = 101101011011010110101$.

How many maximal periodicities can a string contain?

Definition 5.3.38 A maximal periodicity (b, e, ℓ) is called a *run* if its period length ℓ is minimal, i.e., for all $\ell' < \ell$ the triple (b, e, ℓ') is not a maximal periodicity.

The *exponent* of a run (b, e, ℓ) is $\frac{e-b+1}{\ell}$, i.e., it is the length of the run divided by the period length.

The next deep theorem states that the number of runs in S is in $O(n)$.

Theorem 5.3.39 *The following statements hold.*

1. *A string of length n contains at most $O(n)$ runs; an upper bound is $c_1 n = 1.029n$.*
2. *The sum of the exponents of all runs in a string of length n is in $O(n)$; an upper bound is $c_2 n = 4.1n$.*

Proof Both results were first proved by Kolpakov and Kucherov [186], and the upper bounds are due to Crochemore et al. [68, 71]. \square

Our next goal is to characterize the relationship between runs and maximal periodicities. Fine and Wilf's theorem [102] (Theorem 5.3.41) plays a crucial role, as we shall see.

Lemma 5.3.40 *Let ω be a periodicity with period-lengths p and q . If $p < q$, then ω is also a periodicity with period-length $q - p$.*

Proof According to Lemma 5.3.26, we have

1. $\omega[i] = \omega[i + p]$ for all $1 \leq i \leq m - p$ and $2p \leq m$,
2. $\omega[i] = \omega[i + q]$ for all $1 \leq i \leq m - q$ and $2q \leq m$.

It follows

$$\omega[i] = \omega[i + q] = \omega[i + q - p]$$

for all i with $1 \leq i \leq m - q$, and

$$\omega[i] = \omega[i - p] = \omega[i + q - p]$$

for all i with $p + 1 \leq i \leq m - (q - p) = m - q + p$. Moreover, $2(q - p) < 2q \leq m$. Hence ω is a periodicity with period-length $q - p$. \square

Theorem 5.3.41 *If ω is a periodicity with period-lengths p and q , then it is also a periodicity with period-length $\gcd(p, q)$, where $\gcd(p, q)$ denotes the greatest common divisor of p and q .*

Proof This follows from the repeated application of Lemma 5.3.40 in conjunction with the correctness of the well-known Euclidean algorithm, which determines the greatest common divisor $\gcd(p, q)$ of two positive integers p and q ; see Algorithm 5.19. \square

Corollary 5.3.42 *There is a maximal periodicity (b, e, ℓ) if and only if there is a run (b, e, p) so that $\ell = qp$ for some natural number $q \leq \frac{e-b+1}{2p}$.*

Algorithm 5.19 Euclidean algorithm to determine the greatest common divisor $\gcd(p, q)$ of two positive integers p and q .

```

 $\gcd(p, q)$ 
  if  $p = q$  then return  $p$ 
  if  $p < q$  then return  $\gcd(p, q - p)$ 
  else return  $\gcd(p - q, q)$ 

```

Proof The if-direction is straightforward. To prove the only-if-direction, suppose that (b, e, ℓ) is a maximal periodicity, and observe that this implies $\ell \leq \frac{e-b+1}{2}$. Among all maximal periodicities with start position b and end position e , let (b, e, p) be the one with smallest period-length. Clearly, (b, e, p) is a run. According to Theorem 5.3.41, $(b, e, \gcd(p, \ell))$ is also a maximal periodicity. Because p is minimal, it follows that $p = \gcd(p, \ell)$. Therefore, $\ell = qp$ for some natural number q . \square

It follows as a consequence that we can enumerate all maximal periodicities provided that all runs are known: If (b, e, p) is a run, then $(b, e, p), (b, e, 2p), (b, e, 3p), \dots, (b, e, qp)$ are maximal periodicities, where $q = \lfloor \frac{e-b+1}{2p} \rfloor$.

Corollary 5.3.43 *A string S of length n contains at most $\frac{c_2}{2}n$ maximal periodicities, where c_2 is the constant from Theorem 5.3.39.*

Proof Let (b, e, p) be a run in S with exponent $exp = \frac{e-b+1}{p}$. Because there are $q = \lfloor \frac{e-b+1}{2p} \rfloor$ maximal periodicities corresponding to the run (b, e, p) and $2q = 2\lfloor \frac{e-b+1}{2p} \rfloor \leq exp$, it follows that $q \leq \frac{exp}{2}$. According to Theorem 5.3.39, the sum of the exponents of all runs in S is less than c_2n . Consequently, the number of maximal periodicities in S is bounded by $\frac{c_2}{2}n$. \square

Recall that the algorithm for finding all maximal periodicities has the worst-case time complexity $O(n + z)$, where z is the number of all maximal periodicities in S . It follows from Corollary 5.3.43 that z is in $O(n)$, so that the worst-case time complexity of the algorithm is in fact $O(n)$.

If one is interested in all runs of a string, then these can be obtained in $O(n)$ time by first computing all maximal periodicities and then removing all those having a non-minimal period length. A better solution is to modify the algorithm for finding all maximal periodicities as follows: When all maximal periodicities of type 1 have been stored in the lists $L(i)$, $1 \leq i \leq n$, in increasing order of their end positions, then maximal periodicities having the same start position b and the same end position e occur consecutively in the list $L(b)$. Therefore, when we scan the list $L(b)$ and find entries, say $(b, e, \ell_1), (b, e, \ell_2), \dots, (b, e, \ell_k)$, having the same start

and end positions but different period lengths, then we just keep the entry with minimum period length and delete the others from the list. The resulting lists contain all *runs* of type 1. The rest of the algorithm remains the same.

5.4 Comparing two strings

In solving problems that deal with the comparison of strings, it is often desirable to know the lexicographic order of their suffixes. In this section, we focus on two strings, but the material will later be generalized to multiple strings.

5.4.1 Generalized suffix array

In what follows, we are given two strings of length n_1 and n_2 , respectively. Because these strings may share identical suffixes, we append the special character $\#$ to the first and the sentinel character $\$$ to the second string. The resulting strings S^1 and S^2 have lengths $|S^1| = n_1 + 1$ and $|S^2| = n_2 + 1$. We assume $\# < \$$ and that all other characters in the alphabet Σ are larger than $\$$. The *common suffix array* of S^1 and S^2 is an array that stores the lexicographic order of all suffixes of S^1 and S^2 . It can be obtained by merging the suffix arrays SA_1 and SA_2 of S^1 and S^2 ; see Figures 5.31, 5.32, and 5.33. Note that the i -th lexicographically smallest string among all suffixes may be a suffix of S^1 or of S^2 . To deal with this, the common suffix array of S^1 and S^2 actually consists of *two* arrays D and SA' defined as follows: S_p^j is the i -th lexicographically smallest string among all suffixes of S^1 and S^2 if and only if $D[i] = j$ and $SA'[i] = p$. In other words, the *document array* D tells us to which document (string) a suffix belongs, while the array SA' provides its start position in that document.

The common enhanced suffix array of S^1 and S^2 can be obtained by merging their enhanced suffix arrays (SA_1, LCP_1) and (SA_2, LCP_2) . The merging procedure is similar to that of the merge sort algorithm; see e.g. [61]. It uses a subroutine $compare(j_1, j_2, q)$, detailed in Algorithm 5.20, which compares the suffixes $S_{SA_1[j_1]}^1$ and $S_{SA_2[j_2]}^2$ with offset q . More precisely, $compare(j_1, j_2, q)$ compares $S^1[SA_1[j_1] + q..n_1 + 1]$ and $S^2[SA_2[j_2] + q..n_2 + 1]$ character by character, and it increments q for every character match. When the first character mismatch occurs, the procedure $compare$ returns the pair (q, s) consisting of the new offset q and

$$s = \begin{cases} 1 & \text{if } S_{SA_1[j_1]}^1 < S_{SA_2[j_2]}^2 \\ 2 & \text{otherwise} \end{cases}$$

In what follows let $\bar{s} = 3 - s$, i.e., $\bar{s} = 1$ if $s = 2$ and $\bar{s} = 2$ if $s = 1$.

i	SA	LCP	$S_{SA[i]}$
1	10	-1	#
2	3	0	aataatg#
3	6	3	aatg#
4	4	1	ataatg#
5	7	2	atg#
6	1	0	ctaataatg#
7	9	0	g#
8	2	0	taataatg#
9	5	4	taatg#
10	8	1	tg#
11		-1	

Figure 5.31: The enhanced suffix array of the string $S^1 = ctaataatg\#$.

i	SA	LCP	$S_{SA[i]}$
1	11	-1	\$
2	3	0	aaacatat\$
3	4	2	aacatat\$
4	1	1	acaaacatat\$
5	5	3	acatat\$
6	9	1	at\$
7	7	2	atat\$
8	2	0	caaacatat\$
9	6	2	catat\$
10	10	0	t\$
11	8	1	tat\$
12		-1	

Figure 5.32: The enhanced suffix array of the string $S^2 = acaaacatat\$$.

i	$D[i]$	$SA'[i]$	LCP	$S_{SA'[i]}^{D[i]}$
1	1	10	-1	#
2	2	11	0	\$
3	2	3	0	aaacatat\$
4	2	4	2	aacatat\$
5	1	3	2	aataatg#
6	1	6	3	aatg#
7	2	1	1	acaaacatat\$
8	2	5	3	acatat\$
9	2	9	1	at\$
10	1	4	2	ataatg#
11	2	7	3	atat\$
12	1	7	2	atg#
13	2	2	0	caaacatat\$
14	2	6	2	catat\$
15	1	1	1	ctaataatg#
16	1	9	0	g#
17	2	10	0	t\$
18	1	2	1	taataatg#
19	1	5	4	taatg#
20	2	8	2	tat\$
21	1	8	1	tg#
22			-1	

Figure 5.33: The common enhanced suffix array of $S^1 = ctaataatg\#$ and $S^2 = acaaacatat\$$.

Algorithm 5.20 Procedure $compare(j_1, j_2, q)$ compares the suffixes $S_{SA_1[j_1]}^1$ and $S_{SA_2[j_2]}^2$ with offset q .

```

while  $S^1[SA_1[j_1] + q] = S^2[SA_2[j_2] + q]$  do
     $q \leftarrow q + 1$ 
if  $S^1[SA_1[j_1] + q] < S^2[SA_2[j_2] + q]$  then
    return  $(q, 1)$ 
else
    return  $(q, 2)$ 

```

The pseudo-code of the merging procedure is shown in Algorithm 5.21, and we exemplify it by merging the ESAs of the strings $S^1 = ctaataatg\#$ and $S^2 = acaaacatat\$$; see Figures 5.31 and 5.32. Let j_1 and j_2 be the current indices in SA_1 and SA_2 , respectively. After placing $\#$ and $\$$ as the first two entries in the common ESA, we have $j_1 = 2$ and $j_2 = 2$. The procedure call $compare(2, 2, 0)$ yields the pair $(q, s_{new}) = (2, 2)$, showing that $q = |\text{lcp}(S_{SA_1[2]}^1, S_{SA_2[2]}^2)| = 2$ and $S_{SA_1[2]}^1 = aataatg\# > aaacatat\$ = S_{SA_2[2]}^2$. Since $s_{new} = 2 = s$, $\text{LCP}[3]$ is set to $\text{LCP}_2[2] = 0$ and $aaacatat\$$ is placed as the third entry in the common ESA. Furthermore, j and j_2 are incremented by one. The merging procedure then distinguishes between the following cases: $\text{LCP}_s[j_s]$ is smaller than q , larger than q , or equal to q . In our example, the third case applies because $\text{LCP}_2[j_2] = \text{LCP}_2[3] = 2$ equals $q = 2$. In this case, the procedure $compare$ is called with the parameters $j_1 = 2$, $j_2 = 3$, and $q = 2$. Again, it returns the pair $(q, s_{new}) = (2, 2)$, sets $\text{LCP}[4]$ to $\text{LCP}_2[3] = 2$, places $aaacatat\$$ as the fourth entry in the common ESA, and increments j and j_2 by one. Now $\text{LCP}_2[j_2] = \text{LCP}_2[4] = 1$ is smaller than $q = 2$. This implies that $S_{SA_1[j_1]}^1 = S_{SA_1[2]}^1 = aataatg\#$ is lexicographically smaller than $S_{SA_2[j_2]}^2 = S_{SA_2[4]}^2 = acaaacatat\$$. Consequently, $aataatg\#$ will be the fifth entry in the common ESA. Because the previous entry $aaacatat\$$ belongs to the other string, $\text{LCP}[5]$ is set to $q = 2$, q itself gets the new value $\text{LCP}_2[4] = 1$, and \bar{s} and s are swapped (that is, now $s = 1$ and $\bar{s} = 2$). Then, $aataatg\#$ is placed as the fifth entry in the common ESA, and both j and j_1 are incremented by one. Because $\text{LCP}_1[j_1] = \text{LCP}_1[3] = 3 > 1 = q$, it follows that $S_{SA_1[j_1]}^1 = aatg\#$ is also lexicographically smaller than $S_{SA_2[j_2]}^2 = acaaacatat\$$. Thus, $\text{LCP}[6]$ is set to $\text{LCP}_1[3] = 3$, $aatg\#$ becomes the sixth entry in the common ESA, and j as well as j_1 is incremented. The next comparison of $\text{LCP}_1[j_1]$ with q shows that these are equal ($\text{LCP}_1[j_1] = \text{LCP}_1[4] = 1$). This means that the first characters of $S_{SA_1[j_1]}^1 = ataatg\#$ and $S_{SA_2[j_2]}^2 = acaaacatat\$$ coincide, but in order to find out which of these two suffixes is lexicographically smaller than the other, the procedure $compare$ must be called with the parameters $j_1 = 4$, $j_2 = 4$, and $q = 1$. It returns the pair $(q, s_{new}) = (1, 2)$, sets $\text{LCP}[7]$ to $q_{old} = 1$, swaps \bar{s} and s (so that $s = 2$ and $\bar{s} = 1$), places $acaacatat\$$ as

Algorithm 5.21 Merging the suffix arrays SA_1 and SA_2 and the LCP-arrays LCP_1 and LCP_2 of S^1 and S^2 .

```

 $D[1] \leftarrow 1; SA'[1] \leftarrow SA_1[1]; LCP[1] \leftarrow -1$ 
 $D[2] \leftarrow 2; SA'[2] \leftarrow SA_2[1]; LCP[2] \leftarrow 0$ 
 $j \leftarrow 3$ 
 $(j_1, j_2) \leftarrow (2, 2)$ 
 $(\bar{s}, s) \leftarrow (1, 2)$ 
 $q \leftarrow 0$ 
while  $j \leq n_1 + n_2 + 2$  do
    /* Invariants:  $S_{SA_s[j_s-1]}^s < S_{SA_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$  and */
    /*  $q = |\text{lcp}(S_{SA_s[j_s-1]}^s, S_{SA_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})|$ , where  $q = 0$  if  $j_s > n_s + 1$  or  $j_{\bar{s}} > n_{\bar{s}} + 1$  */
     $q_{old} \leftarrow q$ 
    if  $j_1 \leq n_1 + 1$  and  $j_2 \leq n_2 + 1$  then
         $(q, s_{new}) \leftarrow \text{compare}(j_1, j_2, q)$ 
    else
         $(q, s_{new}) \leftarrow (-1, \bar{s})$ 
    if  $s_{new} = s$  then
         $LCP[j] \leftarrow LCP_s[j_s]$ 
    else
         $LCP[j] \leftarrow q_{old}$ 
         $\text{swap}(\bar{s}, s)$ 
     $D[j] \leftarrow s$ 
     $SA'[j] \leftarrow SA_s[j_s]$ 
     $(j, j_s) \leftarrow (j + 1, j_s + 1)$ 
    while  $j_s \leq n_s + 1$  and  $LCP_s[j_s] \neq q$  do
        if  $LCP_s[j_s] > q$  then
             $LCP[j] \leftarrow LCP_s[j_s]$ 
        else
            /*  $LCP_s[j_s] < q$  */
             $LCP[j] \leftarrow q$ 
             $q \leftarrow LCP_s[j_s]$ 
             $\text{swap}(\bar{s}, s)$ 
         $D[j] \leftarrow s$ 
         $SA'[j] \leftarrow SA_s[j_s]$ 
         $(j, j_s) \leftarrow (j + 1, j_s + 1)$ 
     $LCP[n_1 + n_2 + 3] \leftarrow -1$ 

```

the seventh entry in the common ESA, and increments j and j_2 by one. Now $\text{LCP}_2[j_2] = \text{LCP}_2[5] = 3$ is larger than $q = 1$. Hence $S_{\text{SA}_2[5]}^2 = \text{acatat}\$$ is lexicographically smaller than $S_{\text{SA}_1[4]}^1 = \text{ataatg}\#$. It is placed in the common ESA, and so on.

In order to prove the correctness of Algorithm 5.21, let j , j_1 , and j_2 be the current indices in SA, SA_1 , and SA_2 , respectively, and let q be the current offset. Let us assume for a moment that $j_1 \leq n_1 + 1$ and $j_2 \leq n_2 + 1$. We show that both while-loops maintain the following invariants (which obviously hold true when the outer while-loop is reached for the first time):

1. $S_{\text{SA}_s[j_s-1]}^s < S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$ (so $D[j-1] = s$ and $\text{SA}[j-1] = \text{SA}_s[j_s-1]$)
2. $q = |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})|$

The merging procedure stores the value of q in the variable q_{old} and calls the subroutine $\text{compare}(j_1, j_2, q)$, which returns the pair (q, s_{new}) , where $q = |\text{lcp}(S_{\text{SA}_s[j_s]}^s, S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})|$. If $s_{new} = s$, then $S_{\text{SA}_s[j_s]}^s < S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$ and thus $S_{\text{SA}_s[j_s]}^s$ becomes the j -th entry in the common suffix array. Because this suffix belongs to the same string S^s as the $(j-1)$ -th entry in the common suffix array, $\text{LCP}[j]$ must be set to the length of their longest common prefix, which is $\text{LCP}_s[j]$. Otherwise, if $s_{new} = \bar{s}$, then $S_{\text{SA}_s[j_s]}^s > S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$ and thus $S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$ will be the j -th entry in the common suffix array. By the second invariant, $\text{LCP}[j]$ is correctly set to $q_{old} = |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})|$. Then $S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$ is placed into the common suffix array by swapping \bar{s} and s and using the same three lines of code as in the previous case. It is readily verified that the invariants hold when the inner while-loop is reached. The merging procedure proceeds by distinguishing between the cases (1) $\text{LCP}_s[j_s]$ is larger than q , (2) smaller than q , or (3) equal to q .

1. $\text{LCP}_s[j_s] > q$. The combination of the first invariant $S_{\text{SA}_s[j_s-1]}^s < S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$ and the second invariant $q = |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})|$ yields

$$S^s[\text{SA}_s[j_s-1] + q] < S^{\bar{s}}[\text{SA}_{\bar{s}}[j_{\bar{s}}] + q]$$

It further follows from

$$\text{LCP}_s[j_s] = |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_s[j_s]}^s)| > |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})| = q$$

that $|\text{lcp}(S_{\text{SA}_s[j_s]}^s, S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}})| = q$. This, in conjunction with the fact that the characters $S^s[\text{SA}_s[j_s-1] + q]$ and $S^{\bar{s}}[\text{SA}_{\bar{s}}[j_{\bar{s}}] + q]$ coincide, implies that $S_{\text{SA}_s[j_s]}^s$ is lexicographically smaller than $S_{\text{SA}_{\bar{s}}[j_{\bar{s}}]}^{\bar{s}}$. Hence the suffix $S_{\text{SA}_s[j_s]}^s$ can be placed into the common ESA without any further character comparison. Since $S_{\text{SA}_s[j_s]}^s$ is a suffix of the same string as the $(j-1)$ -th entry in the common suffix array, the j -th entry in the common LCP-array must be set to $\text{LCP}_s[j_s]$. After the increment of j_s , both invariants are true.

2. $\text{LCP}_s[j_s] < q$. Similar to the previous case,

$$\text{LCP}_s[j_s] = |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_s[j_s]}^s)| < |\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_{\bar{s}}[j_s]}^{\bar{s}})| = q$$

has $|\text{lcp}(S_{\text{SA}_s[j_s]}^s, S_{\text{SA}_{\bar{s}}[j_s]}^{\bar{s}})| = \text{LCP}_s[j_s]$ as a consequence. Moreover,

$$S^{\bar{s}}[\text{SA}_{\bar{s}}[j_s] + \text{LCP}_s[j_s]] = S^s[\text{SA}_s[j_s - 1] + \text{LCP}_s[j_s]] < S^s[\text{SA}_s[j_s] + \text{LCP}_s[j_s]]$$

Therefore, $S_{\text{SA}_{\bar{s}}[j_s]}^{\bar{s}}$ is lexicographically smaller than $S_{\text{SA}_s[j_s]}^s$ and it becomes the j -th entry in the common suffix array. Clearly, the j -th entry in the common LCP-array must be set to q because q equals $|\text{lcp}(S_{\text{SA}_s[j_s-1]}^s, S_{\text{SA}_{\bar{s}}[j_s]}^{\bar{s}})|$. To ensure that the second invariant holds after the execution of the inner while-loop, q is set to $\text{LCP}_s[j_s]$.

3. $\text{LCP}_s[j_s] = q$. In this case, the inner while-loop is not executed at all. Hence the invariants hold.

The boundary case ($j_1 > n_1 + 1$ or $j_2 > n_2 + 1$) still needs an explanation. Because the last element that was inserted into the common ESA belongs to the string S^s , the boundary case occurs for $j_s > n_s + 1$. That is, all suffixes of S^s have been placed into the common ESA and the remaining suffixes of $S^{\bar{s}}$ must be inserted one by one at the end of the common ESA. The algorithm accomplishes this by setting (q, s_{new}) to $(-1, \bar{s})$, having the effect that the first of the remaining suffixes of $S^{\bar{s}}$ is placed into the common ESA by the lines of code before the inner while-loop, while the others are successively inserted by the inner while-loop.

Although Algorithm 5.21 works well in practice, it has a quadratic worst-case time complexity. The worst case occurs when the algorithm merges the ESAs of the same de Bruijn sequence; see Figure 5.11 (page 142). The proof of this fact is left to the reader. Section 5.5.5 will show that the common ESA of two strings can be obtained by merging the individual suffix arrays in linear time. As we shall see in a moment, the common suffix array of S^1 and S^2 can easily be obtained from the generalized suffix array of S^1 and S^2 , defined as follows.

Definition 5.4.1 Given two strings S^1 and S^2 that do not contain the special characters $\#$ and $\$$, the suffix array SA of the string $S = S^1\#S^2\$$ is called the *generalized suffix array* of S^1 and S^2 . If S^1 and S^2 have $\#$ and $\$$, respectively, at the end (and nowhere else), then their generalized suffix array is the suffix array SA of the string $S = S^1S^2$. If the generalized suffix array of S^1 and S^2 is enhanced with further arrays, e.g. with the LCP-array, then it will be called *generalized enhanced suffix array* or GESA for short.

In the rest of Section 5.4, it is convenient to assume that S^1 and S^2 are already terminated by $\#$ and $\$$.

i	SA	LCP	$S_{SA[i]}$	$D[i]$	$SA'[i]$	$S_{SA'[i]}^{D[i]}$
1	10	-1	# <i>acaaacatat</i> \$	1	10	#
2	21	0	<i>\$</i>	2	11	<i>\$</i>
3	13	0	<i>aaacatat</i> \$	2	3	<i>aaacatat</i> \$
4	14	2	<i>aacatat</i> \$	2	4	<i>aacatat</i> \$
5	3	2	<i>aataatg</i> # <i>acaaacatat</i> \$	1	3	<i>aataatg</i> #
6	6	3	<i>aatg</i> # <i>acaaacatat</i> \$	1	6	<i>aatg</i> #
7	11	1	<i>acaaacatat</i> \$	2	1	<i>acaaacatat</i> \$
8	15	3	<i>acatat</i> \$	2	5	<i>acatat</i> \$
9	19	1	<i>at</i> \$	2	9	<i>at</i> \$
10	4	2	<i>ataatg</i> # <i>acaaacatat</i> \$	1	4	<i>ataatg</i> #
11	17	3	<i>atat</i> \$	2	7	<i>atat</i> \$
12	7	2	<i>atg</i> # <i>acaaacatat</i> \$	1	7	<i>atg</i> #
13	12	0	<i>caaacatat</i> \$	2	2	<i>caaacatat</i> \$
14	16	2	<i>catat</i> \$	2	6	<i>catat</i> \$
15	1	1	<i>ctaataatg</i> # <i>acaaacatat</i> \$	1	1	<i>ctaataatg</i> #
16	9	0	<i>g</i> # <i>acaaacatat</i> \$	1	9	<i>g</i> #
17	20	0	<i>t</i> \$	2	10	<i>t</i> \$
18	2	1	<i>taataatg</i> # <i>acaaacatat</i> \$	1	2	<i>taataatg</i> #
19	5	4	<i>taatg</i> # <i>acaaacatat</i> \$	1	5	<i>taatg</i> #
20	18	2	<i>tat</i> \$	2	8	<i>tat</i> \$
21	8	1	<i>tg</i> # <i>acaaacatat</i> \$	1	8	<i>tg</i> #
22		-1				

Figure 5.34: The suffix array of $S = \textit{ctaataatg}\#\textit{acaaacatat}\$$ coincides with the common suffix array of the strings $S^1 = \textit{ctaataatg}\#$ and $S^2 = \textit{acaaacatat}\$$.

Figure 5.34 shows that the lexicographic order of the suffixes of $S = S^1 S^2$ corresponds to the lexicographic order of all suffixes of S^1 and S^2 . In fact, the common suffix array of S^1 and S^2 can easily be obtained from the suffix array SA of S as follows:

$$D[i] = \begin{cases} 1 & \text{if } SA[i] \leq n_1 + 1 \\ 2 & \text{otherwise} \end{cases}$$

and

$$SA'[i] = \begin{cases} SA[i] & \text{if } D[i] = 1 \\ SA[i] - (n_1 + 1) & \text{if } D[i] = 2 \end{cases}$$

Conversely, given the arrays D and SA' , the array SA can readily be computed in linear time. Thus, the common suffix array and the suffix array

of the concatenated strings are two representations of the same information. In the following, we identify them and call both the generalized suffix array of S^1 and S^2 . In applications, we will always use the representation that best suits our purpose.

5.4.2 Longest common substring

Definition 5.4.2 A *common substring* of two strings S^1 and S^2 is a string that is a substring of both S^1 and S^2 .

The problem of finding a *longest common substring* of two or more strings is a classical problem in computer science. In 1970 Donald E. Knuth conjectured that a linear-time algorithm for solving this problem cannot exist. However, when Weiner [330] showed in 1973 that the suffix tree of a string can be built in linear time, it became clear that he was wrong. A solution to the longest common substring problem that uses the generalized suffix tree of two strings S^1 and S^2 can be found, for example, in [139]. Here we present a solution based on the GESA of S^1 and S^2 . The following theorem contains the key idea of our solution.

Theorem 5.4.3 Given the GESA (of size n) of the strings S^1 and S^2 , define

$$M = \{i \mid 2 \leq i \leq n \text{ and } D[i-1] \neq D[i]\}$$

to be the set of all indices i so that the two consecutive entries $\text{SA}[i-1]$ and $\text{SA}[i]$ belong to different strings. Moreover, let j be an index from M so that $\text{LCP}[j] = \max\{\text{LCP}[i] \mid i \in M\}$. Then $S[\text{SA}[j]..\text{SA}[j] + \text{LCP}[j] - 1]$ is a longest common substring of S^1 and S^2 .

Proof $\text{SA}[j-1]$ and $\text{SA}[j]$ belong to different strings because $j \in M$. It is a property of the ESA of S (the GESA of S^1 and S^2) that the length of the longest common prefix of the suffixes $S_{\text{SA}[j-1]}$ and $S_{\text{SA}[j]}$ of S is $\text{LCP}[j]$. In particular, the string

$$S[\text{SA}[j-1]..\text{SA}[j-1] + \text{LCP}[j] - 1] = S[\text{SA}[j]..\text{SA}[j] + \text{LCP}[j] - 1]$$

is a common substring of S^1 and S^2 . (We observe that if $\text{LCP}[j] = 0$, then $S[\text{SA}[j]..\text{SA}[j] + \text{LCP}[j] - 1]$ is the empty string, and the empty string is a common substring of S^1 and S^2 .) For an indirect proof of the theorem, suppose that there is a longer common substring ω of S^1 and S^2 . Then there exist indices k and l with $1 \leq k < l \leq n$ so that $D[k] \neq D[l]$ and ω is a common prefix of the suffixes $S_{\text{SA}[k]}$ and $S_{\text{SA}[l]}$ of S . Clearly, ω is also a prefix of every suffix $S_{\text{SA}[m]}$ with $k < m < l$ (if such an m exists). Let m be an index so that $k < m \leq l$ and $D[m-1] \neq D[m]$; such an index must exist because $D[k] \neq D[l]$. Consequently, $m \in M$. Moreover, since ω is a

Algorithm 5.22 Finding a longest common substring of S^1 and S^2 .

```

construct the generalized enhanced suffix array of  $S^1$  and  $S^2$ 
/* the first two entries correspond to # and $ */
j ← 3          /* LCP[3] = 0 */
for i ← 4 to n do          /* n = n1 + n2 + 2 */
    if D[i - 1] ≠ D[i] then
        if LCP[i] > LCP[j] then
            j ← i
output S[SA[j]..SA[j] + LCP[j] - 1]

```

common prefix of $S_{SA[m-1]}$ and $S_{SA[m]}$, it follows that $LCP[m] \geq |\omega| > LCP[j]$. This contradicts the fact that $LCP[j] = \max\{LCP[i] \mid i \in M\}$. \square

It is a corollary to Theorem 5.4.3 that Algorithm 5.22 returns a longest common substring of two strings S^1 and S^2 . In the example of Figure 5.34, Algorithm 5.22 returns $S[SA[11]..SA[11] + LCP[11] - 1] = S[17..19] = ata$.

Exercise 5.4.4 A *palindrome* is a string ω that reads the same backwards as forwards, i.e., $\omega = \omega^{rev}$, where ω^{rev} denotes the reverse string of ω .

Given string S of length n , an odd-length substring ω of S is called a *maximal palindrome of radius k with midpoint q* if $\omega = S[q - k..q + k]$ is a palindrome but $S[q - (k + 1)..q + k + 1]$ is not a palindrome. For example, if $S = gtaacacaagtt$, then $\omega = aacacaa$ is a maximal palindrome of radius 3 with midpoint 6. Similarly, an even-length substring ω of S is called a *maximal palindrome of radius k with midpoint q* if $\omega = S[q - k + 1..q + k]$ is a palindrome but $S[q - k..q + k + 1]$ is not a palindrome. For example, if $S = gtaaccaagtt$, then $\omega = aaccaa$ is a maximal palindrome of radius 3 with midpoint 5.

A substring ω of S is called a *maximal palindrome* if it is a maximal palindrome of radius k with midpoint q for some k with $1 \leq k \leq n/2$ and some q with $1 \leq q \leq n$.

Develop an algorithm that identifies all maximal palindromes of a string S of length n in $O(n)$ time.

Hint: Focus on how to find all even-length maximal palindromes. The odd-length maximal palindromes can be found similarly.

- Prove that an even-length string ω is a palindrome if and only if $(\omega[1..m/2])^{rev} = \omega[m/2 + 1..m]$, where $m = |\omega|$.
- For a fixed position q in S , show that the radius k of the (even-length) maximal palindrome with midpoint q (if there is one) can be computed by $k = |\text{lcp}(S_{n-q+1}^{rev}, S_{q+1})|$.
- Build the GESA of S and S^{rev} and use Lemma 4.2.8.

5.4.3 Finding exact matches

In this section, we tackle a problem that has its origin in genome comparisons. Nowadays, the DNA sequences of entire genomes are being determined at a rapid rate. When the genome sequences of closely related organisms become available, one of the first questions researchers ask is how they align. (Alignments will be discussed in Chapter 8.) This alignment may help, for example, in understanding why a strain of a bacterium is pathogenic or resistant to antibiotics while another is not. The starting point for any comparison of large genomes (like mammalian or plant genomes) is the computation of exact matches between their DNA sequences. Exact matches between two strings S^1 and S^2 fall into three basic categories:

- common k -mers (common substrings of a fixed length k),
- maximal unique matches (these occur only once in S^1 and S^2),
- maximal exact matches (these cannot be extended in either direction towards the beginning or end of S^1 and S^2 without allowing for a mismatch).

Here we show how exact matches that belong to the last two categories can be computed. In Section 8.3, these will be used for whole genome alignment. That is the reason why we are not really interested in the matching substrings, but rather in their start positions and lengths.

Definition 5.4.5 An *exact match* between two strings S^1 and S^2 is a triple (ℓ, p_1, p_2) so that $S^1[p_1..p_1 + \ell - 1] = S^2[p_2..p_2 + \ell - 1]$. An exact match is called *right maximal* if $S^1[p_1 + \ell] \neq S^2[p_2 + \ell]$ (note that $S^1[n_1 + 1] = \#$ and $S^2[n_2 + 1] = \$$). It is called *left maximal* if $S^1[p_1 - 1] \neq S^2[p_2 - 1]$.³ A left and right maximal exact match is called *maximal exact match* (MEM).

As an example, consider the exact match $(2, 4, 7)$ between the strings $S^1 = \text{ctaataatg}\#$ and $S^2 = \text{acaaacatat}\$$ (note that $S^1[4..5] = \text{at} = S^2[7..8]$). It is left maximal because $S^1[3] = a \neq c = S^2[6]$ but it is not right maximal as $S^1[6] = a = S^2[9]$. By contrast, the exact match $(2, 4, 9)$ is both left and right maximal because $S^1[3] = a \neq t = S^2[8]$ and $S^1[6] = a \neq \$ = S^2[11]$.

Lemma 5.4.6 A triple (ℓ, p_1, p_2) is an exact match between two strings S^1 and S^2 if and only if $\langle (p_1, p_1 + \ell - 1), (p_2 + n_1 + 1, p_2 + n_1 + \ell) \rangle$ is a repeated pair in the string $S = S^1 S^2$. An exact match (ℓ, p_1, p_2) is (left/right) maximal if and only if the repeated pair $\langle (p_1, p_1 + \ell - 1), (p_2 + n_1 + 1, p_2 + n_1 + \ell) \rangle$ is (left/right) maximal.

³To cope with boundary cases, we tacitly assume that $S^1[0] = \$$ and $S^2[0] = \#$.

Proof Straightforward. \square

Consequently, maximal exact matches can be computed by the algorithm that computes maximal repeated pairs; see Section 5.3.4. (A different algorithm is presented in Section 7.6.2.)

Definition 5.4.7 A *maximal unique match* (MUM) between two strings S^1 and S^2 is a maximal exact match (ℓ, p_1, p_2) between them so that the string $S^1[p_1..p_1 + \ell - 1] = S^2[p_2..p_2 + \ell - 1]$ occurs exactly once in S^1 and once in S^2 .

The exact match $(3, 4, 7)$ between $S^1 = \text{ctaataatg}\#$ and $S^2 = \text{acaaacatat}\$$ is a maximal unique match because $S^1[4..6] = \text{ata} = S^2[7..9]$ occurs exactly once in S^1 and once in S^2 . As a negative example, consider the maximal exact match $(2, 4, 9)$ between S^1 and S^2 : it is not unique.

Lemma 5.4.8 A triple (ℓ, p_1, p_2) is a maximal unique match between two strings S^1 and S^2 if and only if there is an index i in the GESA of S^1 and S^2 so that

1. $\text{SA}[i - 1] = p_1$ and $\text{SA}[i] = p_2 + n_1 + 1$ or vice versa,
2. $\ell = \text{LCP}[i]$,
3. $S[\text{SA}[i - 1] - 1] \neq S[\text{SA}[i] - 1]$,⁴
4. $\text{LCP}[i] > \text{LCP}[i - 1]$ and $\text{LCP}[i] > \text{LCP}[i + 1]$.

Proof “if”: It is a consequence of conditions (1) and (2) that (ℓ, p_1, p_2) is a right maximal exact match. By (3), it is also left maximal, hence maximal. Condition (4) implies that there is no other suffix with prefix $S^1[p_1..p_1 + \ell - 1] = S^2[p_2..p_2 + \ell - 1]$. It follows that (ℓ, p_1, p_2) is a MUM.

“only if”: If (ℓ, p_1, p_2) is a maximal unique match, then the suffixes $S^1_{p_1}$ and $S^2_{p_2}$ share a common prefix ω of length ℓ and none of the other suffixes starts with ω . Therefore, $S^1_{p_1}$ and $S^2_{p_2}$ must occur consecutively in the GESA of S^1 and S^2 . In other words, there is an index $i \geq 1$ so that (1) $\text{SA}[i - 1] = p_1$ and $\text{SA}[i] = p_2 + n_1 + 1$ or vice versa. Because none of the other suffixes starts with ω , we have (4) $\text{LCP}[i] > \text{LCP}[i - 1]$ and $\text{LCP}[i] > \text{LCP}[i + 1]$. Moreover, (2) $\ell = \text{LCP}[i]$ and (3) $S[\text{SA}[i - 1] - 1] \neq S[\text{SA}[i] - 1]$ hold because (ℓ, p_1, p_2) is a maximal exact match. \square

As a corollary, it follows that Algorithm 5.23 outputs all MUMs between two strings S^1 and S^2 .

Exercise 5.4.9 Apply Algorithm 5.23 to the strings $S^1 = \text{ccaacga}\#$ and $S^2 = \text{cagacga}\$$.

Exercise 5.4.10 Modify Algorithm 5.23 so that it outputs only maximal unique matches of length $\geq \ell$, where ℓ is a user-defined length threshold.

⁴To cope with boundary cases, we tacitly assume that $S[0] = \$$.

Algorithm 5.23 Finding maximal unique matches of S^1 and S^2 .

```

construct the generalized enhanced suffix array of  $S^1$  and  $S^2$ 
/* the first two entries correspond to # and $ */
for  $i \leftarrow 4$  to  $n$  do           /*  $n = n_1 + n_2 + 2$  */
    if  $D[i-1] \neq D[i]$  then
        if  $S[SA[i-1]-1] \neq S[SA[i]-1]$  then
            if  $LCP[i] > LCP[i-1]$  and  $LCP[i] > LCP[i+1]$  then
                if  $D[i-1] = 1$  then
                    output  $(LCP[i], SA[i-1], SA[i] - n_1 - 1)$ 
                else
                    output  $(LCP[i], SA[i], SA[i-1] - n_1 - 1)$ 

```

5.5 Traversals with suffix links

Suffix links are a key feature of older linear-time suffix tree construction algorithms [123, 218, 315]. We did not yet need them because the suffix insertion algorithm presented in Section 4.4 constructs a suffix tree with the aid of a suffix array. However, suffix links are very useful in some applications; a prime example is the computation of matching statistics.

5.5.1 Suffix links in the suffix tree

Definition 5.5.1 Let $\overline{a\omega}$ be an internal node in the suffix tree ST of a string S that is terminated by $\$$. A pointer $slink(\overline{a\omega})$ from $\overline{a\omega}$ to the internal node $\overline{\omega}$ is called a *suffix link*. The suffix link of the root node points to the root node itself.

As a matter of fact, it is *a priori* not clear that for any internal node $\overline{a\omega}$, there is also an internal node $\overline{\omega}$ in ST. However, it is not difficult to see that this is indeed the case. Because every internal node $\overline{a\omega}$ in ST is branching, there are two leaves i and j in the subtrees of different children of $\overline{a\omega}$. That is, $a\omega$ is the longest common prefix of the suffixes S_i and S_j of S . Thus, ω is the longest common prefix of the suffixes S_{i+1} and S_{j+1} of S . This implies that $\overline{\omega}$ is an internal node in ST.

As already mentioned, some suffix tree construction algorithms like Ukkonen's online algorithm [315] determine suffix links while the suffix tree is built. Because we constructed suffix trees via suffix arrays, we next provide a linear-time algorithm that establishes suffix links after the suffix tree has been constructed.

Preprocess the suffix tree ST of string S for constant time LCA queries. In a linear-time bottom-up traversal of ST, label each internal node $\overline{a\omega}$ with a pair of leaves (i, j) so that i and j are in the subtrees of different children

of $\overline{a\omega}$. To establish the suffix link from $\overline{a\omega}$, we must find the internal node $\overline{\omega}$. It is clear from the preceding considerations that $\overline{\omega} = \text{LCA}(i+1, j+1)$. Thus, $\overline{\omega}$ can be found by a constant time LCA query provided that the leaves ω with suffix numbers $i+1$ and $j+1$ can be accessed in constant time. This can, for example, be accomplished with an array of n pointers, where the i -th pointer points to the leaf numbered i .

Consequently, all suffix links for the $O(n)$ internal nodes of the suffix tree can be computed in $O(n)$ time. In the next section, we use the same idea in the context of suffix arrays.

5.5.2 Suffix links in the lcp-interval tree

This section shows that suffix links can be computed on the fly (i.e., they are not stored explicitly) if SA, ISA, PSV_{LCP} , and NSV_{LCP} , and RMQ_{LCP} are available. Because we deal with enhanced suffix arrays (not with suffix trees), the string S of length n must not necessarily be terminated by $\$$.

Definition 5.5.2 For every (non-singleton) lcp-interval $\ell\text{-}[i..j]$ representing a string $a\omega$, the ω -interval is called the *suffix link interval* of $\ell\text{-}[i..j]$. The suffix link interval of the lcp-interval $0\text{-}[1..n]$ is $0\text{-}[1..n]$ itself.

Observe that the suffix link interval of an lcp-interval of lcp-value 1 is the lcp-interval $0\text{-}[1..n]$. The following lemma shows that every suffix link interval is in fact a non-singleton lcp-interval.

Lemma 5.5.3 For every lcp-interval $\ell\text{-}[i..j]$ representing a string $a\omega$, where $a \in \Sigma$, there is an lcp-interval of lcp-value $\ell-1$ representing the string ω .

Proof By definition, $a\omega$ is the longest common prefix of $S_{\text{SA}[i]}, \dots, S_{\text{SA}[j]}$. In particular, we have $S[\text{SA}[i]..\text{SA}[i] + \ell - 1] = a\omega = S[\text{SA}[j]..\text{SA}[j] + \ell - 1]$ and $S[\text{SA}[i] + \ell] \neq S[\text{SA}[j] + \ell]$. If we omit the first character a in $a\omega$, then we get $S[\text{SA}[i]+1..\text{SA}[i]+\ell-1] = \omega = S[\text{SA}[j]+1..\text{SA}[j]+\ell-1]$ and $S[\text{SA}[i]+\ell] \neq S[\text{SA}[j]+\ell]$. In other words, ω is the longest common prefix of $S_{\text{SA}[i]+1}, \dots, S_{\text{SA}[j]+1}$. Hence the ω -interval is an lcp-interval with lcp-value $|\omega| = \ell - 1$. \square

The following function will help us to find suffix link intervals.

Definition 5.5.4 For each i with $\text{SA}[i] < n$, define $\psi[i] = \text{ISA}[\text{SA}[i] + 1]$.

So $\psi[i]$ is the index at which the suffix $S_{\text{SA}[i]+1}$ occurs in the suffix array. Let us turn to the problem of finding the suffix link interval of an lcp-interval $\ell\text{-}[i..j]$ representing a string $a\omega$. The suffixes $S_{\text{SA}[i]}$ and $S_{\text{SA}[j]}$ at the indices i and j have $a\omega$ as longest common prefix. The suffixes $S_{\text{SA}[i]+1}$ and $S_{\text{SA}[j]+1}$ in turn have ω as longest common prefix. They can be found at the indices $\psi[i] = \text{ISA}[\text{SA}[i] + 1]$ and $\psi[j] = \text{ISA}[\text{SA}[j] + 1]$. Therefore, the

indices $\psi[i]$ and $\psi[j]$ belong to the ω -interval. We know by Lemma 5.5.3 that this ω -interval is an lcp-interval of lcp-value $\ell - 1$, but we do not know the boundaries of this interval. These boundaries can be identified with the help of the arrays PSV_{LCP} and NSV_{LCP} ; see Definition 4.3.7. First, the range minimum query $\text{RMQ}_{\text{LCP}}(\psi[i] + 1, \psi[j])$ yields an $(\ell - 1)$ index of the ω -interval, say index k , and then we obtain the boundaries of the ω -interval by $p = \text{PSV}_{\text{LCP}}[k]$ and $q = \text{NSV}_{\text{LCP}}[k] - 1$; see Lemma 4.3.8. Algorithm 5.24 shows pseudo-code for this method. It is clear that the computation of one suffix link can be done in constant time.

Algorithm 5.24 Computation of the suffix link of an lcp-interval $[i..j]$.

```

 $k \leftarrow \text{RMQ}_{\text{LCP}}(\psi[i] + 1, \psi[j])$ 
return  $[\text{PSV}_{\text{LCP}}[k].. \text{NSV}_{\text{LCP}}[k] - 1]$ 

```

For example, consider the lcp-interval $2\text{-}[1..2]$ representing the string aa in Figure 5.35. The range minimum query $\text{RMQ}_{\text{LCP}}(\psi[1] + 1, \psi[2]) = \text{RMQ}_{\text{LCP}}(3, 4)$ yields the 1-index 3 and thus the boundaries of the a -interval are $p = \text{PSV}_{\text{LCP}}[3] = 1$ and $q = \text{NSV}_{\text{LCP}}[k] - 1 = 6$. Therefore, the suffix link interval of the lcp-interval $2\text{-}[1..2]$ is $1\text{-}[1..6]$.

Some readers may wish to navigate in the (virtual) lcp-interval tree without range minimum queries. We would like to point out that one can replace the range minimum query $\text{RMQ}_{\text{LCP}}(\psi[i] + 1, \psi[j])$ in the computation of a suffix link interval by a simple binary search. The task of the range minimum query $\text{RMQ}_{\text{LCP}}(\psi[i] + 1, \psi[j])$ is to find an $(\ell - 1)$ -index, from which the boundaries of the suffix link interval can be identified by means of the arrays PSV_{LCP} and NSV_{LCP} . We know that the suffixes $S_{\text{SA}[\psi[i]]}$ and $S_{\text{SA}[\psi[j]]}$ have a common prefix of length $\ell - 1$ and that $c = S[\text{SA}[\psi[i]] + \ell - 1] \neq S[\text{SA}[\psi[j]] + \ell - 1]$. A binary search yields the smallest index k , $\psi[i] + 1 \leq k \leq \psi[j]$, so that $S[\text{SA}[k] + \ell - 1] \neq c$. Clearly, this index k satisfies $\text{LCP}[k] = \ell - 1$.

Exercise 5.5.5 Show that the boundaries of a suffix link interval can be identified without PSV_{LCP} and NSV_{LCP} in $O(\log n)$ time by a binary search on the LCP-array (using range minimum queries), provided that an $(\ell - 1)$ index of the suffix link interval is known.

5.5.3 Computing suffix links space efficiently

In this section, we present a linear-time algorithm that computes and stores all suffix link intervals explicitly. Of course, the previous algorithm can be used for this, but a big drawback is its space consumption. Aside from the arrays SA and LCP , it needs the data structure supporting constant time range minimum queries and three additional arrays ψ , PSV_{LCP} , and NSV_{LCP} .

i	SA	ISA	ψ	LCP	$S_{SA[i]}$	PSV _{LCP}	NSV _{LCP}	slink
1	3	3	2	-1	<i>aaacatat</i>			
2	4	7	4	2	<i>aacatat</i>	1	3	[1..6]
3	1	1	7	1	<i>acaaacatat</i>	1	7	[1..10]
4	5	2	8	3	<i>acatat</i>	3	5	[7..8]
5	9	4	9	1	<i>at</i>	1	7	
6	7	8	10	2	<i>atat</i>	5	7	[9..10]
7	2	6	1	0	<i>caaacatat</i>	1	11	[1..10]
8	6	10	6	2	<i>catat</i>	7	9	[1..6]
9	10	5		0	<i>t</i>	1	11	
10	8	9	5	1	<i>tat</i>	9	11	[1..10]
11				-1				

Figure 5.35: The enhanced suffix array of the string $S = acaaacatat$.

The algorithm that is described below was presented in [165] without a proof of its linear-time complexity. It bears a strong resemblance to the computation of the failure links in the Aho-Corasick algorithm; cf. Section 2.5. Other algorithms to compute suffix links can be found e.g. in [1,209].

First of all, how can suffix links be stored? Remember that in applications the lcp-interval tree is traversed but not really constructed. So in contrast to suffix trees, we do not implement suffix links as pointers. Instead, suffix links are stored in an array *slink*, called *suffix link table*. Of course, it is sufficient to store the left and right boundary of an interval (or alternatively, the left boundary and the size of the interval). As discussed in Section 4.3.3, information coupled to an lcp-interval can be stored at different locations. In the suffix link table *slink*, we store the suffix link interval $[p..q]$ of an lcp-interval $\ell\text{-}[i..j]$ at the first ℓ -index of $[i..j]$; see Figure 5.35 for an example.

Our space-economical algorithm computes suffix links in linear time by a top-down traversal of the lcp-interval tree. In our opinion, the explanation of how the algorithm works is easier to grasp if we use the suffix tree instead of the lcp-interval tree. That is why we start with suffix trees.

Computing suffix links in a suffix tree

The suffix link of the root node points to the root node itself. To compute the other suffix links, we traverse the suffix tree in a top-down fashion. Whenever we encounter an internal node having depth $d \geq 1$, we use the fact that the suffix link of its parent node (which has depth $d - 1$) has already been computed. More precisely, if node $\bar{a}\bar{w}$ is the child of node α ,

then we can follow the suffix link of α and reach node $\text{slink}(\alpha)$ in constant time. There are two cases:

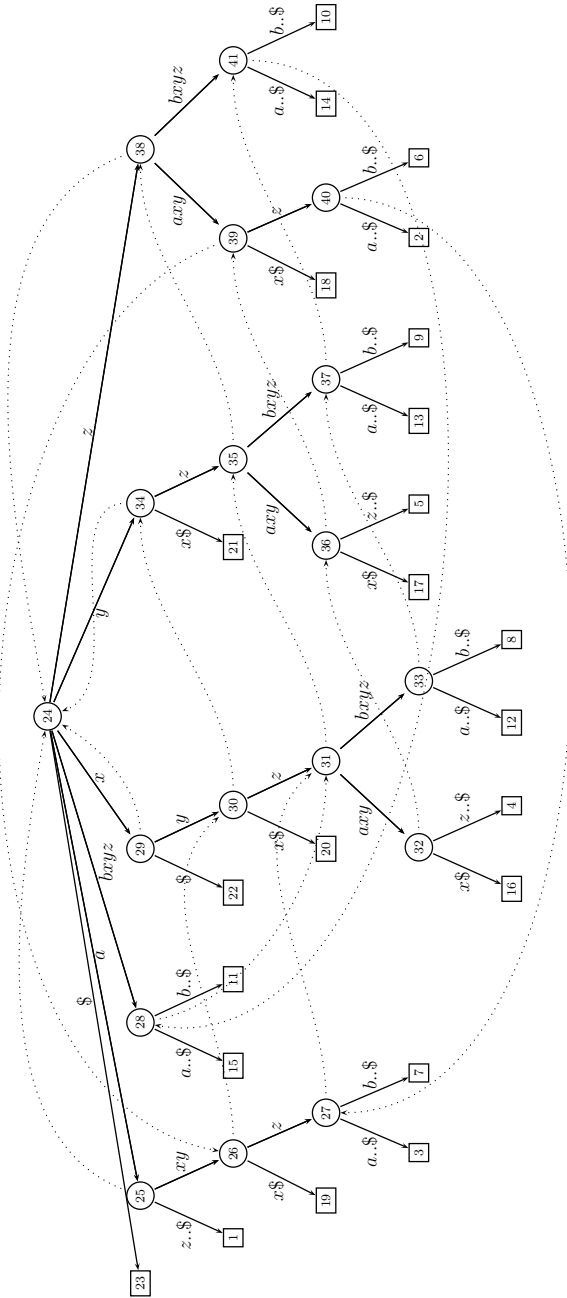
1. If α is not the root node, then (a) $\alpha = \overline{a}u$, where u is a (possibly empty) prefix of ω , (b) $\text{slink}(\alpha) = \overline{u}$, and (c) $\omega = uv$, where v is the label of the edge from $\overline{a}u$ to $\overline{a}v$.
2. If α is the root node, then so is $\text{slink}(\alpha)$.

To deal with both cases simultaneously, in case (2) we set $\overline{u} = \text{root}$ (so $u = \varepsilon$) and $v = \omega$. Now we walk down from node \overline{u} along the path labeled v until the node \overline{v} is found. There is one subtlety, however, which must be made explicit.⁵ Because we know that \overline{v} is a node in the suffix tree, the edge labels on the path from the root to \overline{v} spell out the string $\omega = uv$. Therefore, from node \overline{u} we follow the unique edge whose label, say v_1 , starts with the first character of v and reach the node of $\overline{uv_1}$. This node must be on the path from \overline{u} to \overline{v} and it is unnecessary to inspect the whole label v_1 . Of course, v_1 is a prefix of v and we have $v = v_1v'$ for some string v' . Then we follow the edge whose label v_2 starts with the first character of v' and reach node $\overline{uv_1v_2}$, again without inspecting the rest of v_2 . This process is repeated until the node \overline{v} is found, and the suffix link from $\overline{a}u$ to $\overline{a}v$ is established. It follows from the preceding discussion that the total time to traverse the path is proportional to the number of nodes on it, rather than the number of characters on it. Consequently, the run time of the algorithm that computes all suffix links in this way is proportional to the overall number of nodes traversed.

As an example, we show how the suffix link of the internal node \overline{zaxy} (node 39) in the suffix tree of the string $S = \text{azaxyzaxyzbxyzbxzyxaxys}$ is found; see Figure 5.36. The algorithm follows the suffix link of the parent node \overline{z} (node 38) to the root node, determines the edge whose label starts with the character a , reaches node \overline{a} (node 25), determines the edge whose label starts with the character x , and reaches node \overline{ax} (node 26). Thus, the suffix link of \overline{zaxy} to \overline{axy} is the pointer from node 39 to node 26.

The amortized time analysis of the algorithm goes as follows. In the situation described above, if v is just one character, then the path from node $\text{slink}(\alpha) = \overline{u}$ to node \overline{v} has just one edge. In this case, the identification of \overline{v} takes constant time. The algorithm must traverse extra nodes in the down-walk only if the path from \overline{u} to $\overline{v} = \overline{uv}$ has more than one node. Suppose $v = v_1v_2 \dots v_m$ and the algorithm has to traverse m nodes in the down-walk from node \overline{u} to node \overline{v} , i.e., from \overline{u} to $\overline{uv_1}$, from $\overline{uv_1}$ to $\overline{uv_1v_2}$, etc. We say that v is split into v_1, v_2, \dots, v_m . We charge the split into v_k and v_{k+1} , where $1 \leq k < m$, to the following position p_k in S : Let i be the largest suffix number in the subtree rooted at node $\overline{a}v$; so $S[i..i + |\omega| - 1] = a\omega$.

⁵This is known as the skip and count trick of Ukkonen's suffix tree construction algorithm [315].

Figure 5.36: Suffix tree of $S = azaxyzaxyzbxyzaxyx\$$.

Moreover, let p_k , $i < p_k \leq i + |a\omega| - 1$ be the position in S at which v_{k+1} starts, where $1 \leq k < m$. In other words, the split into v_k and v_{k+1} occurs on the path from the root to the leaf with suffix number $i + 1$ between the positions $p_k - 1$ and p_k (between the characters $S[p_k - 1]$ and $S[p_k]$).

To illustrate this, we go back to the example of Figure 5.36. To establish the suffix link of node \overline{bxyz} (node 28), the algorithm starts at the root and traverses the nodes \overline{x} , \overline{xy} , and \overline{xyz} . In this case, $v = xyz$ is split into $v_1 = x$, $v_2 = y$, and $v_3 = z$. The largest suffix number in the subtree rooted at node \overline{bxyz} is 15 and we have $S[15..18] = bxyz$. The split positions are $p_1 = 17$ (because $S[16..17] = xy$ is split into $v_1 = x$ and $v_2 = y$) and $p_2 = 18$ (because $S[17..18] = yz$ is split into $v_2 = y$ and $v_3 = z$). As another example, consider the construction of the suffix link of node \overline{axy} (node 26). Since the suffix link of the parent node \overline{a} (node 25) points to the root, the algorithm starts at the root and traverses the nodes \overline{x} and \overline{xy} . In this case, the split position is 21 (because $S[20..21] = xy$ is split into x and y).

We claim that for every position p in S , there is at most one split. To substantiate the claim, we need the following lemma.

Lemma 5.5.6 *Let p be a position in S and let $i < p$ be a suffix number so that the label S_i of the path from the root to leaf i is split between the positions $p - 1$ and p , i.e., the characters $S[p - 1]$ and $S[p]$ belong to different edge labels. Then the same is true for every suffix number j with $i < j < p$.*

Proof Let β be the node in the suffix tree that separates the positions $p - 1$ and p on the path from the root to the leaf i . That is, $S[p - 1]$ is the last character of the label of β 's incoming edge and $S[p]$ is the first character of the label of an outgoing edge. It is easy to verify that node $\text{slink}(\beta)$ separates the positions $p - 1$ and p on the path from the root to the leaf $i + 1$. The lemma follows by repeating the argument. \square

Suppose that a split between the positions $p - 1$ and p occurs in the suffix tree, and let i be the smallest leaf number so that there is such a split on the path from the root to leaf i (in other words, on every path from the root to leaf k , where $k < i$, the positions $p - 1$ and p belong to the same edge label). According to Lemma 5.5.6, on every path from the root to leaf j , where $i < j < p$, the positions $p - 1$ and p are split. Obviously, for all l with $p \leq l$, the path from the root to leaf l does not contain the position $p - 1$. In summary, for every position p in S , there is at most one split. It follows as a consequence that the overall number of extra nodes that have to be traversed in the down-walks cannot exceed n , resulting in a worst-case time complexity of $O(n)$.

Computing suffix links on enhanced suffix arrays

From now on we deal with enhanced suffix arrays, so the string S of length n must not necessarily be terminated by $\$$. The algorithm described above

Algorithm 5.25 Suppose the suffix link interval $[p..q]$ of an ℓ -interval $[i..j]$ is already known. *computeLink*(ℓ, p, q, i', j') computes the suffix link interval $[p'..q']$ of a child interval ℓ' - $[i'..j']$ of $[i..j]$, stores it at the first ℓ' -index m' of $[i'..j']$, and returns the quadruple (ℓ', m', p', q') .

```

 $m' \leftarrow \text{RMQ}_{\text{LCP}}(i' + 1, j')$       /* find first  $\ell'$ -index of  $[i'..j']$  */
 $\ell' \leftarrow \text{LCP}[m']$ 
 $[p'..q'] \leftarrow [p..q]$ 
 $c \leftarrow \max\{\ell - 1, 0\}$ 
while  $c \neq \ell' - 1$  do
     $[p'..q'] \leftarrow \text{getInterval}([p'..q'], S[\text{SA}[i'] + 1 + c])$ 
     $c \leftarrow \text{LCP}[\text{RMQ}_{\text{LCP}}(p' + 1, q')]$ 
 $\text{slink}[m'] \leftarrow [p'..q']$ 
return  $(\ell', m', p', q')$ 

```

works on lcp-interval trees as follows (Algorithm 5.26 shows pseudo-code of the final algorithm). In a top-down traversal of the lcp-interval tree (cf. Algorithm 4.8 on page 98), let ℓ - $[i..j]$ be the current lcp-interval and suppose that its suffix link interval $[p..q]$ has already been computed. The suffix link interval of a child interval ℓ' - $[i'..j']$ of $[i..j]$ is computed by the procedure *computeLink* with input ℓ, p, q, i', j' ; see Algorithm 5.25 for pseudo-code.

Algorithm 5.25 determines the first ℓ' -index m' of $[i'..j']$ by the range minimum query $\text{RMQ}_{\text{LCP}}(i' + 1, j')$ (or alternatively, with the help of the child table; cf. Theorem 4.3.25). This takes constant time. Furthermore, the interval $[p'..q']$, which will eventually store the suffix link interval we are searching for, is initially set to $[p..q]$. To determine the suffix link interval of $[i'..j']$, we apply Lemma 5.5.3. Let a_ω be the string that is represented by ℓ' - $[i'..j']$. Then, the suffix link interval of $[i'..j']$ is the ω -interval, where $\omega = S[\text{SA}[i'] + 1.. \text{SA}[i'] + \ell' - 1]$. If $\ell' = 1$, then the procedure returns (ℓ', m', p', q') . Otherwise $\ell' > 1$. In this case, the procedure follows the path in the lcp-interval tree that corresponds to the string $\omega = S[\text{SA}[i'] + 1.. \text{SA}[i'] + \ell' - 1]$, starting at the lcp-interval $[p'..q'] = [p..q]$ with lcp-value $\ell - 1$. In case $\ell - 1 \leq 0$, the node $[p'..q']$ is the root node $[1..n]$ and the procedure *getInterval*($[1..n], S[\text{SA}[i'] + 1]$) (cf. Algorithm 5.1 on page 118) is called to find the $S[\text{SA}[i'] + 1]$ -interval. Otherwise, if $\ell - 1 > 0$, then $[p'..q'] = [p..q]$ is not the root node and the procedure *getInterval*($[p'..q'], S[\text{SA}[i'] + \ell]$) is called to find the $S[\text{SA}[i'] + \ell]$ -interval. In Algorithm 5.25, both cases are treated simultaneously by setting c to $\max\{\ell - 1, 0\}$ and calling *getInterval*($[p'..q'], S[\text{SA}[i'] + c + 1]$). Starting from the lcp-interval returned by that procedure call, Algorithm 5.25 further follows the path in the lcp-interval tree until the suffix link interval of $[i'..j']$ is found (this is the case if $c = \ell' - 1$). This path can be found charac-

Algorithm 5.26 Given an lcp-interval ℓ - $[i..j]$ with first ℓ -index fst , for which the suffix link interval $[p..q]$ has already been computed, the procedure $LinkTopDown(\ell, i, j, fst, p, q)$ recursively constructs the suffix link table of all lcp-intervals in the lcp-interval tree rooted at $[i..j]$.

```

 $k \leftarrow i$ 
 $m \leftarrow fst$ 
repeat
  if  $k \neq m - 1$  then
     $(\ell', m', p', q') \leftarrow computeLink(\ell, p, q, k, m - 1)$ 
     $LinkTopDown(\ell', k, m - 1, m', p', q')$ 
     $k \leftarrow m$  /*  $k$  is left boundary of the next child interval */
  if  $k = j$  then
    return /* there is no more non-singleton child interval */
  else
     $m \leftarrow RMQ(k + 1, j)$  /*  $m$  is the next  $\ell$ -index unless  $LCP[m] \neq \ell$  */
until  $LCP[m] \neq \ell$ 
/*  $[k..j]$  is the last non-singleton child interval of  $[i..j]$  */
 $(\ell', m', p', q') \leftarrow computeLink(\ell, p, q, k, j)$ 
 $LinkTopDown(\ell', k, j, m', p', q')$ 

```

ter by character if one increments c by 1 in each iteration of the while-loop. As explained above, however, it suffices to use the first character (in the suffix tree, one has to find the edge whose label starts with this first character). That is why we set c to the lcp-value of the current lcp-interval in the while-loop of Algorithm 5.25.

Exercise 5.5.7 Consider the enhanced suffix array of the string $S = acaaacatat$ from Figure 5.35 (page 188). Under the assumption that the suffix link interval $[1..10]$ of the lcp-interval 1 - $[1..6]$ has already been computed, the suffix link interval of the child interval 3 - $[3..4]$ can be computed by the procedure call $computeLink(1, 1, 10, 3, 4)$. Execute the procedure by hand.

The procedure $LinkTopDown(\ell, i, j, fst, p, q)$, detailed in Algorithm 5.26, takes an lcp-interval ℓ - $[i..j]$ with first ℓ -index fst and suffix link interval $[p..q]$ as input and recursively constructs the suffix link table of all lcp-intervals in the lcp-interval tree rooted at $[i..j]$. Therefore, the procedure call $LinkTopDown(0, 1, n, RMQ_{LCP}(2, n), 1, n)$ builds the whole suffix link table, except for the root interval 0 - $[1..n]$.⁶ Because the suffix link interval of this interval is 0 - $[1..n]$ itself, the assignment $slink[RMQ_{LCP}(2, n)] \leftarrow [1..n]$ completes the job.

⁶If S is terminated by $\$,$ then 2 is the first 0-index of $[1..n]$ because $LCP[1] = -1$ and $LCP[2] = 0$; so the range minimum query $RMQ_{LCP}(2, n)$ is superfluous in this case.

Although the first ℓ -index fst of an lcp-interval $\ell\text{-}[i..j]$ can be obtained by $\text{RMQ}_{\text{LCP}}(i + 1, j)$, it is better to have it as a parameter of the procedure *LinkTopDown* in Algorithm 5.26 because its renewed computation can be avoided in this way.

Exercise 5.5.8 Compute the suffix links of the enhanced suffix array from Figure 5.35 with the help of Algorithm 5.26, i.e., execute the procedure call *LinkTopDown*(0, 1, 10, $\text{RMQ}_{\text{LCP}}(2, 10)$, 1, 10) manually.

5.5.4 Matching statistics

As in Section 5.4, we consider two strings S^1 and S^2 of lengths $n_1 + 1$ and $n_2 + 1$, where S^1 is terminated by $\#$ and S^2 is terminated by $\$$.⁷

Matching statistics were introduced by Chang and Lawler [54] in the context of approximate string matching. The following definition is a preliminary definition of the matching statistics of S^2 w.r.t. S^1 (this definition will be refined later).

Definition 5.5.9 The *matching statistics* of S^2 w.r.t. S^1 is an array ms of size $n_2 + 1$ so that for every entry $ms[p_2] = k$ the following holds: $S^2[p_2..p_2 + k - 1]$ is the longest prefix of $S^2_{p_2}$ that occurs as a substring of S^1 .

For ease of presentation, we first show how to compute the matching statistics of S^2 w.r.t. S^1 with the help of a suffix tree. First, the suffix tree ST of S^1 is built.

The naive way to compute $ms[p_2]$, where p_2 is a fixed position in S^2 , is to match the initial characters of $S^2_{p_2}$ against ST by following the unique path of character matches until no further matches are possible. The length of the matching path is $ms[p_2]$. To achieve a linear time algorithm, however, we use the naive method only for $p_2 = 1$. Suppose that the algorithm has just computed $ms[p_2 - 1] = k$ by following a matching path for position $p_2 - 1$, where $p_2 \geq 2$. If the path is empty (i.e., $k = 0$) or it ends within the label of an edge outgoing from the root, then the search for $ms[p_2]$ starts at the root. Otherwise, the matching path is $S^1[p_1 - 1..p_1 + k - 2] = a\omega = S^2[p_2 - 1..p_2 + k - 2]$ for some position p_1 in S^1 , some character $a \in \Sigma$, and some string $\omega \in \Sigma^*$, and it either ends at an internal node $\overline{a\omega}$ or within the label of an edge outgoing from an internal node $\overline{a\omega}$. In the former case, $u = \omega$; in the latter case, let v be the string so that $\omega = uv$. To match the next suffix $S^2_{p_2}$ against ST, one follows the suffix link $\text{slink}(\overline{a\omega})$ from node $\overline{a\omega}$ to node \overline{u} . Because $a\omega = auv$ is a prefix of $S^2_{p_2-1}$, the string $\omega = uv$ is a prefix of $S^2_{p_2}$. That is, the search for $ms[p_2]$ can start at node \overline{u} . Moreover, instead of traversing the path labeled v by examining every character on it, the algorithm merely inspects the first character of each edge; cf. Section

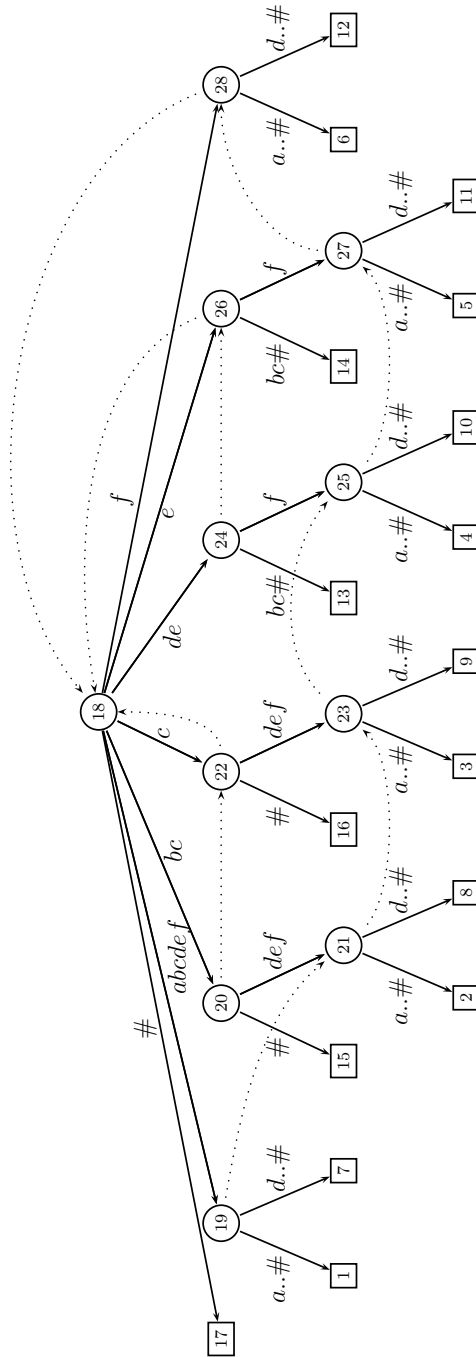
⁷For enhanced suffix arrays, the use of $\#$ and $\$$ is convenient but not really necessary.

5.5.3. This technique is known under the name *skip and count technique*. To elaborate on this, let \bar{u} , $\overline{uv_1}$, $\overline{uv_1v_2}$, \dots , $\overline{uv_1v_2\dots v_m}$ be the path in ST so that $v = v_1v_2\dots v_mv_{m+1}$, where v_{m+1} is either the empty string or there is an edge outgoing from $\overline{uv_1v_2\dots v_m}$ so that v_{m+1} is a non-empty proper prefix of the edge label. For each node $\overline{uv_1v_2\dots v_i}$, one follows the edge whose first character (of the label) coincides with the first character of v_{i+1} . In this manner, the correct child node $\overline{uv_1v_2\dots v_iv_{i+1}}$ is reached in constant time. If $v_{m+1} = \varepsilon$, the matching phase continues by following (character by character) the unique matching path of $S^2_{p_2+k-1}$, starting from node $\bar{\omega} = \overline{uv_1v_2\dots v_m}$. Otherwise $ms[p_2] = k - 1$, because the label of the edge outgoing from $\overline{uv_1v_2\dots v_m}$ with proper prefix v_{m+1} cannot continue with the character $S^2[p_2 + k - 1]$. For an indirect proof of this claim, suppose that the matching path ω continues with the character $S^2[p_2 + k - 1]$. So on the one hand, the string $\omega S^2[p_2 + k - 1]$ occurs in ST. On the other hand, because the matching path of $a\omega = S^2[p_2 - 1..p_2 + k - 2]$ ended in a mismatch $S^1[p_1 + k - 1] = b \neq S^2[p_2 + k - 1]$, it follows that the string ωb also occurs in ST. Consequently, $\bar{\omega}$ must be an internal (branching) node, contradicting the fact that the path ω ended within an edge.

To illustrate the algorithm, let us compute the matching statistics of the string $S^2 = abcdebc\$$ w.r.t. $S^1 = abcdefabcdefdebc\#$. The suffix tree ST of S^1 can be found in Figure 5.37. The unique matching path of $S^2_1 = abcdebc\$$ ends within the edge labeled $abcdef$ outgoing from the root. Since the first five characters $abcde$ match and the first mismatch occurs at position 6 in S^2 , it follows that $ms[1] = 5$. As explained above, the search for $ms[2]$ starts at the root, and it follows the path labeled $bcde$ using the skip and count technique. To be precise, the algorithm follows the edge whose label starts with the character $S^2[2] = b$ and reaches node \bar{bc} (node 20). Then it determines the edge whose label starts with the character $S^2[4] = d$. Because the matching path $bcde$ ends within the edge labeled def , it follows that $ms[2] = 4$. Then, the algorithm follows the suffix link of node \bar{bc} to node \bar{c} (node 22), gets stuck in the edge with label def , and sets $ms[3] = 3$. Thereafter, it first follows the suffix link of node \bar{c} to the root node, then the edge whose label starts with the character $S^2[4] = d$, and reaches node \bar{de} . From there it matches $S^2[6..8] = bc\$$ character by character against the edge with label $bc\#$. Clearly, $ms[4] = 4$ because the matching path $debc$ has length 4. The computation of the rest of the matching statistics is left as an exercise to the reader.

We postpone the time analysis of the algorithm until we have learned how it works on enhanced suffix arrays. The following definition refines our preliminary definition of the matching statistics.

Definition 5.5.10 The *matching statistics* of S^2 w.r.t. S^1 is an array ms of size $n_2 + 1$ so that for every entry $ms[p_2] = (k, [lb..rb])$ the following holds:

Figure 5.37: Suffix tree of $S^1 = abcdefabcdefdebc\#$.

i	SA	LCP	$S_{\text{SA}[i]}$	slink
1	17	-1	#	
2	1	0	abcdefghijkldefdebc#	[1..17]
3	7	6	abcdefghijkldefdebc#	[5..6]
4	15	0	bc#	
5	2	2	bcdefabcdefghijkldefdebc#	[7..9]
6	8	5	bcdefdebc#	[8..9]
7	16	0	c#	
8	3	1	cdefabcdefghijkldefdebc#	[1..17]
9	9	4	cdefdebc#	[11..12]
10	13	0	debc#	
11	4	2	defabcdefghijkldefdebc#	[13..15]
12	10	3	defdebc#	[14..15]
13	14	0	ebc#	
14	5	1	efabcdefghijkldefdebc#	[1..17]
15	11	2	efdebc#	[16..17]
16	6	0	fabcdefghijkldefdebc#	
17	12	1	fdebc#	[1..17]
18		-1		

Figure 5.38: The ESA of the string $S^1 = abcdefabcdefghijkldefdebc\#$.

1. $S^2[p_2..p_2 + k - 1]$ is the longest prefix of $S^2_{p_2}$ that occurs as a substring of S^1 .
2. $[lb..rb]$ is the $S^2[p_2..p_2 + k - 1]$ -interval in the ESA of S^1 .

As an example, consider the ESA of the string $S^1 = abcdefabcdefghijkldefdebc\#$ in Figure 5.38. The matching statistics of $S^2 = abcdebc\$$ w.r.t. S^1 is depicted in Figure 5.39. In the matching statistic $ms[p_2] = (k, [lb..rb])$, the interval $[lb..rb]$ can be a singleton interval; see $ms[4]$ in Figure 5.39. This case occurs if the matching path in the suffix tree ends within an edge to a leaf. If $[lb..rb]$ is a non-singleton lcp-interval, then its lcp-value may differ from k . Consider for instance $ms[1] = (5, [2..3])$ in Figure 5.39. The lcp-interval $[2..3]$ has lcp-value 6 but $k = 5$. This is the case if the matching path in the suffix tree ends within an edge to an internal node. The interval $[lb..rb]$ is an lcp-interval of lcp-value k only if the matching path in the suffix tree ends at an internal node; see $ms[6]$ in Figure 5.39.

If the matching path ends within an edge to a node, it is helpful to have its parent node at hand. In the lcp-interval tree, this parent node is the maximal lcp-interval that matches a prefix of $S^2_{p_2}$, defined as follows.

p_2	1	2	3	4	5	6	7	8
$ms[p_2]$	5, [2..3]	4, [5..6]	3, [8..9]	4, [10..10]	3, [13..13]	2, [4..6]	1, [7..9]	0, [1..17]
interval	[1..17]	[4..6]	[7..9]	[10..12]	[13..15]	[4..6]	[7..9]	[1..17]

Figure 5.39: Matching statistics of the string $S^2 = abcdebc\$$ w.r.t. the string $S^1 = abcdefabcdefdebc\#$; cf. Figure 5.38. The row “interval” contains the maximal lcp-interval that matches $S^2_{p_2}$.

Definition 5.5.11 We say that an lcp-interval ℓ -[$i..j$] in the ESA of S^1 matches (a prefix of) $S^2_{p_2}$ if $S^1[SA[i]..SA[i] + \ell - 1] = S^2[p_2..p_2 + \ell - 1]$. A maximal lcp-interval that matches (a prefix of) $S^2_{p_2}$ is a (non-singleton) lcp-interval ℓ -[$i..j$] that matches (a prefix of) $S^2_{p_2}$, but none of its non-singleton child intervals does.

Below we explain how the algorithm described above works on enhanced suffix arrays. In our running example, we match the string $S^2 = abcdebc\$$ against the ESA of the string $S^1 = abcdefabcdefdebc\#$; see Figures 5.38 and 5.39.

As already mentioned, the naive way to compute matching statistics is to match $S^2_{p_2}$ character by character against the ESA of S^1 until a mismatch occurs, and we use the naive method for position $p_2 = 1$. This is done by calling procedure *match_char_by_char* from Algorithm 5.27 with the parameters $([1..n_1 + 1], 1, 0)$. This procedure call determines the $S^2[1]$ -interval $[lb..rb]$ by means of the procedure call *getInterval*([$1..n_1 + 1$], $S^2[1]$); see Algorithm 5.1 on page 118. If this interval exists (i.e., if $[lb..rb] \neq \perp$), then its lcp-value ℓ is computed, and $S^2[1..\ell - 1]$ is matched character by character against $S^1[SA[lb]..SA[lb] + \ell - 1]$. If a mismatch occurs in this comparison, the procedure *match_char_by_char* returns $\langle (k, [lb..rb]), [1..n_1 + 1] \rangle$, where $k = |\text{lcp}(S^1[SA[lb]..SA[lb] + \ell - 1], S^2[1..\ell - 1])| < \ell$. With regard to the lcp-interval tree, this means that the first mismatch occurred on the edge from the root node 0 -[$1..n_1 + 1$] to the internal node ℓ -[$lb..rb$], and that k characters of the edge label matched. If no mismatch occurs in this comparison, then the interval $[i..j]$ is used to store this last matching lcp-interval $[lb..rb]$, the procedure call *getInterval*([$i..j$], $S^2[\ell]$) determines the $S^2[1..\ell]$ -interval in the ESA of S^1 , and the while-loop is repeated. (In our example, this would happen if we would match—by the procedure call *match_char_by_char*([$1..n_1 + 1$], 2, 0)—the second suffix of S^2 against the ESA of S^1 .) It may happen that during this process a singleton interval is found, and Algorithm 5.27 deals with this case. (In our example, this would happen if we would match the fourth suffix of S^2 against the ESA of S^1 .) In either case, the return value $\langle (k, [lb..rb]), [i..j] \rangle$ of the procedure call *match_char_by_char*([$1..n_1 + 1$], 1, 0) yields the matching statistic $ms[1] = (k, [lb..rb])$ and the maximal lcp-interval $[i..j]$ that matches S^2 .

Algorithm 5.27 Procedure *match_char_by_char*([$i..j$], pos, k) takes an lcp-interval [$i..j$], a position pos in S^2 , and a natural number k as input, where [$i..j$] is the $S^2[pos..pos + k - 1]$ -interval. It returns a tuple $\langle(k', [lb..rb]), [i'..j']\rangle$ so that $S^2[pos..pos + k' - 1]$ is a substring of S^1 , but $S^2[pos..pos + k']$ is not a substring of S^1 , [$lb..rb$] is the $S^2[pos..pos + k' - 1]$ -interval, and [$i'..j'$] is the maximal lcp-interval that matches S^2_{pos} .

```

[ $lb..rb$ ]  $\leftarrow$  getInterval([ $i..j$ ],  $S^2[pos + k]$ )
while [ $lb..rb$ ]  $\neq \perp$ 
  if  $lb \neq rb$  then
     $\ell \leftarrow \text{LCP}[\text{RMQ}_{\text{LCP}}(lb + 1, rb)]$ 
    while  $k < \ell$  and  $S^1[\text{SA}[lb] + k] = S^2[pos + k]$  do
       $k \leftarrow k + 1$ 
    if  $k < \ell$  then
      return  $\langle(k, [lb..rb]), [i..j]\rangle$ 
    [ $i..j$ ]  $\leftarrow$  [ $lb..rb$ ]
    [ $lb..rb$ ]  $\leftarrow$  getInterval([ $i..j$ ],  $S^2[pos + k]$ )
  else /* singleton interval found */
    while ( $S^1[\text{SA}[lb] + k] = S^2[pos + k]$ ) do
       $k \leftarrow k + 1$ 
    return  $\langle(k, [lb..rb]), [i..j]\rangle$ 
return  $\langle(\text{LCP}[\text{RMQ}_{\text{LCP}}(i + 1, j)], [i..j]), [i..j]\rangle$ 

```

Algorithm 5.28 contains pseudo-code for the computation of the matching statistics. It maintains the following invariants: before the for-loop is executed for a position $p_2 \geq 2$, $(k, [lb..rb])$ is the matching statistic for position $p_2 - 1$ and [$i..j$] is the maximal lcp-interval that matches $S^2_{p_2-1}$. If $k = 0$, we use *match_char_by_char*([$1..n_1 + 1$], $p_2, 0$) to match $S^2_{p_2-1}$ against the ESA of S^1 . Otherwise $k > 0$. Because the k -length prefix of $S^2_{p_2-1}$, say $a\omega = S^2[p_2 - 1..p_2 + k - 2]$, matches a substring of S^1 , the $(k - 1)$ -length prefix $\omega = [p_2..p_2 + k - 2]$ of $S^2_{p_2}$ also matches a substring of S^1 . The maximal lcp-interval [$i..j$] that matches $S^2_{p_2-1}$ represents some string au , where u is a prefix of ω . In turn, the u -interval is the suffix link interval [$p..q$] of [$i..j$], and Algorithm 5.28 determines [$p..q$] as well as its lcp-value ℓ . Henceforth, the interval [$i..j$] stores the (currently best) lcp-interval that matches $S^2_{p_2}$. Since the lcp-interval ℓ -[$p..q$] matches $S^2_{p_2}$, Algorithm 5.28 sets [$i..j$] to [$p..q$].

Now, starting from the suffix link interval ℓ -[$p..q$], which is the u -interval (where $u = S^2[p_2..p_2 + \ell - 1]$), the algorithm searches for the ω -interval (where $\omega = S^2[p_2..p_2 + k - 2]$), using the skip and count technique. In the first while-loop, if $\ell < k - 1$ and $p \neq q$, the procedure call *getInterval*([$p..q$], $S^2[p_2 + \ell]$) identifies the child interval of the current lcp-interval [$p..q$] that starts with the characters $S^2[p_2..p_2 + \ell]$. This child interval is the new current lcp-interval, hence it is stored in [$p..q$]. If $p \neq q$, i.e., [$p..q$] is a non-singleton

Algorithm 5.28 Computing matching statistics.

```

 $\langle (k, [lb..rb]), [i..j] \rangle \leftarrow match\_char\_by\_char([1..n_1 + 1], 1, 0)$ 
 $ms[1] \leftarrow (k, [lb..rb])$ 
for  $p_2 \leftarrow 2$  to  $n_2$  do
    /* Invariants:  $ms[p_2 - 1] = (k, [lb..rb])$  and */
    /*  $[i..j]$  is the maximal lcp-interval that matches  $S_{p_2-1}^2$  */
    if  $k = 0$  then
         $\langle (k, [lb..rb]), [i..j] \rangle \leftarrow match\_char\_by\_char([1..n_1 + 1], p_2, 0)$ 
         $ms[p_2] \leftarrow (k, [lb..rb])$ 
    else
         $[p..q] \leftarrow slink[RMQ_{LCP}(i + 1, j)]$  /* suffix link interval of  $[i..j]$  */
         $\ell \leftarrow LCP[RMQ_{LCP}(p + 1, q)]$  /* lcp-value of  $[p..q]$  */
         $[i..j] \leftarrow [p..q]$ 
        while  $\ell < k - 1$  and  $p \neq q$  do /* skip and count */
             $[p..q] \leftarrow getInterval([p..q], S^2[p_2 + \ell])$ 
            if  $p \neq q$  then /* lcp-interval found */
                 $\ell \leftarrow LCP[RMQ_{LCP}(p + 1, q)]$ 
                if  $\ell \leq k - 1$  then
                     $[i..j] \leftarrow [p..q]$ 
            if  $\ell = k - 1$  then
                 $\langle (k, [lb..rb]), [i..j] \rangle \leftarrow match\_char\_by\_char([i..j], p_2, k - 1)$ 
                 $ms[p_2] \leftarrow (k, [lb..rb])$ 
            else
                 $k \leftarrow k - 1$ 
                 $ms[p_2] \leftarrow (k, [p..q])$ 

```

interval, the algorithm determines the lcp-value of $[p..q]$ and stores it in the variable ℓ . If $\ell \leq k - 1$, then $[p..q]$ is the (currently best) lcp-interval that matches $S_{p_2}^2$, and thus it is stored in $[i..j]$. The first while-loop is left either because $\ell < k - 1$ or $p = q$ (both conditions cannot simultaneously be true). We consider the following three mutually exclusive cases:

1. If $\ell = k - 1$, then we match $S_{p_2+k-1}^2$ character by character against the ESA of S^1 until a mismatch occurs. (In the suffix tree, this corresponds to the case in which the matching path ends at an internal node.)
2. If $\ell > k - 1$, then every suffix in the interval $[p..q]$ has the same character b at position $k - 1$, and b is different from $S^2[p_2+k-1]$. Therefore, the matching path cannot be continued with the character $S^2[p_2+k-1]$, and $ms[p_2]$ can be set to $(k - 1, [p..q])$. (In the suffix tree, this corresponds to the case in which the matching path ends within the edge to an internal node.)

3. If $p = q$, then the ω -interval we are looking for is the singleton interval $[p..p]$, that is, $\omega = S^2[p_2..p_2 + k - 2] = S^1[SA[p]..SA[p] + k - 2]$. As in the previous case, we have $S^2[p_2 + k - 1] \neq S^1[SA[p] + k - 1]$ and thus $ms[p_2]$ can be set to $(k - 1, [p..q])$. (In the suffix tree, this corresponds to the case in which the matching path ends within the edge to a leaf.)

We shall illustrate Algorithm 5.28 for $p_2 = 2$. As mentioned above, matching $S^2 = abcdebc\$$ character by character against the enhanced suffix array of the string $S^1 = abcdefabcdefdebc\#$ yields $ms[1] = (5, [2..3])$ and $[1..17]$ as the maximal lcp-interval that matches S^2_1 . Clearly, the suffix link interval of $[1..17]$ is the root interval $0-[1..17]$ itself. Then the skip and count phase starts. The child interval of $[1..17]$ that starts with the character $S^2[2] = b$ is the b -interval $[4..6]$; cf. Figure 5.38. Because the lcp-value $\ell = 2$ of the lcp-interval $[4..6]$ is less than $k - 1 = 4$, we know that every suffix in the interval $[4..6]$ starts with $S^2[p_2..p_2 + \ell - 1] = S^2[2..3] = bc$. Furthermore, the interval $[i..j] = [4..6]$ is the (currently best) lcp-interval that matches S^2_2 , and the while-loop is executed again. The child interval of the bc -interval $[4..6]$ that continues with the character $S^2[4] = d$ is the bcd -interval $[5..6]$; cf. Figure 5.38. Because its lcp-value $\ell = 5$ is greater than $k - 1 = 4$, the while-loop is left without updating $[i..j]$, and according to case (2) we set $ms[2] = (4, [5..6])$. Similarly, we obtain $ms[3] = (3, [8..9])$ and $[7..9]$ as the maximal lcp-interval that matches S^2_3 . Since the suffix link interval of $[7..9]$ is the root interval $0-[1..17]$, the search for $ms[4]$ starts at the node $0-[1..17]$. In the skip and count phase, the algorithm first finds the d -interval $[10..12]$, having lcp-value $\ell = 2$. Because $\ell = k - 1 = 3 - 1$, the algorithm continues as demanded by case (1): it matches S^2_6 character by character against the ESA of S^1 until a mismatch occurs.

The time analysis of the algorithm is based on the depth of a node $\ell-[i..j]$ in the lcp-interval tree of S^1 , which is the length of the path from the root node $0-[1..n_1 + 1]$ to node $\ell-[i..j]$.

Lemma 5.5.12 *Let $\ell-[i..j] \neq 0-[1..n_1 + 1]$ be an lcp-interval and let $(\ell - 1)-[p..q]$ be its suffix link interval. Then the depth of $\ell-[i..j]$ in the lcp-interval tree of S^1 is at most one greater than the depth of $(\ell - 1)-[p..q]$.*

Proof Suppose the nodes on the path from the root node $0-[1..n_1 + 1]$ to node $\ell-[i..j]$ represent the strings $\varepsilon, a\omega_1, a\omega_1\omega_2, \dots, a\omega_1\omega_2 \dots \omega_m$, i.e., the depth of node $\ell-[i..j]$ is m . Note that ω_1 may be the empty string. By Lemma 5.5.3, the intervals representing the strings $\omega_1, \omega_1\omega_2, \dots, \omega_1\omega_2 \dots \omega_m$ are all lcp-intervals, and the lcp-interval $(\ell - 1)-[p..q]$ represents the string $\omega_1\omega_2 \dots \omega_m$. Since $\omega_1\omega_2 \dots \omega_r$ is a prefix of $\omega_1\omega_2 \dots \omega_m$ for every r with $1 \leq r \leq m$, it follows that every $\omega_1\omega_2 \dots \omega_r$ -interval is a node on the path from the root to $(\ell - 1)-[p..q]$. If $\omega_1 = \varepsilon$, then the depth of node $(\ell - 1)-[p..q]$ is at least $m - 1$, otherwise it is at least m . \square

Let us resume the analysis of the worst-case time complexity of the algorithm. The algorithm starts with $p_2 = 1$ by matching S^2 character by character against the enhanced suffix array of S^1 . There are $k + 1$ comparisons if $S^2[1..k]$ matches, but $S^2[1..k + 1]$ does not. Then it determines the suffix link interval of the maximal lcp-interval that matches S^2 , uses the skip and count technique to find the descendant of the suffix link interval corresponding to $S^2[2..k]$, and matches S^2_2 character by character against the ESA of S^1 starting with $S^2[k + 1]$, and so on. In general, it determines the suffix link interval of the current maximal lcp-interval (this takes only constant time), uses the skip and count technique to find the corresponding descendant of the suffix link interval (this takes time proportional to the number of nodes on the path from the suffix link interval to the descendant), and then proceeds with the character-by-character matching (this takes time proportional to the number of character comparisons). Over the entire algorithm, n_2 suffix link intervals have to be determined, taking $O(n_2)$ time. Because the traversal from the current maximal lcp-interval to its suffix link interval decreases the depth by at most one (Lemma 5.5.12), the overall decrements to current depth cannot exceed n_2 . Since the current depth is bounded by n_2 , the overall increments to current depth cannot exceed $2n_2$. Thus, the skip and count phases require $O(n_2)$ time in total. We still have to analyze the overall number of character comparisons in the character-by-character matching phases. Once a character of S^2 matched a character of S^1 , it will not be considered again. Consequently, the number of character matches is bounded by n_2 . Because every character-by-character matching phase ends with a mismatch, there are $n_2 + 1$ mismatches over the entire algorithm. Thus, the character-by-character matching phases also require only $O(n_2)$ time in total.

Exercise 5.5.13 Suppose that the suffix array of the string S^1 is given and its LCP-array is preprocessed so that range minimum queries can be answered in constant time. Furthermore, let the inverse suffix array and the matching statistics of the string S^2 w.r.t. S^1 be known. Show that $|\text{lcp}(S^1_p, S^2_q)|$ can be computed in constant time for all $1 \leq p \leq n_1 + 1$ and $1 \leq q \leq n_2 + 1$.

Exercise 5.5.14 The *bidirectional matching statistics* of S^2 w.r.t. S^1 is an array $bms[1..n_2 + 1]$ so that for each entry $bms[i] = (k, p)$ the string $S^2[p..p + k - 1]$ is a longest substring of S^2 containing position i (i.e., $p \leq i \leq p + k - 1$) that matches a substring somewhere in S^1 . Describe an $O(n_1 + n_2)$ time algorithm that computes the bidirectional matching statistics.

Exercise 5.5.15 Let the generalized suffix array of the two strings S^1 and S^2 in form of the arrays $D[i]$ and SA' be given. The *mutual matching statistics* of S^1 and S^2 is an array mms so that an entry $mms[i] = k$ implies:

1. If $D[i] = 1$, then $S^1[SA'[i]..SA'[i] + k - 1]$ is the longest prefix of $S^1_{SA'[i]}$ that occurs as a substring of S^2 .
2. If $D[i] = 2$, then $S^2[SA'[i]..SA'[i] + k - 1]$ is the longest prefix of $S^2_{SA'[i]}$ that occurs as a substring of S^1 .

Show that the array *mms* can be computed in $O(n_1 + n_2)$ time.

Hint: It may be instructive to have a second look at the reasoning used in Section 5.2.1.

Exercise 5.5.16 Suppose that S^1 and S^2 are two strings with $|S^1| \leq |S^2|$. Use the technique of matching S^2 against the enhanced suffix array of S^1 to compute a longest common substring of S^1 and S^2 . (This solution to the longest common substring problem is more space efficient than the previous one because it just needs space for the suffix array of the smaller of the two strings.)

Exercise 5.5.17 Use the technique of matching S^2 against the enhanced suffix array of S^1 to compute all maximal unique matches (MUMs) between S^1 and S^2 .

Exercise 5.5.18 Given a threshold t , a maximal exact match is *rare* if it occurs at most t times in each of the strings S^1 and S^2 . (For $t = 1$, rare MEMs coincide with MUMs.) Outline an algorithm that computes rare MEMs by matching S^2 against the ESA of S^1 .

5.5.5 Merging two suffix arrays in linear time

Jeon et al. [165] showed that the generalized suffix array SA of two strings S^1 and S^2 can be obtained by merging the suffix arrays SA_1 and SA_2 of S^1 and S^2 in linear time. As usual, we assume that S^1 is terminated by $\#$, S^2 is terminated by $\$$, $|S^1| = n_1 + 1$, and $|S^2| = n_2 + 1$. The algorithm uses an auxiliary array *cnt* of size $n_1 + 2$. Initially, $cnt[i] = 0$ for all i with $1 \leq i \leq n_1 + 2$. Using suffix links, we match S^2 against SA_1 in $O(n_2)$ time. During the matching phase, $cnt[i]$ is incremented by one if a suffix of S^2 is detected to be lexicographically larger than $S^1_{SA_1[i-1]}$ and smaller than $S^1_{SA_1[i]}$ (if it is larger than the last suffix $S^1_{SA_1[n_1+1]}$ in SA_1 , then $cnt[n_1 + 2]$ is incremented by one). Of course, we have to modify the procedures *getInterval*, *match_char_by_char*, and (a variant of) the computation of the matching statistics for this task; see Exercise 5.5.20. For example, if we match the string $S^2 = acaaacatat\$$ against the suffix array of the string $S^1 = ctaataatg\#$ (shown in Figure 5.40), then we find that the suffix *aatg\#* at index 3 is lexicographically smaller than *acaacatat\\$*, whereas the suffix *ataatg\#* at index 4 is lexicographically larger than *acaacatat\\$*. Consequently, $cnt[4]$ is incremented by 1 for S^2_1 . Subsequently, $cnt[6]$ is incremented by 1 for

i	SA	LCP	$S_{SA[i]}$	cnt
1	10	-1	#	0
2	3	0	aataatg#	3
3	6	3	aatg#	0
4	4	1	ataatg#	3
5	7	2	atg#	1
6	1	0	ctaataatg#	2
7	9	0	g#	0
8	2	0	taataatg#	1
9	5	4	taatg#	0
10	8	1	tg#	1
11		-1		0

Figure 5.40: ESA of the string $S^1 = ctaataatg\#$ with cnt -array.

S_2^2 , and so on. The final state of the cnt -array is depicted in Figure 5.40. The generalized suffix array SA of S^1 and S^2 is then built incrementally as follows: Since there are $cnt[1]$ suffixes in SA_2 that are lexicographically smaller than $S_{SA_1[1]}^1$, the suffixes in the interval $[1..cnt[1]]$ of SA_2 are stored as the first suffixes in SA. After this, the suffix $S_{SA_1[1]}^1$ is placed at index $cnt[1] + 1$. Similarly, because there are $cnt[2]$ suffixes in SA_2 that are lexicographically larger than $S_{SA_1[1]}^1$ and smaller than $S_{SA_1[2]}^1$, the suffixes in the interval $[cnt[1] + 1..cnt[1] + cnt[2]]$ of SA_2 are stored at the interval $[cnt[1] + 2..cnt[1] + cnt[2] + 1]$ in SA, and so on. In general, if $PS[i]$ denotes the prefix sum $\sum_{j=1}^i cnt[j]$, then the suffix $S_{SA_1[i]}^1$ is placed at index $PS[i] + i$ in SA and the suffixes in the interval $[PS[i] + 1..PS[i + 1]]$ of SA_2 are stored at the interval $[PS[i] + i + 1..PS[i + 1] + i]$ in SA (where $PS[0] = 0$). It is rather obvious that the worst-case time complexity of the merging algorithm is $O(n_1 + n_2)$ time.

As a matter of fact, the combined LCP-array of the arrays LCP_1 and LCP_2 of S^1 and S^2 , respectively, can also be computed in linear time with the merging algorithm. The next lemma from [190] is the key.

Lemma 5.5.19 Suppose suffix S_j^2 is placed between the suffixes $S_{SA_1[i-1]}^1$ and $S_{SA_1[i]}^1$ in the merging algorithm, and let $ms[j] = (\ell, [p..q])$.

1. If $p \leq i \leq q$, then

- a) $|lcp(S_j^2, S_{SA_1[i]}^1)| = \ell$
- b) $|lcp(S_{SA_1[i-1]}^1, S_j^2)| = LCP_1[i]$

2. Otherwise $i - 1 = q$ and

$$a) |\text{lcp}(S_{SA_1[i-1]}^1, S_j^2)| = \ell$$

$$b) |\text{lcp}(S_j^2, S_{SA_1[i]}^1)| = \text{LCP}_1[i]$$

Proof The longest common prefix ω of $S_{SA_1[i-1]}^1$ and $S_{SA_1[i]}^1$ has length $\text{LCP}_1[i]$. Because suffix S_j^2 is placed between them, ω must also be a prefix of S_j^2 . Let ωa , ωb , and ωc be the length $(\text{LCP}_1[i] + 1)$ prefixes of $S_{SA_1[i-1]}^1$, S_j^2 , and $S_{SA_1[i]}^1$, respectively:

$$\begin{array}{c|c} i-1 & \omega a \dots = S_{SA_1[i-1]}^1 \\ & \omega b \dots = S_j^2 \\ i & \omega c \dots = S_{SA_1[i]}^1 \end{array}$$

Clearly, $a \neq c$ and there are three mutually exclusive cases.

- (i) If $b = a$, then $i - 1 = q$ and $|\text{lcp}(S_{SA_1[i-1]}^1, S_j^2)| = \ell$. Since $b \neq c$, we have $|\text{lcp}(S_j^2, S_{SA_1[i]}^1)| = |\omega| = \text{LCP}_1[i]$.
- (ii) If $b = c$, then $i = p$ and $|\text{lcp}(S_j^2, S_{SA_1[i]}^1)| = \ell$. Because $a \neq b$, it follows that $|\text{lcp}(S_{SA_1[i-1]}^1, S_j^2)| = |\omega| = \text{LCP}_1[i]$.
- (iii) If $a < b < c$, then $p \leq i - 1 < i \leq q$ and $\ell = |\omega| = \text{LCP}_1[i]$. Hence $|\text{lcp}(S_j^2, S_{SA_1[i]}^1)| = \ell$ and $|\text{lcp}(S_{SA_1[i-1]}^1, S_j^2)| = \text{LCP}_1[i]$. \square

As in the merging algorithm, the auxiliary array *cnt* is computed by matching S^2 against SA_1 . This time, however, the matching statistics *ms* are calculated as well. In the merging phase, the LCP-array can be computed with the help of *cnt* and *ms* as follows: Suppose that a suffix $S_{SA_2[j]}^2$ must be placed in between the suffixes $S_{SA_1[i-1]}^1$ and $S_{SA_1[i]}^1$, say at index k , and let $ms[j] = (\ell, [p..q])$. Recall that there are $cnt[i]$ many suffixes of S^2 that must be placed in between the suffixes $S_{SA_1[i-1]}^1$ and $S_{SA_1[i]}^1$.

- If $S_{SA_2[j]}^2$ is the first of these, then by Lemma 5.5.19

$$\text{LCP}[k] = \begin{cases} \text{LCP}_1[i] & \text{if } p \leq i \leq q \\ \ell & \text{otherwise} \end{cases}$$

- If $S_{SA_2[j]}^2$ is not the first of these, then $\text{LCP}[k] = \text{LCP}_2[j]$.
- If $S_{SA_2[j]}^2$ is the last one, then $SA[k+1] = SA_1[i]$ and by Lemma 5.5.19

$$\text{LCP}[k+1] = \begin{cases} \ell & \text{if } p \leq i \leq q \\ \text{LCP}_1[i] & \text{otherwise} \end{cases}$$

Exercise 5.5.20 Modify the procedures *getInterval* (Algorithm 5.1 on page 118), *match_char_by_char* (Algorithm 5.27 on page 199), and the computation of the matching statistics (Algorithm 5.28 on page 200) so that they compute the *cnt*-array.

5.6 Comparing multiple strings

In this section, we will compare more than two strings. Again, generalized enhanced suffix arrays play the key role.

5.6.1 Generalized suffix array

Let S^1, S^2, \dots, S^m be strings of sizes n_1, n_2, \dots, n_m , respectively. We are interested in the lexicographic order of all suffixes

$$S_1^1, \dots, S_{n_1}^1, S_1^2, \dots, S_{n_2}^2, \dots, S_1^m, \dots, S_{n_m}^m$$

of these strings. Note that two suffixes S_p^j and S_q^k with $j \neq k$ may coincide, i.e., $S_p^j = S_q^k$ is possible. (In this case, it is natural to demand that the suffix with the smaller superscript shall appear before the suffix with the larger superscript.) Because the strings may share identical suffixes, we use m pairwise distinct characters $\#_1, \#_2, \dots, \#_m$ to tell the suffixes apart. To be precise, for each j with $1 \leq j \leq m$, we obtain the string $S^j \#_j$ of length $n_j + 1$ by appending the special character $\#_j$ to S^j . This ensures that each suffix can uniquely be assigned to one of the m strings: if the suffix ends with $\#_j$, then it belongs to S^j . If we assume that $\#_1 < \#_2 < \dots < \#_m$ and that all other characters in the alphabet Σ are larger than these symbols, then the suffixes of the strings $S^1 \#_1, S^2 \#_2, \dots, S^m \#_m$ are not only pairwise distinct, but we also have $S_p^j \#_j < S_q^k \#_k$ if and only if either $S_p^j < S_q^k$ or $S_p^j = S_q^k$ and $j < k$.

The *common suffix array* of the strings $S^1 \#_1, S^2 \#_2, \dots, S^m \#_m$ consists of *two* arrays, the *document array* D and the array SA' , having the following properties:

- For every suffix $S_k^j \#_j$, there is an index i so that $j = D[i]$ and $k = SA'[i]$.
- $S_{SA'[i]}^{D[i]} \#_{D[i]} < S_{SA'[i+1]}^{D[i+1]} \#_{D[i+1]}$ for all i with $1 \leq i \leq m - 1 + \sum_{j=1}^m n_j$.

In other words, the arrays D and SA' specify the lexicographic order of all the suffixes of the m strings. An example can be found in (the two rightmost columns of) Figure 5.41.

The common suffix array of multiple strings can be obtained from their generalized suffix array, defined as follows.

Definition 5.6.1 Given m strings S^1, S^2, \dots, S^m , the suffix array SA of the string $S = S^1 \#_1 S^2 \#_2 \dots S^m \#_m$ is called the *generalized suffix array* of S^1, S^2, \dots, S^m . If the generalized suffix array of the strings is enhanced with further arrays, e.g. with the LCP-array, then it will be called *generalized enhanced suffix array* or GESA for short.

i	SA	ISA	LCP	$S_{SA[i]}$	$S_{SA[i]}^\#$	D	SA'
1	5	9	-1	$\#_1aac\#_2caac\#_3$	$\#_1$	1	5
2	9	14	0	$\#_2caac\#_3$	$\#_2$	2	4
3	14	6	0	$\#_3$	$\#_3$	3	5
4	6	10	0	$aac\#_2caac\#_3$	$aac\#_2$	2	1
5	11	1	3	$aac\#_3$	$aac\#_3$	3	2
6	3	4	1	$ac\#_1aac\#_2caac\#_3$	$ac\#_1$	1	3
7	7	7	2	$ac\#_2caac\#_3$	$ac\#_2$	2	2
8	12	11	2	$ac\#_3$	$ac\#_3$	3	3
9	1	2	2	$acac\#_1aac\#_2caac\#_3$	$acac\#_1$	1	1
10	4	13	0	$c\#_1aac\#_2caac\#_3$	$c\#_1$	1	4
11	8	5	1	$c\#_2caac\#_3$	$c\#_2$	2	3
12	13	8	1	$c\#_3$	$c\#_3$	3	4
13	10	12	1	$caac\#_3$	$caac\#_3$	3	1
14	2	3	2	$cac\#_1aac\#_2caac\#_3$	$cac\#_1$	1	2
15			-1				

Figure 5.41: The enhanced suffix array of $S = acac\#_1aac\#_2caac\#_3$ coincides with the common enhanced suffix array of the strings $acac\#_1$, $aac\#_2$, and $caac\#_3$ (consisting of the two arrays D and SA').

Figure Figure 5.41 shows an example. Note that the string S has length $m + \sum_{j=1}^m n_j$.

Given the inverse ISA of the generalized suffix array SA of S^1, S^2, \dots, S^m , Algorithm 5.29 constructs the common suffix array of these strings in linear time. Conversely, given the arrays D and SA' , the suffix array SA can easily be computed in linear time; see Exercise 5.6.2. Therefore, the common suffix array and the generalized suffix array are two representations of the same information. In the following, we identify them and call both the generalized suffix array. In applications, we will always use the representation that best suits our purpose. Given $S = S^1\#_1S^2\#_2\dots S^m\#_m$ and its suffix array SA, $S_{SA[i]}^\#$ henceforth denotes the prefix of $S_{SA[i]}$ that ends with the first separator symbol. In other words, $S_{SA[i]}^\# = S_{SA'[i]}^{D[i]}\#_{D[i]}$; see Figure 5.41. In fact, we will often identify $S_{SA[i]}$ with $S_{SA[i]}^\#$. That is, when we write $S_{SA[i]}$, we often actually mean $S_{SA[i]}^\#$.

Exercise 5.6.2 Let the common suffix array of $S^1\#_1, S^2\#_2, \dots, S^m\#_m$ in form of the two arrays D and SA' be given. Give a linear-time algorithm that computes the generalized suffix array SA of the strings S^1, S^2, \dots, S^m .

Algorithm 5.29 Computation of the arrays SA' and D from ISA .

```

offset  $\leftarrow$  0
for  $j \leftarrow 1$  to  $m$  do
  for  $k \leftarrow$  offset + 1 to offset +  $n_j$  + 1 do
     $D[ISA[k]] \leftarrow j$ 
     $SA'[ISA[k]] \leftarrow k - \text{offset}$ 
  offset  $\leftarrow$  offset +  $n_j$  + 1

```

Exercise 5.6.3 What are the time complexities of the following two algorithms? (Both share the first phase, and merging is done as in Section 5.5.5.)

- (1) For every j , $1 \leq j \leq m$, build the suffix array SA_j of string $S^j \#_j$.
- (2a) Construct the common suffix array of $S^1 \#_1, S^2 \#_2, \dots, S^m \#_m$ by successively merging the suffix arrays SA_1, SA_2, \dots, SA_m . That is, merge SA_1 with SA_2 , yielding the common suffix array SA_{12} of $S^1 \#_1$ and $S^2 \#_2$, merge SA_{12} with SA_3 , and so on.
- (2b) Merge SA_1 with SA_2 , yielding the common suffix array SA_{12} of $S^1 \#_1$ and $S^2 \#_2$, merge SA_3 with SA_4 , yielding the common suffix array SA_{34} of $S^3 \#_3$ and $S^4 \#_4$ etc., then merge SA_{12} with SA_{34} , yielding the common suffix array of $S^1 \#_1, S^2 \#_2, S^3 \#_3$, and $S^4 \#_4$ etc., until the common suffix array of $S^1 \#_1, S^2 \#_2, \dots, S^m \#_m$ is obtained.

5.6.2 Longest common substring

In Section 5.4.2, we have solved the problem of finding a longest common substring of two strings. It is left as an exercise for the reader to develop a linear-time algorithm that finds a longest substring common to multiple strings S^1, \dots, S^m . A non-obvious algorithm can be found in Exercise 5.6.9. Here, we focus on the following generalization of the longest common substring problem.

Definition 5.6.4 Given m strings S^1, \dots, S^m , the k -common substring problem is to simultaneously find, for all k with $2 \leq k \leq m$, a longest substring common to at least k of the strings.

In the following, let ℓ_k denote the length of the longest substrings that are common to at least k of the strings. Furthermore, let SA and LCP be the suffix array and the lcp-array of the length n string $S = S^1 \#_1 S^2 \#_2 \dots S^m \#_m$.

Theorem 5.6.5 The string ω is a longest substring common to at least k of the strings S^1, \dots, S^m if and only if there exist indices p and q with $1 \leq p < q \leq n$ so that

1. $|\{D[p], D[p+1], \dots, D[q]\}| \geq k$,
2. ω is common prefix of $S_{SA[p]}, S_{SA[p+1]}, \dots, S_{SA[q]}$ and $|\text{lcp}(S_{SA[p]}, S_{SA[q]})| = |\omega|$,
3. $\text{LCP}[p] < |\omega|$ and $\text{LCP}[q+1] < |\omega|$ hold true.
4. $|\omega| = \ell_k$.

Proof “if”: (1) and (2) imply that ω is a substring common to at least k strings and (4) implies that ω is a longest string with this property.

“only if”: If ω is a longest substring common to at least k strings, then $|\omega| = \ell_k$ and there are indices p' and q' with $1 \leq p' < q' \leq n$ so that $|\{D[p'], D[p'+1], \dots, D[q']\}| \geq k$, ω is a common prefix of $S_{SA[p']}, S_{SA[p'+1]}, \dots, S_{SA[q']}$, and $|\text{lcp}(S_{SA[p]}, S_{SA[q]})| = |\omega|$. Let p be the largest index with $p \leq p'$ and $\text{LCP}[p] < |\omega|$, and let q be the smallest index with $q' \leq q$ and $\text{LCP}[q+1] < |\omega|$. These indices p and q satisfy (1)–(4). \square

Of course, we do not know ℓ_k . That is why we successively compute all strings having properties (1)–(3) and keep track of the currently longest string with these properties.

Exercise 5.6.6 Use the notion of lcp-intervals to reformulate Theorem 5.6.5. Give an algorithm that solves the k -common substring problem by a bottom-up traversal of the lcp-interval tree. Analyze its worst-case time complexity.

A naive solution

We use an array A of size $m - 1$ that stores for each k , $2 \leq k \leq m$, a pair (lcs, idx) , where lcs is the length of a (currently) longest substring ω common to at least k strings and idx is an index so that $S[SA[idx]..SA[idx] + lcs - 1] = \omega$. In what follows, we denote the first and second component of an array element $A[k]$ by $A[k].lcs$ and $A[k].idx$, respectively. Initially, $A[k] = (0, \perp)$ for all k with $2 \leq k \leq m$.

Moreover, we employ a doubly linked list consisting of exactly m elements. For each string S^j , $1 \leq j \leq m$, there is exactly one element in the list and a pointer $\text{strptr}[j]$ to that element. Furthermore, there is a pointer LV to the last element in the list. (The name LV is an acronym for *last visited* because—as we shall see later— LV points to the element that corresponds to the last visited index in the suffix array.) Every element e in the list is a pair (lcp, idx) , where lcp is an lcp-value and idx is a position in the suffix array. We denote the first and second component of a list element e by $e.lcp$ and $e.idx$, respectively. Initially, the list has the form $[(0, 1), (0, 2), \dots, (0, m - 1), (|S_{SA[m]}|, m)]$ and for all j , $1 \leq j \leq m$, $\text{strptr}[j]$ points

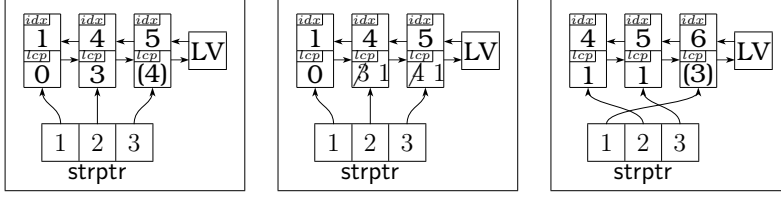


Figure 5.42: Left: List after $i = 5$ has been processed. The last suffixes of the strings 1, 2, and 3 are $S_{SA[1]} = \#_1$, $S_{SA[4]} = aac\#_2$, and $S_{SA[5]} = aac\#_3$. The longest common prefixes of these suffixes with $S_{SA[i]}$ have length 0, 3, and 4. Furthermore, $A[2] = A[3] = (0, \perp)$. Middle: In step $i = 6$, the function *lcp_update* is called with *lcp*-value $LCP[6] = 1$ and the *lcp*-values of list elements are being updated. Moreover, $A[2]$ is set to $(3, 4)$. Right: List after the procedure call *list_update*(6).

to the element with second component j . This is because the m lexicographically smallest suffixes are $\#_1, \#_2, \dots, \#_m$; cf. Figure 5.41. Note that $|S_{SA[m]}| = |\#_m| = 1$.

Our algorithm linearly scans the generalized enhanced suffix array starting at index $m + 1$, and for each i with $m + 1 \leq i \leq n$, it first calls the procedure *lcp_update* with parameter $LCP[i]$ and then the procedure *list_update* with parameter i .

- *lcp_update*($LCP[i]$) linearly scans the list from right to left (the rightmost element can be found with the *LV* pointer) and compares the value $e.lcp$ of the current element e with $LCP[i]$. Suppose that e is the k -th element from right to left. If $e.lcp > LCP[i]$, then we compare the (currently best) value $A[k].lcs$ with $e.lcp$. In case $A[k].lcs \leq e.lcp$, we update $A[k]$ by the assignment $A[k] \leftarrow e$. Furthermore, in the case $e.lcp > LCP[i]$, the value $e.lcp$ is updated by the assignment $e.lcp \leftarrow LCP[i]$, and the next list element is considered. Otherwise (i.e., $e.lcp \leq LCP[i]$) the procedure stops the scan of the list.
- *list_update*(i) updates the list by standard list operations as follows: It deletes the element to which $strptr[D[i]]$ points from the list and adds a new element $(|S_{SA[i]}|, i)$ at the end of the list. The pointers *LV* and $strptr[D[i]]$ are updated so that they both point to the added element.

Figures 5.42 and 5.43 illustrate the algorithm for the example from Figure 5.41.

Each execution of the procedure *lcp_update* takes $O(m)$ time, while each execution of the procedure *list_update* takes constant time. Because these

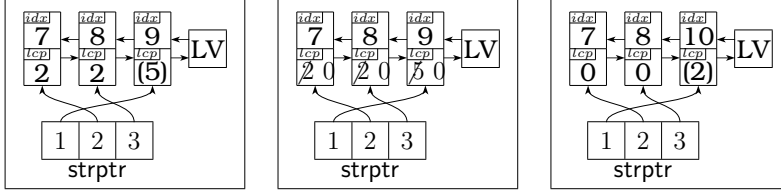


Figure 5.43: Left: List after $i = 9$ has been processed. Middle: In iteration $i = 10$, the function `lcp_update` is called with `lcp`-value $LCP[10] = 0$ and $A[3]$ is set to $(2, 7)$. Right: List after the procedure call `list_update(10)`.

procedures are called $O(n)$ times, the worst-case time complexity of the naive algorithm is in $O(m \cdot n)$.

We show that for each i with $m \leq i \leq n$ the algorithm maintains the following invariants:

1. The idx -values of the list elements are in strict ascending order (left-to-right).
2. If `strptr[j]` points to the element e , then $D[e.idx] = j$ and $D[p] \neq j$ for all p with $idx < p \leq i$. In words, the element e to which `strptr[j]` points corresponds to the last suffix seen so far that belongs to string S^j .
3. The lcp -values of the list elements are in ascending order (from left to right).
4. Each element e in the list satisfies $e.lcp = |\text{lcp}(S_{SA[e.idx]}, S_{SA[i]})|$.

It is straightforward to verify that our algorithm maintains the invariants (1)-(3). We prove by induction that the crucial property (4) is an invariant and make use of the following fact: for all indices p and q with $1 \leq p < q \leq n$ the equality $|\text{lcp}(S_{SA[p]}, S_{SA[q]})| = \min_{p < l \leq q} \{LCP[l]\}$ holds true. For $i = m$ property (4) holds. The induction hypothesis states that after index $i - 1$ (where $i - 1 \geq m$) has been considered, every element e in the list satisfies $e.lcp = |\text{lcp}(S_{SA[e.idx]}, S_{SA[i-1]})| = \min_{e.idx < l \leq i-1} \{LCP[l]\}$. After the procedure call `lcp_update(LCP[i])`, we have

$$e.lcp = \min\{|\text{lcp}(S_{SA[e.idx]}, S_{SA[i-1]})|, LCP[i]\} = \min_{e.idx < l \leq i} \{LCP[l]\} = |\text{lcp}(S_{SA[e.idx]}, S_{SA[i]})|$$

Hence property (4) is satisfied. This is also true after the procedure call `list_update(i)` because for the new element $e_{new} = (i, |S_{SA[i]}|)$, we have $e_{new}.lcp = |S_{SA[i]}| = |\text{lcp}(S_{SA[e_{new}.idx]}, S_{SA[i]})|$.

Theorem 5.6.7 *The algorithm solves the k -common substring problem.*

Proof Recall that ℓ_k denotes the length of a longest substring common to at least k strings, where $2 \leq k \leq m$. Furthermore, let q be the largest index for which there is an index p , $1 \leq p < q \leq n$, so that the suffixes $S_{SA[p]}, S_{SA[p+1]}, \dots, S_{SA[q]}$ have a common prefix of length ℓ_k and $|\{D[p], D[p+1], \dots, D[q]\}| \geq k$. Because q is the largest index with this property, it follows that $LCP[q+1] < \ell_k$. Now consider the list before $lcp_update(LCP[q+1])$ is called. By the fourth invariant, we know that each element e in the list satisfies $e.lcp = |\text{lcp}(S_{SA[e.idx]}, S_{SA[q]})|$. Let e' be k -th element in the list (from right to left). As the suffixes $S_{SA[p]}, S_{SA[p+1]}, \dots, S_{SA[q]}$ have a common prefix of length ℓ_k and $|\{D[p], D[p+1], \dots, D[q]\}| \geq k$, the first k elements (from right to left) in the list have an lcp -value $\geq \ell_k$. Moreover, $e'.lcp = \ell_k$ because otherwise there would be a longer substring common to k strings. Consequently, when the procedure $lcp_update(LCP[q+1])$ is called, $A[k]$ is correctly updated by $A[k] \leftarrow e'$. By the choice of index q , $A[k]$ remains unchanged from that point on. \square

A linear-time solution

To obtain a linear-time solution, we combine all elements having the same lcp -value into intervals. This implies that every interval can be identified by its unique lcp -value. Because the lcp -values of the *elements* are in ascending order (from left to right) in the list, it follows that the lcp -values of the *intervals* are also in strict ascending order (from left to right) in the list. To represent intervals, each list element now has the five components $(lcp, idx, begin, end, size)$.

- *begin*: Pointer (from the last element of the interval) to the first element of the interval.
- *end*: Pointer (from the first element of the interval) to the last element of the interval.
- *size*: Number of elements in the interval.

Although each list element has five components $(lcp, idx, begin, end, size)$, they are only relevant for the first and last element of an interval. To be precise, the four components $(lcp, idx, end, size)$ are solely relevant for the first element of an interval and the component *begin* is solely relevant for the last element of an interval.

We can access all intervals by following the *begin* pointers as follows: We start with the *LV* pointer and find the last element of the first interval I_1 (from right to left). Following the *begin* pointer of that last element, we reach the first element e_1 of I_1 . Note that the size information can be used to compute the position k_1 (from right to left) of e_1 in the list, namely $k_1 = e_1.size$. Then we find the element left to it with the help of the usual

links of the doubly linked list. This element is the last element of the second interval I_2 and we can reach the first element e_2 of I_2 by following the begin pointer of the last element of I_2 . The position k_2 (from right to left) of e_2 in the list is $k_2 = k_1 + e_2.size$. In this way, we can proceed until all intervals have been found.

It will also be necessary to access an interval of a certain *lcp*-value in constant time. To this end, we use an array *intptr*[1..*n*] of *interval pointers*. To be precise, *intptr*[*j*] points to the first element of the interval with *lcp*-value *j*.

Initially, the list contains the elements $(0, 1), (0, 2), \dots, (0, m-1), (|S_{SA[m]}|, m)$ divided into two intervals: The first $m-1$ elements (from left to right) form an interval and the second interval solely consists of the last element.

The linear-time algorithm has the same structure as the naive algorithm. It satisfies invariants (1)–(2) for all elements and invariants (3)–(4) for all elements that are the first element of an interval. The algorithm linearly scans the enhanced suffix array starting at index $m+1$, and for each i with $m+1 \leq i \leq n$, it first calls the procedure *lcp_update* and then the procedure *list_update*.

- *lcp_update*(LCP[*i*]) linearly scans the list of intervals from right to left as described above. Let e be the first element of the current interval and let k be its position (from right to left) in the list. If $e.lcp \geq \text{LCP}[i]$, then we compare the (currently best) value $A[k].lcs$ with $e.lcp$. In case $A[k].lcs \leq e.lcp$, we update $A[k]$ by the assignment $A[k] \leftarrow (e.lcp, e.idx)$. Then the next interval (provided it exists) is considered. If its *lcp*-value is greater than or equal to LCP[*i*], it will be set to be the current interval and so on. Otherwise, if the *lcp*-value of the next interval is strictly smaller than LCP[*i*], then e is the first element of a new interval with *lcp*-value LCP[*i*] and size k . Consequently, we update these values by $e.lcp \leftarrow \text{LCP}[i]$ and $e.size \leftarrow k$. Furthermore, we set $e.end \leftarrow e'$ and $e'.begin \leftarrow e$, where e' is the last element of the list (LV points to it). Finally, the interval pointer *intptr*[LCP[*i*]] must point to e and it is updated accordingly.
- *list_update*(*i*) deletes the element \bar{e} from the list to which *strptr*[*D*[*i*]] points. However, this must be done with care. First, the size of the interval to which \bar{e} belongs must be decreased by one. Second, if \bar{e} is the first or the last element of its interval, then *begin* and *end* pointers must be modified accordingly. Nevertheless, these updates can be done in constant time provided that the first and last element of the interval to which \bar{e} belongs can be found in constant time. According to invariant (4), this interval has *lcp*-value $|\text{lcp}(S_{SA[\bar{e}.idx]}, S_{SA[i]})|$, and the *lcp*-value can be identified in constant time by $\text{LCP}[\text{RMQ}(\bar{e}.idx + 1, i)]$. Then, one finds the first element \bar{e} of the interval to which \bar{e} belongs by following the interval pointer *intptr*[LCP[RMQ($\bar{e}.idx + 1, i$)]], while the

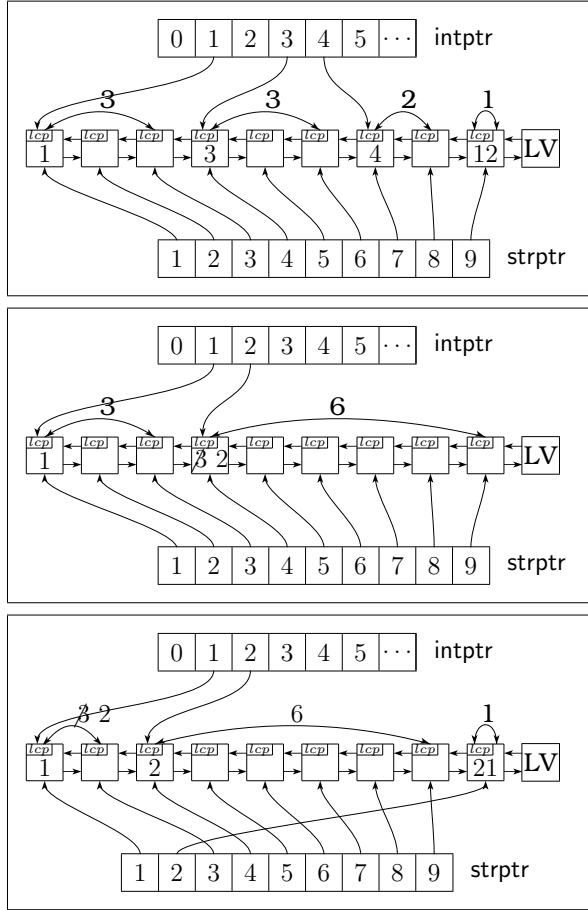


Figure 5.44: In this fictitious example, there are nine strings and the upper figure depicts the point of departure. For each element e in the list, the component $e.idx$ is omitted. Furthermore, the component $e.lcp$ is only shown for the relevant elements, viz. the first element of intervals. The *begin* and *end* pointers of an interval are drawn as an arc with two arrowheads, and the size of the interval is represented by the number above this arc. In the next iteration i , we have $LCP[i] = 2$, $D[i] = 2$, and $|S_{SA[i]}| = 21$. The middle figure shows the situation after the procedure call `lcp_update(2)`, while the lower figure depicts the situation after the procedure call `list_update(i)`.

last element of that interval can be found with the help of the pointer $\bar{e}.end$. Moreover, the procedure $list_update(i)$ adds a new interval at the end of the list. This interval solely consists of the element $e = (|S_{SA[i]}|, i, e, e, 1)$, i.e., both pointers $e.begin$ and $e.end$ point to e itself. Finally, the pointers LV , $strptr[D[i]]$, and $intptr[|S_{SA[i]}|]$ are updated so that they all point to the added element.

Figure 5.44 illustrates how the algorithm works.

In contrast to the naive algorithm, when the algorithm updates an entry in the array A , say $A[k]$, then it does not check whether other entries $A[k']$ with $k' < k$ have to be updated as well. This is because the algorithm directly jumps from the last element of an interval to the first element and skips the elements in between them. Consequently, in the final state of the array A , there may be entries $A[k]$ and $A[k']$ with $2 \leq k' < k \leq m$ so that $\ell_k = A[k].lcs > A[k'].lcs$. This means that the algorithm has found a string common to k strings that is longer than the (currently longest) string common to k' strings, where $k' < k$. Therefore, in a final phase, the algorithm scans the array A from right to left and for $j = m$ down to $j = 3$ it tests whether $A[j].lcs > A[j-1].lcs$ holds true. If so, it updates $A[j-1]$ by the assignment $A[j-1] \leftarrow A[j]$.

Let us turn to the overall complexity of the algorithm. In each iteration, at most two intervals are created. Thus, the algorithm creates at most $2n$ intervals. When the procedure $lcp_update(LCP[i])$ reads an interval, it either overwrites this interval with a new interval or it stops at this interval. Clearly, every interval can be overwritten only once. Thus, in all iterations the procedure lcp_update can overwrite at most $2n$ intervals. Hence it takes $O(n)$ time. Since the same is true for the procedure $list_update$, the overall running time of the algorithm is $O(n)$.

The algorithm described in this section stems from [18]. There, it is also shown that it can be modified so that it solves the more general k -common repeated substring problem in linear time.

Definition 5.6.8 Given S^1, S^2, \dots, S^m and positive integers x_1, x_2, \dots, x_m , the k -common repeated substring problem is to find, for all k with $1 \leq k \leq m$, a longest string ω for which there are at least k strings $S^{i_1}, S^{i_2}, \dots, S^{i_k}$ ($1 \leq i_1 < i_2 < \dots < i_k \leq m$) so that ω occurs at least x_{i_j} times in S^{i_j} for each j with $1 \leq j \leq k$.

Exercise 5.6.9 Consider Algorithm 5.30.

1. Apply it to the example from Figure 5.41.
2. Explain how it works.
3. Analyze its worst-case time complexity.

Algorithm 5.30 Exercise 5.6.9: Show that this algorithm (taken from [17]) computes a longest common substring of the strings S^1, \dots, S^m .

```

construct the GESA of  $S^1, \dots, S^m$ 
b  $lb \leftarrow m + 1$ 
    $rb \leftarrow m + 1$ 
    $str\_cnt[1..m] \leftarrow [0..0]$ 
    $nos \leftarrow 0$  /*  $nos$  is an acronym for number of strings */
    $lcs \leftarrow 0$ 
    $\omega \leftarrow \varepsilon$ 
repeat
  while  $nos \neq m$  and  $rb \leq n$  do
    if  $str\_cnt[D[rb]] = 0$  then
       $nos \leftarrow nos + 1$ 
       $str\_cnt[D[rb]] \leftarrow str\_cnt[D[rb]] + 1$ 
       $rb \leftarrow rb + 1$ 
    if  $rb \leq n$  then
       $idx \leftarrow \text{RMQ}_{\text{LCP}}(lb + 1, rb)$ 
      if  $\text{LCP}[idx] > lcs$  then
         $lcs \leftarrow \text{LCP}[idx]$ 
         $\omega \leftarrow \text{SA}[idx.idx + lcs - 1]$ 
      while  $lb < idx$  do
         $str\_cnt[D[lb]] \leftarrow str\_cnt[D[rb]] - 1$ 
        if  $str\_cnt[D[lb]] = 0$  then
           $nos \leftarrow nos - 1$ 
           $lb \leftarrow lb + 1$ 
    until  $rb = n + 1$ 
return  $\omega$ 

```

5.6.3 Document frequency

Let \mathcal{D} be a database (or library) of strings (or documents) S^1, \dots, S^m on the alphabet Σ . The *document frequency* of a string $\phi \in \Sigma^+$ is defined as the number of strings (documents) in \mathcal{D} that have ϕ as a substring. Formally,

$$df(\phi, \mathcal{D}) = |\{S^j \in \mathcal{D} \mid \phi \text{ is a substring of } S^j\}|$$

If the database \mathcal{D} is clear from the context, then we will simply write $df(\phi)$ instead of $df(\phi, \mathcal{D})$.

The problem is to preprocess the database \mathcal{D} so that document frequency queries $df(\phi)$ can be answered quickly. A straightforward solution is to precompute the GESA of S^1, \dots, S^m in $O(n)$ time. Then, for any string ϕ , $df(\phi)$ can be determined by finding the ϕ -interval $[p..q]$ in the GESA and computing $|\{j \mid D[i] = j \text{ for some } i \text{ with } p \leq i \leq q\}|$ in a scan of the interval. This, however, takes time proportional to the size of the interval $[p..q]$.

Another simple solution can be obtained by counting how often a suffix from S^j appeared up to position i in the GESA of S^1, \dots, S^m . We use an array *count* of dimension $n \times m$ so that

$$\text{count}[i, j] = |\{q \mid 1 \leq q \leq i \text{ and } D[q] = j\}|$$

Clearly, the array *count* can be computed in $O(m \cdot n)$ time. To compute $df(\phi)$, we first determine the ϕ -interval $[p..q]$ in the GESA of S^1, \dots, S^m in $O(|\phi|)$ time. Obviously,

$$df(\phi) = |\{j \mid 1 \leq j \leq m, \text{count}[q, j] - \text{count}[p-1, j] > 0\}|$$

can then be computed in $O(m)$ time. In summary, after an $O(m \cdot n)$ time preprocessing phase, this simple method computes $df(\phi)$ in $O(|\phi| + m)$ time.

Quite surprisingly, there is a method that is able to compute $df(\phi)$ in $O(|\phi|)$ time, using only $O(n)$ time in the preprocessing phase. This is achieved with the help of the so-called correction terms devised by Hui [159]. Let us introduce some useful notations.

Definition 5.6.10 For a string ϕ , define

$$\begin{aligned} \chi_j(\phi) &= \begin{cases} 1 & \text{if } \phi \text{ is a substring of } S^j \\ 0 & \text{otherwise} \end{cases} \\ N_j(\phi) &= |\{i \mid 1 \leq i \leq n_j, S^j[i..i + |\phi| - 1] = \phi\}| \\ CT_j(\phi) &= \begin{cases} N_j(\phi) - 1 & \text{if } N_j(\phi) > 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

So $\chi_j(\phi)$ indicates whether or not ϕ is a substring of S^j , $N_j(\phi)$ is the numbers of occurrences of ϕ in the string S^j , and $CT_j(\phi)$ “corrects” this number in the following sense.

Lemma 5.6.11 *The following equality holds:*

$$\chi_j(\phi) = N_j(\phi) - CT_j(\phi)$$

Proof Straightforward. □

Lemma 5.6.12 *Let $N(\phi) = \sum_{j=1}^m N_j(\phi)$ and $CT(\phi) = \sum_{j=1}^m CT_j(\phi)$. Then*

$$df(\phi) = N(\phi) - CT(\phi)$$

Proof $df(\phi) = \sum_{j=1}^m \chi_j(\phi) = \sum_{j=1}^m (N_j(\phi) - CT_j(\phi)) = N(\phi) - CT(\phi)$. □

$CT(\phi)$ is called *correction term* because it “corrects” $N(\phi)$ in the sense that ϕ occurs in $N(\phi) - CT(\phi)$ of the m strings S^1, \dots, S^m .

Algorithm 5.31 Computation of CT'_j

```

 $last \leftarrow j$       /* index of  $\#_j$  */
for  $i \leftarrow 1$  to  $n$  do
     $CT'_j[i] \leftarrow 0$ 
    for  $i \leftarrow m+1$  to  $n$  do
        if  $D[i] = j$  then
             $l \leftarrow \text{RMQ}_{\text{LCP}}(last+1, i)$ 
             $CT'_j[l] \leftarrow CT'_j[l] + 1$ 
             $last \leftarrow i$ 

```

$N(\phi)$, the number of occurrences of ϕ in the strings S^1, \dots, S^m , equals the size of the ϕ -interval in the GESA of S^1, \dots, S^m , and we can determine this interval in $O(|\phi|)$ time. Moreover, by Hui's [159] method we are able to attach information to the boundaries of the ϕ -interval so that the correction term $CT(\phi)$ can be computed in constant time. Below we present Hui's [159] method, whose preprocessing phase takes only $O(n)$ time.

The trick of the improved algorithm is to count in a different way as in the simple algorithm described above. We explain the new way of counting first for a fixed j , $1 \leq j \leq m$. Let us assume that the LCP-array of the GESA of S^1, \dots, S^m has been preprocessed in linear time so that range minimum queries can be answered in constant time. We use an array CT'_j of size n . Initially $CT'_j[i] = 0$ for all $1 \leq i \leq n$. During a linear scan of the GESA, we keep track of the last index $last$ with $D[last] = j$. Whenever a new index i is considered, we test whether or not $D[i] = j$ holds true. If so, and $last$ is not undefined, then we increment $CT'_j[\text{RMQ}_{\text{LCP}}(last+1, i)]$ by one. Of course, we must provide a reason for this. According to Lemma 4.2.8, $\ell = \text{LCP}[\text{RMQ}_{\text{LCP}}(last+1, i)]$ is the length of the longest common prefix ω of $S_{\text{SA}[last]}$ and $S_{\text{SA}[i]}$. To account for the fact that an(other) occurrence of ω in S^j has been detected, we increment an entry in the array CT'_j . This entry should be within the lcp-interval $\ell-[p..q]$ that represents ω but not within one of its child intervals. In other words, this entry should be an ℓ -index, and $\text{RMQ}_{\text{LCP}}(last+1, i)$ is an ℓ -index.

The pseudo-code for the computation of CT'_j is shown in Algorithm 5.31 and an example can be found in Figure 5.45.

The next lemma provides the link between array CT'_j and $CT_j(\omega)$.

Lemma 5.6.13 *If the lcp-interval $[p..q]$ represents the string ω , then we have $CT_j(\omega) = \sum_{i=p+1}^q CT'_j[i]$.*

Proof Suppose $N_j(\omega) > 1$ (the case $N_j(\omega) \leq 1$ follows by a similar reasoning), and let $r = N_j(\omega)$. By definition, $CT_j(\omega) = r - 1$. Let p_1, p_2, \dots, p_r be the indices in the interval $[p..q]$ so that $D[p_l] = j$, $1 \leq l \leq r$. Without loss of generality, we may assume $p \leq p_1 < p_2 < \dots < p_r \leq q$. For every l with $1 \leq l < r$,

i	$SA[i]$	$ISA[i]$	$LCP[i]$	$S_{SA[i]}^\#$	$D[i]$	$CT'_1[i]$	$CT'_2[i]$	$CT'_3[i]$	$CT'[i]$
1	5	9	-1	$\#_1$	1	0	0	0	0
2	9	14	0	$\#_2$	2	1	0	0	1
3	14	6	0	$\#_3$	3	0	1	0	1
4	6	10	0	$aac\#_2$	2	0	0	1	1
5	11	1	3	$aac\#_3$	3	0	0	0	0
6	3	4	1	$ac\#_1$	1	0	1	1	2
7	7	7	2	$ac\#_2$	2	1	0	0	1
8	12	11	2	$ac\#_3$	3	0	0	0	0
9	1	2	2	$acac\#_1$	1	0	0	0	0
10	4	13	0	$c\#_1$	1	1	1	1	3
11	8	5	1	$c\#_2$	2	1	0	0	1
12	13	8	1	$c\#_3$	3	0	0	0	0
13	10	12	1	$caac\#_3$	3	0	0	1	1
14	2	3	2	$cac\#_1$	1	0	0	0	0
15			-1						

Figure 5.45: The generalized enhanced suffix array of the strings *acac*, *aac*, and *caac* and the arrays CT'_j .

Algorithm 5.31 increments the value of $CT'_j[\text{RMQ}_{\text{LCP}}(p_l + 1, p_{l+1})]$ by one (this happens for $i = p_{l+1}$ in the for-loop). Since $p < p_l + 1 \leq p_{l+1} \leq q$, it follows that Algorithm 5.31 increments the values of $CT'_j[p + 1], \dots, CT'_j[q]$ at least $r - 1$ times, i.e., $CT'_j(\omega) = r - 1 \leq \sum_{i=p+1}^q CT'_j[i]$. To see that the inequality is in fact an equality, consider any pair of indices $i_1 \neq i_2$ with $D[i_1] = j = D[i_2]$ and $D[i] \neq j$ for all $i_1 < i < i_2$, so that at least one of them does not lie in the interval $[p..q]$. If $i_1 < i_2 \leq p$, then $\text{RMQ}_{\text{LCP}}(i_1 + 1, i_2) \leq p$. Hence none of the values $CT'_j[p + 1], \dots, CT'_j[q]$ is incremented. Analogously, if $q \leq i_1 < i_2$, then $q < \text{RMQ}_{\text{LCP}}(i_1 + 1, i_2)$. Again, none of the values $CT'_j[p + 1], \dots, CT'_j[q]$ is incremented. Finally, if $i_1 < p$ or $q < i_2$ (or both), then the longest common prefix of the suffixes $S_{SA[i_1]}$ and $S_{SA[i_2]}$ must be shorter than ω . Because the ω -interval $[p..q]$ is an lcp-interval of lcp-value $|\omega|$, every index i with $p < i \leq q$ satisfies $LCP[i] \geq |\omega|$. It follows as a consequence that the index $\text{RMQ}_{\text{LCP}}(i_1 + 1, i_2)$ cannot lie in the interval $[p + 1..q]$, i.e., none of the values $CT'_j[p + 1], \dots, CT'_j[q]$ is incremented. \square

The attentive reader might have noticed already that one does not need m arrays CT'_1, \dots, CT'_m , but just one array CT' to compute correction terms. The CT' array is the accumulation of the arrays CT'_1, \dots, CT'_m , i.e., $CT'[i] = \sum_{j=1}^m CT'_j[i]$ for every index i , $1 \leq i \leq n$.

Algorithm 5.32 Computation of CT' .

```

for  $j \leftarrow 1$  to  $m$  do
     $last[j] \leftarrow j$  /* index of  $\#_j$  */
for  $i \leftarrow 1$  to  $n$  do
     $CT'[i] \leftarrow 0$ 
for  $i \leftarrow m+1$  to  $n$  do
     $j \leftarrow D[i]$ 
     $l \leftarrow \text{RMQ}_{\text{LCP}}(last[j] + 1, i)$ 
     $CT'[l] \leftarrow CT'[l] + 1$ 
     $last[j] \leftarrow i$ 

```

Lemma 5.6.14 *If the lcp-interval $[p..q]$ represents the string ω , then we have $CT(\omega) = \sum_{i=p+1}^q CT'[i]$.*

Proof The equalities

$$CT(\omega) = \sum_{j=1}^m CT_j(\omega) = \sum_{j=1}^m \sum_{i=p+1}^q CT'_j[i] = \sum_{i=p+1}^q \sum_{j=1}^m CT'_j[i] = \sum_{i=p+1}^q CT'[i]$$

prove the lemma, where the second equality follows from Lemma 5.6.13 and the last equality follows from the definition $CT'[i] = \sum_{j=1}^m CT'_j[i]$. \square

Pseudo-code for the linear-time computation of CT' can be found in Algorithm 5.32, and an example can be found in Figure 5.45.

The final trick is to use the prefix sum array CT'' , defined by $CT''[q] = \sum_{i=1}^q CT'[i]$, instead of the array CT' . This allows us to compute $CT(\omega) = \sum_{i=p+1}^q CT'[i] = CT''[q] - CT''[p]$ in constant time for every ω -interval $[p..q]$. Clearly, the array CT'' can be precomputed in $O(n)$ time.

Algorithm 5.33 summarizes the whole method.

A direct application

As a direct application of the correction term method, we give an alternative solution to the k -common substring problem. The algorithm is based on the same preprocessing phase as Algorithm 5.33. Furthermore, as with the algorithms in Section 5.6.2, it uses an array A of size $m-1$ that stores for each k , $2 \leq k \leq m$, a pair (lcs, idx) , where lcs is the length of a (currently) longest substring ω common to at least k strings and idx is an index so that $S[\text{SA}[idx].. \text{SA}[idx] + lcs - 1] = \omega$, where $S = S^1 \#_1 S^2 \#_2 \dots S^m \#_m$. Initially, $A[k] = (0, \perp)$ for all k with $2 \leq k \leq m$.

Then, the algorithm traverses the lcp-interval tree of S . Here, we will exemplify the method for a bottom-up traversal as described in Algorithm 4.6 (page 94), but other kinds of traversals are possible. As discussed in Section 4.3.2, we merely have to specify the procedure call

Algorithm 5.33 An algorithm to solve the document frequency problem.
 Let \mathcal{D} be a database (or library) of strings (or documents) S^1, \dots, S^m .

- Preprocessing phase:
 - Construct the GESA of S^1, \dots, S^m .
 - Preprocess the LCP-array so that range minimum queries can be answered in constant time.
 - Compute the array CT'' .
 - Computation of $df(\phi)$ for a string ϕ :
 - Determine the ϕ -interval $[p..q]$ by a string matching algorithm.
 - Compute $df(\phi) = N(\phi) - CT(\phi) = q - p + 1 - (CT''[q] - CT''[p])$.
-

$process(\langle \ell, p, q, childList \rangle)$ in Algorithm 4.6, where $\ell[p..q]$ is an lcp-interval representing a string ω . The algorithm proceeds as follows: It computes the number

$$k = df(\omega) = N(\omega) - CT(\omega) = q - p + 1 - (CT''[q] - CT''[p])$$

and compares the (currently best) value $A[k].lcs$ with ℓ . In case $A[k].lcs < \ell$, it updates $A[k]$ by the assignment $A[k] \leftarrow (\ell, p)$. This takes constant time. Hence the entire traversal requires only $O(n)$ time. As in the linear-time algorithm presented in Section 5.6.2, the array A must be updated in a final phase. As explained there, the algorithm scans the array A from right to left and for $j = m$ down to $j = 3$ it tests whether $A[j].lcs > A[j-1].lcs$ holds true. If so, it updates $A[j-1]$ by the assignment $A[j-1] \leftarrow A[j]$.

In contrast to the $O(n)$ time algorithm presented in Section 5.6.2, it seems that this algorithm *cannot* be modified so that it also solves the more general k -common repeated substring problem in $O(n)$ time.

5.6.4 Document retrieval

In Section 5.6.3, we have seen how the document frequency $df(\phi)$ of a string ϕ in a database (or library) \mathcal{D} of strings (or documents) S^1, \dots, S^m can be computed. A related problem is the *document listing problem*, whose task is to retrieve all documents that contain ϕ . Formally, we wish to compute the set

$$dl(\phi) = \{j \mid \phi \text{ is a substring of } S^j, 1 \leq j \leq m\}$$

As in Section 5.6.3, a straightforward solution to the document listing problem is to compute the ϕ -interval $[p..q]$ in the GESA of S^1, \dots, S^m and,

Algorithm 5.34 An algorithm to solve the document listing problem.

Let \mathcal{D} be a database (or library) of strings (or documents) S^1, \dots, S^m .

- Preprocessing phase:
 - Construct the GESA of S^1, \dots, S^m .
 - Preprocess the D -array so that distinct elements range queries can be answered in optimal time.
 - Computation of $dl(\phi)$ for a string ϕ :
 - Determine the ϕ -interval $[p..q]$ by a string matching algorithm.
 - Compute $dl(\phi) = \text{derq}_D(p, q)$.
-

in a scan of the interval, compute $\{j \mid D[i] = j \text{ for some } i \text{ with } p \leq i \leq q\}$. However, we have already seen that this takes time proportional to the size of the interval $[p..q]$. Given the ϕ -interval, we are interested in an algorithm that retrieves all $df(\phi)$ documents containing ϕ in optimal time $O(df(\phi))$. Muthukrishnan [233] showed that this can be done by solving a more general problem, namely the distinct elements range query problem.

Definition 5.6.15 Given an array $A[1..n]$ of elements from the set $\{1, \dots, m\}$, $m \leq n$ and two indices i and j with $1 \leq i \leq j \leq n$, a *distinct elements range query* $\text{derq}_A(i, j)$ on the interval $[i..j]$ returns the set of all distinct elements in $A[i..j]$.

We will show that after a linear-time preprocessing, every query $\text{derq}_A(i, j)$ can be answered in optimal $O(d)$ time, where $d = |\text{derq}_A(i, j)|$ is the number of distinct elements in the subarray $A[i..j]$. With this result, the document listing problem can be solved as shown in Algorithm 5.34. Note that the preprocessing phase of Algorithm 5.34 needs only $O(n)$ time and $dl(\phi)$ can be computed in $O(|\phi| + df(\phi))$ time.

It remains to provide a solution to the distinct elements range query problem. Define an array $\text{Prev}[1..n]$ by

$$\text{Prev}[i] = \max (\{j \mid 1 \leq j < i \text{ and } A[j] = A[i]\} \cup \{0\})$$

In other words, among all indices j with $j < i$ and $A[j] = A[i]$, $\text{Prev}[i]$ is the largest index. If there is no index j with $j < i$ and $A[j] = A[i]$, then $\text{Prev}[i] = 0$. Figure 5.46 shows an example. It is left as an exercise to the reader to show that the array Prev can be computed in linear time. Moreover, Prev is preprocessed in linear time so that range minimum queries can be answered in constant time.

The solution to the problem relies on the following simple lemma.

i	SA	LCP	$S_{SA[i]}^\#$	D	Prev
1	4	-1	$\#_1$	1	0
2	8	0	$\#_2$	2	0
3	12	0	$\#_3$	3	0
4	15	0	$\#_4$	4	0
5	20	0	$\#_5$	5	0
6	3	0	$a\#_1$	1	1
7	14	1	$a\#_4$	4	4
8	19	1	$a\#_5$	5	5
9	2	1	$aa\#_1$	1	6
10	18	2	$aa\#_5$	5	8
11	1	2	$aaa\#_1$	1	9
12	17	3	$aaa\#_5$	5	10
13	5	2	$aac\#_2$	2	2
14	9	2	$aag\#_3$	3	3
15	6	1	$ac\#_2$	2	13
16	10	1	$ag\#_3$	3	14
17	7	0	$c\#_2$	2	15
18	11	0	$g\#_3$	3	16
19	13	1	$ga\#_4$	4	7
20	16	2	$gaaa\#_5$	5	12
21		-1			

Figure 5.46: The generalized enhanced suffix array of the strings $S^1 = aaa$, $S^2 = aac$, $S^3 = aag$, $S^4 = ga$, and $S^5 = gaaa$ and the array Prev.

Lemma 5.6.16 *An element $e \in \{1, \dots, m\}$ occurs in $A[i..j]$ if and only if there is exactly one index k with $i \leq k \leq j$ so that $A[k] = e$ and $\text{Prev}[k] < i$.*

Proof “if:” Obvious.

“only if:” Let $k_1 < k_2 < \dots < k_q$ be all indices in the interval $[i..j]$ so that $A[k_p] = e$, where $1 \leq p \leq q$. Clearly, $\text{Prev}[k_1] < i$ because otherwise there would be a smaller index k_0 in $[i..j]$ with $A[k_0] = e$. Moreover, by the definition of Prev, we have $\text{Prev}[k_p] = k_{p-1} > i$ for all $2 \leq p \leq q$. Thus, the lemma follows. \square

It is a consequence of Lemma 5.6.16 that $\text{derq}_A(i, j) = \{A[k] \mid k \in \mathcal{I}\}$, where $\mathcal{I} = \{k \mid i \leq k \leq j \text{ and } \text{Prev}[k] < i\}$. Thus, it is sufficient to compute the set \mathcal{I} . This is done by an application of procedure *PrevIndices* from Algorithm 5.35: *PrevIndices*(i, i, j, \emptyset) returns the set \mathcal{I} in $O(|\mathcal{I}|)$ time.

Algorithm 5.35 Procedure $PrevIndices(b, i, j, set)$.

```

 $q \leftarrow RMQ_{Prev}(i, j)$ 
if  $Prev[q] < b$  then
     $set \leftarrow set \cup \{q\}$ 
     $set \leftarrow PrevIndices(b, i, q - 1, set)$ 
     $set \leftarrow PrevIndices(b, q + 1, j, set)$ 
return  $set$ 

```

5.6.5 Shortest unique substrings

A primer is an oligonucleotide (an oligonucleotide is a relatively short single-stranded DNA or RNA molecule) that serves as a starting point for DNA synthesis. Primers are required because DNA polymerases, the enzymes that catalyze replication, can only add new nucleotides to an existing strand of DNA. The polymerase starts replication at the 3'-end of the primer, and copies the opposite strand. In most cases of natural DNA replication, the primer for DNA synthesis and replication is a short strand of RNA. Many of the laboratory techniques of biochemistry and molecular biology that involve DNA polymerase, such as DNA sequencing and the polymerase chain reaction (PCR), require DNA primers. These primers are chemically synthesized oligonucleotides, with a length of about twenty bases. They are hybridized to a target DNA (i.e., they form base pairs with a complementary region of the target DNA), which is then copied by the polymerase.

The polymerase chain reaction (PCR) is a scientific technique in molecular biology to amplify a single or few copies of a piece of DNA across several orders of magnitude, generating thousands to millions of copies of a particular DNA sequence. The method relies on thermal cycling, consisting of cycles of repeated heating and cooling of the reaction for DNA melting and enzymatic replication of the DNA. Primers containing sequences complementary to the target region along with a DNA polymerase (after which the method is named) are key components to enable selective and repeated amplification. As PCR progresses, the DNA generated is itself used as a template for replication, setting in motion a chain reaction in which the DNA template is exponentially amplified.

PCR is done with two primers, one for each end of the piece of DNA to be amplified. Here we confine ourselves to finding potential primers on one strand because the search for primers on the opposite strand can be done analogously. Ideally, the nucleotide sequence to which the primer hybridizes should not appear elsewhere in the long DNA sequence. So we can formulate the exact matching version of the *primer selection problem*: For every position i in a given string (DNA sequence) S , compute the

Algorithm 5.36 For each position i in string S , this algorithm finds the shortest substring that begins at i and appears nowhere else in S . Moreover, it outputs all shortest unique substrings of S .

```

construct the ESA of  $S$ 
for  $i \leftarrow 1$  to  $n$  do
     $sus[i] \leftarrow \infty$  /* initialize shortest unique substring array */
     $minlen \leftarrow n$  /*  $S$  itself is a unique substring */
    for  $i \leftarrow 1$  to  $n$  do
         $cur \leftarrow 1 + \max\{LCP[i], LCP[i + 1]\}$ 
        if  $|S_{SA[i]}| \geq cur$  then
             $sus[SA[i]] \leftarrow cur$ 
            if  $cur < minlen$  then
                 $minlen \leftarrow cur$ 
        if  $minlen < n$  then
            for  $i \leftarrow 1$  to  $n$  do
                if  $sus[i] = minlen$  then
                    output  $S[i..i + minlen - 1]$  /* a shortest unique substring */
            else
                output  $S$  itself is the shortest unique substring

```

shortest substring (if any) that begins at i and does not appear anywhere else in S .

In fact, one can determine the length of the shortest unique substring starting at position $SA[i]$ by simply looking at $LCP[i]$ and $LCP[i + 1]$. Because the longest common prefix of $S_{SA[i-1]}$ and $S_{SA[i]}$ has length $LCP[i]$, the length $LCP[i]$ prefix of $S_{SA[i]}$ is not unique. However, the length $LCP[i] + 1$ prefix of $S_{SA[i]}$ is unique among all suffixes of S that are lexicographically smaller than $S_{SA[i]}$. Analogously, we infer that the length $LCP[i + 1] + 1$ prefix of $S_{SA[i]}$ is unique among all suffixes of S that are lexicographically larger than $S_{SA[i]}$, and it is the shortest prefix with this property. There is one caveat though. It may happen that $S_{SA[i]}$ is a prefix of $S_{SA[i+1]}$, in which case $|S_{SA[i]}| = LCP[i + 1]$, so that there is no length $LCP[i + 1] + 1$ prefix of $S_{SA[i]}$ (note, however, that $S_{SA[i]}$ cannot be a prefix of $S_{SA[i-1]}$). The cur -length prefix of $S_{SA[i]}$, where $cur = 1 + \max\{LCP[i], LCP[i + 1]\}$, is the shortest unique substring of S that starts at position $SA[i]$ unless $|S_{SA[i]}| + 1 = cur$. Algorithm 5.36 computes and stores the lengths of these locally shortest unique substrings in the array $sus[1..n]$.

Moreover, Algorithm 5.36 provides an alternative method to compute all globally shortest unique substrings of S (in Section 4.3.3, these have been determined by a breadth-first traversal of the lcp-interval tree). Of course, in this context S must not be terminated by the sentinel character $\$$ because otherwise $\$$ itself would be the globally shortest unique substring.

Without loss of generality, assume that S contains at least two different characters (otherwise S itself is the shortest unique substring). When the first for-loop of Algorithm 5.36 is finished, then the length of all shortest unique substrings of S is minlen , and the second for-loop outputs them.

Exercise 5.6.17 Analyze the worst-case time complexity of Algorithm 5.36. Apart from the output, it is quite obvious that the run-time is $O(n)$. So the question is, how big is the output in the worst case?

DNA microarrays are a widely used tool to monitor gene expression. An oligonucleotide array (“chip”) is a plastic or glass slide containing many spots, each consisting of many copies of a known oligonucleotide (a short DNA sequence) attached to the chip. In an experiment, an mRNA sample is extracted from several cells, reverse-transcribed into complementary DNA (cDNA), and fluorescently labeled. The labeled cDNA is then allowed to bind to (hybridize with) the oligonucleotides on the array. Whenever the Watson-Crick complementary sequence of an oligonucleotide is present in a cDNA sequence, that cDNA will hybridize to the oligonucleotide. Unhybridized cDNA is washed off the chip, and the amount of hybridized cDNA at each spot can be measured via the fluorescent dye intensity. The idea behind this procedure is that each spot represents one gene and that the amount of hybridized cDNA at a spot is a measure of the gene’s transcript abundance in the cell sample, which is often interpreted as the gene expression level or its “activity” in the cell sample. Oligonucleotides should be gene-specific, i.e., only the transcripts of a single gene should hybridize to a given oligonucleotide, so the measurement taken at the oligonucleotide’s spot can be interpreted as the corresponding gene’s expression level in the sample. Note that a cDNA need not contain the perfect Watson-Crick complement of an oligonucleotide to hybridize.

We formulate the exact matching version of the *oligonucleotide selection problem*: Given a set $\mathcal{D} = \{S^1, \dots, S^m\}$ of cDNA sequences (transcribed genes), for every k with $1 \leq k \leq m$, find a shortest substring of S^k that does not appear as a substring of any of the other strings.

Definition 5.6.18 A string $\phi \in \Sigma^*$ is called *string-specific* if $df(\phi) = 1$, i.e., if ϕ occurs in exactly one of the strings S^1, \dots, S^m . Furthermore, ϕ is said to be *S^k -specific* if it is string-specific and a substring of S^k .

With this definition, the oligonucleotide selection problem reads as: Given a set $\mathcal{D} = \{S^1, \dots, S^m\}$ of strings, find a shortest S^k -specific string for every k with $1 \leq k \leq m$.

We solve the oligonucleotide selection problem by a two-phase algorithm. In the first phase, we compute the GESA of S^1, \dots, S^m and preprocess it so that the document frequency of each string that is represented by an lcp-interval can be determined in constant time. This

first phase takes linear time ($O(n)$ time, where $|S| = n = m + \sum_{k=1}^m n_k$ and $|S^k| = n_k$). The second phase consists of a breadth-first traversal of the lcp-interval tree of the GESA of S^1, \dots, S^m , using a queue. Initially, the queue contains only the root interval $0-[1..n]$. The algorithm maintains the following invariant: If, during the traversal, an lcp-interval representing a string u is added to the queue, then $df(u) > 1$, i.e., u is not string-specific. Our algorithm keeps two arrays $minlen[1..m]$ and $minpos[1..m]$, where initially $minlen[k] = \infty = minpos[k]$ for all k with $1 \leq k \leq m$. The entry $minlen[k]$ stores the length of the currently shortest S^k -specific string, while $minpos[k]$ stores a position in S^k at which a currently shortest S^k -specific string starts.

Suppose that $\ell-[i..j]$ is removed from the front of the queue, i.e., it is the lcp-interval that is processed next during the breadth-first traversal. By the invariant, $[i..j]$ represents a string u with $df(u) > 1$, i.e., u is not string-specific. The algorithm computes all child intervals of $[i..j]$ and proceeds by case distinction:

1. If a singleton child interval $[p..p]$ of $[i..j]$ is detected, then it tests whether $S_{SA[p]}^\#[\ell + 1]$ is the separator symbol $\#_k$, where $k = D[p]$. This is the case if and only if $\ell = n_k$.
 - a) If $S_{SA[p]}^\#[\ell + 1] = \#_k$, then all prefixes of $S_{SA[p]}^\#$ without the last character $\#_k$ are not string-specific. Thus, nothing needs to be done.
 - b) If $S_{SA[p]}^\#[\ell + 1] \neq \#_k$, then the length $\ell + 1$ prefix of $S_{SA[p]}^\#$ is S^k -specific. So the algorithm tests whether $\ell + 1 < minlen[k]$, and if the test is positive, it sets $minlen[k] \leftarrow \ell + 1$ and $minpos[k] \leftarrow SA'[p]$.
2. If an lcp-interval $[lb..rb]$ is detected, then the algorithm checks whether the string ω that is represented by $[lb..rb]$ is string-specific.
 - a) If $df(\omega) = 1$, the length $\ell + 1$ prefix of $S_{SA[lb]}^\#$ is S^k -specific, where $k = D[lb]$. If it is shorter than the currently shortest S^k -specific string, the algorithm sets $minlen[k] \leftarrow \ell + 1$ and $minpos[k] \leftarrow SA'[lb]$. Clearly, it is superfluous to inspect the subtree rooted at the interval $[lb..rb]$, so the lcp-interval $[lb..rb]$ is not added to the queue.
 - b) If $df(\omega) > 1$, then $[lb..rb]$ is added to the back of the queue.

Then, the algorithm proceeds with the next lcp-interval at the front of the queue, as described above, until the queue is empty.

It is not difficult to see that the algorithm takes time proportional to the number of processed intervals. In the worst case, this is $O(n)$.

5.6.6 A distance measure for genomes

In a series of papers, Haubold et al. used matching statistics to compute pairwise distances of genome sequences for phylogenetic reconstruction; see [82, 148] and the references therein. (Phylogenetic reconstruction will be discussed in Chapter 10.) As usual, we consider m strings S^1, \dots, S^m of lengths n_1, \dots, n_m . In [82], Domazet-Lošo and Haubold showed that the (non-symmetric) matrix⁸

$$\ell_{ij} = \sum_{k=1}^{n_i} ms_{ij}[k]$$

can be computed in $O(m \cdot n)$ time on a generalized suffix tree. Recall that the matching statistics ms_{ij} of S^i w.r.t. S^j is an array so that for every k with $1 \leq k \leq n_i$ we have $ms_{ij}[k] = \ell$ if and only if $S^i[k..k + \ell - 1]$ is the longest prefix of S^i_k that occurs as a substring of S^j . Since the average lengths ℓ_{ij}/n_i and ℓ_{ji}/n_j do not necessarily coincide, Domazet-Lošo and Haubold [82] used

$$K_r^{ij} = \max\{\ell_{ij}/n_i, \ell_{ji}/n_j\}$$

as a distance measure for phylogenetic reconstruction.

The matrix ℓ_{ij} can be computed as follows. For each i with $1 \leq i \leq m$ do:

1. Construct the ESA of string S^i in $O(n_i)$ time.
2. For each $j \neq i$, compute the matching statistics ms_{ij} of S^i w.r.t. S^j in $O(n_j)$ time by matching S^j against the lcp-interval tree of S^i as in Section 5.5.4. Of course, the matching statistics need not be stored because $\ell_{ij} = \sum_{k=1}^{n_i} ms_{ij}[k]$ can be obtained by successively summing up the values.

For a fixed i , this requires $O(n)$ time. Thus, the overall worst-case time complexity of this method is $O(m \cdot n)$.

In practice, it is faster to compute the matrix ℓ_{ij} on the GESA of S^1, \dots, S^m provided that the GESA fits into main memory. To develop the algorithm that does this, we need the next lemma.

Lemma 5.6.19 *Let idx be the index in the GESA of S^1, \dots, S^m at which one can find S^i_k , the k -th suffix of S^i . For $j \neq i$, $ms_{ij}[k]$ is the lcp-value of the smallest lcp-interval that contains both idx and an index idx' with $D[idx'] = j$ (i.e., the interval contains S^i_k and a suffix of S^j).*

Proof Let $\ell\text{-}[lb..rb]$ be the smallest lcp-interval that contains both idx and an index idx' with $D[idx'] = j$. If $S^j_{k'}$ is the suffix of S^j that occurs at

⁸In fact, they used $\ell_{ij} = \sum_{k=1}^{n_i} (1 + ms_{ij}[k]) = n_i + \sum_{k=1}^{n_i} ms_{ij}[k]$ and mention that their method is based on “locally shortest unique substrings.” This is because the shortest prefix of S^i_k that is not a substring of S^j has length $1 + ms_{ij}[k]$.

idx' , then $u = S^i[k..k + \ell - 1]$ is the longest common prefix of S_k^i and $S_{k'}^j$. Hence u is a prefix of S_k^i that occurs as a substring of S^j . So $ms_{ij}[k] \geq \ell$. We claim that equality holds. For a proof by contradiction, suppose that $ms_{ij}[k] > \ell$. Then, there must be a prefix $\omega = uv$ of S_k^i that has length $|\omega| = ms_{ij}[k]$ and occurs as a substring of S^j . Obviously, the ω -interval is an lcp-interval of lcp-value $|\omega|$. It is readily verified that this lcp-interval is embedded in $[lb..rb]$. This, however, contradicts our assumption that ℓ - $[lb..rb]$ is the smallest lcp-interval that contains both idx and an index idx' with $D[idx'] = j$. \square

The following algorithm computes the matrix ℓ_{ij} in $O(m \cdot n)$ time. (It is a simple exercise to show that the worst-case time complexity of this algorithm is indeed $O(m \cdot n)$.)

1. Initialize a matrix ℓ_{ij} containing zeros.
2. Construct the GESA of S^1, \dots, S^m in $O(n)$ time.
3. In a bottom-up traversal of the lcp-interval tree, for each lcp-interval ℓ - $[lb..rb]$ compute $occ_{[lb..rb]}(S^1), \dots, occ_{[lb..rb]}(S^m)$, where

$$occ_{[lb..rb]}(S^j) = |\{k \mid lb \leq k \leq rb \text{ and } D[k] = j\}|$$

is the number of suffixes in the interval $[lb..rb]$ that belong to string S^j . Clearly, when an lcp-interval is reached in the bottom-up traversal, the corresponding values of all its child intervals are known. During the traversal, if the current lcp-interval ℓ - $[lb..rb]$ has a

- a) singleton child interval $[idx..idx]$ with $D[idx] = i$, then for all $j \neq i$ with $occ_{[lb..rb]}(S^j) > 0$ set

$$\ell_{ij} \leftarrow \ell_{ij} + \ell$$

- b) non-singleton child interval $[p..q]$ so that $occ_{[p..q]}(S^j) = 0$ but $occ_{[lb..rb]}(S^j) > 0$, then for all $i \neq j$ with $occ_{[p..q]}(S^i) \neq 0$ set

$$\ell_{ij} \leftarrow \ell_{ij} + \ell \cdot occ_{[p..q]}(S^i)$$

We use Figures 5.47 and 5.48 to exemplify the algorithm. Suppose that the lcp-interval 1-[4..9] is processed during the bottom-up traversal of the lcp-interval tree. Its child intervals are [4..4], 2-[5..6], and 2-[7..9]. The non-singleton child intervals 2-[5..6] and 2-[7..9] have already been processed and the current state of the matrix ℓ_{ij} is shown in Figure 5.48 at the upper right corner ($\ell_{ij} = 2$ for all $i \neq j$, except for $\ell_{12} = 4$). Furthermore, we have $D[4] = 3$, $occ_{[5..6]}(S^1) = 0$, $occ_{[5..6]}(S^2) = 1$, $occ_{[5..6]}(S^3) = 1$, $occ_{[7..9]}(S^1) = 2$, $occ_{[7..9]}(S^2) = 1$, and $occ_{[7..9]}(S^3) = 0$. Therefore, $occ_{[4..9]}(S^1) = 2$, $occ_{[4..9]}(S^2) = 2$, and $occ_{[4..9]}(S^3) = 2$. For the singleton child interval [4..4], $\ell = 1$ is added

i	LCP	$S_{SA[i]}^\#$	lcp-intervals		
1	-1	$\#_1$	0	1	2
2	0	$\#_2$			
3	0	$\#_3$			
4	0	$a\#_3$			
5	1	$aa\#_3$			
6	2	$aac\#_2$			
7	1	$ac\#_1$			
8	2	$ac\#_2$			
9	2	$acac\#_1$			
10	0	$c\#_1$			
11	1	$c\#_2$			
12	1	$caa\#_3$			
13	2	$cac\#_1$			
14	-1				

Figure 5.47: The LCP-array and the lcp-interval tree (singleton intervals are not shown) of the generalized suffix array of the strings $acac$, aac , and $caac$.

to ℓ_{31} and ℓ_{32} because $D[4] = 3$, $occ_{[4..9]}(S^1) = 2$, and $occ_{[4..9]}(S^2) = 2$. Since $occ_{[5..6]}(S^1) = 0$ and $occ_{[4..9]}(S^1) = 2$, $\ell \cdot occ_{[5..6]}(S^2) = 1 \cdot 1$ is added to ℓ_{21} and $\ell \cdot occ_{[5..6]}(S^3) = 1 \cdot 1$ is added to ℓ_{31} . Analogously, $\ell \cdot occ_{[7..9]}(S^1) = 1 \cdot 2$ is added to ℓ_{13} and $\ell \cdot occ_{[7..9]}(S^2) = 1 \cdot 1$ is added to ℓ_{23} because $occ_{[7..9]}(S^3) = 0$ and $occ_{[4..9]}(S^3) = 2$; see Figure 5.48.

We still have to argue that the algorithm correctly computes the matrix ℓ_{ij} . To this end, fix i and j with $i \neq j$. Let idx be the index in the GESA at which one can find S_k^i . On the path from the singleton interval $[idx..idx]$ to the root $0-[1..n]$ of the lcp-interval tree, let $\ell-[lb..rb]$ be the first lcp-interval (lowest node in the tree) so that $occ_{[lb..rb]}(S^j) > 0$. This means that $\ell-[lb..rb]$ is the smallest lcp-interval that contains idx and an index idx' with $D[idx'] = j$. According to Lemma 5.6.19, $ms_{ij}[k] = \ell$. If $\ell-[lb..rb]$ is the parent interval of $[idx..idx]$, then ℓ is correctly added to ℓ_{ij} by case 3a). If $\ell-[lb..rb]$ is not the parent interval of $[idx..idx]$, then the non-singleton child interval $[p..q]$ of $[lb..rb]$ that lies on the path to $[idx..idx]$ must satisfy $occ_{[p..q]}(S^j) = 0$. So when $\ell-[lb..rb]$ is processed, then $\ell \cdot occ_{[p..q]}(S^i)$ is added to ℓ_{ij} by case 3b). Clearly, at that point in time $ms_{ij}[k] = \ell$ is accounted for. Therefore, when the bottom-up traversal terminates at the root node, we have $\ell_{ij} = \sum_{k=1}^{n_i} ms_{ij}[k]$.

2-[5..6]	1	2	3
1	—	0	0
2	0	—	0 + 2
3	0	0 + 2	—

2-[7..9]	1	2	3
1	—	0 + 2 + 2	0
2	0 + 2	—	2
3	2	2	—

1-[4..9]	1	2	3
1	—	4	0 + 1 · 2
2	2 + 1 · 1	—	2 + 1 · 1
3	2 + 1 + 1 · 1	2 + 1	—

2-[12..13]	1	2	3
1	—	4	2 + 2
2	3	—	3
3	4 + 2	3	—

1-[10..13]	1	2	3
1	—	4 + 1 + 1 · 1	4 + 1
2	3 + 1	—	3 + 1
3	6	3 + 1 · 1	—

0-[1..13]	1	2	3
1	—	6	5
2	4	—	4
3	6	4	—

Figure 5.48: State of the matrix ℓ_{ij} after the lcp-interval shown in the upper left corner has been processed.

5.6.7 All-pairs suffix-prefix matching

The all-pairs suffix-prefix matching problem is a key problem that arises in the context of DNA sequencing. It is formally defined as follows.

Definition 5.6.20 Given m strings S^1, S^2, \dots, S^m the *all-pairs suffix-prefix matching problem* is the problem of finding, for all $j \neq k$ with $1 \leq j \leq m$ and $1 \leq k \leq m$, the *longest* suffix of S^j that is a prefix of S^k .

Why is the all-pairs suffix-prefix matching problem important in the context of DNA sequencing? DNA sequencers cannot read whole genomes in one go, but rather produce short DNA fragments, called reads. To determine the order of bases within a long DNA sequence, the following strategy can be used. The DNA sequence is amplified and randomly fractured into millions of small fragments (shotgun step). These fragments are then sequenced and the original DNA sequence is reconstructed from the reads (assembly step). This strategy can be compared to taking many copies of a book, passing them all through a shredder, and piecing the text of the book back together just by looking at the shredded pieces. Finding all pairs of overlapping reads is a key task in sequence assembly because the source is reconstructed by merging overlapping reads. It must be stressed, however, that the assembly process is exacerbated by

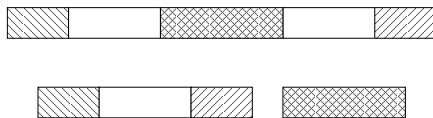


Figure 5.49: Schematic view of a mis-assembled genomic region. The top of the figure shows the true structure of the genomic region, where the white regions represent (nearly) identical copies of the same DNA sequence. If the repeat is long enough, then the assembler will not have a read containing the entire repeat and its unique flanking sequences (the left and right region, respectively). Because the copies of the repeat appear identical to the assembler, the result of the genome assembly (shown on the lower half of the figure, left-hand side) has just one copy of the repeat and the region in between the two copies of the repeat is being lost (lower half, right-hand side).

- the sheer volume of data (next-generation sequencing technologies produce billions of bases in a single run at a relatively low cost),
- repeats (identical and nearly identical sequences), and
- sequencing errors.

Repeats confuse the assembly process because reads originating from distinct copies of the repeat appear identical to the assembler. Figure 5.49 illustrates this phenomenon; for more details see e.g. [277]. Techniques to deal with sequencing errors can be found e.g. in [292, 316].

The all-pairs suffix-prefix matching problem was first solved in optimal time by Gusfield et al. [140]; see also [139]. They used suffix trees for this purpose. In this section, we present an algorithm that solves the all-pairs suffix-prefix matching problem with the help of enhanced suffix arrays; it originates from [249]. Other approaches include [128, 291]. The method of Simpson and Durbin [291] will be discussed in Section 7.7.3.

Now let us address the problem itself. We build the GESA of the strings S^1, S^2, \dots, S^m , i.e., the ESA of the string $S = S^1 \#_1 S^2 \#_2 \dots S^m \#_m$, which has length $n = m + \sum_{l=1}^m n_l$; see Figure 5.50 for an example. A solution to the problem will be stored in a matrix ov (ov is called overlap matrix because $ov[j, k]$ contains the longest overlap of the end of string S^j with the beginning of string S^k).

The following simple lemma is the basis of the solution to the all-pairs suffix-prefix matching problem.

i	SA	LCP	$S_{SA[i]}^\#$	D	SA'	
1	4	-1	$\#_1$	1	4	
2	8	0	$\#_2$	2	4	
3	12	0	$\#_3$	3	4	
4	15	0	$\#_4$	4	3	
5	20	0	$\#_5$	5	5	
6	3	0	$a\#_1$	1	3	
7	14	1	$a\#_4$	4	2	
8	19	1	$a\#_5$	5	4	
9	2	1	$aa\#_1$	1	2	
10	18	2	$aa\#_5$	5	3	
11	1	2	$aaa\#_1$	1	1	i_1
12	17	3	$aaa\#_5$	5	2	
13	5	2	$aac\#_2$	2	1	i_2
14	9	2	$aag\#_3$	3	1	i_3
15	6	1	$ac\#_2$	2	2	
16	10	1	$ag\#_3$	3	2	
17	7	0	$c\#_2$	2	3	
18	11	0	$g\#_3$	3	3	
19	13	1	$ga\#_4$	4	1	i_4
20	16	2	$gaaa\#_5$	5	1	i_5
21		-1				

Figure 5.50: The generalized enhanced suffix array of the strings $S^1 = aaa$, $S^2 = aac$, $S^3 = aag$, $S^4 = ga$, and $S^5 = gaaa$. The indices i_1, i_2, \dots, i_5 satisfy $SA'[i_j] = 1$.

Lemma 5.6.21 *In the generalized enhanced suffix array of S^1, S^2, \dots, S^m , let i be an index with $SA'[i] = 1$ and $D[i] = k$, i.e., $S_{SA[i]}^\# = S^k\#_k$. If the suffix S_p^j of string S^j is a prefix of string S^k , where $j \neq k$, then either*

- $S_p^j\#_j$ appears before index i in the GESA of S^1, S^2, \dots, S^m , or
- $S_p^j\#_j$ appears at an index q with $i < q \leq i + j - k$ in the GESA of S^1, S^2, \dots, S^m (in this case $k < j$ and all suffixes—without their last character—between i and q are identical with S^k).

Proof If S_p^j is a proper prefix of S^k , then (by the definition of the lexicographic order) S_p^j is lexicographically smaller than S^k . It is quite obvious that this implies that $S_p^j\#_j$ is lexicographically smaller than $S^k\#_k$. Therefore, it appears before index i .

Algorithm 5.37 All-pairs suffix-prefix matching

```

for  $i \leftarrow m + 1$  to  $n$  do
   $k \leftarrow D[i]$ 
  if  $SA'[i] = 1$  then
    for  $j \leftarrow 1$  to  $m$  do
      if  $j \neq k$  then
         $Ov[j, k] \leftarrow top(stack[j])$ 
        /* The next five lines handle suffixes identical with  $S^k$  */
      if  $i < n$  then
         $q \leftarrow i + 1$ 
        while  $q \leq n$  and  $LCP[q] = n_k$  and  $|S_{SA'[q]}^{D[q]}| = n_k$  do
           $Ov[D[q], k] \leftarrow SA'[q]$ 
           $q \leftarrow q + 1$ 
      if  $i < n$  then
        if  $LCP[i + 1] < LCP[i]$  then
          for  $\ell \leftarrow LCP[i]$  downto  $LCP[i + 1] + 1$  do
            for each  $j$  in  $list[\ell]$  do
               $pop(stack[j])$ 
               $remove(list[\ell], j)$ 
          else if  $|S_{SA'[i]}^k| = LCP[i + 1]$  then
            /*  $|S_{SA'[i]}^k| = LCP[i + 1]$  implies that  $S_{SA'[i]}^k$  is a prefix of  $S_{SA'[i+1]}^{D[i+1]}$  */
             $push(stack[k], SA'[i])$ 
             $add(list[LCP[i + 1]], k)$ 

```

Otherwise, S_p^j is a non-proper prefix of S^k , i.e., $S_p^j = S^k$. We further distinguish two cases.

(1) If $j < k$, then $S_p^j \#_j$ appears before index i because $\#_j < \#_k$, i.e., $S_p^j \#_j$ is lexicographically smaller than $S^k \#_k$.

(2) If $k < j$, then $S_p^j \#_j$ is lexicographically larger than $S^k \#_k$. We observe that only the length $n_k + 1$ suffixes of $S^{k+1} \#_{k+1}, S^{k+2} \#_{k+2}, \dots, S^{j-1} \#_{j-1}$ can possibly lie lexicographically in between $S^k \#_k$ and $S_p^j \#_j$. Moreover a length $n_k + 1$ suffix of $S^h \#_h$, where $k < h < j$ and $n_k \leq n_h$, lies lexicographically in between $S^k \#_k$ and $S_p^j \#_j$ if and only if the length n_k suffix of S^h is identical with S^k . Thus, $S_p^j \#_j$ appears at an index q with $i < q \leq i + j - k$. \square

Algorithm 5.37 scans the generalized enhanced suffix array from index $m + 1$ to index n . During the scan, it keeps track of all suffixes seen so far that are a *prefix* of the current suffix. This is achieved by the usage of m stacks $stack[1], stack[2], \dots, stack[m]$, one for each string S^1, S^2, \dots, S^m . To efficiently administer stacks, the algorithm uses at most $\max\{n_1, n_2, \dots, n_m\}$ lists $list[1], list[2], \dots$. Initially, all stacks and lists are empty.

2 (aa)				3 (aa)
3 (a)			2 (a)	4 (a)
<i>stack</i> [1]	<i>stack</i> [2]	<i>stack</i> [3]	<i>stack</i> [4]	<i>stack</i> [5]

Figure 5.51: The stacks before the outer for-loop of Algorithm 5.37 is executed for $i = 11$. For example, the top element of *stack*[1] is 2 (the corresponding suffix $S_2^1 = aa$ of S^1 is supplied in parentheses).

We apply Algorithm 5.37 to the strings from Figure 5.50. Before the outer for-loop is executed for $i = 11$, the stacks contain the elements depicted in Figure 5.51 and we have $list[1] = [1, 4, 5]$ and $list[2] = [1, 5]$. In the execution of the outer for-loop for $i = 11$, one has $SA'[11] = 1$ and $k = D[11] = 1$, so that the first column of the *Ov*-matrix is filled in by the assignments $Ov[2, 1] \leftarrow \perp$ (the operation *top* applied to an empty stack returns \perp), $Ov[3, 1] \leftarrow \perp$, $Ov[4, 1] \leftarrow 2$, and $Ov[5, 1] \leftarrow 3$. However, the suffix $S_2^5 = aaa$ of S^5 equals S^1 and $S_2^5 \#_5$ directly follows $S^1 \#_1$ in the generalized suffix array. Algorithm 5.37 detects this by means of the subsequent five lines of code: the suffix S_p^j , where $j = D[q]$ and $p = SA'[q]$ equals S^k if and only if $LCP[q] = n_k$ and $|S_p^j| = n_k$. Note that $|S_p^j|$ can be determined in constant time because $|S_p^j| = n_j - p + 1$. In our example, $LCP[12] = 3 = n_1$ and $|S_2^5| = n_5 - 2 + 1 = 3 = n_1$ and thus Algorithm 5.37 assigns $SA'[12] = 2$ to $Ov[5, 1]$. Because the else-if-condition is true, $SA'[11] = 1$ is pushed onto *stack*[1] and the string number 1 is added to $list[LCP[12]] = list[3]$ (which was empty). In the execution of the outer for-loop for $i = 12$, the top element is removed from the stack *stack*[1] (hence the stacks again have the elements as in Figure 5.51) and the string number 1 is removed from $list[3]$ (hence $list[3]$ is empty again). During the execution of the outer for-loop for $i = 13$, the second column of the *Ov*-matrix is filled in (see Figure 5.52); for $i = 14$, the third column of the *Ov*-matrix is filled in, etc.

To prove the correctness of Algorithm 5.37, we show that the following properties are invariants of the outer for-loop: before the for-loop is executed for some i with $m + 1 \leq i \leq n$, the stacks *stack*[1], *stack*[2], ..., *stack*[m] contain exactly the suffixes⁹ (in increasing order of string length) seen so far that are a prefix of $S_{SA'[i]}^{D[i]}$. Furthermore, j is in $list[\ell]$ if and only if *stack*[j] contains the suffix $S_{n_j-\ell+1}^j$ (the suffix of S^j that has length ℓ).

Under the assumption that the above-mentioned properties hold before the for-loop is executed for some i , we show that the properties also hold before the for-loop is executed for $i + 1$. We proceed by case analysis.

⁹As a matter of fact, the stacks do not contain suffixes but suffix numbers. For example, if $p = top(stack[j])$, then the suffix number p corresponds to the suffix S_p^j of S^j .

ov	1	2	3	4	5
1	—	2 (aa)	2 (aa)	\perp	\perp
2	\perp	—	\perp	\perp	\perp
3	\perp	\perp	—	3 (g)	3 (g)
4	2 (a)	2 (a)	2 (a)	—	1 (ga)
5	2 (aaa)	3 (aa)	3 (aa)	\perp	—

Figure 5.52: The overlap matrix ov after Algorithm 5.37 was applied to the strings of Figure 5.50. For example, $ov[4, 1] = 2$ means that $S_2^4 = a$ is the longest suffix of $S^4 = ga$ that matches a prefix of $S^1 = aaa$.

If $LCP[i+1] \geq LCP[i]$, then all suffixes seen so far that are a prefix of $S_{SA'[i]}^{D[i]}$ are also a prefix of $S_{SA'[i+1]}^{D[i+1]}$. Moreover, the else-if-condition tests whether $S_{SA'[i]}^k$ (where $k = D[i]$) is a prefix of $S_{SA'[i+1]}^{D[i+1]}$. If this is the case, the suffix number $SA'[i]$ representing the suffix $S_{SA'[i]}^k$ is pushed onto $stack[k]$ and the string number k is added to $list[|S_{SA'[i]}^k|]$. Therefore, the properties also hold before the for-loop is executed for $i+1$.

Otherwise, if $LCP[i+1] < LCP[i]$, then all suffixes of length ℓ with $LCP[i] \geq \ell \geq LCP[i+1] + 1$ must be removed from the stacks because these suffixes are not a prefix of $S_{SA'[i+1]}^{D[i+1]}$. For each ℓ (in decreasing order) we use the list $list[\ell]$ to find all stacks that have a suffix of length ℓ on top and successively remove these top elements from the stacks and the string numbers from $list[\ell]$ until $list[\ell]$ is empty. Again, the properties also hold before the for-loop is executed for $i+1$. (Observe that in this case $S_{SA'[i]}^k$ cannot be a prefix of $S_{SA'[i+1]}^{D[i+1]}$ because $|S_{SA'[i]}^k| \geq LCP[i] > LCP[i+1]$.)

The correctness of Algorithm 5.37 now follows from the invariant properties. If the for-loop is executed for some i with $SA'[i] = 1$ and $D[i] = k$, then the topmost element of $stack[j]$, say $t = top(stack[j])$, represents the longest suffix S_t^j of S^j seen so far that is a prefix of S^k . Lemma 5.6.21 implies that S_t^j is the longest suffix of S^j that is a prefix of S^k , unless $k < j$ and there is a suffix S_p^j of S^j so that $S_p^j = S^k$. However, all suffixes that are identical with S^k directly (precede or) succeed S^k in the suffix array and the algorithm deals with this special case.

Let us analyze the worst-case time complexity of Algorithm 5.37. Each of the n suffixes is pushed at most once onto a stack. This implies that there are at most n elements added to the lists. Consequently, there are at most $2n$ push and pop operations on the stacks, and at most $2n$ add and remove operations on the lists. There are m indices i_1, i_2, \dots, i_m so that $SA'[i_j] = 1$. For every i_j , the inner for-loop is executed m times. Furthermore, there are at most $m-1$ suffixes that are identical with the

current suffix S^k (excluding S^k itself). Thus, for every i_j the algorithm takes $O(m)$ time. This sums up to $O(m^2)$ time for all m indices i_1, i_2, \dots, i_m . Altogether, the worst-case time complexity of Algorithm 5.37 is $O(n + m^2)$. This is optimal because n is the size of the input and m^2 is the size of the output.

Exercise 5.6.22 Formulate the *all-pairs prefix-prefix matching problem* and give two algorithms that solve it: one with range minimum queries and one without range minimum queries. Analyze the worst-case time complexity of the algorithms.

Exercise 5.6.23 Define the *all-pairs suffix-suffix matching problem* and show that it is equivalent to the all-pairs prefix-prefix matching problem.

Exercise 5.6.24 Formulate the *all-pairs longest common substring problem* and give an $O(m \cdot n)$ time algorithm that solves it, where m is the number of input strings and n is the sum of their lengths. (It is an open problem whether an $O(n + m^2)$ time solution exists.)

5.7 String kernels

5.7.1 Machine learning

If a classification problem is non-linear, one can map the input data into a high (possibly infinite) dimensional feature space, where each coordinate corresponds to one feature of the data items. A map $\phi : \mathcal{X} \rightarrow \mathcal{F}$ from the input space \mathcal{X} to a feature space \mathcal{F} (endowed with an inner product $\langle \cdot, \cdot \rangle$) is called a feature map. In the feature space \mathcal{F} , a variety of methods can be used to find relations in the data, and a linear relation in \mathcal{F} usually corresponds to a non-linear relation in the input space \mathcal{X} . Kernel methods owe their name to the use of kernel functions $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ that enable them to operate in the feature space \mathcal{F} without ever computing the coordinates of the data in that space, but rather by simply computing $k(x, y) = \langle \phi(x), \phi(y) \rangle$. This operation is often computationally more efficient than the explicit computation of the coordinates. Among other things, kernel functions have been introduced for strings, graphs, and images. Algorithms capable of operating with kernels include support vector machines, linear discriminant analysis, principal components analysis, and many others; see [285].

Support vector machines (SVMs) are a set of related supervised learning methods used for classification. Let $\{(x_1, y_1), \dots, (x_m, y_m)\}$ be given, where each x_i is a training example from the input space \mathcal{X} belonging to one of two classes, and each $y_i \in \{-1, 1\}$ indicates to which class x_i belongs. Roughly speaking, SVMs learn a linear decision boundary to discriminate

between the two classes. To be more precise, they use the training data $\{(x_1, y_1), \dots, (x_m, y_m)\}$ and the kernel matrix with the entries $k(x_i, x_j)$, where $i, j \in \{1, \dots, m\}$, to construct a classifier $f(x) = \text{sgn}(\sum_{i=1}^m \alpha_i y_i k(x_i, x) + b)$, by finding optimal¹⁰ coefficients α_i and offset b . In the classifier, sgn is the signum function, and the set $\{x_i \mid 1 \leq i \leq m, \alpha_i \neq 0\}$ is called the set of *support vectors*. The classifier $f(x)$ can then be used to classify new test examples: it predicts whether a new example x falls into one class or the other.

To implement this discriminative approach to classification, one has to solve the following subproblems:

1. Compute the kernel matrix, i.e., compute $k(x_i, x_j)$ for $i, j \in \{1, \dots, m\}$.
2. Find the coefficients α_i and offset b (learning phase).
3. For $x \in \mathcal{X}$, compute $f(x) = \text{sgn}(\sum_{i=1}^m \alpha_i y_i k(x_i, x) + b)$ (classification).

In the following, we show how steps (1) and (3) can be implemented for string kernels. Problem (1) is tackled in Sections 5.7.2 and 5.7.3, and a solution to problem (3) is sketched in Section 5.7.4. Solutions to problem (2) are discussed e.g. in [63, 285].

We would like to point out that the approach was successfully applied to the protein classification problem by Leslie et al. [201], albeit with a different sequence-similarity kernel (called spectrum kernel).

5.7.2 Calculating a string kernel

In our context, the input space \mathcal{X} is the set Σ^* of all finite strings on an alphabet Σ , and the feature map $\phi : \mathcal{X} \rightarrow \mathbb{R}^\infty$ maps a string $S \in \Sigma^*$ to the vector $(\sqrt{W(\omega)} \cdot \text{occ}_\omega(S))_{\omega \in \Sigma^*}$, where $W : \Sigma^* \rightarrow \mathbb{R}$ is a *weight function* with $W(\omega) \geq 0$, and $\text{occ}_\omega(S)$ denotes the number of substring occurrences of ω in S . The *string kernel* (or weighted all-substrings kernel) of two strings S^1 and S^2 is defined by

$$k(S^1, S^2) = \langle \phi(S^1), \phi(S^2) \rangle = \sum_{\omega \in \Sigma^*} W(\omega) \text{occ}_\omega(S^1) \text{occ}_\omega(S^2)$$

For example, the linear string kernel of $S^1 = \text{acaaacatat}$ and $S^2 = \text{caact}$ for the constant weight function $W(\omega) = 1$ is $k(S^1, S^2) = 26$; see Figure 5.53 (in which strings $\omega \in \Sigma^+$ with $\text{occ}_\omega(S^1) = 0$ or $\text{occ}_\omega(S^2) = 0$ are omitted).

Vishwanathan, Smola, and Teo [307, 320] have shown that such a string kernel can be computed in linear time with the help of suffix arrays and matching statistics, and we will follow their approach. The following lemma is the key to the solution.

¹⁰See e.g. [63] for formulations of SVM optimization problems.

$\omega \in \Sigma^+$	a	c	t	aa	ac	ca	aac	caa
$occ_\omega(S^1) = occ_\omega(acaacatat)$	6	2	2	2	2	2	1	1
$occ_\omega(S^2) = occ_\omega(caact)$	2	2	1	1	1	1	1	1
$occ_\omega(S^1) \cdot occ_\omega(S^2)$	12	4	2	2	2	2	1	1

Figure 5.53: $k(S^1, S^2) = \sum_{\omega \in \Sigma^+} occ_\omega(S^1) occ_\omega(S^2) = 26$.

Lemma 5.7.1 For strings $\omega, S \in \Sigma^*$, we write $\omega \sqsubset S$ if ω is a prefix of S . Then,

$$k(S^1, S^2) = \sum_{p_2=1}^{n_2} \sum_{\omega \sqsubset S_{p_2}^2} W(\omega) occ_\omega(S^1)$$

Proof Of course, a summand $W(\omega) occ_\omega(S^1) occ_\omega(S^2)$ is zero if ω does not occur in S^1 or S^2 . Thus, it suffices to sum over all $\omega \in \Sigma^*$ that are substrings of both S^1 and S^2 . Fix such a string ω . We claim that

$$occ_\omega(S^2) = |\{p_2 \mid 1 \leq p_2 \leq n_2, \omega \sqsubset S_{p_2}^2\}|$$

To verify the claim, suppose that $occ_\omega(S^2) = k$, and let i_1, \dots, i_k be all the positions in S^2 at which ω occurs, i.e., $\omega = S^2[i_j..i_j + |\omega| - 1]$ for all j with $1 \leq j \leq k$. In other words, i_1, \dots, i_k are the positions in S^2 so that ω is a prefix of $S_{i_j}^2$ for all j with $1 \leq j \leq k$. Now the claim follows:

$$occ_\omega(S^2) = k = |\{i_1, \dots, i_k\}| = |\{p_2 \mid 1 \leq p_2 \leq n_2, \omega \sqsubset S_{p_2}^2\}|$$

Consequently,

$$\sum_{\omega \in \Sigma^*} W(\omega) occ_\omega(S^1) occ_\omega(S^2) = \sum_{p_2=1}^{n_2} \sum_{\omega \sqsubset S_{p_2}^2} W(\omega) occ_\omega(S^1)$$

□

The next two lemmata generalize Lemma 4.3.18.

Lemma 5.7.2 Let $[lb..rb]$ be an lcp-interval and let $[b..e]$ be its parent interval in the lcp-interval tree of S^1 . Suppose $[b..e]$ represents u and $[lb..rb]$ represents uv , where u and uv are substrings of S^1 with $v \neq \varepsilon$. For each prefix p of v we have

$$\sum_{\omega \sqsubset up} W(\omega) occ_\omega(S^1) = \sum_{\omega \sqsubset u} W(\omega) occ_\omega(S^1) + \left(\sum_{\omega \sqsubset up, \omega \not\sqsubset u} W(\omega) \right) (rb - lb + 1)$$

Proof Let ω be a substring of S^1 so that $\omega \sqsubset up$ but $\omega \not\sqsubset u$. The key observation is that the ω -interval coincides with the uv -interval. In other words, ω occurs as often in S^1 as uv does, namely $(rb - lb + 1)$ times. Thus,

$$\begin{aligned}
 \sum_{\omega \sqsubset up} W(\omega) \text{occ}_\omega(S^1) &= \sum_{\omega \sqsubset u} W(\omega) \text{occ}_\omega(S^1) + \sum_{\omega \sqsubset up, \omega \not\sqsubset u} W(\omega) \text{occ}_\omega(S^1) \\
 &= \sum_{\omega \sqsubset u} W(\omega) \text{occ}_\omega(S^1) + \sum_{\omega \sqsubset up, \omega \not\sqsubset u} W(\omega) \text{occ}_{uv}(S^1) \\
 &= \sum_{\omega \sqsubset u} W(\omega) \text{occ}_\omega(S^1) + \left(\sum_{\omega \sqsubset up, \omega \not\sqsubset u} W(\omega) \right) (rb - lb + 1)
 \end{aligned}$$

□

There are some interesting weight functions for which $W(\omega)$ can be computed in constant time. To keep the presentation simple, we here focus on length-dependent weights. Nevertheless, the subsequent considerations also apply, with a grain of salt, to many other weights; see [320] for details. A weight function W is called *length-dependent* if, for all strings $u, v \in \Sigma^+$, $|u| = |v|$ implies $W(u) = W(v)$. In other words, the weight of a string solely depends on its length; so strings of the same length get the same weight. As an example, consider $W(\omega) = \lambda^{-|\omega|}$ for some constant $0 < \lambda < 1$. Since a length dependent weight function $W : \Sigma^* \rightarrow \mathbb{R}$ can be viewed as a function $W : \mathbb{N} \rightarrow \mathbb{R}$, we will also write $W(|\omega|)$ instead of $W(\omega)$. Clearly, the *prefix sum* of all weights of strings up to length j can be precomputed by $\text{PS}(j) = \sum_{i=0}^j W(i)$.

Lemma 5.7.3 *Let W be a length-dependent weight function. In the situation of Lemma 5.7.2, let $[b..e]$ (the parent interval of $[lb..rb]$) have lcp-value ℓ (i.e., $|u| = \ell$). Then, we have*

$$\sum_{\omega \sqsubset up} W(\omega) \text{occ}_\omega(S^1) = \sum_{\omega \sqsubset u} W(\omega) \text{occ}_\omega(S^1) + (\text{PS}(|up|) - \text{PS}(\ell)) (rb - lb + 1)$$

Proof This follows from Lemma 5.7.2 because $\sum_{\omega \sqsubset up, \omega \not\sqsubset u} W(\omega)$ equals

$$\sum_{k=|u|+1}^{|up|} W(k) = \sum_{k=\ell+1}^{|up|} W(k) = \sum_{k=1}^{|up|} W(k) - \sum_{k=1}^{\ell} W(k) = \text{PS}(|up|) - \text{PS}(\ell)$$

□

i	SA	LCP	$S_{SA[i]}$	VAL
1	3	-1	<i>aaacatat</i>	
2	4	2	<i>aacatat</i>	8
3	1	1	<i>acaaacatat</i>	6
4	5	3	<i>acatat</i>	10
5	9	1	<i>at</i>	6
6	7	2	<i>atat</i>	8
7	2	0	<i>caaacatat</i>	0
8	6	2	<i>catat</i>	4
9	10	0	<i>t</i>	0
10	8	1	<i>tat</i>	2
11		-1		

Figure 5.54: The enhanced suffix array of $S = acaaacatat$ with VAL array.

In the above situation, if $val([b..e]) = \sum_{\omega \sqsubseteq u} W(\omega) occ_{\omega}(S^1)$ is already known, then we can compute the value $val([lb..rb]) = \sum_{\omega \sqsubseteq uv} W(\omega) occ_{\omega}(S^1)$ by

$$val([lb..rb]) = val([b..e]) + (PS(r) - PS(\ell)) (rb - lb + 1) \quad (5.2)$$

in constant time, where $r = |uv|$ is the lcp-value of $[lb..rb]$. It follows as a consequence that we can compute the value val for each lcp-interval by a top-down traversal of the lcp-interval tree in $O(n_1)$ time (the value $val([1..n_1])$ of the root interval is set to 0). For the constant weight function $W(\omega) = 1$, Section 4.3.3 contains an algorithm that computes these values and stores them in an array VAL; it is easy to modify the algorithm so that it can cope with length-dependent weight functions. Recall that for an lcp-interval ℓ - $[b..e]$, $val([b..e])$ is stored at each of its ℓ -indices in the array VAL; see Figure 5.54.

The second component of a linear-time algorithm to calculate the string kernel $k(S^1, S^2)$ is the linear-time algorithm that computes the matching statistics of S^2 w.r.t. S^1 in $O(n_2)$ time; see Algorithm 5.28 (page 200). Recall that the matching statistics of S^2 w.r.t. S^1 is an array ms so that for every entry $ms[p_2] = (q, [lb..rb])$, $1 \leq p_2 \leq n_2$, the following holds:

1. $S^2[p_2..p_2 + q - 1]$ is the longest prefix of $S^2_{p_2}$ that occurs as a substring of S^1 .
2. $[lb..rb]$ is the $S^2[p_2..p_2 + q - 1]$ -interval in the ESA of S^1 .

According to Lemma 5.7.1,

$$k(S^1, S^2) = \sum_{p_2=1}^{n_2} \sum_{\omega \sqsubseteq S^2_{p_2}} W(\omega) occ_{\omega}(S^1)$$

p_2	1	2	3	4	5
$ms[p_2]$	3, [7..7]	3, [2..2]	2, [3..4]	1, [7..8]	1, [9..10]
summand	$4 + 1 \cdot 1$	$8 + 1 \cdot 1$	$6 + 1 \cdot 2$	$0 + 1 \cdot 2$	$0 + 1 \cdot 2$

Figure 5.55: The second row shows the matching statistics of $S^2 = caact$ w.r.t. $S^1 = acaaacatat$. cf. Figure 5.54. The last row shows the summands $\sum_{\omega \sqsubset S_{p_2}^2} occ_\omega(S^1)$.

Therefore, when Algorithm 5.28 determines the matching statistic $ms[p_2] = (q, [lb..rb])$, the summand $\sum_{\omega \sqsubset S_{p_2}^2} W(\omega) occ_\omega(S^1)$ can be computed by Lemma 5.7.3:

$$\sum_{\omega \sqsubset S_{p_2}^2} W(\omega) occ_\omega(S^1) = val([b..e]) + (PS(q) - PS(\ell)) (rb - lb + 1) \quad (5.3)$$

where $\ell - [b..e]$ is the parent interval of $[lb..rb]$. Each summand can be obtained in constant time by Lemma 4.3.9:

1. If $LCP[lb] = LCP[rb + 1]$, then $\ell = LCP[lb] = LCP[rb + 1]$ and $val([b..e]) = VAL[lb] = VAL[rb + 1]$ because both lb and $rb + 1$ are ℓ -indices of the parent interval of $[lb..rb]$.
2. If $LCP[lb] > LCP[rb + 1]$, then $\ell = LCP[lb]$ and $val([b..e]) = VAL[lb]$ because lb is the last ℓ -index of the parent interval of $[lb..rb]$.
3. If $LCP[lb] < LCP[rb + 1]$, then $\ell = LCP[rb + 1]$ and $val([b..e]) = VAL[rb + 1]$ because $rb + 1$ is the first ℓ -index of the parent interval of $[lb..rb]$.

As an example consider Figures 5.54 and 5.55 and the constant weight function $W(\omega) = 1$. The first summand is computed with the help of $ms[1] = (q, [lb..rb]) = (3, [7..7])$ as follows: The comparison of $LCP[lb]$ with $LCP[rb + 1]$ yields $LCP[7] = 0 < 2 = LCP[8]$, so the third case applies. Thus,

$$\sum_{\omega \sqsubset S_1^2} W(\omega) occ_\omega(S^1) = VAL[rb + 1] + (q - LCP[rb + 1]) (rb - lb + 1) = 4 + 1 \cdot 1 = 5$$

The second summand is computed with $ms[2] = (q, [lb..rb]) = (3, [2..2])$. Because $LCP[2] = 2 > 1 = LCP[3]$, the second case applies. Therefore,

$$\sum_{\omega \sqsubset S_2^2} W(\omega) occ_\omega(S^1) = VAL[lb] + (q - LCP[lb]) (rb - lb + 1) = 8 + 1 \cdot 1 = 9$$

All summands are listed in the last row of Figure 5.55. Because there are n_2 summands, each of which can be computed in constant time, the string kernel $k(S^1, S^2)$ can be computed in $O(n_2)$ time—after an $O(n_1)$ time pre-processing phase. (Of course the matching statistics need not be stored.)

5.7.3 Calculating the kernel matrix

Given m strings S^1, \dots, S^m on a constant-size alphabet Σ , their $m \times m$ string kernel matrix can be calculated as follows.

For each i with $1 \leq i \leq m$ do:

1. Construct the ESA of string S^i in $O(n_i)$ time, where $n_i = |S^i|$.
2. Precompute the array VAL_i by a top-down traversal of the lcp-interval tree of S^i in $O(n_i)$ time as in Section 4.3.3.
3. For each $j \neq i$, compute $k(S^i, S^j)$ in $O(n_j)$ time by matching S^j against the lcp-interval tree of S^i as described in the previous section.

For a fixed i , this requires $O(n)$ time: $O(n_i)$ time for the first two phases, and $\sum_{j \neq i} O(n_j) = O(n - n_i)$ time for the last phase. Because the three phases must be applied to each of the m strings, the overall worst-case time complexity of calculating the $m \times m$ string kernel matrix is $O(m \cdot n)$.

However, this method of Vishwanathan, Smola, and Teo [307, 320] cannot handle the important TF-IDF weighting scheme. We will see in Section 5.7.5 how to deal efficiently with this weighting scheme.

5.7.4 Classification

To determine to which class a new string S of length N belongs, we have to compute $f(S) = \text{sgn}(\sum_{i=1}^m \alpha_i y_i k(S^i, S) + b)$. It will be shown how to calculate $g(S) = \sum_{i=1}^m \alpha_i y_i k(S^i, S)$ efficiently. The computation of $f(S) = \text{sgn}(g(S) + b)$ is then straightforward.

According to Section 5.7.2, we have

$$\begin{aligned} g(S) &= \sum_{i=1}^m \alpha_i y_i \left(\sum_{p=1}^N \sum_{\omega \sqsubset S_p} W(\omega) \text{occ}_{\omega}(S^i) \right) \\ &= \sum_{p=1}^N \sum_{\omega \sqsubset S_p} W(\omega) \left(\sum_{i=1}^m \alpha_i y_i \text{occ}_{\omega}(S^i) \right) \end{aligned}$$

It follows as a consequence that we can proceed exactly as in Section 5.7.2 provided that we work with the GESA of the support vectors S^1, \dots, S^m and replace the factor $rb - lb + 1$ in Equations (5.2) and (5.3) by the factor $\sum_{k=lb}^{rb} \alpha_{D[k]} y_{D[k]}$ (if an index k in the ω -interval $[lb..rb]$ belongs to string S^i , i.e., $D[k] = i$, then the summand must be $\alpha_{D[k]} y_{D[k]} = \alpha_i y_i$). Using the same prefix-sum-trick as for length-dependent weights, this factor can be computed in constant time. More precisely, define $\text{PS}'(0) = 0$ and precompute

$PS'(j) = \sum_{k=1}^j \alpha_{D[k]} y_{D[k]}$ for $1 \leq j \leq n$. Then,

$$\sum_{k=lb}^{rb} \alpha_{D[k]} y_{D[k]} = PS'(rb) - PS'(lb - 1)$$

With these modifications, the same techniques as in Section 5.7.2 can be used to compute $g(S)$ in $O(N)$ time. So one can classify a string S in linear time, independent of the size of the support vectors.

5.7.5 The TF-IDF weighting scheme

Given a database (library) \mathcal{D} of strings (documents) S^1, \dots, S^m on a constant-size alphabet Σ and a string $\omega \in \Sigma^*$, the *term frequency* (TF) is the raw frequency of the string (the term) ω inside string S^j . In other words, it is $occ_\omega(S^j)$, the number of times ω occurs in S^j . The *document frequency* $df(\omega)$ is the number of strings (documents) in \mathcal{D} in which ω occurs at least once. We would like to point out that Yamamoto and Church [332] first showed how the term frequency and the document frequency of all substrings in a database can be computed with the help of suffix arrays.

For every string ω occurring in \mathcal{D} , the *inverse document frequency* (IDF) is defined by

$$idf(\omega) = \log \frac{m}{df(\omega)}$$

Using this as weight function, we obtain the so-called TF-IDF score

$$occ_\omega(S^j) \log \frac{m}{df(\omega)}$$

of the string ω with respect to the document S^j .

The kernel matrix for the TF-IDF weighting scheme can be computed in $O(m \cdot n)$ time as follows:

1. Construct the GESA of S^1, \dots, S^m in $O(n)$ time, where $n = m + \sum_{i=1}^m n_i$.
2. In a bottom-up traversal of the corresponding lcp-interval tree, compute the values $occ_\omega(S^1), \dots, occ_\omega(S^m)$ and

$$df(\omega) = |\{k \mid 1 \leq k \leq m \text{ and } occ_\omega(S^k) > 0\}|$$

for every string ω that is represented by an lcp-interval (and store these values; for example, at all lcp-indices of the ω -interval). This takes $O(m \cdot n)$ time.

3. In a top-down traversal, simultaneously precompute the functions val_1, \dots, val_m and store them in the arrays VAL_1, \dots, VAL_m (one for each

i	LCP	$S_{SA[i]}^\#$	lcp-intervals		
1	-1	$\#_1$	0	1	2
2	0	$\#_2$			
3	0	$\#_3$			
4	0	$a\#_3$			
5	1	$aa\#_3$			
6	2	$aac\#_2$			
7	1	$ac\#_1$			
8	2	$ac\#_2$			
9	2	$acac\#_1$			
10	0	$c\#_1$			
11	1	$c\#_2$			
12	1	$caa\#_3$			
13	2	$cac\#_1$			
14	-1				

Figure 5.56: The LCP-array of the strings $acac\#_1$, $aac\#_2$, and $caa\#_3$ and its lcp-interval tree (singleton intervals are not shown).

string). Initially, $val_k([1..n]) = 0$ for each k with $1 \leq k \leq m$. In the top-down traversal, let $[lb..rb]$ be the current lcp-interval representing the string ω of length q and let $\ell\text{-}[b..e]$ be its parent interval. Then set

$$val_k([lb..rb]) = val_k([b..e]) + (q - \ell) \left(\log \frac{m}{df(\omega)} \right) occ_\omega(S^k)$$

Again, this takes $O(m \cdot n)$ time.

- During the top-down traversal, if the lcp-interval $[lb..rb]$ has a singleton child interval, say belonging to string S^j , then add $val_i([lb..rb])$ to $k(S^i, S^j)$ for each $i \neq j$. Since there are n singleton intervals, this computation also takes $O(m \cdot n)$ time.

As an example, consider the GESA of the strings $acac\#_1$, $aac\#_2$, and $caa\#_3$ shown in Figure 5.56. The corresponding occ , df , and val values can be found in Figure 5.57, while Figure 5.58 contains the kernel matrix. In an actual implementation, each $val_k([lb..rb])$ is stored in the array VAL_k at all ℓ -indices of $[lb..rb]$; cf. Section 5.7.2. Note that the occ arrays can be overwritten by the VAL arrays.

Theorem 5.7.4 *The algorithm correctly computes the kernel matrix for the TF-IDF weighting scheme.*

ℓ -[$lb..rb$]	0-[1..13]	1-[4..9]	1-[10..13]	2-[5..6]	2-[7..9]	2-[12..13]
ω	ε	a	c	aa	ac	ca
$occ_\omega(S^1)$	5	2	2	0	2	1
$occ_\omega(S^2)$	4	2	1	1	1	0
$occ_\omega(S^3)$	4	2	1	1	0	1
$df(\omega)$	3	3	3	2	2	2
$val_1([lb..rb])$	0	0	0	0	$2 \cdot \log_2(3/2)$	$\log_2(3/2)$
$val_2([lb..rb])$	0	0	0	$\log_2(3/2)$	$\log_2(3/2)$	0
$val_3([lb..rb])$	0	0	0	$\log_2(3/2)$	0	$\log_2(3/2)$

Figure 5.57: $occ_\omega(S^1), \dots, occ_\omega(S^m)$ and $df(\omega)$ for every string ω that is represented by an lcp-interval $[lb..rb]$, and the corresponding values $val_k([lb..rb])$. Note that $\log_2(3/2) = 0.585$.

k	1	2	3
1	—	$2 \log_2(3/2)$	$\log_2(3/2)$
2	$\log_2(3/2) + \log_2(3/2)$	—	$\log_2(3/2)$
3	$\log_2(3/2)$	$\log_2(3/2)$	—

Figure 5.58: The kernel matrix of the strings $acac\#_1$, $aac\#_2$, and $caa\#_3$ (only the non-zero summands are shown).

Proof According to Lemma 5.7.1, we have

$$k(S^i, S^j) = \sum_{p=1}^{n_j} \sum_{\omega \sqsubset S_p^j} idf(\omega) occ_\omega(S^i)$$

Since every suffix S_p^j of S^j is taken into account in the top-down traversal (when the singleton child interval corresponding to S_p^j is encountered), it suffices to show that

$$\sum_{\omega \sqsubset S_p^j} idf(\omega) occ_\omega(S^i) = val_i([lb..rb])$$

where $[lb..rb]$ is the parent interval of the singleton interval representing S_p^j . The proof of this claim is by induction on the depth d of node $[lb..rb]$ in the lcp-interval tree of the GESA of S^1, \dots, S^m . If $d = 0$, then $[lb..rb]$ is the root, i.e., $[lb..rb] = [1..n]$. In this case, the empty string ε is the longest prefix of S_p^j that is a substring of S^i . In other words, for each non-empty prefix ω of S_p^j , we have $occ_\omega(S^i) = 0$. Therefore, $\sum_{\omega \sqsubset S_p^j} idf(\omega) occ_\omega(S^i) = 0 = val_i([1..n])$.

Now suppose that node $[lb..rb]$ has depth $d + 1$. Clearly, the parent interval $[b..e]$ of $[lb..rb]$ has depth d . Let $[b..e]$ represent the string u of length ℓ and let $[lb..rb]$ represent the string uv of length q . Note that uv is a prefix of S_p^j . By the inductive hypothesis, the equality

$$\sum_{\omega \sqsubseteq u} idf(\omega) occ_{\omega}(S^i) = val_i([b..e])$$

holds true. Furthermore,

$$\begin{aligned} \sum_{\omega \sqsubseteq S_p^j} idf(\omega) occ_{\omega}(S^i) &= \sum_{\omega \sqsubseteq uv} idf(\omega) occ_{\omega}(S^i) \\ &= \sum_{\omega \sqsubseteq u} idf(\omega) occ_{\omega}(S^i) + \sum_{\omega \sqsubseteq uv, \omega \not\sqsubseteq u} idf(\omega) occ_{\omega}(S^i) \\ &= val_i([b..e]) + \sum_{\omega \sqsubseteq uv, \omega \not\sqsubseteq u} idf(\omega) occ_{\omega}(S^i) \\ &= val_i([b..e]) + \sum_{\omega \sqsubseteq uv, \omega \not\sqsubseteq u} \left(\log \frac{m}{df(uv)} \right) occ_{\omega}(S^i) \\ &= val_i([b..e]) + (q - \ell) \left(\log \frac{m}{df(uv)} \right) occ_{uv}(S^i) \\ &= val_i([lb..rb]) \end{aligned}$$

and the theorem is proven. \square

5.8 String mining

In string mining problems, one is given m databases $\mathcal{D}_1, \dots, \mathcal{D}_m$ of strings and searches for unknown strings that fulfill certain constraints, which are usually specified by the user. To study the problem, let us use an example from the medical field. Suppose a genetic disease, e.g. *Huntington's disease*, is suspected of being caused by a defect on a certain locus of a certain chromosome, say on the short arm of chromosome 4. To find the cause of the disease, a possible approach would be to sequence that segment of the DNA molecules of many healthy individuals and ill persons. Then one database contains the DNA sequences of the healthy individuals, while the second databases contains the DNA sequences of the ill individuals. Now, one searches for all strings that occur frequently (or always) in one of the database and not too often (or never) in the other database. If one finds, for example, that the string CAGCAGCAG...CAG (in which the codon CAG—coding for the amino acid glutamine—is tandemly repeated more than 36 times) occurs frequently in the database of ill persons, but not too often in the database of healthy persons, then this gives a hypothesis for the cause of the disease. This example is an instance of the frequent string mining problem.

Definition 5.8.1 The *frequent string mining problem* is defined as follows: Given m databases $\mathcal{D}_1, \dots, \mathcal{D}_m$ of strings on the alphabet Σ and m pairs of positive frequency thresholds $(\min f_1, \max f_1), \dots, (\min f_m, \max f_m)$, find all strings $\phi \in \Sigma^+$ that satisfy $\min f_i \leq df(\phi, \mathcal{D}_i) \leq \max f_i$ for all $1 \leq i \leq m$.

Recall from Section 5.6.3 that $df(\phi, \mathcal{D}_i) = |\{\omega \in \mathcal{D}_i : \phi \text{ is substring of } \omega\}|$. A string $\phi \in \Sigma^+$ satisfying $\min f_i \leq df(\phi, \mathcal{D}_i) \leq \max f_i$ is called *relevant substring* of \mathcal{D}_i . The solution to the frequent string mining problem is the intersection of the relevant substrings of the databases $\mathcal{D}_1, \dots, \mathcal{D}_m$. As an example, let \mathcal{D}_1 consist of the strings *acac*, *aac*, and *caac*, while \mathcal{D}_2 contains the strings *aaaa*, *caaac*, and *aca*. Both pairs of frequency thresholds are $(2, 2)$, i.e., $2 \leq df(\phi, \mathcal{D}_i) \leq 2$ for both $i = 1$ and $i = 2$. In other words, we are searching for all strings that occur as a substring of exactly two strings in both databases. The relevant substrings of \mathcal{D}_1 are *aa*, *aac* and *ca*; those of \mathcal{D}_2 are *aa*, *aaa*, *ac*, *c*, and *ca*. Consequently, the solution to this instance of the frequent string mining problem are the strings *aa* and *ca*.

Algorithm 5.38 gives an overview of the solution to the frequent string mining problem. The phases of the algorithm, which originates from [190], are subsequently explained in more detail. It builds on the work of Fischer et al. [109], who first presented an algorithm that solves the frequent string mining problem in optimal time, i.e., in time linear in the size of the input (the databases) and the output (the strings that satisfy the constraints). Other solutions to the problem include [79, 111, 329].

5.8.1 Extraction phase

The extraction of the relevant substrings of a database \mathcal{D} is done by a bottom-up traversal of the lcp-interval tree of the string $S^{\mathcal{D}}$. We store relevant substrings implicitly and process them later in lexicographic order.

Each relevant substring of \mathcal{D} is a prefix of at least one suffix of $S^{\mathcal{D}}$, i.e., it is possible to assign each relevant substring to exactly one suffix that has this substring as a prefix. We assign a relevant substring to the lexicographically largest suffix that has this substring as a prefix.

Lemma 5.8.2 Let $S_j^{\mathcal{D}}$ be a suffix with at least one assigned relevant substring. Let a be the minimum length and b the maximum length of all relevant substrings assigned to $S_j^{\mathcal{D}}$. Then, for each ℓ with $a \leq \ell \leq b$, there exists a relevant substring of length ℓ that is assigned to $S_j^{\mathcal{D}}$.

Proof For an indirect proof, suppose that there is an ℓ with $a \leq \ell \leq b$ so that no relevant substring of length ℓ was assigned to $S_j^{\mathcal{D}}$. The string $S^{\mathcal{D}}[j..j + \ell - 1]$ must be a relevant substring because $\min f \leq df(S^{\mathcal{D}}[j..j + b - 1], \mathcal{D}) \leq df(S^{\mathcal{D}}[j..j + \ell - 1], \mathcal{D}) \leq df(S^{\mathcal{D}}[j..j + a - 1], \mathcal{D}) \leq \max f$. If the prefix $S^{\mathcal{D}}[j..j + \ell - 1]$ of $S_j^{\mathcal{D}}$ has not been assigned to $S_j^{\mathcal{D}}$, it must have been assigned

Algorithm 5.38 An algorithm that solves the frequent string mining problem.

- For each database $\mathcal{D} \in \{\mathcal{D}_1, \dots, \mathcal{D}_m\}$ of size $k = |\mathcal{D}|$ do:
 - Preprocessing phase:
 - * Set $S^{\mathcal{D}} = S^1 \#_1 \dots \#_{k-1} S^k \#_k$, where $\mathcal{D} = \{S^1, \dots, S^k\}$.
 - * Construct the suffix array SA and the LCP-array of $S^{\mathcal{D}}$.
 - * Preprocess the LCP-array so that range minimum queries can be answered in constant time.
 - Extraction phase:
 - * Calculate the array $CT''_{\mathcal{D}}$; see Section 5.6.3.
 - * For each lcp-interval $\ell\text{-}[i..j]$ representing a string ω , compute $df(\omega, \mathcal{D}) = j - i + 1 - CT_{\mathcal{D}}(\omega)$, where $CT_{\mathcal{D}}(\omega) = CT''_{\mathcal{D}}[j] - CT''_{\mathcal{D}}[i]$ is the correction term.
 - * Store each relevant substring ϕ (i.e., $\min f \leq df(\phi, \mathcal{D}) \leq \max f$) at the lexicographically largest suffix that has ϕ as a prefix.
 - Iteratively calculate the intersection of the relevant substrings of the databases \mathcal{D}_1 and \mathcal{D}_2 , then the intersection of the result with the relevant substrings of \mathcal{D}_3 , and so on.
 - Intersection of relevant substrings of two databases \mathcal{D}_1 and \mathcal{D}_2 :
 - * Compute the matching statistics of $S^{\mathcal{D}_2}$ w.r.t. $S^{\mathcal{D}_1}$ as in Section 5.5.4. While matching the string $S^{\mathcal{D}_2}$ against the enhanced suffix array of $S^{\mathcal{D}_1}$, also compute the auxiliary array cnt , which can be used to merge the enhanced suffix arrays of $S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$; see Section 5.5.5.
 - * Process all suffixes of $S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$ in lexicographic order by *simulating* the merging of the enhanced suffix arrays of $S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$ and compute common relevant substrings.
-

to a lexicographically larger suffix. But then $S^{\mathcal{D}}[j..j+a-1]$ can be assigned to the same suffix. This contradicts the fact that $S^{\mathcal{D}}[j..j+a-1]$ has been assigned to the lexicographically largest suffix that has $S^{\mathcal{D}}[j..j+a-1]$ as a prefix. \square

In other words, for each suffix, the lengths of assigned relevant substrings form a (possibly empty) interval $[a..b]$. Let us call this interval the *results interval*. We use two arrays a and b of size $n = |S^{\mathcal{D}}|$ to store the left and right boundaries of the results intervals. Initially, all results intervals are empty: $a[k] = \infty$ and $b[k] = 0$ for $1 \leq k \leq n$. In a bottom-up traversal of the lcp-interval tree, when the procedure *process* of Algorithm 4.6 (page 94) is applied to an lcp-interval $\ell-[i..j]$ representing the string ω , it executes the following code:

```
if  $\min f \leq j - i + 1 - (CT''_{\mathcal{D}}[j] - CT''_{\mathcal{D}}[i]) \leq \max f$  then
  if  $a[j] = \infty$  then  $b[j] \leftarrow \ell$ 
   $a[j] \leftarrow \max\{\text{LCP}[i], \text{LCP}[j+1]\} + 1$ 
```

The first if-statement tests whether ω is relevant, i.e., whether $\min f \leq df(\omega, \mathcal{D}) \leq \max f$ is true. If so, the second if-statement tests whether the results interval $[a..b]$ is still empty, i.e., whether $a[j] = \infty$ holds. If this is also true, then $b[j]$ is set to ℓ . Note that j is the largest index with the property that ω is a prefix of $S^{\mathcal{D}}_{\text{SA}[j]}$ because it is the right boundary of the ω -interval. Since the algorithm traverses the lcp-interval tree in a bottom-up fashion, $b[j]$ remains unchanged once it is set (every ancestor of the ω -interval in the lcp-interval tree represents a string that is a proper prefix of ω). Furthermore, the left-boundary $a[j]$ of the results interval $[a..b]$ must be set to $k+1$, where k is the lcp-value of the parent interval of $\ell-[i..j]$. This is because the frequency constraint does not necessarily hold for the parent interval (during the bottom-up traversal, the frequency constraint will be checked separately for the parent interval). According to Corollary 4.3.10, the parent interval of $\ell-[i..j]$ has lcp-value $\max\{\text{LCP}[i], \text{LCP}[j+1]\}$.

Continuing our small example, Figures 5.59 and 5.60 show the results intervals of the databases \mathcal{D}_1 and \mathcal{D}_2 .

5.8.2 Intersection phase

To perform the intersection phase, we match the string $S^{\mathcal{D}_2}$ against the enhanced suffix array of $S^{\mathcal{D}_1}$ and compute the matching statistics of $S^{\mathcal{D}_2}$ w.r.t. $S^{\mathcal{D}_1}$ (see Figure 5.60) as well as the auxiliary array *cnt* (see Figure 5.59). With this information, it is possible to merge the enhanced suffix arrays SA_1 and SA_2 of $S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$ in linear time; cf. Section 5.5.5. However, since we are not interested in the common enhanced suffix array of SA_1 and SA_2 , we merely simulate the merging without actually building the common ESA. This simulation allows us to process all suffixes of

i	$SA_1[i]$	$LCP_1[i]$	$S_{SA_1[i]}^{D_1}$	$D_1[i]$	$CT''_{D_1}[i]$	$a_1[i]$	$b_1[i]$	$cnt[i]$
1	5	-1	# ₁	1	0	∞	0	0
2	9	0	# ₂	2	1	∞	0	0
3	14	0	# ₃	3	2	∞	0	0
4	6	0	<i>aac</i> # ₂	2	3	∞	0	9
5	11	3	<i>aac</i> # ₃	3	3	2	3	0
6	3	1	<i>ac</i> # ₁	1	5	∞	0	1
7	7	2	<i>ac</i> # ₂	2	6	∞	0	0
8	12	2	<i>ac</i> # ₃	3	6	∞	0	0
9	1	2	<i>acac</i> # ₁	1	6	∞	0	2
10	4	0	<i>c</i> # ₁	1	9	∞	0	0
11	8	1	<i>c</i> # ₂	2	10	∞	0	0
12	13	1	<i>c</i> # ₃	3	10	∞	0	0
13	10	1	<i>caac</i> # ₃	3	11	∞	0	3
14	2	2	<i>cac</i> # ₁	1	11	2	2	0
15		-1						0

Figure 5.59: Generalized enhanced suffix array of the strings *acac*, *aac*, and *caac* with the array *cnt* and results intervals for the frequency constraint $2 \leq df(\phi, \mathcal{D}_1) \leq 2$.

i	$SA_2[i]$	$LCP_2[i]$	$S_{SA_2[i]}^{D_2}$	$D_2[i]$	$CT''_{D_2}[i]$	$a_2[i]$	$b_2[i]$	$ms[SA_2[i]]$
1	5	-1	$\$_1$	1	0	∞	0	\perp
2	11	0	$\$_2$	2	1	∞	0	\perp
3	15	0	$\$_3$	3	2	∞	0	\perp
4	4	0	<i>a</i> $\$_1$	1	3	∞	0	(1,[4..9])
5	14	1	<i>a</i> $\$_3$	3	4	∞	0	(1,[4..9])
6	3	1	<i>aa</i> $\$_1$	1	5	∞	0	(2,[4..5])
7	2	2	<i>aaa</i> $\$_1$	1	6	∞	0	(2,[4..5])
8	1	3	<i>aaaa</i> $\$_1$	1	7	∞	0	(2,[4..5])
9	7	3	<i>aaac</i> $\$_2$	2	7	3	3	(2,[4..5])
10	8	2	<i>aac</i> $\$_2$	2	8	2	2	(3,[4..5])
11	9	1	<i>ac</i> $\$_2$	2	9	∞	0	(2,[6..9])
12	12	2	<i>aca</i> $\$_3$	3	9	2	2	(3,[9..9])
13	10	0	<i>c</i> $\$_2$	2	11	∞	0	(1,[10..14])
14	13	1	<i>ca</i> $\$_3$	3	12	∞	0	(2,[13..14])
15	6	2	<i>caaac</i> $\$_2$	2	12	1	2	(3,[13..13])

Figure 5.60: Generalized enhanced suffix array of the strings *aaaa*, *caaac*, and *aca* with matching statistics and results intervals for the frequency constraint $2 \leq df(\phi, \mathcal{D}_2) \leq 2$.

$S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$ in lexicographic order, and to compute all common relevant substrings.

Recall from Section 5.5.5 that $\text{PS}[p_1 - 1]$ denotes the sum $\sum_{j=1}^{p_1-1} \text{cnt}[j]$ of the first $p_1 - 1$ entries of the array *cnt* (by definition, $\text{PS}[0] = 0$), and the suffix $S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}$ is placed at index $\text{PS}[p_1 - 1] + p_1 - 1$ in the common suffix array SA of $S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$ because the suffixes $S_{\text{SA}_1[1]}^{\mathcal{D}_1}, \dots, S_{\text{SA}_1[p_1-2]}^{\mathcal{D}_1}$ and the suffixes $S_{\text{SA}_2[1]}^{\mathcal{D}_2}, \dots, S_{\text{SA}_2[\text{PS}[p_1-1]]}^{\mathcal{D}_2}$ are lexicographically smaller than $S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}$. The suffixes in the interval $[\text{PS}[p_1 - 1] + 1.. \text{PS}[p_1]]$ of SA₂ are stored at the interval $[\text{PS}[p_1 - 1] + p_1.. \text{PS}[p_1] + p_1 - 1]$ in SA, then suffix $S_{\text{SA}_1[p_1]}^{\mathcal{D}_1}$ is placed at index $\text{PS}[p_1] + p_1$, and so on. In what follows, if $ms[\text{SA}_2[p_2]] = (k, [i..j])$, then $ms[\text{SA}_2[p_2]].lcp = k$, $ms[\text{SA}_2[p_2]].lb = i$, and $ms[\text{SA}_2[p_2]].rb = j$. According to Lemma 5.5.19, if suffix $S_{\text{SA}_2[p_2]}^{\mathcal{D}_2}$ is placed between the suffixes $S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}$ and $S_{\text{SA}_1[p_1]}^{\mathcal{D}_1}$, then we have:

1. If $ms[\text{SA}_2[p_2]].lb \leq p_1 \leq ms[\text{SA}_2[p_2]].rb$, then

- a) $|\text{lcp}(S_{\text{SA}_2[p_2]}^{\mathcal{D}_2}, S_{\text{SA}_1[p_1]}^{\mathcal{D}_1})| = ms[\text{SA}_2[p_2]].lcp$
- b) $|\text{lcp}(S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}, S_{\text{SA}_2[p_2]}^{\mathcal{D}_2})| = \text{LCP}_1[p_1]$

2. Otherwise

- a) $|\text{lcp}(S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}, S_{\text{SA}_2[p_2]}^{\mathcal{D}_2})| = ms[\text{SA}_2[p_2]].lcp$
- b) $|\text{lcp}(S_{\text{SA}_2[p_2]}^{\mathcal{D}_2}, S_{\text{SA}_1[p_1]}^{\mathcal{D}_1})| = \text{LCP}_1[p_1]$

Algorithm 5.39 processes the suffixes of $S^{\mathcal{D}_1}$ and $S^{\mathcal{D}_2}$ in lexicographic order. For each relevant substring ϕ of \mathcal{D}_2 , it locates the lexicographically largest suffix of $S^{\mathcal{D}_1}$ that has ϕ as a prefix (if there is such a suffix of $S^{\mathcal{D}_1}$). Algorithm 5.39 does this by maintaining a set of relevant substrings of \mathcal{D}_2 that are a prefix of the currently processed suffix of $S^{\mathcal{D}_1}$. When the body of the outer for-loop is executed for a certain value of the loop variable p_1 , the processed suffix is $S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}$. The set consists of all relevant substrings of \mathcal{D}_2 assigned to suffixes of $S^{\mathcal{D}_2}$ that are lexicographically smaller than $S_{\text{SA}_1[p_1]}^{\mathcal{D}_1}$. It can be shown as in Lemma 5.8.2 that this set can be represented as an interval, which will be denoted by $[a_{cur}..b_{cur}]$. Exercise 5.8.4 asks you to show that the following property (P) is an invariant of the outer for-loop of Algorithm 5.39:

- (P) Before the loop body is executed for a certain value of the loop variable p_1 , we have: The length ℓ prefix of the suffix $S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}$ is a relevant substring of \mathcal{D}_2 assigned to a suffix of $S^{\mathcal{D}_2}$ that is lexicographically smaller than $S_{\text{SA}_1[p_1-1]}^{\mathcal{D}_1}$ if and only if $a_{cur} \leq \ell \leq b_{cur}$.

With the help of the loop invariant and the next lemma, we are able to show that Algorithm 5.39 correctly computes common relevant substrings.

Algorithm 5.39 Intersection of result intervals, based on the following input: suffix arrays SA_1 and SA_2 , lcp-arrays LCP_1 and LCP_2 , result-interval-arrays a_1, b_1, a_2, b_2 , matching statistics ms , and the cnt -array.

```

 $a_{cur} \leftarrow \infty$ 
 $b_{cur} \leftarrow 0$ 
 $a_{next} \leftarrow \infty$ 
 $b_{next} \leftarrow 0$ 
 $p_2 \leftarrow 1$ 
for  $p_1 \leftarrow k_1 + 1$  to  $n_1 + 1$  do           /*  $\mathcal{D}_1$  contains  $k_1$  strings */
    if  $LCP_1[p_1] \geq a_{cur}$  then
         $b_{next} \leftarrow \min\{b_{cur}, LCP_1[p_1]\}$ 
    else
         $a_{next} \leftarrow \infty$ 
         $b_{next} \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $cnt[p_1]$  do
        if  $a_2[p_2] \leq b_2[p_2]$  then           /* results interval is not empty */
            if  $ms[SA_2[p_2]].lb \leq p_1 \leq ms[SA_2[p_2]].rb$  then
                 $lcp_1 \leftarrow LCP_1[p_1]$ 
                 $lcp_2 \leftarrow ms[SA_2[p_2]].lcp$ 
            else
                 $lcp_1 \leftarrow ms[SA_2[p_2]].lcp$ 
                 $lcp_2 \leftarrow LCP_1[p_1]$ 
            /* now  $lcp_1 = |\text{lcp}(S_{SA_1[p_1-1]}^{\mathcal{D}_1}, S_{SA_2[p_2]}^{\mathcal{D}_2})|$  and  $lcp_2 = |\text{lcp}(S_{SA_2[p_2]}^{\mathcal{D}_2}, S_{SA_1[p_1]}^{\mathcal{D}_1})|$  */
            if  $lcp_1 \geq a_2[p_2]$  then
                 $a_{cur} \leftarrow \min\{a_{cur}, a_2[p_2]\}$ 
                 $b_{cur} \leftarrow \max\{b_{cur}, \min\{lcp_1, b_2[p_2]\}\}$ 
            if  $lcp_2 \geq a_2[p_2]$  then
                 $a_{next} \leftarrow \min\{a_{next}, a_2[p_2]\}$ 
                 $b_{next} \leftarrow \max\{b_{next}, \min\{lcp_2, b_2[p_2]\}\}$ 
         $p_2 \leftarrow p_2 + 1$ 
    end for
     $a_1[p_1 - 1] \leftarrow \max\{a_{cur}, a_1[p_1 - 1]\}$ 
     $b_1[p_1 - 1] \leftarrow \min\{b_{cur}, b_1[p_1 - 1]\}$ 
     $a_{cur} \leftarrow a_{next}$ 
     $b_{cur} \leftarrow b_{next}$ 
end for

```

		a_{cur}	b_{cur}
$p_1 - 1$	$S_{SA_1[p_1-1]}^{\mathcal{D}_1}$	$a_1[p_1 - 1]$	$b_1[p_1 - 1]$
	$S_{SA_2[p_2]}^{\mathcal{D}_2}$	$a_2[p_2]$	$b_2[p_2]$
	\dots	\dots	\dots
p_1	$S_{SA_1[p_1]}^{\mathcal{D}_1}$	$a_1[p_1]$	$b_1[p_1]$

Figure 5.61: To compute common relevant substrings of \mathcal{D}_1 and \mathcal{D}_2 that must be stored at index $p_1 - 1$, one must consider relevant substrings of \mathcal{D}_2 represented by the interval $[a_{cur}..b_{cur}]$ and those that are assigned to suffixes of $S^{\mathcal{D}_2}$ that are lexicographically in between $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ and $S_{SA_1[p_1]}^{\mathcal{D}_1}$.

Lemma 5.8.3 *Let ϕ be a relevant substring of \mathcal{D}_2 and let $S_{SA_2[p_2]}^{\mathcal{D}_2}$ be the lexicographically largest suffix in SA_2 that has ϕ as a prefix, i.e., ϕ is stored at index p_2 . Furthermore, suppose that in a (simulated) merging of SA_1 and SA_2 , the suffix $S_{SA_2[p_2]}^{\mathcal{D}_2}$ is placed in between $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ and $S_{SA_1[p_1]}^{\mathcal{D}_1}$, i.e., $S_{SA_1[p_1-1]}^{\mathcal{D}_1} < S_{SA_2[p_2]}^{\mathcal{D}_2} < S_{SA_1[p_1]}^{\mathcal{D}_1}$. If ϕ is also a relevant substring of \mathcal{D}_1 , then it is stored at an index $\geq p_1 - 1$.*

Proof For an indirect proof, suppose that ϕ is stored at an index $k < p_1 - 1$, i.e., $S_{SA_1[k]}^{\mathcal{D}_1}$ is the lexicographically largest suffix in SA_1 that has ϕ as a prefix. Note that ϕ is a common prefix of $S_{SA_1[k]}^{\mathcal{D}_1}$ and $S_{SA_2[p_2]}^{\mathcal{D}_2}$. Because $S_{SA_1[k]}^{\mathcal{D}_1} < S_{SA_1[p_1-1]}^{\mathcal{D}_1} < S_{SA_2[p_2]}^{\mathcal{D}_2}$, it follows that ϕ is also a prefix of $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$. This, however, contradicts the fact that $S_{SA_1[k]}^{\mathcal{D}_1}$ is the lexicographically largest suffix in SA_1 that has ϕ as a prefix. \square

In order to compute common relevant substrings of \mathcal{D}_1 and \mathcal{D}_2 that must be stored at index $p_1 - 1$, it suffices to consider—apart from the relevant substrings of \mathcal{D}_2 represented by the interval $[a_{cur}..b_{cur}]$ —relevant substrings of \mathcal{D}_2 assigned to suffixes of $S^{\mathcal{D}_2}$ that are lexicographically in between $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ and $S_{SA_1[p_1]}^{\mathcal{D}_1}$; see Figure 5.61. This is because the others will be stored at an index $\geq p_1$ by Lemma 5.8.3.

To show the correctness of Algorithm 5.39, we prove the following claim: Upon termination of the inner for-loop, the length ℓ prefix of the suffix $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ is a relevant substring of \mathcal{D}_2 assigned to a suffix of $S^{\mathcal{D}_2}$ that is lexicographically smaller than $S_{SA_1[p_1]}^{\mathcal{D}_1}$ if and only if $a_{cur} \leq \ell \leq b_{cur}$. It then follows that the common relevant substrings of \mathcal{D}_1 and \mathcal{D}_2 that must be stored at index $p_1 - 1$ are represented by the intersection of the intervals $[a_{cur}..b_{cur}]$ and $[a_1[p_1 - 1]..b_1[p_1 - 1]]$. That is, the left boundary of the common results interval is $\max\{a_{cur}, a_1[p_1 - 1]\}$ and the right boundary is $\min\{b_{cur}, b_1[p_1 - 1]\}$.

By property (P), the claim is true for all suffixes of $S^{\mathcal{D}_2}$ that are lexicographically smaller than $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$. In addition to those, it suffices to consider the suffixes of $S^{\mathcal{D}_2}$ that are lexicographically in between $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ and $S_{SA_1[p_1]}^{\mathcal{D}_1}$. This is done in the inner for-loop of Algorithm 5.39. Whenever the results interval stored at the current index p_2 is non-empty, the algorithm computes $lcp_1 = |\text{lcp}(S_{SA_1[p_1-1]}^{\mathcal{D}_1}, S_{SA_2[p_2]}^{\mathcal{D}_2})|$ and tests whether $lcp_1 \geq a_2[p_2]$. If so, then the length ℓ prefix of the suffix $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ is a relevant substring of \mathcal{D}_2 assigned to the suffix $S_{SA_2[p_2]}^{\mathcal{D}_2}$ if and only if $a[p_2] \leq \ell \leq \min\{lcp_1, b[p_2]\}$. Consequently, the interval $[a_{cur}..b_{cur}]$ is widened by the assignments $a_{cur} \leftarrow \min\{a_{cur}, a_2[p_2]\}$ and $b_{cur} \leftarrow \max\{b_{cur}, \min\{lcp_1, b_2[p_2]\}\}$. Therefore, upon termination of the inner for-loop, the length ℓ prefix of the suffix $S_{SA_1[p_1-1]}^{\mathcal{D}_1}$ is a relevant substring of \mathcal{D}_2 assigned to a suffix of $S^{\mathcal{D}_2}$ that is lexicographically smaller than $S_{SA_1[p_1]}^{\mathcal{D}_1}$ if and only if $a_{cur} \leq \ell \leq b_{cur}$. In other words, the claim and thus the correctness of Algorithm 5.39 is proven.

Continuing the example from Figures 5.59 and 5.60, let us consider the situation before the body of the outer for-loop in Algorithm 5.39 is executed for $p_1 = 13$. At that point in time, we have $a_{cur} = \infty$, $b_{cur} = 0$, $a_{next} = \infty$, $b_{next} = 0$, and $p_2 = 13$. Then, a_{next} is set to ∞ and b_{next} is set to 0 because $\text{LCP}_1[p_1] = 1 \not\geq \infty = a_{cur}$. In the inner for-loop, only $p_2 = 15$ satisfies $a_2[p_2] = 1 \leq 2 = b_2[p_2]$. In this case, $ms[SA_2[p_2]].lb = 13 \leq p_1 \leq 13 = ms[SA_2[p_2]].rb$. So lcp_1 is set to $\text{LCP}_1[p_1] = 1$ and lcp_2 is set to $ms[SA_2[p_2]].lcp = 3$. When the inner for-loop terminates, we have $a_{cur} = 1$, $b_{cur} = 1$, $a_{next} = 1$, and $b_{next} = 2$ because $lcp_1 \geq a_2[p_2]$ and $lcp_2 \geq a_2[p_2]$. At the end of the body of the outer for-loop, $a_1[p_1 - 1] = a_1[12]$ is set to $\max\{a_{cur}, a_1[p_1 - 1]\} = \infty$, $b_1[p_1 - 1] = b_1[12]$ is set to $\min\{b_{cur}, b_1[p_1 - 1]\} = 0$, and both variables a_{cur} and b_{cur} are updated. Before the body of the outer for-loop is executed for $p_1 = 14$, we have $a_{cur} = a_{next} = 1$, $b_{cur} = b_{next} = 2$, and $p_2 = 16$. Then, b_{next} is set to $\min\{b_{cur}, \text{LCP}_1[p_1]\} = 2$ because $\text{LCP}_1[p_1] = 2 \geq 1 = a_{cur}$. Since $cnt[p_1] = 0$, the body of the inner for-loop is not executed. Again, $a_1[p_1 - 1]$ is set to ∞ , $b_1[p_1 - 1]$ is set to 0, and both variables a_{cur} and b_{cur} are updated. Before the body of the outer for-loop is executed for $p_1 = 15$, we have $a_{cur} = a_{next} = 1$, $b_{cur} = b_{next} = 2$, and $p_2 = 16$. Then, a_{next} is set to ∞ and b_{next} is set to 0 because $\text{LCP}_1[p_1] = -1$. Again, the body of the inner for-loop is not executed at all. Finally, $a_1[p_1 - 1] = a_1[14]$ is set to $\max\{a_{cur}, a_1[p_1 - 1]\} = 2$, and $b_1[p_1 - 1] = b_1[14]$ is set to $\min\{b_{cur}, b_1[p_1 - 1]\} = 2$. This means that ca , the length 2 prefix of $S_{SA_1[14]}^{\mathcal{D}_1}$ is $cac\#_1$, is a common relevant substring of \mathcal{D}_1 and \mathcal{D}_2 .

How much time does it take to intersect the results intervals of all databases $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_m$? We successively intersect \mathcal{D}_2 with \mathcal{D}_1 , then \mathcal{D}_3 with \mathcal{D}_1 , and so on until we have intersected \mathcal{D}_m with \mathcal{D}_1 . Since intersecting the results intervals of database \mathcal{D}_i with the results intervals of the database \mathcal{D}_1 takes $\mathcal{O}(n_1 + n_i)$ time, the overall time complexity is $\mathcal{O}((m-1) \cdot n_1 + \sum_{i=2}^m n_i)$. Without loss of generality, we may assume that

\mathcal{D}_1 is the smallest database (i.e., n_1 is smaller than or equal to n_i for all $1 < i \leq m$). It follows that $(m-1) \cdot n_1 + \sum_{i=2}^m n_i \leq 2 \cdot \sum_{i=1}^m n_i$, so the overall time complexity to intersect all databases is linear in the total size of all databases.

We conclude this section with two important remarks. Since a database contains many strings, the use of pairwise different separator symbols blows up the alphabet. For this reason, it is valuable to know that just one separator symbol suffices; see [190] for details. Moreover, the restriction that all frequency thresholds $\min f_i$, $1 \leq i \leq m$, must be positive is unnecessarily severe. In fact, it is enough to demand that at least one of these values is positive—the others may be 0; again see [190] for details.

Exercise 5.8.4 Prove that property (P) is really an invariant of the outer for-loop of Algorithm 5.39.

Making the Components of Enhanced Suffix Arrays Smaller

Up to this point, we did not trouble ourselves much about memory requirements. However, in some applications of enhanced suffix arrays we needed more than just one or two arrays. In large scale applications, the total memory usage of these arrays may be prohibitive. Fortunately, the various components of enhanced suffix arrays can be made smaller. We shall learn in this chapter how this can be done for the suffix array and the LCP-array. Moreover, we shall see that essentially one small data structure suffices to support the following tasks:

- find the parent interval of an lcp-interval,
- find all child intervals of an lcp-interval,
- find the suffix link interval of an lcp-interval,
- answer a range minimum query.

These results rely on the fact that *rank* and *select* queries can be answered in constant time. From now on, we state the alphabet size explicitly in complexity analyses.

6.1 Constant time *rank* and *select* queries

Virtually all methods to compress the components of enhanced suffix arrays take advantage of the following theorem.

Theorem 6.1.1 *A bit vector $B[1..n]$ can be preprocessed in linear time so that the following operations are supported in constant time:*

- $\text{rank}_b(B, i)$: *returns the number of occurrences of bit b in $B[1..i]$.*

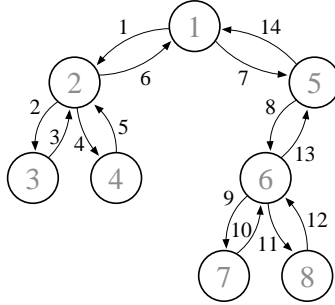


Figure 6.1: A preorder traversal of the tree follows the consecutively numbered arrows. Each node is labeled with its preorder index i , i.e., node i is the i -th node visited in the preorder traversal.

- $select_b(B, i)$: returns the position of the i -th occurrences of bit b in $B[1..n]$.

The bit vector $B[1..n]$ uses n bits and the supporting data structures use only $o(n)$ bits.

For the bit vector $B = 1110100111010000$, we have for example $rank_1(B, 6) = 4$ and $select_0(B, 4) = 11$. If it is clear from the context which bit vector B is meant, we write $rank_b(i)$ and $select_b(i)$ instead of $rank_b(B, i)$ and $select_b(B, i)$.

It is beyond the scope of this book to provide a proof of the theorem. The reader is referred to the seminal work of Jacobson [162, 163] who showed that attaching a dictionary of size $o(n)$ bits to the bit vector $B[1..n]$ is enough to answer *rank* queries in constant time. His solution for the *select* operation was not optimal, but later Clark [58] proved that $o(n)$ bits are enough to answer *select* queries in constant time; see also [228].

Jacobson [163] was interested in the *rank* and *select* operations because they allowed him to simulate tree traversals on bit vectors. There are several possibilities to represent a (static) tree by a bit vector. Here we will review just one of them [230]: the tree T is represented by a balanced sequence of parentheses that is obtained by a preorder (depth-first) traversal of T ; cf. Figure 6.1. To be precise, start with the root of T and do the following:

1. Write an opening parenthesis.
2. For each child node (from left to right), write its balanced parentheses sequence.
3. Write a closing parenthesis.

((()	())	((()	())))
1	2	3	3	4	4	2	5	6	7	7	8	8	6	5	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 6.2: The balanced parentheses sequence BPS_{pre} of the tree from Figure 6.1. The lower row of numbers shows the positions of the parentheses in the sequence. The row below the balanced parentheses sequence is only for illustrative purposes: the opening parentheses are numbered consecutively and a closing parenthesis has the number of its matching opening parenthesis.

The resulting balanced sequence of parentheses is denoted by BPS_{pre} . The i -th opening parenthesis in BPS_{pre} represents the node with preorder index i , which is defined as follows.

Definition 6.1.2 Let v be a node in a rooted tree T . If v is the i -th node visited in a preorder traversal of T , then we say that the preorder index of v is i .

We exemplify the procedure by the example of Figure 6.1. The preorder traversal of the tree starts at the root (node 1) and writes an opening parenthesis. It visits its left child (node 2), writes an opening parenthesis, visits node 3 (the left child of node 2), and again writes an opening parenthesis. Because node 3 does not have children it then writes a closing parenthesis, etc. The resulting nested balanced sequence of parentheses BPS_{pre} is shown in Figure 6.2.

Simulations of tree traversals that are based on this representation make use of the following theorem.

Theorem 6.1.3 Given a balanced sequence of n opening and n closing parentheses (requiring $2n$ bits), the following operations can be supported in constant time with only $o(n)$ bits of extra space:

- $\text{rank}_{(i)}$: returns the number of opening parentheses up to and including position i . The operation $\text{rank}_{)}(i)$ is defined analogously.
- $\text{select}_{(i)}$: returns the position of the i -th opening parenthesis. The operation $\text{select}_{)}(i)$ is defined analogously.
- $\text{findclose}(i)$: returns the position of the closing parenthesis matching the opening parenthesis at position i . The operation $\text{findopen}(i)$ is defined analogously.

- *enclose*(i): given a parenthesis pair whose opening parenthesis is at position i , it returns the position of the opening parenthesis corresponding to the closest matching parenthesis pair enclosing i .
- *double-enclose*(i, j): given two opening parentheses at positions i and j with $\text{findclose}(i) < j$, the operation *double-enclose*(i, j) returns the position k of an opening parenthesis so that the parenthesis pair $(k, \text{findclose}(k))$ most tightly encloses the parenthesis pairs $(i, \text{findclose}(i))$ and $(j, \text{findclose}(j))$.
- *rr-enclose*(i, j) (an acronym for range-restricted-enclose): given two opening parentheses at positions i and j so that $\text{findclose}(i) < j$, the operation *rr-enclose*(i, j) returns the smallest position, say k , of an opening parenthesis so that $\text{findclose}(i) < k < j$ and $\text{findclose}(j) < \text{findclose}(k)$.

We are not going to prove the theorem, but instead refer to the original literature. The *rank* and *select* operations have already been dealt with (representing an opening parenthesis by 1 and a closing parenthesis by 0 yields a bit vector). Jacobson [163] also studied the operations *findclose* and *findopen*, but it was Munro and Raman [229, 230] who provided the first constant time and $o(n)$ extra space solution for *findclose*, *findopen*, and *enclose*. Later, Geary et al. [121] gave a simpler solution for these operations. Based on [121], Gog and Fischer [125] showed how the *double-enclose* and *rr-enclose* operations can be implemented. (The paper [229] also contained a solution for the *double-enclose* operation, but it was dropped in the journal version [230] because it was erroneous). Sadakane and Navarro [275] proposed a simple and flexible data structure, called the range min-max tree, that supports all operations in constant time with only $o(n)$ bits of extra space.

Given the balanced parentheses sequence BPS_{pre} of the tree T and the position pos of an opening parenthesis in BPS_{pre} , the preorder index i of the node represented by that parenthesis is

$$i = \text{rank}_l(pos)$$

Conversely, given the preorder index i of node v , the opening parenthesis that represents v is the i -th opening parenthesis in BPS_{pre} . In what follows, $_i($ denotes the i -th opening parenthesis and $)_i$ denotes its matching closing parenthesis in BPS_{pre} . It is easy to see that $_i($ can be found at position

$$ipos = \text{select}_l(i)$$

and $)_i$ can be found at position

$$cipos = \text{findclose}(ipos)$$

For example, node 2—the node with preorder index 2—in the tree of Figure 6.1 is represented in the BPS_{pre} of Figure 6.2 by the opening parenthesis at position $\text{select}_l(2) = 2$. Its matching closing parenthesis can be found at position $\text{findclose}(2) = 7$.

One can find the parent, leftmost child, next sibling, and the subtree size of v in constant time on the BPS_{pre} as follows:

- The parent node of v is given by the closest parenthesis pair $_k(\dots)_k$ enclosing the pair $_i(\dots)_i$. Thus, if v is not the root, then $\text{enclose}(\text{ipos})$ returns the position of $_k($. In our example, $\text{enclose}(2) = 1$ yields the position of the opening parenthesis that represents the parent of node 2 in BPS_{pre} . The parent node has preorder index $\text{rank}_l(1) = 1$.
- If the parenthesis next to $_i($ (at position $\text{ipos} + 1$) is a closing parenthesis, then it must be $)_i$ because BPS_{pre} is balanced. Hence v has no children because it is a leaf. Otherwise, if the parenthesis next to $_i($ is an opening parenthesis, then it must be $_{i+1}($ and the pair $_{i+1}(\dots)_{i+1}$ represents the leftmost child of v . In our example, at position $\text{ipos} + 1 = 3$ in BPS_{pre} there is an opening parenthesis, so the leftmost child of node 2 has preorder index $\text{rank}_l(3) = 3$.
- If the parenthesis next to $)_i$ (at position $\text{cpos} + 1$) is a closing parenthesis, then v has no sibling. Otherwise, if the parenthesis next to $)_i$ is an opening parenthesis, say $_k($, then the pair $_k(\dots)_k$ represents the next sibling of v . In our example, at position $\text{cpos} + 1 = 8$ in BPS_{pre} there is an opening parenthesis, so the next sibling of node 2 has preorder index $\text{rank}_l(8) = 5$.
- The size of the subtree of T rooted at node v can be determined by $\frac{\text{cpos} - \text{ipos} + 1}{2}$. In our example, the size of the subtree rooted at node 2 is $\frac{7 - 2 + 1}{2} = 3$.

Furthermore, given the preorder index j of another node w , the lowest common ancestor $u = \text{LCA}(v, w)$ of v and w can be computed in constant time as follows. Without loss of generality, suppose that $\text{ipos} < \text{jpos} = \text{select}_l(j)$. If the pair $_i(\dots)_i$ encloses the pair $_j(\dots)_j$, then $u = v$. Otherwise, if $\text{cpos} < \text{jpos}$, then the position of the opening parenthesis representing u is $\text{pos} = \text{double-enclose}(\text{ipos}, \text{jpos})$. That is, the preorder index of $u = \text{LCA}(v, w)$ is $\text{rank}_l(\text{pos})$. In our example, the lowest common ancestor of the nodes 2 and 6 is represented by the opening parenthesis at position $\text{double-enclose}(2, 9) = 1$; its preorder index is $\text{rank}_l(1) = 1$.

The proof of these facts is left to the reader because we will not use the balanced parentheses sequence BPS_{pre} in this book (instead, we will use a different balanced parentheses sequence).

The number of different binary trees with n nodes is $C_n = \frac{1}{n+1} \binom{2n}{n}$; cf. Exercise 3.4.7. For large n this is about 2^{2n} (use Stirling's approxima-

tion as in Section 3.3.1), so asymptotically $\log_2 C_n \approx 2n$ bits are necessary to encode a binary tree. Clearly, the same is true for non-binary trees. Consequently, the $2n + o(n)$ bit representation of a tree sketched above uses only sublinear space on top of the information-theoretic lower bound, while at the same time supporting navigation on the tree in constant time. A data structure that uses an amount of space that is so close to the information-theoretic lower bound, but still allows for efficient operations, is called *succinct data structure*.

6.2 Compressed suffix and LCP-arrays

6.2.1 Compressed suffix array

In large scale applications, the space consumption of the index structure is often a bottleneck. Thus, scientists started to investigate how index structures like the suffix array can be compressed. One line of research [135, 270] uses the Ψ -function for this purpose as it tends to be easier to compress than the suffix array itself. This is because the ψ -values in every c -interval, $c \in \Sigma$, form an increasing integer sequence; see Lemma 6.2.1. Therefore, the ψ -array consists of σ increasing integer sequences.

Lemma 6.2.1 *If $i < j$ and $S[\text{SA}[i]] = S[\text{SA}[j]]$, then $\psi[i] < \psi[j]$.*

Proof The suffixes of S appear in lexicographic order in the suffix array, so $i < j$ if and only if $S_{\text{SA}[i]} < S_{\text{SA}[j]}$. With $S[\text{SA}[i]] = c = S[\text{SA}[j]]$, we have $S_{\text{SA}[i]} = cS_{\text{SA}[i]+1}$ and $S_{\text{SA}[j]} = cS_{\text{SA}[j]+1}$.¹ If we drop the first character c , we get the suffixes $S_{\text{SA}[i]+1}$ and $S_{\text{SA}[j]+1}$. Clearly, these suffixes appear in the same lexicographic order as $S_{\text{SA}[i]}$ and $S_{\text{SA}[j]}$. Hence $S_{\text{SA}[i]+1} < S_{\text{SA}[j]+1}$. We conclude that $S_{\text{SA}[\psi[i]]} < S_{\text{SA}[\psi[j]]}$ because $\text{SA}[\psi[i]] = \text{SA}[i] + 1$ by Definition 5.5.4. Thus, $\psi[i] < \psi[j]$. \square

Increasing integer sequences can be compressed by so-called *gap encoding* methods. For instance, Sadakane [270, 272] used Elias δ coding [88] to encode ψ differentially (i.e., $\psi[i+1] - \psi[i]$ is encoded) within the areas where it is increasing. It is beyond the scope of this book to discuss the different approaches; the reader is referred to the overview article [238] instead. Here, we assume that the ψ -function is available in a compressed form so that its values can be accessed in constant time. The main purpose of an index is to allow efficient exact string matching, and we will now show that constant time access to ψ is enough to answer *decision queries* (“Is P a substring of S ?”) and *counting queries* (“How often does P occur in S ?”) in $O(m \log n)$ by the same binary search as in Algorithm

¹To cope with boundary cases, we tacitly assume that S is terminated by $\$$.

Algorithm 6.1 Store every s -th suffix of the string S in the array SSA.

```

 $j \leftarrow 1$ 
for  $i \leftarrow 1$  to  $n$  do
    if  $(SA[i] \bmod s) = 0$  then
         $B_s[i] \leftarrow 1$ 
         $SSA[j] \leftarrow SA[i]$ 
         $j \leftarrow j + 1$ 

```

5.4 (page 121). The key observation is that one must merely be able to lexicographically compare the pattern P with the length m prefix of the suffixes of S at certain indices. In other words, given index i , it suffices to retrieve $S[SA[i]..SA[i] + m - 1]$ from the compressed ψ -function. This can be done with the help of the C -array (if we consider all characters in Σ that are smaller than c , then $C[c]$ is the overall number of their occurrences in S).

Because the first characters of all suffixes in the suffix array are in increasing order, the first character $S[SA[i]]$ of $S[SA[i]..SA[i] + m - 1]$ must be the character c satisfying $C[c] < i \leq C[c + 1]$. Clearly, c can be found in $O(\log \sigma)$ time by a binary search on C , but it is also possible to determine c in constant time as follows. Let $B_C[1..n]$ be a bit vector so that $B_C[k] = 1$ if and only if $k = C[a] + 1$ for a character $a \in \Sigma$. In other words, the ones in the bit vector B_C mark indices at which the lexicographically ordered suffixes of S change their first character. If B_C is enhanced with an $o(n)$ space data structure that supports constant time *rank* queries, then $c = \Sigma[rank_1(B_C, i)]$ can be found in constant time.² The second character of $S[SA[i]..SA[i] + m - 1]$ is $S[SA[i] + 1] = S[SA[\psi(i)]] = \Sigma[rank_1(B_C, \psi(i))]$ because $SA[\psi(i)] = SA[i] + 1$, the third character is $\Sigma[rank_1(B_C, \psi(\psi(i)))]$, and so on. To sum up, $S[SA[i]..SA[i] + m - 1]$ can be determined in $O(m)$ time from the compressed ψ -function and hence a binary search delivers the P -interval in SA in $O(m \log n)$ time.

However, to answer *enumeration queries* (“Where are all z occurrences of P in S ?”) we need to retrieve occurrence positions. Again, there are several ways to do this [238], and we present only the simplest solution. This method uses a sampling parameter s and stores every s -th position (suffix) of the string S in the array SSA. The array SSA is called *sparse suffix array*; see e.g. [176, 179, 322]. To be able to decide whether a suffix is sampled or not, we use a bit vector $B_s[1..n]$ with $B_s[i] = 1$ if and only if the suffix at index i is sampled, or equivalently, if and only if $SA[i] \bmod s = 0$; see Algorithm 6.1.

²Here, we assume that all characters of the alphabet Σ occur in S . If this is not the case, then $c = \Sigma'[rank_1(B_C, i)]$, where Σ' denotes the ordered subalphabet of Σ that contains exactly the characters of S .

Algorithm 6.2 Find $SA[i]$ in the array SSA.

```

 $d \leftarrow 0$ 
while  $B_s[i] = 0$  do
     $i \leftarrow \psi(i)$ 
     $d \leftarrow d + 1$ 
 $k \leftarrow SSA[rank_1(B_s, i)] - d$ 
if  $SSA[rank_1(B_s, i)] > d$  then
    return  $k$ 
else
    return  $n + k$ 

```

Algorithm 6.2 shows how the value $SA[i]$ can be retrieved from the array SSA and the bit vector B_s . Given an index i , if $B_s[i] = 1$, then $SA[i]$ is sampled and it can be found at index $rank_1(B_s, i)$ in the array SSA. Otherwise, we compute $\psi(i), \psi^2(i), \dots, \psi^d(i)$ until a d with $B_s[\psi^d(i)] = 1$ is found. This means that the suffix $SA[\psi^d(i)]$ is sampled, and $SA[\psi^d(i)] = SSA[rank_1(B_s, \psi^d(i))]$. If $SA[\psi^d(i)] > d$, then $SA[i] = SA[\psi^d(i)] - d$ because $SA[\psi(i)] = SA[i] + 1, \dots, SA[\psi^d(i)] = SA[i] + d$. Otherwise, if $SA[\psi^d(i)] \leq d$, then $SA[i] = n + SA[\psi^d(i)] - d$.

Exercise 6.2.2 Show that Algorithm 6.2 has a worst-case time complexity of $O(s)$.

Exercise 6.2.3 The sparse suffix array stores every s -th *position* of the text. Give an algorithm in pseudo-code that takes every s -th *entry* of the suffix array SA as sample, and an algorithm that retrieves $SA[i]$ from this sampled suffix array. What advantages and disadvantages does this approach have, compared to the previous method?

6.2.2 Compressed LCP-array

We recall from Corollary 4.2.5 that $PLCP[i-1] - 1 \leq PLCP[i]$ for $2 \leq i \leq n$. If we add i on both sides of the inequality, we get

$$(i-1) + PLCP[i-1] \leq i + PLCP[i]$$

In other words, the sequence $1 + PLCP[1], 2 + PLCP[2], \dots, n + PLCP[n]$ is increasing. We define $PLCP[0] = 0$ and add the element $0 = 0 + PLCP[0]$ at the beginning of the sequence. Furthermore, we define a function d that computes the difference between two consecutive elements of the extended sequence. To be precise,

$$d(i) = i + PLCP[i] - (i-1 + PLCP[i-1]) = PLCP[i] - PLCP[i-1] + 1$$

Algorithm 6.3 Retrieve LCP[j] from SA and the encoded bit vector B_d .

```

 $i \leftarrow \text{SA}[j]$       /* now retrieve PLCP[ $i$ ] from  $B_d$  */
 $ipos \leftarrow \text{select}_1(B_d, i)$ 
return  $ipos - 2i$ 

```

for $1 \leq i \leq n$. Now we use a unary code to encode each value $d(i)$ in the increasing sequence of natural numbers $d(1), d(2), \dots, d(n)$. That is, $d(i)$ is encoded by $d(i)$ zeros followed by a one. For example, 0 is encoded as 1, 1 is encoded as 01, 2 is encoded as 001, and so on. The encoded sequence B_d consists of n ones and at most $n - 1$ zeros because $\text{PLCP}[k] \leq n - 1$ for all k with $1 \leq k \leq n$. It has the following crucial property: before the position $ipos$ of the i -th one in B_d , there are

$$z(i) = \sum_{k=1}^i d(k) = \sum_{k=1}^i (\text{PLCP}[k] - \text{PLCP}[k-1] + 1) = \text{PLCP}[i] + i$$

many zeros. We can retrieve $\text{PLCP}[i] = z(i) - i$ by the equation $\text{PLCP}[i] = ipos - 2i$ because the number of zeros $z(i)$ before the position $ipos$ is $z(i) = ipos - i$. Clearly, the position $ipos$ of the i -th one in B_d can be found in constant time, provided that we use a data structure that supports select_1 queries on the encoded sequence in constant time.

So the good news is that at most $2n - 1$ bits are required to encode the PLCP-array, plus $o(n)$ bits to retrieve a value $\text{PLCP}[i]$ in constant time. The bad news is that we usually do not need a value $\text{PLCP}[i] = \text{LCP}[\text{ISA}[i]]$ but a value $\text{LCP}[j]$. In order to retrieve $\text{LCP}[j]$ from the encoded sequence, we must first determine $i = \text{SA}[j]$ and then retrieve $\text{PLCP}[i]$ because $\text{PLCP}[i] = \text{LCP}[\text{ISA}[i]] = \text{LCP}[\text{ISA}[\text{SA}[j]]] = \text{LCP}[j]$; see Algorithm 6.3. This is a disadvantage because the retrieval of $\text{SA}[j]$ is relatively slow on a compressed suffix array. Moreover, the compressed representation does not benefit from sequential access to the LCP-array because lcp-values are stored in text order and not in suffix array order. Sadakane [271], the inventor of the compression technique, has already stated the disadvantages.

6.3 The balanced parentheses sequence of the LCP-array

This section introduces a balanced parentheses sequence of the LCP-array, which originates from [247, 248]. Although it can be constructed without the Super-Cartesian tree (Definition 4.3.23) of the LCP-array, it is instructive to introduce it with the help of this tree. The reader should

i	SA	LCP	$S_{SA[i]}$
1	3	-1	<i>aaacatat</i>
2	4	2	<i>aacatat</i>
3	1	1	<i>acaaacatat</i>
4	5	3	<i>acatat</i>
5	9	1	<i>at</i>
6	7	2	<i>atat</i>
7	2	0	<i>caaacatat</i>
8	6	2	<i>catat</i>
9	10	0	<i>t</i>
10	8	1	<i>tat</i>
11		-1	

Figure 6.3: The LCP-array of the string $S = acaaacatat$.

keep in mind that a node in a Super-Cartesian tree has either a right sibling or a right child but not both. As an example, consider the LCP-array in Figure 6.3 and its Super-Cartesian tree in Figure 6.4.

The Super-Cartesian tree of an array can be represented by a balanced parentheses sequence BPS. To obtain the BPS do the following:

1. Write the balanced parentheses sequence of the left child.
2. Write an opening parenthesis.
3. Write the balanced parentheses sequence of the right child/sibling.
4. Write a closing parenthesis.

Figure 6.5 shows an example. However, two different Super-Cartesian trees may have the same sequence of balanced parentheses. (For instance, consider the LCP-arrays of the strings *acg* and *acc*.) This is because the cases “right child” and “right sibling” are treated in the same fashion. To compensate for the lack of information as to whether there is a right child or a right sibling, we enhance the BPS with a bitstring B . This bitstring B is obtained by replacing step (4) of the procedure above with

- 4'. Write a closing parenthesis. If the node under consideration is a right sibling, append 0 to B ; otherwise append 1 to B .

Again, see Figure 6.5 for an example. The Super-Cartesian tree is only conceptual. To be precise, the BPS with the additional bitstring B , which we call *enhanced* BPS, can be obtained solely based on the LCP-array; see

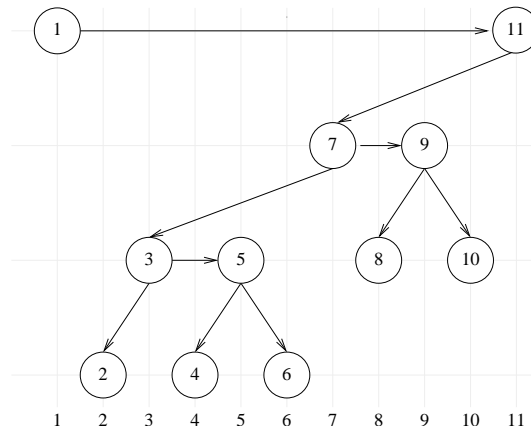


Figure 6.4: The Super-Cartesian tree of the LCP-array from Figure 6.3.

		1		1			1	0	1			1			1	0	1		0	1	
(()	(()	(()))	(()	(()))	())
1	2	2	3	4	4	5	6	6	5	3	7	8	8	9	10	10	9	7	11	11	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Figure 6.5: Enhanced BPS of the Super-Cartesian tree of Figure 6.4. The lower row of numbers shows the positions of the parentheses in the sequence. The row below the BPS is only for illustrative purposes: the opening parentheses are numbered consecutively and a closing parenthesis has the number of its matching opening parenthesis. The row above the BPS shows the bitstring B .

Algorithm 6.4 Construction of the enhanced BPS based on the LCP-array.

```

push(1)          /* LCP[1] = -1 */
B ← ε           /* bitstring B is initially empty */
write '('
for i ← 2 to n + 1 do
  while LCP[i] < LCP[top()] do
    last ← pop()
    write ')'
  if LCP[last] = LCP[top()] then
    append a 0 to the bitstring
  else
    append a 1 to the bitstring
  push(i) and write '('
write ')' and append 01 to the bitstring /* for LCP[1] and LCP[n + 1] */

```

Algorithm 6.4. It is worth mentioning that the stack can be implemented with $n + o(n)$ bits [105, Section 4.2].

Obviously, the BPS has $2n + 2$ parentheses and thus can be represented with $2n + 2$ bits. With a linear-time preprocessing and $o(n)$ bits of extra space, we obtain a data structure that supports the operations described in Section 6.1 in constant time. Furthermore, the bitstring B of length $n + 1$ is preprocessed so that *rank/select* queries can be answered in constant time (again, this requires only $o(n)$ bits of extra space). Consequently, the enhanced BPS with the above-mentioned operations thereon needs $3n + o(n)$ bits.

Each index i , $1 \leq i \leq n + 1$, in the LCP-array is represented by the i -th opening parenthesis (and the matching closing parenthesis) in the BPS. The position $ipos$ of the i -th opening parenthesis in the BPS can be found by $ipos = select_i(i)$. Conversely, given the position pos of an opening parenthesis in the BPS, we can determine the index i to which this opening parenthesis corresponds by $i = rank_i(pos)$.

Let us denote the i -th opening parenthesis by $_i($ and the matching closing parenthesis by $)_i^b$, where b is its mark (bit in B). Note that $_i($ and $)_i^b$ occur at positions $ipos = select_i(i)$ and $cipos = findclose(ipos)$ in the BPS. Moreover, b occurs at position $bcipos = rank_b(cipos)$ in the bitstring B . Conversely, given the position $bpos$ in B , its corresponding index in the LCP-array can be determined by $rank_i(findopen(select_b(bpos)))$.

With the help of the BPS, we can find NSV_{LCP} -values in constant time. As in Section 4.3.1, we write NSV instead of NSV_{LCP} .

Lemma 6.3.1 *For each index i with $1 < i < n + 1$, $\text{NSV}[i]$ can be determined on the BPS in constant time by*

$$\text{NSV}[i] = \text{rank}_l(\text{findclose}(\text{select}_l(i))) + 1$$

Proof The i -th opening parenthesis is written when the for-loop of Algorithm 6.4 is executed for i . Apart from that, i is pushed onto the stack. It is popped from the stack when the first index j with $\text{LCP}[j] < \text{LCP}[i]$ is encountered. Clearly, $j = \text{NSV}[i]$. Therefore, the closing parenthesis matching the i -th opening parenthesis (this is $\text{findclose}(\text{select}_l(i))$) is written when the for-loop of Algorithm 6.4 is executed for j . Moreover, in the same execution of the for-loop, j is pushed onto the stack and the j -th opening parenthesis is written. In other words, between $\text{findclose}(\text{select}_l(i))$ and $\text{select}_l(j)$ there is no opening parenthesis. Consequently, $j = \text{rank}_l(\text{findclose}(\text{select}_l(i))) + 1$. \square

In the example of Figures 6.3 and 6.5, $\text{NSV}[5]$ is computed as follows: $\text{select}_l(5)$ returns the position 7 of the fifth opening parenthesis. Then, $\text{findclose}(7)$ returns the position 10 of the closing parenthesis matching the fifth opening parenthesis. Because there are six opening parentheses up to position 10, $\text{NSV}[5] = \text{rank}_l(10) + 1 = 6 + 1 = 7$.

Lemma 6.3.2 *For an lcp-interval $[i..j]$, we have*

1. $\text{LCP}[i] > \text{LCP}[j + 1]$ if and only if $\text{NSV}[i] = j + 1$,
2. $\text{LCP}[i] \leq \text{LCP}[j + 1]$ if and only if $\text{NSV}[i] > j + 1$.

Proof Because $[i..j]$ is an lcp-interval, we have

- $\text{LCP}[k] > \text{LCP}[i]$ for all k with $i + 1 \leq k \leq j$,
- $\text{LCP}[k] > \text{LCP}[j + 1]$ for all k with $i + 1 \leq k \leq j$.

Hence $\text{NSV}[i] \geq j + 1$.

(1) Clearly, if $\text{NSV}[i] = j + 1$, then $\text{LCP}[i] > \text{LCP}[j + 1]$. Conversely, $\text{LCP}[i] > \text{LCP}[j + 1]$ in conjunction with $\text{NSV}[i] \geq j + 1$ implies $\text{NSV}[i] = j + 1$.

(2) This follows from $\text{NSV}[i] \geq j + 1$ and (1). \square

Lemmata 6.3.1 and 6.3.2 together imply that we can test in constant time whether $\text{LCP}[i] > \text{LCP}[j + 1]$ or $\text{LCP}[i] \leq \text{LCP}[j + 1]$. This test can be done on the BPS without access to the LCP-array.

Algorithm 6.5 Computing the parent interval of an lcp-interval $[i..j]$.

```

if NSV[ $i$ ] >  $j + 1$  then      /* LCP[ $i$ ] ≤ LCP[ $j + 1$ ] */
    return [PSV[ $j + 1$ ]..NSV[ $j + 1$ ] - 1]
else                          /* LCP[ $i$ ] > LCP[ $j + 1$ ] */
    return [PSV[ $i$ ].. $j$ ]

```

6.3.1 Finding the parent interval

The parent interval of an lcp-interval can be determined by Algorithm 6.5 with the help of PSV and NSV-values; cf. Lemma 4.3.9. We have seen that NSV[i] can be computed in constant time on the BPS. We will show in this section that PSV[i] can also be computed in constant time, provided we use the enhanced BPS (the BPS with the additional bitstring B) instead of the plain BPS. This implies that the parent interval of an lcp-interval can be found in constant time.

In terms of the balanced parentheses sequence, PSV[i] is the largest index k with $k < i$ so that the parenthesis pair $_k(\dots)_k$ encloses the pair $_i(\dots)_i$ and $\text{LCP}[k] \neq \text{LCP}[i]$. Without the LCP-array, however, we cannot directly test whether or not $\text{LCP}[k] \neq \text{LCP}[i]$. For this reason, we first show how PSEV[i] can be determined on the BPS in constant time, where

$$\text{PSEV}[i] = \max\{j \mid 1 \leq j < i \text{ and } \text{LCP}[j] \leq \text{LCP}[i]\}$$

Lemma 6.3.3 *Let i be an index with $1 < i < n + 1$. PSEV[i] can be determined on the BPS in constant time by*

$$\text{PSEV}[i] = \text{rank}_\ell(\text{enclose}(\text{select}_\ell(i)))$$

Proof When the for-loop of Algorithm 6.4 is executed for i , i is pushed onto the stack and the i -th opening parentheses is written. At that moment, let i_1, \dots, i_m, i be the elements on the stack. Clearly, we have $i_1 < \dots < i_m < i$ and $\text{LCP}[i_1] \leq \dots \leq \text{LCP}[i_m] \leq \text{LCP}[i]$. Moreover, the non-matched opening parentheses before position $\text{select}_\ell(i)$ are those at the positions $\text{select}_\ell(i_1), \dots, \text{select}_\ell(i_m)$. Note that the parenthesis pairs corresponding to i_1, \dots, i_m are exactly the pairs enclosing the pair $_i(\dots)_i$. Clearly, $\text{PSEV}[i] = i_m$ because each index k with $i_m < k < i$ satisfies $\text{LCP}[k] > \text{LCP}[i]$. Since the pair $_{i_m}(\dots)_{i_m}$ most tightly encloses the pair $_i(\dots)_i$, the index i_m can be found by $\text{rank}_\ell(\text{enclose}(\text{select}_\ell(i)))$. All in all, $\text{PSEV}[i] = \text{rank}_\ell(\text{enclose}(\text{select}_\ell(i)))$. \square

At this point the bitstring B of the enhanced BPS comes into play.

Lemma 6.3.4 *Let i be an index with $1 < i < n + 1$. PSV[i] can be determined in constant time on the enhanced BPS by Algorithm 6.6.*

Algorithm 6.6 Computing $\text{PSV}[i]$ in constant time on the enhanced BPS.

```

bcipos = rank1(findclose(select1(i)))
if B[bcipos] = 1 then
    PSV[i] ← PSEV[i]
else /* determine the position bpos of the next 1 in the bitstring B */
    bpos ← select1(rank1(bcipos) + 1)
    ip ← rank1(findopen(select1(bpos)))
    PSV[i] ← PSEV[ip]
    
```

Proof Let $j = \text{NSV}[i]$. Suppose that the elements on the stack are i_1, \dots, i_m when the for-loop of Algorithm 6.4 is executed for index j . Since i is on the stack, we have $i = i_k$ for some $1 \leq k \leq m$. Note that $i_1 < i_2 < \dots < i_m$ as well as $\text{LCP}[i_1] \leq \text{LCP}[i_2] \leq \dots \leq \text{LCP}[i_m]$. Let i_p, \dots, i_q be the indices with $\text{LCP}[i_p] = \text{LCP}[i] = \text{LCP}[i_q]$. Clearly, $i = i_k$ is one of these indices, so $p \leq k \leq q$. Because $\text{LCP}[j] < \text{LCP}[i] = \text{LCP}[i_p]$, it follows that $i_p, \dots, i_q, \dots, i_m$ (and possibly more elements) are popped from the stack. The crucial observation is that $)_{i_q}^0 \dots)_{i_{p+1}}^0)_{i_p}^1$ form a contiguous subsequence of the enhanced BPS. Moreover, if we can determine the index i_p , then $\text{PSV}[i] = \text{PSEV}[i_p]$.

As observed above, $i = i_k$ for some k with $p \leq k \leq q$. That is, the closing parenthesis $)_i^b$ at position $\text{cipos} = \text{findclose}(\text{select}_1(i))$ is one of the closing parentheses in $)_{i_q}^0 \dots)_{i_{p+1}}^0)_{i_p}^1$. Algorithm 6.6 computes the position of the mark b in the bitstring B by $\text{bcipos} = \text{rank}_1(\text{cipos})$, i.e., $b = B[\text{bcipos}]$. If $b = 1$, then $i = i_p$ and $\text{PSV}[i] = \text{PSEV}[i]$. Otherwise, $b = 0$ and $i \neq i_p$. In a left-to-right scan of the enhanced BPS, starting at position cipos , $)_{i_p}$ is the first closing parenthesis that is marked with 1. Thus, the position of $)_{i_p}$ in the BPS can be obtained by determining the position bpos of the next 1 in the bitstring B and finding the corresponding closing parenthesis by $\text{select}_1(\text{bpos})$. The index i_p itself is $\text{rank}_1(\text{findopen}(\text{select}_1(\text{bpos})))$. Now, $\text{PSV}[i] = \text{PSEV}[i_p]$.

It is obvious that Algorithm 6.6 runs in constant time. \square

Exercise 6.3.5 Prove that Algorithm 6.7 computes $\text{PSV}[i]$ in $O(\sigma)$ time. Note that the algorithm needs access to the LCP-array and the BPS, but not to the bitstring B .

Algorithm 6.7 Computing $\text{PSV}[i]$ in $O(\sigma)$ time.

```

j ← PSEV[i]
while LCP[j] = LCP[i] do
    j ← PSEV[j]
PSV[i] ← j
    
```

Algorithm 6.8 For an lcp-interval $\ell\text{-}[i..j]$, this procedure returns its first ℓ -index solely based on the BPS.

```

if NSV[i] > j + 1 then           /* LCP[i] ≤ LCP[j + 1] */
    return rank((findopen(select((j + 1) - 1))
else
    return rank((findopen(findclose(select((i)) - 1))

```

Exercise 6.3.6 Devise an algorithm that computes PSV[i] in $O(\log(\sigma))$ time by a binary search. The algorithm is allowed to access the LCP-array and the BPS, but not the bitstring B .

6.3.2 Finding child intervals

According to Lemma 4.3.5, determining the child intervals of an ℓ -interval $[i..j]$ boils down to finding its ℓ -indices. On the BPS, the first ℓ -index of $[i..j]$ can be found by Algorithm 6.8.

Lemma 6.3.7 *Algorithm 6.8 takes an lcp-interval $\ell\text{-}[i..j]$ as input and determines its first ℓ -index k in constant time.*

Proof Let $i_1 < i_2 < \dots < i_m$ be the ℓ -indices of $[i..j]$. Algorithm 6.8 first tests whether or not $\text{LCP}[i] \leq \text{LCP}[j + 1]$. By Lemma 6.3.2, we know that $\text{LCP}[i] \leq \text{LCP}[j + 1]$ if and only if $\text{NSV}[i] > j + 1$.

If $\text{LCP}[i] \leq \text{LCP}[j + 1]$, then $(i_m \dots i_2)_{i_1 \ j+1}$ is a contiguous subsequence of the BPS. That is, the closing parenthesis corresponding to the first ℓ -index i_1 of $[i..j]$ directly precedes the opening parenthesis corresponding to $j + 1$. Therefore, $i_1 = \text{rank}_{(}$ (findopen(select₍(j + 1) - 1)).

If $\text{LCP}[i] > \text{LCP}[j + 1]$, then $(i_m \dots i_2)_{i_1} i$ is a contiguous subsequence of the BPS. That is, the closing parenthesis corresponding to the first ℓ -index i_1 of $[i..j]$ directly precedes the closing parenthesis corresponding to i . Thus, $i_1 = \text{rank}_{(}$ (findopen(findclose(select₍(i)) - 1)). \square

Once we have found the first ℓ -index of an lcp-interval $[i..j]$, it is possible to determine all ℓ -indices $i_1 < i_2 < \dots < i_m$ of $[i..j]$. In essence, this is because $(i_m^0 \dots i_2^0)_{i_1^1}^1$ is a contiguous subsequence of the enhanced BPS. With the help of the ℓ -indices, the k -th child interval of $[i..j]$ (if it exists) can be found in constant time. Algorithm 6.9 gives the pseudo-code.

Lemma 6.3.8 *Algorithm 6.9 takes an lcp-interval $[i..j]$ as input and computes its k -th child interval in constant time (where $1 \leq k \leq \sigma$). If $[i..j]$ has less than k child intervals, it returns \perp .*

Algorithm 6.9 For an lcp-interval $[i..j]$, this procedure returns the k -th child interval based on the enhanced BPS, where $1 \leq k \leq \sigma$. If $[i..j]$ has less than k child intervals, it returns \perp .

```

if NSV[ $i$ ] >  $j + 1$  then           /* LCP[ $i$ ] ≤ LCP[ $j + 1$ ] */
     $ci_{1pos} = select_{\ell}(j + 1) - 1$ 
else
     $ci_{1pos} = findclose(select_{\ell}(i)) - 1$ 
 $bci_{1pos} \leftarrow rank_{\gamma}(ci_{1pos})$ 
if BPS[ $ci_{1pos} - 1$ ] = "(" or (BPS[ $ci_{1pos} - 1$ ] = ")") and  $B[bci_{1pos} - 1] = 1$  then
     $ci_{mpos} \leftarrow ci_{1pos}$ 
else           /* BPS[ $ci_{1pos} - 1$ ] = ")" and  $B[bci_{1pos} - 1] = 0$  */
     $bpos \leftarrow select_1(rank_1(bci_{1pos}) - 1) + 1$ 
     $ci_{mpos} \leftarrow select_{\gamma}(bpos)$ 
 $m \leftarrow ci_{1pos} - ci_{mpos} + 1$            /* the number of  $\ell$ -indices is  $m$  */
if  $m = 1$  then
     $i_1 \leftarrow rank_{\ell}(findopen(ci_{1pos}))$ 
    return  $[i..i_1 - 1]$ 
else if  $2 \leq k \leq m$  then
     $i_{k-1} \leftarrow rank_{\ell}(findopen(ci_{1pos} - (k - 1) + 1))$ 
     $i_k \leftarrow rank_{\ell}(findopen(ci_{1pos} - k + 1))$ 
    return  $[i_{k-1}..i_k - 1]$ 
else if  $k = m + 1$  then
     $i_m \leftarrow rank_{\ell}(findopen(ci_{mpos}))$ 
    return  $[i_m..j]$ 
else
    return  $\perp$ 

```

Proof Let $i_1 < i_2 < \dots < i_m$ be the ℓ -indices of $[i..j]$, where $m \geq 1$. After the execution of the first if-then-else statement in Algorithm 6.9, ci_{1pos} is the position of the closing parenthesis corresponding to the first ℓ -index i_1 of $[i..j]$; cf. Lemma 6.3.7. In the enhanced BPS, the closing parenthesis $)_{i_1}$ is marked with a 1, and the position of its mark in the bitstring B is $bci_{1pos} = rank_{\gamma}(ci_{1pos})$. Now, if $)_{i_1}^1$ is directly preceded by an opening parenthesis or a closing parenthesis marked with a 1, then i_1 is the only ℓ -index of $[i..j]$ (hence $m = 1$). In this case, the position ci_{mpos} of the closing parenthesis corresponding to the last ℓ -index i_m coincides with ci_{1pos} . Otherwise, $)_{i_1}^1$ is directly preceded by a closing parenthesis marked with a 0. Because $)_{i_m}^0 \dots)_{i_2}^0)_{i_1}^1$ is a contiguous subsequence of the enhanced BPS, it follows that $m > 1$, i.e., there is more than one ℓ -index. Note that the closing parenthesis preceding $)_{i_m}^0$ (if it exists) must be marked with a 1. In this case, we compute $bpos = select_1(rank_1(bci_{1pos}) - 1) + 1$, the position directly right to the position of the first 1 in B that is left to bci_{1pos} (to

ensure that there is always such a 1 we set $B[0] = 1$, i.e., an additional 1 is added at the beginning of B). The position ci_mpos of the closing parenthesis corresponding to the last ℓ -index i_m is $ci_mpos \leftarrow select_1(bpos)$. Because $)_{i_m}^0 \dots)_{i_2}^0)_{i_1}^1$ is a contiguous subsequence of the enhanced BPS, the number of ℓ -indices of $[i..j]$ is $m = ci_1pos - ci_mpos + 1$. Moreover the k -th ℓ -index i_k , $1 \leq k \leq m$, satisfies $i_k = rank_1(findopen(ci_1pos - k + 1))$. According to Lemma 4.3.5, Algorithm 6.9 returns the k -th child interval of $[i..j]$ (if it exists). Since each of its statements can be done in constant time, the whole algorithm takes only constant time. \square

6.3.3 Computing $getInterval([i..j], c)$

In exact string matching, the procedure $getInterval$ (Algorithm 5.1 on page 118) searches for a specific child interval. To be precise, if the lcp-interval $[i..j]$ represents the string ω , the procedure call $getInterval([i..j], c)$ computes the lcp-interval $[lb..rb]$ that represents the string ωc , where $c \in \Sigma$.

Using Algorithm 6.9, we can enumerate all child intervals $[lb..rb]$ of $[i..j]$ until the one with $S[SA[lb] + |\omega|] = \dots = S[SA[rb] + |\omega|] = c$ is found. In other words, the procedure $getInterval$ can be implemented on a data structure using $3n + o(n)$ bits. It needs access to the suffix array SA and the string S , but not to the LCP-array.

As a matter of fact, the enhanced BPS allows us to implement the procedure $getInterval$ in such a way that the specific child interval can be found by a binary search in $O(\log \sigma)$ time. It is plain to see that in a binary search on $1, \dots, \sigma$ the respective child interval can be found by Algorithm 6.9. However, a more efficient implementation can be obtained by a simple modification of that algorithm. This goes as follows. As in Algorithm 6.9, we determine the positions ci_1pos and ci_mpos of the closing parentheses corresponding to the first and the last ℓ -index i_1 and i_m , respectively. Then, $m = ci_1pos - ci_mpos + 1$ is the number of ℓ -indices and $)_{i_m}^0 \dots)_{i_2}^0)_{i_1}^1$ is a contiguous subsequence of the enhanced BPS, where $i_1 < i_2 < \dots < i_m$ are the ℓ -indices of $[i..j]$. Therefore, the binary search can be done on the sequence $)_{i_m}^0 \dots)_{i_2}^0)_{i_1}^1$ of closing parenthesis. Let k be the current value in the binary search (initially, $k = \lfloor (1 + m)/2 \rfloor$). Recall that the k -th ℓ -index i_k , $1 \leq k \leq m$, can be found by $i_k = rank_1(findopen(ci_1pos - k + 1))$. Furthermore, $i_k - 1$ is the right boundary of the k -th child interval and i_k is the left boundary of the $(k + 1)$ -th child interval. Thus, if $S[SA[i_k - 1] + |\omega|] = c$ or $S[SA[i_k] + |\omega|] = c$, then we know one boundary of the interval we are searching for and we can compute the other boundary as in Algorithm 6.9. Otherwise, if $S[SA[i_k - 1] + |\omega|] < c$, the binary search proceeds with the right half of the current sequence of closing parenthesis, and if $S[SA[i_k] + |\omega|] > c$, the binary search proceeds with the left half.

6.3.4 Answering RMQs in constant time

Algorithm 6.10 shows that a range minimum query can also be answered in constant time on the BPS. Its correctness is proved in Theorem 6.3.9.

Algorithm 6.10 Constant time calculation of $\text{RMQ}(i, j)$ based on the BPS.

```

if  $i = j$  then
    return  $i$ 
 $ipos \leftarrow \text{select}_l(i)$ 
 $jpos \leftarrow \text{select}_l(j)$ 
 $cipos \leftarrow \text{findclose}(ipos)$ 
if  $jpos < cipos$  then
    return  $i$ 
else
     $pos \leftarrow \text{rr\_enclose}(ipos, jpos)$ 
    if  $pos = \perp$  then
        return  $j$ 
    else
        return  $\text{rank}_l(pos)$ 

```

Theorem 6.3.9 Algorithm 6.10 computes $\text{RMQ}(i, j)$ in constant time for any two indices i and j with $1 \leq i \leq j \leq n$.

Proof We use a case differentiation. If $i = j$, then obviously $\text{RMQ}(i, j) = i$. Otherwise $i < j$. In this case, let $ipos = \text{select}_l(i)$ and $jpos = \text{select}_l(j)$. If $jpos < \text{findclose}(ipos)$, then $ipos < jpos < \text{findclose}(jpos) < \text{findclose}(ipos)$, i.e., the parenthesis pair $i(\dots)_i$ encloses the pair $j(\dots)_j$. By the construction of the BPS, this means that $\text{LCP}[i] \leq \text{LCP}[k]$ for all k with $i < k \leq j$. Hence $\text{RMQ}(i, j) = i$. Otherwise, we have $\text{findclose}(ipos) < jpos$. If $\text{rr_enclose}(ipos, jpos) = \perp$, then there is no parenthesis pair with opening parenthesis at a position greater than $\text{findclose}(i)$ that encloses the parenthesis pair $j(\dots)_j$. It follows from the construction of the BPS that $\text{LCP}[j] < \text{LCP}[k]$ for all k with $i \leq k < j$. Thus, $\text{RMQ}(i, j) = j$.

In the last case $\text{rr_enclose}(ipos, jpos) = pos$, where pos is the smallest position of an opening parenthesis so that $\text{findclose}(ipos) < pos < jpos$ and $\text{findclose}(jpos) < \text{findclose}(pos)$. So we have

$$ipos < \text{findclose}(ipos) < pos < jpos < \text{findclose}(jpos) < \text{findclose}(pos)$$

Let $p = \text{rank}_l(pos)$. The inequality $\text{findclose}(ipos) < pos$ implies that $\text{LCP}[p] < \text{LCP}[k]$ for all k with $i \leq k < p$. Furthermore, because the parenthesis pair $p(\dots)_p$ encloses the pair $j(\dots)_j$, we conclude that $\text{LCP}[p] \leq \text{LCP}[k]$ for all k with $p < k \leq j$. All in all, $\text{RMQ}_{\text{LCP}}(i, j) = p$. \square

Exercise 6.3.10 Given two lcp-intervals, show that their LCA in the lcp-interval tree can be computed in constant time based on the BPS.

It should be stressed that Algorithm 6.10 does not rely on a specific property of the LCP-array. Consequently, it can be used to answer range minimum queries in constant time on the BPS of an arbitrary array A (provided, of course, that the array elements stem from a totally ordered set). A similar statement is true for the computation of NSV and PSV-values, where the latter relies on the additional bitstring B . Thus, the enhanced BPS of an array A —together with a data structure that supports the operations described in Section 6.1—occupies only $3n + o(n)$ bits and allows us to compute $\text{RMQ}(i, j)$, $\text{PSV}[i]$, and $\text{NSV}[i]$ on A in constant time; cf. [247]. This practical solution is not far away from the theoretically optimal solution: Fischer [106] has shown that $2.54n + o(n)$ bits suffice for this task.

6.3.5 Computing suffix link intervals

As explained in Section 5.5.2, the suffix link interval $[lb..rb]$ of an lcp-interval $[i..j]$ can be determined by the following pseudo-code (Algorithm 5.24 on page 187):

$$\begin{aligned} k &\leftarrow \text{RMQ}(\psi[i] + 1, \psi[j]) \\ [lb..rb] &\leftarrow [\text{PSV}[k].. \text{NSV}[k] - 1] \end{aligned}$$

In the previous sections, we have seen that a value of PSV, NSV, and RMQ can be computed in constant time on the enhanced BPS. With the compressed suffix array discussed in Section 6.2.1, ψ -values can be computed easily. In this book, however, we will use a different data structure: a wavelet tree. We shall see in Section 7.4 that a ψ -value can be computed in $O(\log \sigma)$ time on a wavelet tree (and we shall also learn that a wavelet tree supports many more operations). Consequently, with a wavelet tree and the enhanced BPS, a suffix link interval can be determined in $O(\log \sigma)$ time, using the pseudo-code above.

6.3.6 Attaching additional information

In some applications, we had to attach additional information to lcp-intervals. For example, in Section 4.3.3 an additional array VAL was introduced that helped to compute string kernels efficiently; cf. Section 5.7.2. For an ℓ -interval $[i..j]$ with ℓ -indices $i_1 < i_2 < \dots < i_k$ the same information $\text{VAL}[i_1]$ was stored k times, namely at each ℓ -index. Another more prominent example is the LCP-array itself: the same value $\text{LCP}[i_1]$ is stored at each ℓ -index of the interval $[i..j]$. In this section, we show how to avoid the redundant information at the indices i_2, \dots, i_k . We will

		2		3		2		1		2		1		0		-1					
(()	(()	(())	(()	(())	()			
1	2	2	3	4	4	5	6	6	5	3	7	8	8	9	10	10	9	7	11	11	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Figure 6.6: The row above the BPS shows the values of the LCP'-array corresponding to the LCP-array shown in Figure 6.3 (page 266). Hence $\text{LCP}'[1..8] = [2, 3, 2, 1, 2, 1, 0, -1]$. The row below the BPS is only for illustrative purposes: the opening parentheses are numbered consecutively and a closing parenthesis has the number of its matching opening parenthesis. The lower row of numbers shows the positions of the parentheses in the BPS.

do this for the LCP-array, but the same technique applies to other arrays like the VAL-array as well. Since the information is kept only once for each lcp-interval, we use an array LCP' of size n' , where n' is the number of (non-singleton) lcp-intervals (to put it differently, n' is the number of internal nodes in the suffix tree of S). Recall that $n' < n$. Now we make use of the fact that there is a bijection between the set of closing parentheses that are marked with a 1 in the enhanced BPS of S and the set of first lcp-indices of all lcp-intervals. To be precise, we store the values of first lcp-indices in the order in which they occur from left to right in the BPS. For example, in Figure 6.5 (page 267) the left-to-right order of closing parentheses that are marked with a 1 corresponds to the sequence of indices 2, 4, 6, 3, 8, 10, 7, 1. The corresponding sequence of lcp-values (cf. Figure 6.3 on page 266) $\text{LCP}[2] = 2$, $\text{LCP}[4] = 3$, $\text{LCP}[6] = 2$, $\text{LCP}[3] = 1$, $\text{LCP}[8] = 2$, $\text{LCP}[10] = 1$, $\text{LCP}[7] = 0$, $\text{LCP}[1] = -1$ is stored in an array LCP' ; see Figure 6.6. So given the enhanced BPS and the LCP' -array, it remains to address the following two problems:

1. Given an index i , how to retrieve the value $\text{LCP}[i]$.
2. Given an lcp-interval $[i..j]$, how to retrieve its lcp-value.

Algorithm 6.11 (originating from [126]) solves the first and Algorithm 6.12 solves the second problem.

Lemma 6.3.11 *Given an index i , Algorithm 6.11 returns the value $\text{LCP}[i]$ in constant time, based on the enhanced BPS and the LCP' -array.*

Proof Algorithm 6.11 determines the position cpos of the closing parenthesis matching the i -th opening parenthesis, and the corresponding position bcpos in the bit vector B . Furthermore, it calculates the number $k = \text{rank}_1(B, \text{bcpos})$ of ones up to position bcpos in the bit vector B . Now,

Algorithm 6.11 For an index i , calculate $\text{LCP}[i]$ using BPS and LCP' .

```

 $ipos \leftarrow \text{select}_\ell(i)$ 
 $cipos \leftarrow \text{findclose}(ipos)$ 
 $bcipos \leftarrow \text{rank}_\gamma(cipos)$ 
 $k \leftarrow \text{rank}_1(B, bcipos)$ 
if  $B[bcipos] = 1$  then
    return  $\text{LCP}'[k]$ 
else
    return  $\text{LCP}'[k + 1]$ 

```

Algorithm 6.12 For an lcp-interval $[i..j]$, this procedure returns its lcp-value based on the BPS and the LCP' -array.

```

if  $\text{NSV}[i] > j + 1$  then           /*  $\text{LCP}[i] \leq \text{LCP}[j + 1]$  */
     $cpos \leftarrow \text{select}_\ell(j + 1) - 1$ 
else
     $cpos \leftarrow \text{findclose}(\text{select}_\ell(i)) - 1$ 
     $bcpos \leftarrow \text{rank}_\gamma(cpos)$ 
     $k \leftarrow \text{rank}_1(B, bcpos)$ 
return  $\text{LCP}'[k]$ 

```

if $B[bcipos] = 1$, then i is the first lcp-index in its lcp-interval and its LCP-value can be found at index k in the array LCP' . Otherwise i is not the first lcp-index in its lcp-interval, say $[lb..rb]$, and there are more closing parentheses immediately to the right of $)_i$. Of those, the first marked with a 1 is the first lcp-index of $[lb..rb]$ and hence its LCP-value can be found at index $k + 1$ in the array LCP' . \square

In Algorithm 6.11, $bcipos$ can be determined without $\text{rank}_\gamma(cipos)$ because

$$\text{rank}_\gamma(cipos) = \underbrace{\text{rank}_\gamma(ipos)}_{=ipos-i} + \underbrace{\text{rank}_\gamma(cipos) - \text{rank}_\gamma(ipos)}_{=\frac{cipos-ipos+1}{2}} = \frac{ipos + cipos + 1}{2} - i$$

This can be seen as follows. We know from $ipos = \text{select}_\ell(i)$ that there are i opening parentheses up to position $ipos$. Thus, $\text{rank}_\gamma(ipos) = ipos - i$. Furthermore, since $cipos$ is the position of the closing parenthesis matching the i -th opening parenthesis, we infer that $\text{BPS}[ipos..cipos]$ is a balanced parentheses sequence. It follows as a consequence that the number of closing parentheses in it is $\frac{cipos-ipos+1}{2}$.

Lemma 6.3.12 Algorithm 6.12 takes an lcp-interval $[i..j]$ as input and determines its lcp-value in constant time.

Proof The proof is similar to the proof of Lemma 6.3.7. Let $i_1 < i_2 < \dots < i_m$ be the ℓ -indices of $[i..j]$. Algorithm 6.12 first tests whether or not $\text{LCP}[i] \leq \text{LCP}[j+1]$. By Lemma 6.3.2, we know that $\text{LCP}[i] \leq \text{LCP}[j+1]$ if and only if $\text{NSV}[i] > j+1$.

If $\text{LCP}[i] \leq \text{LCP}[j+1]$, then $)_{i_m} \dots)_{i_2})_{i_1} j+1($ is a contiguous subsequence of the BPS. That is, the closing parenthesis corresponding to the first ℓ -index i_1 of $[i..j]$ directly precedes the opening parenthesis corresponding to $j+1$, and it can be found at position $\text{cpos} = \text{select}_\ell(j+1) - 1$.

If $\text{LCP}[i] > \text{LCP}[j+1]$, then $)_{i_m} \dots)_{i_2})_{i_1})_i$ is a contiguous subsequence of the BPS. That is, the closing parenthesis corresponding to the first ℓ -index i_1 of $[i..j]$ directly precedes the closing parenthesis corresponding to i , and it can be found at position $\text{cpos} = \text{findclose}(\text{select}_\ell(i)) - 1$. \square

Compressed Full-Text Indexes

Until now, we have used the suffix array as an index data structure, and exact string matching was done in forward direction. This chapter is dedicated to index data structures and applications in which a forward search is replaced with a backward search. It is organized as follows. First, we review the Burrows-Wheeler transform [48]—a well-known technique employed in lossless data compression—on which backward search is based. Second, we describe the search algorithm discovered by Ferragina and Manzini [100]. Third, we introduce the wavelet tree invented by Grossi et al. [134]. This data structure supports backward search and has many other virtues. In subsequent sections, we shed new light on solutions to problems faced in Chapter 5, such as developing new algorithms that address space efficiency issues, as well as problems related to bidirectional searches and approximate string matching.

7.1 The components of a compressed full-text index

Many variations of *compressed suffix trees* (CSTs) have been proposed in the literature (see e.g. [273]), and these do not all have the same functionality. Because we focus on backward search, which is normally not supported by a CST, we prefer the term “compressed full-text index”.

Definition 7.1.1 A *compressed full-text index* of a string S is a space-efficient data structure that supports (at least) the following operations:

1. backward search,
2. access to the suffix array of S ,
3. access to the LCP-array of S ,
4. navigation on the (virtual) suffix tree of S .

The compressed full-text index of the string S that is used throughout this book consists of the following four components:

1. the wavelet tree of the Burrows-Wheeler transformed string of S ,
2. the sparse suffix array of S from Section 6.2.1,
3. the compressed LCP-array as explained in Section 6.2.2,
4. the balanced parentheses sequence BPS of the LCP-array that was introduced in Section 6.3.

We emphasize that each of the four components can be replaced with another component that has the same functionality. For example, the wavelet tree can be substituted by the compressed suffix array [135, 270] sketched in Section 6.2.1 because backward search can be done with the ψ -function; see Exercise 7.3.3. Further alternatives are described in [238]. However, the wavelet tree has many sophisticated properties that make it most suitable for many applications. Alternative compressed representations of the LCP-array are discussed in [126], among which is a representation that is based on the array LCP' from Section 6.3.6. There are also alternatives to the BPS of the LCP-array, most notably the BPS_{pre} introduced in Section 6.1; cf. [231, 273]. We refer to [124] for an in-depth experimental study of the various incarnations of compressed full-text indexes.

7.2 The Burrows-Wheeler transform

The Burrows-Wheeler transform was introduced in a technical report written by David Wheeler and Michael Burrows [48]; see the historical notes in Adjero et al. [6]. In practice, the Burrows-Wheeler transformed string tends to be easier to compress than the original string; see e.g. [48, Section 3] and [215] for reasons why the transformed string compresses well.

Here we assume that the string S of length n is terminated by the sentinel character $\$$. Although this is not necessary for the Burrows-Wheeler transform to work correctly (cf. [48]), in virtually all practical cases the file to be compressed is terminated by a special symbol, the EOF (end of file) character. Moreover, it allows us to use a fast suffix sorting algorithm to compute the transformed string.

7.2.1 Encoding

The Burrows-Wheeler transform transforms a string S in three steps:

1. Form a conceptual matrix M' whose rows are the cyclic shifts of the string S .

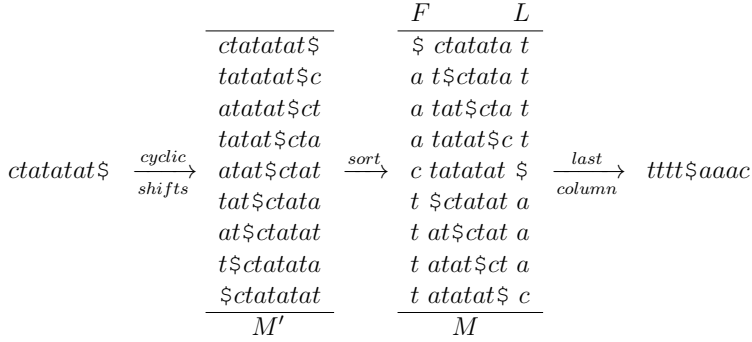


Figure 7.1: The Burrows and Wheeler transform applied to the string $S = ctatatat\$$ yields the output $L = tttt\$aaac$.

2. Compute the matrix M by sorting the rows of M' lexicographically.
3. Output the last column L of M .

An example can be found in Figure 7.1. We next show that computing the Burrows-Wheeler transformed string of S boils down to sorting the suffixes of S , or more precisely, the output L of the Burrows-Wheeler transform can be derived in linear time from the suffix array SA. To this end, we define a string BWT and show that it coincides with L .

Definition 7.2.1 For a string S of length n having the sentinel character at the end (and nowhere else), the string $\text{BWT}[1..n]$ is defined by $\text{BWT}[i] = \$$ if $\text{SA}[i] = 1$ and $\text{BWT}[i] = S[\text{SA}[i] - 1]$ if $\text{SA}[i] \neq 1$.

Obviously, the string $\text{BWT}[1..n]$ can be derived in linear time from the suffix array SA; see Algorithm 7.1.

Algorithm 7.1 Computing BWT from SA and the string S .

```

for  $i \leftarrow 1$  to  $n$  do
  if  $\text{SA}[i] = 1$  then  $\text{BWT}[i] \leftarrow \$$ 
  else  $\text{BWT}[i] \leftarrow S[\text{SA}[i] - 1]$ 

```

If we truncate each string in the matrix M after the sentinel $\$,$ then the truncated strings are still lexicographically ordered; see Figure 7.2. Since these truncated strings are exactly the suffixes of S , the string BWT coincides with the string L (this crucially relies on the fact that S is terminated by $\$$; see Exercise 7.2.2).

F	L		F	L		BWT	F	L
$\$ ctatata t$			$\$$	t		t	$\$$	t
$a t\$ctata t$			$a t\$$	t		t	$at\$$	t
$a tat\$cta t$			$a tat\$$	t		t	$atat\$$	t
$a tatat\$c t$			$a tatat\$$	t		t	$atatat\$$	t
$c tatatat \$$	$\xrightarrow[\text{after } \$]{\text{truncate}}$		$c tatatat \$$	$\$$	$\xrightarrow[\text{L=BWT}]{\text{observe}}$	$\$$	$ctatatat\$$	$\$$
$t \$ctatat a$			$t \$$	a		a	$t\$$	a
$t at\$ctat a$			$t at\$$	a		a	$tat\$$	a
$t atat\$ct a$			$t atat\$$	a		a	$tatat\$$	a
$t atatat\$ c$			$t atatat\$$	c		c	$tatatat\$$	c

Figure 7.2: Truncate the strings (rows) after the sentinel character, and observe that $L = \text{BWT}$.

i	1	2	3	4	5	6	7	8	9
$L[i]$	t	t	t	t	$\$$	a	a	a	c
$LF(i)$	6	7	8	9	1	2	3	4	5
$F[i]$	$\$$	a	a	a	c	t	t	t	t

Figure 7.3: LF maps the last column L to the first column F .

Exercise 7.2.2 For a string S of length n without the sentinel character at the end, define $\text{BWT}[i] = S[n]$ if $\text{SA}[i] = 1$ and $\text{BWT}[i] = S[\text{SA}[i] - 1]$ if $\text{SA}[i] \neq 1$. Find a string S (without sentinel) for which $\text{BWT} \neq L$.

7.2.2 Decoding

It is not obvious how the string BWT can be retransformed into the original string S . The key to this back-transformation is the so-called LF -mapping.

Definition 7.2.3 Let F and L be the first and last column in the matrix M ; cf. Figure 7.1. The function $LF : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ is defined as follows: If $L[i] = c$ is the k -th occurrence of character c in L , then $LF(i) = j$ is the index so that $F[j]$ is the k -th occurrence of c in F .

The function LF is called last-to-first mapping because it maps the last column L to the first column F ; see Figure 7.3 for an example. In the following, when we regard the LF -mapping as an array, we will use the notation $LF[i]$ instead of $LF(i)$.

Algorithm 7.2 Computing LF from BWT and the C -array.

```

for all  $c \in \Sigma$  do
   $count[c] \leftarrow C[c]$ 
for  $i \leftarrow 1$  to  $n$  do
   $c \leftarrow BWT[i]$ 
   $count[c] \leftarrow count[c] + 1$ 
   $LF[i] \leftarrow count[c]$ 

```

Next, we develop a linear-time algorithm that computes LF . To achieve this goal, we must be able to find the k -th occurrence of a character $c \in \Sigma$ in F . Employing the C -array (if we consider all characters in Σ that are smaller than c , then $C[c]$ is the overall number of their occurrences in S), the index of the first occurrence of character c in the array F is $C[c] + 1$. Therefore, the k -th occurrence of c in F can be found at index $C[c] + k$.

Algorithm 7.2 shows the pseudo-code for the computation of LF . It scans the BWT from left to right and counts how often each character appeared already. The algorithm uses an auxiliary array *count* of size σ . Initially, $count[c] = C[c]$. Each time character c appears during the scan of BWT, $count[c]$ is incremented by one. As discussed above, if the algorithm finds the k -th occurrence of character c at index i in BWT, then the k -th occurrence of character c in F appears at index $count[c] = C[c] + k$. In other words, the index $LF[i]$ we are searching for is $count[c]$.

It remains to compute the original string S from BWT and LF . Lemma 7.2.4 states the crucial property of the LF -mapping that makes this possible.

Lemma 7.2.4 *The first row of the matrix M contains the suffix $S_n = \$$. If row i , $2 \leq i \leq n$, of the matrix M contains the suffix S_j , then row $LF(i)$ of M contains the suffix S_{j-1} .*

Proof Since $\$$ is the smallest character in Σ , the first row of M contains $\$$, which is the n -th suffix of S . Let $c \neq \$$ be a character in S , and let $i_1 < i_2 < \dots < i_m$ be all the indices with $BWT[i_k] = c$, $1 \leq k \leq m$. (So if we would number the m occurrences of c in $L = BWT$ as c_1, c_2, \dots, c_m , then $BWT[i_k] = c_k$.) Because the suffixes in M are ordered lexicographically, we have $S_{SA[i_1]} < S_{SA[i_2]} < \dots < S_{SA[i_m]}$. Obviously, this implies $cS_{SA[i_1]} < cS_{SA[i_2]} < \dots < cS_{SA[i_m]}$. (With the occurrence numbers as subscripts, $c_1S_{SA[i_1]} < c_2S_{SA[i_2]} < \dots < c_mS_{SA[i_m]}$.) By definition, $LF(i_k)$ is the index so that $F[LF(i_k)]$ is the k -th occurrence of c in F . Since $cS_{SA[i_k]} = S_{SA[i_k]-1}$, it follows that row $LF(i_k)$ of M contains the suffix $S_{SA[i_k]-1}$. \square

Theorem 7.2.5 *If $L = BWT$ is the output of the Burrows-Wheeler transform applied to the string S , and LF is the corresponding last-to-first mapping, then Algorithm 7.3 computes S .*

Algorithm 7.3 Computing the string S from BWT and LF .

```

 $S[n] \leftarrow \$$ 
 $j \leftarrow 1$ 
for  $i \leftarrow n - 1$  downto 1 do
     $S[i] \leftarrow \text{BWT}[j]$ 
     $j \leftarrow LF(j)$ 

```

Proof Initially, the algorithm assigns $\$$ to $S[n]$. This is correct because $\$$ is the last character of S . Since $\$$ is the smallest character in Σ , row $j = 1$ of the matrix M contains the suffix $S_n = \$$. Now $L[1] = \text{BWT}[1] = S[n - 1]$ implies that the $(n - 1)$ -th character of S is correctly decoded in the first iteration of the for-loop. After the assignment $j \leftarrow LF(j)$, row j contains the suffix S_{n-1} . In the second iteration of the for-loop, the $(n - 2)$ -th character of S is correctly decoded because $L[j] = \text{BWT}[j] = S[n - 2]$, and so on. \square

Exercise 7.2.6 Extend Algorithm 7.3 so that it also computes the suffix array of the string S . Is it possible to overwrite the LF -array with the suffix array? (This would save space if the LF -array is no longer needed.)

An alternative way to retransform the BWT into the original string S uses the ψ -function instead of the LF -mapping. We are already familiar with the ψ -function: For a string of length n (without the sentinel character $\$$ at the end), $\psi(i) = \text{ISA}[\text{SA}[i] + 1]$ for all i with $\text{SA}[i] < n$; see Definition 5.5.4. Here, we assume that the string under consideration is terminated by $\$$. If S is a string of length n having the sentinel character at the end (and nowhere else), then $\text{SA}[1] = n$ because $\$$ is the lexicographically smallest suffix of S . So with the previous definition of the ψ -function, the value $\psi(1)$ is undefined. Definition 7.2.7 provides a value for $\psi(1)$ so that ψ becomes a permutation.

Definition 7.2.7 The function $\psi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ is defined by $\psi(i) = \text{ISA}[\text{SA}[i] + 1]$ for all i with $2 \leq i \leq n$ and $\psi(1) = \text{ISA}[1]$.

The next two lemmata reveal the close relationship between the functions LF and ψ .

Lemma 7.2.8 We have $LF(i) = \text{ISA}[\text{SA}[i] - 1]$ for all i with $\text{SA}[i] \neq 1$ and $LF(i) = 1$ for the index i so that $\text{SA}[i] = 1$.

Proof If $\text{SA}[i] = 1$, then $\text{BWT}[i] = \$$. Since $\$$ occurs at index 1 in the array F , we have $LF(i) = 1$. Now suppose that $\text{SA}[i] \neq 1$. According to Lemma 7.2.4, if $\text{SA}[i] = j$, then $\text{SA}[LF(i)] = j - 1$. So the equation $\text{SA}[LF(i)] = \text{SA}[i] - 1$ holds true. Thus, $LF(i) = \text{ISA}[\text{SA}[i] - 1]$. \square

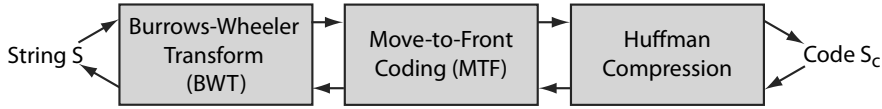


Figure 7.4: The main phases of the bzip2 compression program.

Lemma 7.2.9 *The functions LF and ψ are inverse of each other.*

Proof We will show $LF(\psi(i)) = i$ for all i with $1 \leq i \leq n$. (The equality $\psi(LF(i)) = i$ similarly follows.) If $i = 1$, then $\psi(1) = \text{ISA}[1]$ is the index so that $\text{SA}[\text{ISA}[1]] = 1$. Hence $LF(\psi(1)) = 1$ by Lemma 7.2.8. For $i > 1$, it follows from Lemma 7.2.8 and Definition 7.2.7 that $LF(\psi(i)) = \text{ISA}[\text{SA}[\psi(i)] - 1] = \text{ISA}[\underbrace{\text{SA}[\text{ISA}[\text{SA}[i] + 1]]}_{\text{cancel}} - 1] = \text{ISA}[\text{SA}[i] + 1 - 1] = i$. \square

Exercise 7.2.10 This exercise makes clear that LF can be replaced with ψ in BWT-decoding.

- Modify Algorithm 7.2 so that it computes the ψ -array from BWT. You may assume that the index `index_of_$`, at which the character `$` occurs in the string BWT, is known (it can easily be computed during the Burrows-Wheeler transform).
- Modify Algorithm 7.3 so that it computes the string S from BWT, `index_of_$`, and ψ .
- Show how to compute the suffix array SA from BWT, `index_of_$`, and ψ . Is it possible to overwrite the ψ -array with the suffix array? (This would save space if the ψ -array is no longer needed.)

7.2.3 Data compression

The Burrows-Wheeler transform is used in many lossless data compression programs, of which the best known is Julian Seward's bzip2. Figure 7.4 shows bzip2's main phases. (Its ancestor bzip used arithmetic coding [267] instead of Huffman coding [158]. The change was made because of a software patent restriction.) It is possible to further use a run-length encoder (RLE) in between move-to-front (MTF) and Huffman coding, or to replace MTF with RLE. As a matter of fact, many more variations of the coding scheme are possible. The reader is referred to Adjero et al. [6] for a detailed introduction to the current state of knowledge about data compression with the Burrows-Wheeler transform.

An application of the coding scheme from Figure 7.4 to the string $S = ctatatat\$$ yields the code $S_c = 0111100010111$. The intermediate steps are

$$S = ctatatat\$ \xRightarrow{\text{BWT}} L = ttt\$aaac \xRightarrow{\text{MTF}} R = 300012003 \xRightarrow{\text{Huffman}} S_c = 011110000011101$$

We have already seen how the Burrows-Wheeler transform works, so we now turn to the other two steps: move-to-front and Huffman coding.

Move-to-front coding

Bentley et al. [36] introduced the *move-to-front* transform in 1986 but the method was already described in 1980 by Ryabko; see [269]. The MFT is an encoding of a string designed to improve the performance of entropy encoding techniques of compression like Huffman coding [158] and arithmetic coding [267]. The idea is that each character in the string is replaced by its rank in a list of recently used characters. After a replacement, the character is moved to the front of the list of characters. Algorithm 7.4 makes this precise.

Algorithm 7.4 Move-to-front coding of a string $L \in \Sigma^n$.

```
Initialize a list containing the characters from  $\Sigma$  in increasing order.
for  $i \leftarrow 1$  to  $n$  do
     $R[i] \leftarrow$  number of characters preceding character  $L[i]$  in list
    move character  $L[i]$  to the front of list
```

Figure 7.5 shows the application of Algorithm 7.4 to the string $L = ttt\$aaac$; note that '0' occurs more often in the resulting string R than t or a do in L . As you can see, every *run* (a run is a substring of identical characters) is replaced by a sequence of zeros (except for the first rank). Because a Burrows-Wheeler transformed string usually has many runs, the proportion of zero ranks after MTF has been applied is relatively high.

Pseudo-code for the decoding of the rank vector R is shown in Algorithm 7.5, and Figure 7.6 illustrates the behavior of this algorithm applied to the rank vector $R = 300012003$.

Algorithm 7.5 Move-to-front decoding of R .

```
Initialize a list containing the characters from  $\Sigma$  in increasing order.
for  $i \leftarrow 1$  to  $n$  do
     $L[i] \leftarrow$  character at position  $R[i] + 1$  in list (numbering elements from 1)
    move character  $L[i]$  to the front of list
```

i	$list$	$L[i]$	$R[i]$
1	$\$act$	t	3
2	$t\$ac$	t	0
3	$t\$ac$	t	0
4	$t\$ac$	t	0
5	$t\$ac$	$\$$	1
6	$\$tac$	a	2
7	$a\$tc$	a	0
8	$a\$tc$	a	0
9	$a\$tc$	c	3

Figure 7.5: Move-to-front coding of a string $L = ttt\$aaac$.

i	$list$	$R[i]$	$L[i]$
1	$\$act$	3	t
2	$t\$ac$	0	t
3	$t\$ac$	0	t
4	$t\$ac$	0	t
5	$t\$ac$	1	$\$$
6	$\$tac$	2	a
7	$a\$tc$	0	a
8	$a\$tc$	0	a
9	$a\$tc$	3	c

Figure 7.6: Move-to-front decoding of $R = 300012003$ with $\Sigma = \{\$, a, c, t\}$.

character	'0'	'1'	'2'	'3'
frequency	5/9	1/9	1/9	2/9
Huffman code	1	000	001	01
fixed-length code	00	01	10	11

Figure 7.7: Encoding $R = 300012003$ with a Huffman code takes 15 bits. Encoding it with a fixed-length code would require 18 bits.

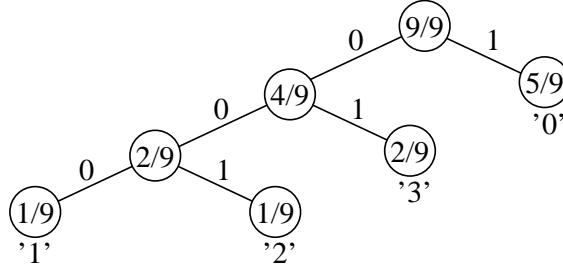
Huffman coding

Huffman coding is an encoding algorithm developed by David A. Huffman [158], which is used for lossless data compression. The algorithm works by creating the so-called Huffman tree and Huffman code in a bottom-up fashion as follows:

1. Initially, there are only leaf nodes, one for each character appearing in the string (file) to be encoded. Besides a character c , a leaf node contains a weight, which equals the frequency of c in the string.
2. The algorithm repeatedly creates a new node whose left child has the smallest weight, whose right child has the second smallest weight (from that point on, these two nodes are no longer considered), and whose weight is the sum of the weights of its children. This is done until only one node remains, the root of the Huffman tree.
3. The codeword of a character c can be read off the path from the root to the leaf that contains c : a 0 means “follow the left child” and a 1 means “follow the right child.”

In this way, a variable-length code—a Huffman code—for encoding characters from the string is obtained. As an example, consider the string $R = 300012003$ on the alphabet $\{'0', '1', '2', '3'\}$ and Figure 7.7. First, leaf '1' (with weight $1/9$) becomes the left child and leaf '2' (with weight $1/9$) becomes the right child of a new node v_1 , whose weight is $2/9$. Second, the node, v_1 (with weight $2/9$) becomes the left child and leaf '3' (with weight $2/9$) becomes the right child of another new node, v_2 , whose weight is $4/9$. Third, the algorithm creates the root of the Huffman tree, whose left child is v_2 and whose right child is the leaf '0'. Figure 7.8 shows the resulting Huffman tree and Figure 7.7 shows the codewords.

Encoding a string is very simple: just replace each character in the string by its codeword. For instance, $R = 300012003$ is encoded by $S_c = 011110000011101$. A Huffman code is a prefix-free code, that is, the codeword representing some particular character is never a prefix of the codeword representing any other character. (Instead of the more accurate

Figure 7.8: Huffman tree and code of $R = 300012003$.

term “prefix-free code,” the term “prefix code” is standard in the literature.) This property makes decoding also simple: the character that is represented by the initial codeword of the encoded string can be read off the path from the root to a leaf in the Huffman tree, where 0 means “go to the left child” and 1 means “go to the right child.” Then, the initial codeword is removed from the encoded string and the decoding process is repeated on the remainder of the encoded string.

For example, the decoding process for $S_c = 011110000011101$ starts at the root of the Huffman tree, goes to the left (since $S_c[1] = 0$) and then to the right (since $S_c[2] = 1$). Because the leaf ‘3’ is encountered, ‘3’ is the character that is represented by the codeword 01. Then, the decoding process continues with 1110000011101, the rest of S_c .

Huffman codes are so important because they are optimal in the sense of Definition 7.2.11; see e.g. [61] for a proof of this fact.

Definition 7.2.11 A prefix-free binary code pc for an alphabet Σ and a frequency function $f : \Sigma \rightarrow [0 \dots 1]$ with $\sum_{c \in \Sigma} f(c) = 1$ is *optimal* if its expected codeword length

$$\sum_{c \in \Sigma} f(c) |pc(c)|$$

is minimum among all prefix-free binary codes for Σ and f .

7.2.4 Direct construction of the BWT

In the last few years, several algorithms have been proposed that construct the BWT either directly or by first constructing the suffix array and then deriving the BWT in linear time from it; see e.g. [173, 206, 254, 293]. The latter approach has a major drawback: all known SACAs require at least $n \log n + n \log \sigma$ bits of main memory ($n \log n$ bits for the suffix array and $n \log \sigma$ bits for the string S). If one has to deal with large datasets, it is

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S		i	m	i	m	m	m	i	s	i	s	m	i	s	i	s	s	i	i	p	i	\$
type		S	L	S	L	L	L	S	L	S	L	L	S	L	S	L	L	S	S	L	L	S

Figure 7.9: Phase II of the induced sorting algorithm.

therefore advantageous to construct the BWT more space efficiently.¹ For example, Okanohara and Sadakane [254] have shown that the induced sorting algorithm devised by Nong et al. [244] (the SACA from Section 4.1.2) can be modified so that it directly constructs the BWT in linear time.

In this section, we discuss (a variant of) the algorithm presented in [254], which we call algorithm BWTbyIS (direct computation of the BWT by induced sorting). Algorithm BWTbyIS shares the same structure with the induced sorting algorithm: it is also divided in two phases. First, we explain how phase II of algorithm BWTbyIS works. We briefly recall phase II of the induced sorting algorithm because this will be the basis of our explanation. The example of Figure 7.9 will serve as an illustration (this is the same example as in Section 4.1.2).

Phase II of the induced sorting algorithm starts with the sequence of sorted LMS-positions (i.e., the order corresponds to the increasing lexicographic order of the suffixes starting at these LMS-positions). In the example of Figure 7.9, this is the sequence 21, 17, 3, 7, 12, 9, 14.

In step 1 of phase II, the sequence of sorted LMS-positions is scanned from right to left (hence in decreasing order) and the positions are moved to their buckets in such a way that they appear in increasing order in the S-type regions of the buckets; see Figure 7.9.

In step 2 of phase II, the array A is scanned from left to right. If, for an element $A[i]$, the position $A[i] - 1$ is of type L (i.e., $T[A[i] - 1] = L$), then $A[i] - 1$ is moved to the current front of its bucket. In the example of Figure 7.9, what happens when the LMS-position 21 is encountered? The position 20 is moved to the current front of the i-bucket because $T[20] = L$ and $S[20] = i$. When position 20 is reached during the scan, position 19 is moved to the current front of the p-bucket because $T[19] = L$ and $S[19] = p$. So step

¹To deal with massive data, one has to resort to external-memory algorithms; see e.g. [30, 99].

2 handles the transitions from LMS-positions to L-positions (LMS→L) as well as the transitions from L-positions to L-positions (L→L). Within a bucket, L→L transitions are dealt with before LMS→L transitions because in a left-to-right scan the L-type region appears before the S-type region.

In step 3 of phase II, the array A is scanned from right to left. If, for an element $A[i]$, the position $A[i] - 1$ is of type S (i.e., $T[A[i] - 1] = S$), then $A[i] - 1$ is moved to the current end of its bucket. In the example of Figure 7.9, what happens when position 19 is encountered? Position 18 is moved to the current end of the i-bucket because $T[18] = S$ and $S[18] = i$. When position 18 is reached during the scan, position 17 is moved to the current end of the i-bucket because $T[17] = S$ and $S[17] = i$. So step 3 handles the transitions from L-positions to S-positions (L→S) as well as the transitions from S-positions to S-positions (S→S). Within a bucket, S→S transitions are dealt with before L→S transitions because in a right-to-left scan the S-type region appears before the L-type region.

Algorithm BWTbyIS does not work with LMS-positions but with LMS-substrings. It starts with the sequence of LMS-substrings *ending* at the sorted LMS-positions. In our example, this is the sequence

iipi\$, issi, \$imi, immmi, ismi, isi, isi

The reader may wonder why the string \$imi appears in this sequence, and not the string imi. Strictly speaking, the string $S[1..3]$ ending at the first LMS-position 3 is not an LMS-string. For reasons that will become clear below, we prepend \$ to this string and say that the resulting string is the LMS-string ending at the first LMS-position of S (i.e., we interpret S as a cyclic string).

In step 1 of phase II, the sequence of sorted LMS-substrings is scanned from right to left. In contrast to the induced sorting algorithm, algorithm BWTbyIS uses queues instead of the buckets. To be precise, for each character $c \in \Sigma$, there is one queue $\text{LMS-queue}[c]$ (which is initially empty). When an LMS-substring uc is encountered during the right-to-left scan, the string u is added to the queue $\text{LMS-queue}[c]$. In our example, we have $\text{LMS-queue}[\$] = [\text{iipi}]$ and $\text{LMS-queue}[i] = [\text{iss}, \$\text{im}, \text{imm}, \text{ism}, \text{is}, \text{is}]$ (the remaining three queues are empty).

Since positions are not available any more, transitions cannot be detected by looking up the types of positions in the type array T , but they can be inferred from the LMS-substrings. To exemplify the idea, let us have a look on the string iipi. Because it is in $\text{LMS-queue}[\$]$, we know that iipi\$ is an LMS-string and that the position at which the character \$ appears is an LMS-position. In what follows, we say that a character is of type L (S, respectively) if the position at which it occurs is of type L (S, respectively). So the character \$ is of type S and the character preceding it is of type L. Given a character c of type L, it is possible to infer the type

transition	implementation
LMS→L	remove from an LMS-queue, add to an L-queue
L→L	remove from an L-queue, add to an L-queue
L→S	remove from an L-stack, add to an S-queue
S→S	remove from an S-queue, add to an S-queue

Figure 7.10: Implementation of the four types of transitions.

of the preceding character b : if $b < c$, then b is of type S; otherwise it is of type L. In our example, this yields

i	i	p	i	\$
	S	L	L	LMS

Once an L→S transition is observed, it is clear from the type structure of an LMS-substring that the types of the remaining characters must be S.

To simulate the transitions in steps 2 and 3 of the induced sorting algorithm, algorithm BWTbyIS employs the following data structures. For each character $c \in \Sigma$, there are two queues $L\text{-queue}[c]$ and $S\text{-queue}[c]$ as well as a stack $L\text{-stack}[c]$ (all of which are initially empty). An LMS→L (L→L, respectively) transition in step 2 is implemented by dequeuing an element from an LMS-queue (L-queue, respectively) and enqueueing an element to an L-queue. When an L→S transition is detected in step 2, it must be postponed to step 3. Furthermore, since step 2 scans from left to right, but step 3 scans from right to left, the order in which L→S transitions are processed must be reversed. That is why L→S transitions are stored in a stack, and not in a queue. In step 3, an L→S (S→S, respectively) transition is implemented by popping an element from an L-stack (dequeuing an element from an S-queue, respectively) and enqueueing an element to an S-queue. Figure 7.10 summarizes the implementation of the transitions.

Now we have all the ingredients to achieve the main goal, namely to compute the BWT. In the algorithm BWTbyIS, the BWT is implemented as an array with the bucket structure known from the induced sorting algorithm. When a character c of type L is processed in step 2, its preceding character b is moved to the current front of the c -bucket of the array BWT (initially, the front of the c -bucket is the index $C[c] + 1$) and the current front of the c -bucket is shifted by one position to the right. Analogously, when a character c of type S is processed in step 3, its preceding character b is moved to the current end of the c -bucket of the array BWT (initially, the end of the c -bucket is the index $C[c + 1]$) and the current end of the c -bucket is shifted by one position to the left. There is one caveat though: when a last character of type S is reached (this is the leftmost character of an initial LMS-substring), the preceding character is not available. The solution to this problem relies on the fact that the position of such a

character is an LMS-position. Recall that phase II of the induced sorting algorithm starts with sorted LMS-positions, and that the relative order of these positions remains unchanged in step 3. In the example of Figure 7.9, the sequence of sorted LMS-positions is 21, 17, 3, 7, 12, 9, 14, and the sequence of characters at the preceding positions 20, 16, 2, 6, 11, 8, 13 is i, s, m, m, m, s, s. In the right-to-left scan of step 3, these characters must be accessed in the reverse order, that is why they are stored on the stack *char-stack*.

Algorithm 7.6 shows pseudo-code of phase II of algorithm BWTbyIS, which runs in linear time. It needs $n \log \sigma$ bits for all LMS-substrings of S , $n \log \sigma$ bits for the BWT, and some auxiliary data structures. The stack *char-stack* can be emulated by using the S-type regions of the array BWT; see Exercise 7.2.13. In step 2, the algorithm merely needs the *front* pointers, the LMS-*queues*, the L-*queues*, and the L-*stacks*. This is the amount of extra memory (besides the $2n \log \sigma$ bits) needed by the algorithm because step 3 requires less memory.

Exercise 7.2.12 Apply Algorithm 7.6 to the example of Figure 7.9.

Exercise 7.2.13 Show how to emulate the stack *char-stack* in Algorithm 7.6 by using the S-type regions of the array BWT. This saves memory. (Note that only the L-type regions of BWT are filled in step 2, and that the S-type regions of BWT are filled from right to left in step 3.)

We now discuss phase I of algorithm BWTbyIS. As in the induced sorting algorithm, the lexicographic names of LMS-substrings can be computed by an application of (a modified version of) Algorithm 7.6 to unsorted LMS-substrings; see Exercise 7.2.17. In contrast to the induced sorting algorithm, however, it is not straightforward to obtain the string \bar{S} (the string that is obtained from S by replacing each LMS-substring with its lexicographic name; recall that the induced sorting algorithm proceeds recursively with \bar{S} unless the LMS-substrings are pairwise distinct). This is because the positions at which the LMS-substrings start are not available. A possible solution would be to store the LMS-substrings in an array ILN so that $\text{ILN}[k]$ is the k -th lexicographically smallest LMS-substring. Then, one determines the LMS-substrings of S from left to right. For each LMS-substring ω encountered, a binary search (see Section 5.1.3) yields the index k with $\text{ILN}[k] = \omega$. This index is the lexicographic name of ω , and it is appended to the growing string \bar{S} . However, this possible solution has a non-linear runtime. Using algorithm engineering techniques, it is possible to implement a space-efficient linear-time algorithm, but we will present a conceptually simpler approach.

Algorithm 7.6 Phase II of the induced sorting algorithm.

```

initialize an empty stack char-stack
for each c in  $\Sigma$  do
    initialize empty queues LMS-queue[c], L-queue[c], and S-queue[c]
    initialize an empty stack L-stack[c]
    front[c]  $\leftarrow C[c] + 1$       /* front of the c-bucket */
    end[c]  $\leftarrow C[c + 1]$       /* end of the c-bucket */

/* step 1: right-to-left scan */
Scan the sorted sequence of LMS-substrings ending at LMS-positions
from right to left. For each LMS-substring uc encountered in the scan,
enqueue the string u to the queue LMS-queue[c].
/* If  $S[i..j] = uc$ , then  $T[i - 1] = L$ ,  $T[i..j - 1] = [S, \dots, S, L, \dots, L]$ ,  $T[j] = S$  */

/* step 2: left-to-right scan, L-type characters before S-type characters */
for each c in  $\Sigma$  do      /* in increasing order */
    while L-queue[c] is not empty do      /* c is L-type */
         $\omega b \leftarrow \text{dequeue}(\text{L-queue}[c])$  /* b is the last character of the string */
        BWT[front[c]]  $\leftarrow b$ 
        front[c]  $\leftarrow \text{front}[c] + 1$ 
        if  $b < c$  then      /* b is S-type */
            push(L-stack[c],  $\omega b$ )      /* store L→S transition */
        else      /* b is L-type */
            enqueue(L-queue[b],  $\omega$ )      /* L→L transition */
        while LMS-queue[c] is not empty do      /* c is S-type */
             $\omega b \leftarrow \text{dequeue}(\text{LMS-queue}[c])$  /* b is L-type */
            push(char-stack, b)      /* store character left of LMS-position */
            enqueue(L-queue[b],  $\omega$ )      /* LMS→L transition */

/* step 3: right-to-left scan, S-type characters before L-type characters */
for each c in  $\Sigma$  do      /* in decreasing order */
    while S-queue[c] is not empty do      /* c is S-type */
        v  $\leftarrow \text{dequeue}(\text{S-queue}[c])$ 
        if  $v = \varepsilon$  then      /* v is the empty string */
            b  $\leftarrow \text{pop}(\text{char-stack})$       /* b precedes c in S, b is L-type */
        else
             $\omega b \leftarrow v$       /* decompose v, its last character b is S-type */
            enqueue(S-queue[b],  $\omega$ )      /* S→S transition */
        BWT[end[c]]  $\leftarrow b$ 
        end[c]  $\leftarrow \text{end}[c] - 1$ 
    while L-stack[c] is not empty do      /* c is L-type */
         $\omega b \leftarrow \text{pop}(\text{L-stack}[c])$       /* b is S-type */
        enqueue(S-queue[b],  $\omega$ )      /* L→S transition */

```

Phase I:

1. In a left-to-right scan of the string S and the type array T , successively determine the LMS-substrings of S and incrementally build the trie of all LMS-substrings (in which the outgoing edges of a node are ordered alphabetically); see Figure 7.11. Mark nodes at which an LMS-substring ends.² As shown in Section 2.5, this takes $O(n)$ time. Furthermore, store the first LMS-position j_1 and count how many different LMS-substrings appear in S ; let m denote this number. Initialize an array $\text{LMS-array}[1..m]$ (in the end, this array will contain the sequence of LMS-substrings with which phase II starts).
- 2a. If all the LMS-substrings are pairwise distinct (in this case m equals the number of LMS-substrings), then proceed as follows: In a post-order traversal of the trie, number the marked nodes from 1 to m in the order of their appearance. In a left-to-right scan of the string S , walk through the trie and compute LMS-substrings as follows: Start with the root of the trie and follow the edge whose label is the first character $S[j_1]$ of the first LMS-substring, then the edge whose label is the second character $S[j_1+1]$ and so on, until a marked node in the trie is reached. Let k_1 be its number. Note that the concatenation of the edge labels on the path from the root to node k_1 spells out the first LMS-substring $S[j_1 \dots j_2]$ of S . Set $\text{LMS-array}[k_1] = S[1 \dots j_1]$.³ Then, start again with the root of the trie and follow the path corresponding to a prefix of $S[j_2 \dots n]$ until a marked node k_2 in the trie is reached. Clearly, the concatenation of the edge labels on the path from the root to node k_2 spells out the second LMS-substring $S[j_2 \dots j_3]$ of S . Set $\text{LMS-array}[k_2] = S[j_1 \dots j_2]$. Again, start at the root of the trie and follow the path corresponding to a prefix of $S[j_3 \dots n]$ until a marked node k_3 in the trie is reached, set $\text{LMS-array}[k_3] = S[j_2 \dots j_3]$, etc. Upon termination of this process, the LMS-array contains the sequence of LMS-substrings with which phase II starts.
- 2b. If not all LMS-substrings are pairwise distinct, then—as in phase I of the induced sorting algorithm—the algorithm must be applied recursively to the string \bar{S} , which is obtained from S by replacing each LMS-substring with its lexicographic name. This string \bar{S} can be computed as follows: Initialize an array $\text{ILN}[1..m]$ (in the end, this will be the inverse of the lexicographic naming, i.e., of the array LN). In a postorder traversal of the trie, number the marked nodes from 1 to m in the order of their appearance (see Figure 7.11) and simultaneously fill the array ILN : if the concatenation of the edge labels on the

²In fact, it suffices to mark the internal nodes at which an LMS-substring ends. For ease of presentation, however, we also mark leaves.

³The first LMS-substring is a special case.

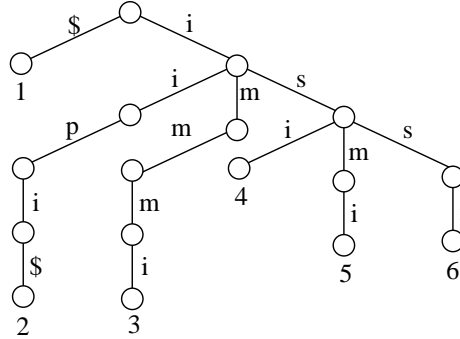


Figure 7.11: The trie of the LMS-substrings \$, iipi\$, immmi, isi, ismi, and issi with the lexicographic names.

current path from the root to node k spells out the LMS-substring ω , then set $\text{ILN}[k] = \omega$ because the lexicographic name of ω is k . In a left-to-right scan of the string S (starting at the first LMS-position j_1), walk through the trie and compute LMS-substrings as in step (2a). Whenever a marked node is encountered during this process, append its number to the string \bar{S} (initially, $\bar{S} = \varepsilon$). Then, recursively compute the Burrows-Wheeler transform BWT of the string \bar{S} . Using BWT , fill the LMS-array: for i from 1 to n do

$$\text{LMS-array}[i] = \begin{cases} \$S[1 \dots j_1] & \text{if } \text{ILN}[\overline{\text{BWT}}[i]] = \$ \\ \text{ILN}[\text{BWT}[i]] & \text{otherwise} \end{cases}$$

Exercise 7.2.14 Show that algorithm BWTbyIS runs in linear time.

Exercise 7.2.15 Apply phase I of algorithm BWTbyIS to the example from Figure 7.9 (page 292).

Exercise 7.2.16 Explain why the marked nodes of the trie are numbered in a postorder traversal (and not in a preorder traversal).
Hint: Use an example in which an LMS-substring is a proper prefix of another LMS-substring.

Exercise 7.2.17 Modify Algorithm 7.6 in such a way that it computes the lexicographic names of (initially unsorted) LMS-substrings.

BWT	<i>t</i>	<i>c</i>	<i>a</i>	§	<i>a</i>	<i>t</i>	<i>c</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
B^s	0	0	0	1	0	0	0	0	0	0	0
B^a	0	0	1	0	1	0	0	1	1	1	1
B^c	0	1	0	0	0	0	1	0	0	0	0
B^t	1	0	0	0	0	1	0	0	0	0	0

Figure 7.12: Indicator bit vectors of BWT = *tca\$atcaaaa*.

7.3 Backward search

Ferragina and Manzini [100] showed that it is possible to search a pattern $P = P[1..m]$ backwards in the suffix array SA of string S , without storing SA. A backward search means that we first search for the $P[m]$ -interval, then for the $P[m-1..m]$ -interval, and so on, until the whole pattern $P[1..m]$ is found. In the computer science literature, any data structure that allows to search a pattern P backwards in the (conceptual) suffix array of a string S is called an *FM-index* of S . Before showing how a backward search works, we introduce a simple FM-index consisting of the C -array and certain indicator bit vectors. In Section 7.4 we will become acquainted with another FM-index: the wavelet tree.

7.3.1 A simple FM-index

Definition 7.3.1 Given a string (text) T of length n on the alphabet Σ ,

- $\text{rank}_c(T, i)$ returns the number of occurrences of character $c \in \Sigma$ in the prefix $T[1..i]$,
- $\text{select}_c(T, i)$ returns the position of the i -th occurrences of character $c \in \Sigma$ in T .

It what follows, we are interested in data structures that support these kinds of queries efficiently. Since we are mainly interested in the Burrows-Wheeler transform of a string S , we fix $T = \text{BWT}$. However, the techniques developed below work for arbitrary strings T .

The easiest method to support $\text{rank}_c(\text{BWT}, i)$ and $\text{select}_c(\text{BWT}, i)$ queries is to use σ many indicator bit vectors of length n . For each character $c \in \Sigma$, the bit vector B^c is defined by $B^c[i] = 1$ if and only if $\text{BWT}[i] = c$; see Figure 7.12. Clearly, $\text{rank}_c(\text{BWT}, i) = \text{rank}_1(B^c, i)$ and $\text{select}_c(\text{BWT}, i) = \text{select}_1(B^c, i)$. Therefore, the problem is reduced to the problem of answering rank and select queries on bit vectors. This can be done in constant time with a total of $n\sigma + o(n\sigma)$ bits of space.

Given the ability to answer $\text{rank}_c(\text{BWT}, i)$ and $\text{select}_c(\text{BWT}, i)$ queries in constant time, it is possible to compute $\text{LF}(i)$ and $\psi(i)$ in constant time as well. This can be seen as follows. According to Definition 7.2.3, if

F	\$	a	a	a	a	a	a	c	c	t	t
B_F	1	1	0	0	0	0	0	1	0	1	0

Figure 7.13: The bit vector B_F for $F = \$aaaaaacctt$.

$\text{BWT}[i] = c$ is the k -th occurrence of character c in the BWT-array, then $LF(i) = j$ is the index so that $F[j]$ is the k -th occurrence of c in the array F . Furthermore, we have seen that the k -th occurrence of c in F can be found at index $C[c] + k$. It follows as a consequence that

$$LF(i) = C[c] + \text{rank}_c(\text{BWT}, i), \text{ where } c = \text{BWT}[i] \quad (7.1)$$

Note that in the computer science literature, $\text{Occ}(c, i)$ is often used instead of $\text{rank}_c(\text{BWT}, i)$.

Because the ψ -function is the inverse of the LF -mapping (Lemma 7.2.9), it follows that if $F[i] = c$ is the k -th occurrence of c in the array F , then $\psi(i) = j$ is the index so that $\text{BWT}[j] = c$ is the k -th occurrence of character c in the BWT-array. So once we know c and k , $\psi(i) = j$ can be obtained by $j = \text{select}_c(\text{BWT}, k)$. In fact, it is sufficient to know c because $k = i - C[c]$. Clearly, c can be obtained from F because $F[i] = c$. However, storing F would be a waste of memory because we can use the array C instead. By doing a binary search on C , we can determine in $O(\log \sigma)$ time the character c with $C[c] < i \leq C[c + 1]$, i.e., $c = \max\{a \in \Sigma \mid C[a] < i\}$. Alternatively, c can be determined in constant time with a *rank* data structure on the bit vector B_F defined by $B_F[1] = 1$ and, for all l with $2 \leq l \leq n$, $B_F[l] = 1$ if and only if $F[l - 1] \neq F[l]$; see Figure 7.13 for an example. This is because $c = \Sigma[m]$, where $m = \text{rank}_1(B_F, i)$. All in all, we have

$$\psi(i) = \text{select}_c(\text{BWT}, i - C[c]), \text{ where } c = \Sigma[\text{rank}_1(B_F, i)] \quad (7.2)$$

Note that the array C can be completely replaced with the bit vector B_F because $C[c] = \text{select}_1(B_F, m) - 1$.

In summary, if we use the indicator bit vectors and the bit vector B_F , then $LF(i)$ and $\psi(i)$ can be computed in constant time, using $n(1 + \sigma) + o(n(1 + \sigma))$ bits of space. If we use the C -array instead of the bit vector B_F , then $LF(i)$ can also be computed in constant time, but the computation of $\psi(i)$ takes $O(\log \sigma)$ time. In this case, $\sigma \log n + n\sigma + o(n\sigma)$ bits of space are required.

Exercise 7.3.2 Give a linear-time algorithm that takes the string BWT as input and returns the bit vector B_F .

7.3.2 The search algorithm

Let us return to the issue of backward search. As already mentioned, a backward search means that we first search for the $P[m]$ -interval, then for

i	BWT	$S_{SA[i]}$
1	t	$\$$
→ 2	c	$aaacatat\$$
3	a	$aacatat\$$
4	$\$$	$acaaacatat\$$
5	a	$acatat\$$
6	t	$at\$$
→ 7	c	$atat\$$
8	a	$caaacatat\$$
9	a	$catat\$$
10	a	$t\$$
11	a	$tat\$$

i	BWT	$S_{SA[i]}$
1	t	$\$$
→ 2	c	$aaacatat\$$
→ 3	a	$aacatat\$$
4	$\$$	$acaaacatat\$$
5	a	$acatat\$$
6	t	$at\$$
7	c	$atat\$$
8	a	$caaacatat\$$
9	a	$catat\$$
10	a	$t\$$
11	a	$tat\$$

Figure 7.14: Searching pattern aa backwards in $S = acaaacatat\$$. Given the a -interval $[2..7]$, one backward search step determines the aa -interval $[i..j]$ by $i = C[a] + \text{rank}_a(\text{BWT}, 2 - 1) + 1 = 1 + 0 + 1 = 2$ and $j = C[a] + \text{rank}_a(\text{BWT}, 7) = 1 + 2 = 3$.

the $P[m-1..m]$ -interval, and so on, until the whole pattern $P[1..m]$ is found. For example, the a -interval in the suffix array of the string $S = acaaacatat\$$ is $[2..7]$; see Figure 7.14. That is, $S_{SA[2]}, S_{SA[3]}, \dots, S_{SA[7]}$ are the only suffixes in S that start with an a . Consequently, if we search for the suffixes starting with aa , then $S_{SA[2]-1}, S_{SA[3]-1}, \dots, S_{SA[7]-1}$ are the sole candidates because only these suffixes have an a at the second position. Note that these candidates can be found in the suffix array at $LF(2), LF(3), \dots, LF(7)$. Out of these candidates only those that have an a at first position belong to the aa -interval. Because $S[SA[i] - 1] = a$ if and only if $\text{BWT}[i] = a$, the suffix $S_{SA[i]-1}$ at index $LF(i)$ belongs to the aa -interval if and only if $S_{SA[i]}$ belongs to the a -interval and $\text{BWT}[i] = a$. As a matter of fact, it suffices to know the first index p and the last index q with $2 \leq p \leq q \leq 7$ and $\text{BWT}[p] = a = \text{BWT}[q]$. (This is because the suffixes $S_{SA[2]}, S_{SA[3]}, \dots, S_{SA[7]}$ are ordered lexicographically and if one prepends the same character to all of them, then the resulting strings will occur in the same lexicographic order.) In our example, we have $p = 3$ and $q = 5$. Hence the boundaries of the aa -interval are $LF(3) = 2$ and $LF(5) = 3$. The crucial question is how to find p and q efficiently. Observe that a linear scan of the BWT array would result in a bad worst-case running time. In fact, we do not have to know p and q , as we shall see below.

Suppose in general that we know the ω -interval $[i..j]$ of some suffix ω of P , say $\omega = P[b..m]$. Next, we have to determine the $c\omega$ -interval, where

Algorithm 7.7 Given an ω -interval $[i..j]$ and a character c , this procedure returns the $c\omega$ -interval if it exists; otherwise, it returns \perp .

```

backwardSearch( $c, [i..j]$ )
     $i \leftarrow C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1$ 
     $j \leftarrow C[c] + \text{rank}_c(\text{BWT}, j)$ 
    if  $i \leq j$  then
        return interval  $[i..j]$ 
    else
        return  $\perp$ 

```

$c = P[b - 1]$. Assume for a moment that the $c\omega$ -interval is non-empty, i.e., $c\omega$ is a substring of S . Let p and q be the smallest and largest index with $i \leq p \leq q \leq j$ and $\text{BWT}[p] = c = \text{BWT}[q]$. As discussed above, the $c\omega$ -interval is the interval $[LF(p)..LF(q)]$. According to Equation 7.1 (page 300) we have

$$\begin{aligned}
 LF(p) &= C[c] + \text{rank}_c(\text{BWT}, p) \\
 &= C[c] + \text{rank}_c(\text{BWT}, p - 1) + 1 \\
 &= C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1
 \end{aligned}$$

where the last equality follows from the fact that p is the index of the first occurrence of c in $\text{BWT}[i..j]$. Analogously,

$$\begin{aligned}
 LF(q) &= C[c] + \text{rank}_c(\text{BWT}, q) \\
 &= C[c] + \text{rank}_c(\text{BWT}, j)
 \end{aligned}$$

because q is the index of the last occurrence of c in $\text{BWT}[i..j]$. We conclude that the $c\omega$ -interval $[C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1..C[c] + \text{rank}_c(\text{BWT}, j)]$ can be determined without knowing p and q . Pseudo-code for one backward search step can be found in Algorithm 7.7. In the preceding discussion, we assumed that the $c\omega$ -interval is non-empty. What happens if it is empty? Then, $\text{rank}_c(\text{BWT}, i - 1) = \text{rank}_c(\text{BWT}, j)$. This implies that $C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1 > C[c] + \text{rank}_c(\text{BWT}, j)$ and thus Algorithm 7.7 returns the undefined value \perp .

Pseudo-code for searching the whole pattern P is given in Algorithm 7.8.

Exercise 7.3.3 Show that backward search can be accomplished in $O(m \log n)$ time, solely based on the ψ -array of S (cf. Definition 7.2.7).

Algorithm 7.8 Given a pattern P , this procedure returns the P -interval if it exists; otherwise, it returns \perp .

backwardSearch(P)

```

 $i \leftarrow 1$ 
 $j \leftarrow n$ 
 $k \leftarrow m$ 
while  $i \leq j$  and  $k \geq 1$  do
     $c \leftarrow P[k]$ 
     $i \leftarrow C[c] + \text{rank}_c(\text{BWT}, i - 1) + 1$ 
     $j \leftarrow C[c] + \text{rank}_c(\text{BWT}, j)$ 
     $k \leftarrow k - 1$ 
if  $i \leq j$  then
    return interval  $[i..j]$ 
else
    return  $\perp$ 

```

7.4 Wavelet trees

The *wavelet tree* was introduced by Grossi et al. [134]. In a very general sense, a wavelet tree is a binary tree⁴ that has exactly σ many leaves and there is a bijection between the set of leaves and Σ (i.e., each of the leaves corresponds to a distinct character from the alphabet Σ). Moreover, every internal node v stores a bit vector B^v equipped with rank and select data structures.

The conceptually easiest way to introduce wavelet trees goes as follows. We say that an interval $[l..r]$ is an *alphabet interval* if it is a subinterval of $[1..\sigma]$, where $\sigma = |\Sigma|$. For an alphabet interval $[l..r]$, the string $\text{BWT}^{[l..r]}$ is obtained from the Burrows-Wheeler transformed string BWT of S by deleting all characters in BWT that do not belong to the subalphabet $\Sigma[l..r]$ of $\Sigma[1..\sigma]$. As an example, consider the string $\text{BWT} = tca\$atcaaaa$ and the alphabet interval $[1..2]$. The string $\text{BWT}^{[1..2]}$ is obtained from $tca\$atcaaaa$ by deleting the characters c and t . Thus, $\text{BWT}^{[1..2]} = a\$aaaaa$. Each node v of the tree corresponds to a string $\text{BWT}^{[l..r]}$, where $[l..r]$ is an alphabet interval. The root of the tree corresponds to the string $\text{BWT} = \text{BWT}^{[1..\sigma]}$. If $l = r$, then v has no children. Otherwise, v has two children: its left child v_L corresponds to the string $\text{BWT}^{[l..m]}$ and its right child v_R corresponds to the string $\text{BWT}^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$. In this case, v stores a bit vector, denoted by B^v or $B^{[l..r]}$, whose i -th entry is 0 if the i -th character in $\text{BWT}^{[l..r]}$ belongs to the subalphabet $\Sigma[l..m]$ and 1 if it belongs to the subalphabet $\Sigma[m+1..r]$. To put it differently, an entry in the bit vector is 0 if the corresponding character belongs to the left subtree and 1 if it

⁴That is, every node in the tree is either a leaf or has exactly two children.

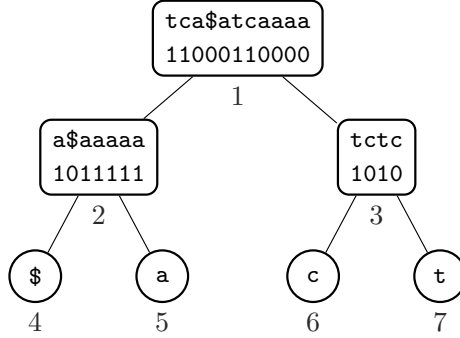


Figure 7.15: Conceptual illustration of the wavelet tree of the string $\text{BWT} = \text{tca\$atcaaaa}$. Only the bit vectors are stored; the corresponding strings are shown for clarity.

belongs to the right subtree; see Figure 7.15. Each bit vector B^v in the tree is preprocessed so that rank and select queries can be answered in constant time. Because the wavelet tree has height $\lceil \log \sigma \rceil$ (the simple proof of this fact is left to the reader), the wavelet tree essentially uses $n \lceil \log \sigma \rceil + o(n \log \sigma)$ bits of space. We shall see in Section 7.4.4 that two additional bit vectors, each of size $\leq \sigma$, are required.

To devise a simple construction algorithm for the wavelet tree is rather straightforward and left to the reader. A more space efficient (but slower) construction algorithm can be found e.g. in [311].

We would like to point out that the leaves of the wavelet tree described above appear in alphabetic order. More precisely, the k -th leaf visited in a depth-first traversal of the wavelet tree corresponds to the character $\Sigma[k]$. This is important in some applications (e.g. bidirectional search).

7.4.1 Answering *rank* and *select* queries

The queries $\text{rank}_c(\text{BWT}, i)$ and $\text{select}_c(\text{BWT}, i)$ can be answered by a traversal of the wavelet tree. We exemplify this by computing $\text{rank}_a(\text{BWT}, 7)$ and $\text{select}_t(\text{BWT}, 2)$ on the wavelet tree of $\text{BWT} = \text{tca\$atcaaaa}$ of Figure 7.15.

To compute $\text{rank}_a(\text{BWT}, 7)$, we start at the root (node 1 in Figure 7.15) and move down the tree as detailed in Algorithm 7.9. Because a belongs to the subinterval $\Sigma[1..2]$ of the ordered alphabet Σ , the occurrences of a correspond to zeros in the bit vector at the root, and they go to the left child; this is node 2 in Figure 7.15. Now the number of a 's in $\text{BWT}^{[1..4]} = \text{tca\$atcaaaa}$ up to (and including) position 7 equals the number of a 's in the string $\text{BWT}^{[1..2]} = \text{a\$aaaaa}$ up to position $\text{rank}_0(B^{[1..4]}, 7)$. So we compute $\text{rank}_0(B^{[1..4]}, 7) = 3$. Because a belongs to the subinterval $\Sigma[2..2]$ of Σ , the

Algorithm 7.9 For a character c , an index i , and an alphabet interval $[l..r]$, the procedure $\text{rank}_c(\text{BWT}, i, [l..r])$ returns the number of occurrences of c in the string $\text{BWT}^{[l..r]}[1..i]$, unless $l = r$ (in this case, it returns i).

```

 $\text{rank}_c(\text{BWT}, i, [l..r])$ 
  if  $l = r$  then return  $i$ 
  else
     $m = \lfloor \frac{l+r}{2} \rfloor$ 
    if  $c \leq \Sigma[m]$  then
      return  $\text{rank}_c(\text{BWT}, \text{rank}_0(B^{[l..r]}), i, [l..m])$ 
    else
      return  $\text{rank}_c(\text{BWT}, \text{rank}_1(B^{[l..r]}), i, [m+1..r])$ 

```

occurrences of a correspond to ones in the bit vector at node 2, and they go to the right child; this is node 5 in Figure 7.15. The number of a 's in the string $\text{BWT}^{[1..2]} = a\$aaaaa$ up to position 3 equals the number of a 's in $\text{BWT}^{[2..2]} = aaaaaa$ up to position $\text{rank}_1(B^{[1..2]}, 3) = 2$. Since $\text{BWT}^{[2..2]}$ consists solely of a 's, the number of a 's in $\text{BWT}^{[2..2]}$ up to position 2 is 2. All in all, $\text{rank}_a(\text{BWT}, 7) = 2$.

To compute $\text{select}_t(\text{BWT}, 2)$, we start at the leaf corresponding to t (node 7 in Figure 7.15) and move up the tree. Since this leaf is the right child of its parent (node 3 in Figure 7.15), the second t in $\text{BWT}^{[3..4]} = tctc$ can be found at position $\text{select}_1(B^{[3..4]}, 2) = 3$. The node 3 is the right child of its parent node 1, so the position of the second t in $\text{BWT}^{[1..4]} = tca\$atcaaaa$ is $\text{select}_1(B^{[1..4]}, 3) = 6$. Since node 1 is the *root*, we conclude that the second occurrence of t in $\text{BWT} = tca\$atcaaaa$ can be found at position 6.

In fact, the wavelet tree of the string BWT allows us to determine $\text{BWT}[i]$ *without accessing* BWT itself. (Thus, the string BWT need not be stored.) The procedure is very similar to that of $\text{rank}_c(\text{BWT}, i)$. For example, to determine the fourth character of the string BWT we must follow the path from the root to the leaf corresponding to the character $\text{BWT}[4]$. Because $B^{[1..4]}[4] = 0$, we move from the root to its left child (node 2 in Figure 7.15) and now seek the character at position $\text{rank}_0(B^{[1..4]}, 4) = 2$ in the string $\text{BWT}^{[1..2]} = a\$aaaaa$. Now $B^{[1..2]}[2] = 0$ tells us to move to the left child and search for the character at position $\text{rank}_0(B^{[1..2]}, 2) = 1$ in the string $\text{BWT}^{[1..1]} = \$$. Because this is a leaf, we report the character corresponding to this leaf, which in our example is $\$$.

It should be stressed that the procedures to compute $\text{rank}_c(\text{BWT}, i)$, $\text{select}_c(\text{BWT}, i)$, and $\text{BWT}[i]$ solely use the bit vectors equipped with rank and select data structures. Their worst-case time complexity is $O(\log \sigma)$ because the height of the wavelet tree is $O(\log \sigma)$.

Exercise 7.4.1 We have seen in Section 7.3 that backward search can be done with the C -array and $\text{rank}_c(\text{BWT}, i)$ queries. The wavelet tree of

the BWT supports the latter in $O(\log \sigma)$ time. Show that $C[c]$ can also be computed in $O(\log \sigma)$ time on the wavelet tree. This means that we do not need the C -array when the wavelet tree is available. Argue that in practice one would nevertheless use the C -array.

7.4.2 Retrieval of $SA[i]$ and the string starting at $SA[i]$

If the suffix array SA is available, then $SA[i]$ can be determined in constant time. However, if we want to save space, we cannot afford to keep the whole suffix array in main memory. Instead, we can use the sparse suffix array SSA of S . Recall that Algorithm 6.2 (page 264) used the ψ -function to retrieve $SA[i]$ from the sparse suffix array SSA . It is not difficult to see, however, that the retrieval of $SA[i]$ can also be done with the LF -mapping (this is left as an exercise for the reader). In practice, one should prefer LF over ψ because the former is implemented with *rank*-queries, whereas the latter uses *select*-queries, and *rank*-queries can be answered more quickly than *select*-queries. According to Equation 7.1 (page 300), we have

$$LF(i) = C[c] + \text{rank}_c(\text{BWT}, i), \text{ where } c = \text{BWT}[i]$$

and the wavelet tree suffices to compute $LF(i)$ in $O(\log \sigma)$ time because all three values $C[c]$, $\text{rank}_c(\text{BWT}, i)$, and $\text{BWT}[i]$ can be computed on it within that time bound. Therefore, it takes $O(s \log \sigma)$ time in the worst case to retrieve $SA[i]$ from the sparse suffix array SSA , where s is the sampling parameter.

In some applications, it is necessary to output the length ℓ substring of S starting at position $SA[i]$, but only the index i is known. Clearly, one can retrieve $SA[i]$ from the sparse suffix array SSA and then output $S[SA[i]..SA[i] + \ell - 1]$ by accessing S itself. However, this substring can also be retrieved without S and without the (sparse) suffix array of S . First of all, note that $S[SA[i]] = F[i]$, $S[SA[i] + 1] = F[\psi(i)]$, $S[SA[i] + 2] = F[\psi(\psi(i))]$, etc. In other words, $S[SA[i]..SA[i] + \ell - 1]$ coincides with the string $F[i] F[\psi(i)] \dots F[\psi^{\ell-1}(i)]$. If we use the bit vector B_F of F as defined in Section 7.3.1, then the character $S[SA[i]] = F[i]$ can be calculated in constant time by $m = \text{rank}_1(B_F, i)$ and $c = \Sigma[m]$. Moreover, by Equation 7.2 (page 300)

$$\psi(i) = \text{select}_c(\text{BWT}, i - (\text{select}_1(B_F, m) - 1))$$

and this value can be calculated in $O(\log \sigma)$ on the wavelet tree. So it takes $O(\ell \log \sigma)$ time to compute $S[SA[i]..SA[i] + \ell - 1]$ in that way. Of course, it is possible to replace the bit vector B_F with the C -array; see Exercise 7.4.2.

Exercise 7.4.2 Show that $S[SA[i]..SA[i] + \ell - 1]$ can be retrieved in $O(\ell \log \sigma)$ time using the C -array and the wavelet tree of the BWT of S . Devise an algorithm for the same task that solely uses the wavelet tree and analyze its worst-case time complexity.

Algorithm 7.10 For character c and position i , procedure $\text{rank}_c(\text{BWT}, i)$ returns the number of occurrences of c in BWT up to (and including) i .

```

let  $h = \log \sigma$  be the height of the wavelet tree
let  $A[1..2^h - 1]$  be the array of bit vectors
 $l \leftarrow 1$ 
 $r \leftarrow \sigma$ 
 $j \leftarrow 1$  /*  $j$  is root */
while  $j \leq 2^h - 1$  do /* while  $j$  is not a leaf */
     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
    if  $c \leq \Sigma[m]$  then /* go to left child */
         $i \leftarrow \text{rank}_0(A[j], i)$ 
         $j \leftarrow 2j$ 
         $r \leftarrow m$ 
    else /* go to right child */
         $i \leftarrow \text{rank}_1(A[j], i)$ 
         $j \leftarrow 2j + 1$ 
         $l \leftarrow m + 1$ 
return  $i$ 

```

7.4.3 Implementation: If σ is a power of 2

If $\sigma = 2^h$ for some natural number h , then the wavelet tree can be implemented by a perfect binary tree, which is defined as follows.

Definition 7.4.3 A *perfect binary tree* is a rooted tree in which every internal node has exactly two children and all leaves are at the same depth (same level).

Clearly, a perfect binary tree of height h has 2^h leaves and $s_I = 2^h - 1$ internal nodes. Hence its size is $s = 2^{h+1} - 1$. It is well known that such a perfect binary tree can be represented by an array $A[1..s]$. The root of the tree can be found at index 1 and, given the index j of a node, the index of its

- left child is $2j$,
- right child is $2j + 1$,
- parent is $\lfloor \frac{j}{2} \rfloor$.

In a wavelet tree, only the internal nodes carry information, namely the bit vectors. Therefore, we use an array $A[1..s_I]$ of size $s_I = 2^h - 1$ to store them. More precisely, we store the bit vector $B^{[1..2^h]}$ at $A[1]$, $B^{[1..2^{h-1}]}$ at $A[2]$, $B^{[2^{h-1}+1..2^h]}$ at $A[3]$, and so on. Algorithm 7.10 gives pseudo-code for the

Algorithm 7.11 For a position i , this procedure returns $\text{BWT}[i]$.

```

let  $h = \log \sigma$  be the height of the wavelet tree
let  $A[1..2^h - 1]$  be the array of bit vectors
 $j \leftarrow 1$  /*  $j$  is root */
while  $j \leq 2^h - 1$  do /* while  $j$  is not a leaf */
    if  $A[j][i] = 0$  then /* go to left child */
         $i \leftarrow \text{rank}_0(A[j], i)$ 
         $j \leftarrow 2j$ 
    else /* go to right child */
         $i \leftarrow \text{rank}_1(A[j], i)$ 
         $j \leftarrow 2j + 1$ 
return  $\Sigma[j - (2^h - 1)]$ 

```

computation of $\text{rank}_c(\text{BWT}, i)$. The algorithm starts at the root node with the alphabet interval $[1..\sigma]$. If it reaches a node $j \leq 2^h - 1$ with the alphabet interval $[l..r]$, then it tests whether $c \leq \Sigma[m]$, where $m = \lfloor \frac{l+r}{2} \rfloor$. If so, it must proceed with the left child $2j$ of j . Thus, it sets i to $\text{rank}_0(A[j], i)$ —notice that $A[j] = B^{[l..r]}$ —and r to m . Otherwise, it must proceed with the right child $2j + 1$ of j , so it sets i to $\text{rank}_1(A[j], i)$ and l to $m + 1$. At some point in time, it must reach a leaf, i.e., the current node j satisfies $2^h \leq j \leq 2^{h+1} - 1$. In this case $l = r$ because the wavelet tree is a perfect binary tree. Consequently, the algorithm returns the correct value i .

The wavelet tree of Figure 7.15 (page 304) is a perfect binary tree of height $h = 2$. Its bit vectors are stored in the array A : $A[1] = B^{[1..4]}$, $A[2] = B^{[1..2]}$, and $A[3] = B^{[3..4]}$. Algorithm 7.10 computes $\text{rank}_a(\text{BWT}, 7)$ as follows: In the first iteration of the while-loop, it computes $m = \lfloor \frac{1+4}{2} \rfloor = 2$. Since $a \leq \Sigma[2] = a$, it then determines $i = \text{rank}_0(A[1], 7) = 3$ and moves to the left child of the root by setting $j = 2 \cdot 1$. Furthermore, r is set to 2. In the second iteration of the while-loop, $m = \lfloor \frac{1+2}{2} \rfloor = 1$. Now $a > \Sigma[1] = \$$, so the algorithm computes $i = \text{rank}_1(A[2], 3) = 2$ and moves to the right child by setting $j = 2 \cdot 2 + 1$. Since $j = 5$ is greater than $2^h - 1 = 3$, Algorithm 7.10 returns $i = 2$.

The computation of $\text{BWT}[i]$ on the wavelet tree proceeds in a similar fashion; see Algorithm 7.11. If it reaches a node $j \leq 2^h - 1$ with the alphabet interval $[l..r]$, then $A[j] = B^{[l..r]}$. If $B^{[l..r]}[i] = 0$, then the character we are searching for is in the left subtree of j ; otherwise, it is in the right subtree. When the algorithm reaches a leaf, the current node j satisfies $2^h \leq j \leq 2^{h+1} - 1$. Since the characters of Σ occur in alphabetic order at the leaves $2^h, 2^h + 1, \dots, 2^{h+1} - 1$, the current node j corresponds to the character $\Sigma[j - (2^h - 1)]$. Again, we use the example from Figure 7.15 and compute $\text{BWT}[4]$ by Algorithm 7.11. In the first iteration of the while-loop, we have $A[1][4] = 0$. Therefore, the algorithm computes $i =$

Algorithm 7.12 For a character c and a natural number i , the procedure $select_c(\text{BWT}, i)$ returns the position of the i -th occurrence of c in BWT.

```

let  $h = \log \sigma$  be the height of the wavelet tree
let  $c = \Sigma[k]$  be the  $k$ -th character in  $\Sigma$ 
 $j \leftarrow 2^h - 1 + k$ 
while  $j > 1$  do
  if  $j$  is even then      /*  $j$  is left child */
     $b \leftarrow 0$ 
  else                  /*  $j$  is right child */
     $b \leftarrow 1$ 
   $j \leftarrow \lfloor \frac{j}{2} \rfloor$  /* go to parent */
   $i \leftarrow select_b(A[j], i)$ 
return  $i$ 

```

$rank_0(A[1], 4) = 2$ and moves to the left child of the root by setting $j = 2 \cdot 1$. In the second iteration of the while-loop, we have $A[2][2] = 0$. So the algorithm determines $i = rank_0(A[2], 2) = 1$ and moves to the left child by setting $j = 2 \cdot 2$. Because $j = 4$ is greater than $2^h - 1 = 3$, Algorithm 7.11 returns $\Sigma[j - (2^h - 1)] = \Sigma[1] = \$$.

It remains for us to address the problem of computing $select_c(\text{BWT}, i)$. Algorithm 7.12 starts at the leaf j corresponding to the character $c = \Sigma[k]$ and walks upwards to the root of the tree. Because the wavelet tree is a perfect binary tree, the characters of Σ occur in alphabetic order at the leaves $2^h, 2^h + 1, \dots, 2^{h+1} - 1$. Thus, the algorithm starts at the leaf $j = 2^h - 1 + k$. If j is even, then it is the left child of its parent node $\lfloor \frac{j}{2} \rfloor$ and the algorithm sets $b = 0$ to remember this. Otherwise, j is the right child of its parent node; so $b = 1$. Then, the algorithm moves to the parent node and updates i appropriately. When it reaches the root node $j = 1$, it returns i . To illustrate this, let us compute $select_t(\text{BWT}, 2)$ in the example from Figure 7.15. Algorithm 7.12 starts at the leaf $j = 2^h - 1 + 4 = 7$ because t is the fourth character of the alphabet Σ . Since $j = 7$ is odd in the first iteration of the while-loop, it is the right child of its parent and the algorithm visits the parent node by the assignment $j \leftarrow \lfloor \frac{7}{2} \rfloor$. Moreover, i gets the new value $select_1(A[3], 2) = 3$. In the second iteration of the while-loop, $j = 3$ is odd again. Therefore, Algorithm 7.12 moves to the parent node by the assignment $j \leftarrow \lfloor \frac{3}{2} \rfloor$ and i gets the new value $select_1(A[1], 3) = 6$. Then the while-loop terminates and the algorithm returns $i = 6$.

Exercise 7.4.4 Show that $rank_c(\text{BWT}, i)$ and $\text{BWT}[i]$ can be computed simultaneously.

7.4.4 Implementation: If σ is not a power of 2

The following lemma implies that navigation in a wavelet tree for σ characters, where σ is not a power of 2, can be based on the results of the previous section.

Lemma 7.4.5 *If S is a string having σ distinct characters, where $2^{h-1} < \sigma < 2^h$, then the wavelet tree for S is full up to (and including) level $h - 1$.*

Proof We prove the lemma by induction on h . In the base case $h = 2$, we have $\sigma = 3$ because $2 = 2^1 < \sigma < 2^2 = 4$. It is readily verified that the binary tree is full up to level 1. In the inductive step, we prove the lemma for $h > 2$. If σ is even, then both the left and the right subtree of the root represent a string with $\frac{\sigma}{2}$ distinct characters and $2^{h-2} < \frac{\sigma}{2} < 2^{h-1}$. According to the inductive hypothesis, these subtrees are full up to level $h - 2$. So the whole tree is full up to level $h - 1$. If σ is odd, then the left subtree of the root represents a string with $\frac{\sigma+1}{2}$ distinct characters, whereas the right subtree represents a string with $\frac{\sigma-1}{2}$ distinct characters. In case of the left subtree, we have $2^{h-2} < \frac{\sigma+1}{2} \leq 2^{h-1}$. If $\frac{\sigma+1}{2} = 2^{h-1}$, then the left subtree is full up to level $h - 1$. Otherwise the inductive hypothesis tells us that it is full up to level $h - 2$. In case of the right subtree, we have $2^{h-2} \leq \frac{\sigma-1}{2} < 2^{h-1}$ and again the tree is full up to level $h - 2$. As above, we conclude that whole tree is full up to level $h - 1$. \square

We have seen that the height of the wavelet tree of a string with σ distinct characters is $h = \lceil \log \sigma \rceil$. According to Lemma 7.4.5, the wavelet tree is full up to (and including) level $\lfloor \log \sigma \rfloor$. If σ is a power of two, these two values coincide because then the wavelet tree is a perfect binary tree. It was already shown that perfect trees are easy to handle, so from now on we assume that σ is not a power of two. Consequently, the wavelet tree is full up to and including level $h - 1$, but it is not full at level h . So at level $h - 2$ there are solely internal nodes, but at level $h - 1$ there is at least one leaf. Figure 7.16 shows an example. If s denotes the size of the tree, then $s_I = s - \sigma$ is the number of internal nodes. Again, the wavelet tree is implemented as an array $A[1..s_I]$ of bit vectors, where each bit vector $A[j]$ is equipped with rank and select data structures. The bit vectors corresponding to the internal nodes up to and including level $h - 2$ are stored in the usual order in $A[1..2^{h-1} - 1]$ and those at level $h - 1$ are stored in left-to-right order in $A[2^{h-1}..s_I]$. In our example from Figure 7.16, the bit vectors at the nodes 1 through 7 are stored in $A[1]$ through $A[7]$. The bit vectors at nodes 8, 10, and 12 are stored in $A[8]$, $A[9]$, and $A[10]$.

Algorithm 7.13 computes $\text{rank}_c(\text{BWT}, i)$. Since the wavelet tree is full up to and including level $h - 1$, the algorithm navigates as in a perfect tree until the current node j is on level $h - 1$ of the tree. If the current alphabet interval $[l..r]$ is a singleton interval (i.e., $l = r$), then j is a leaf and the

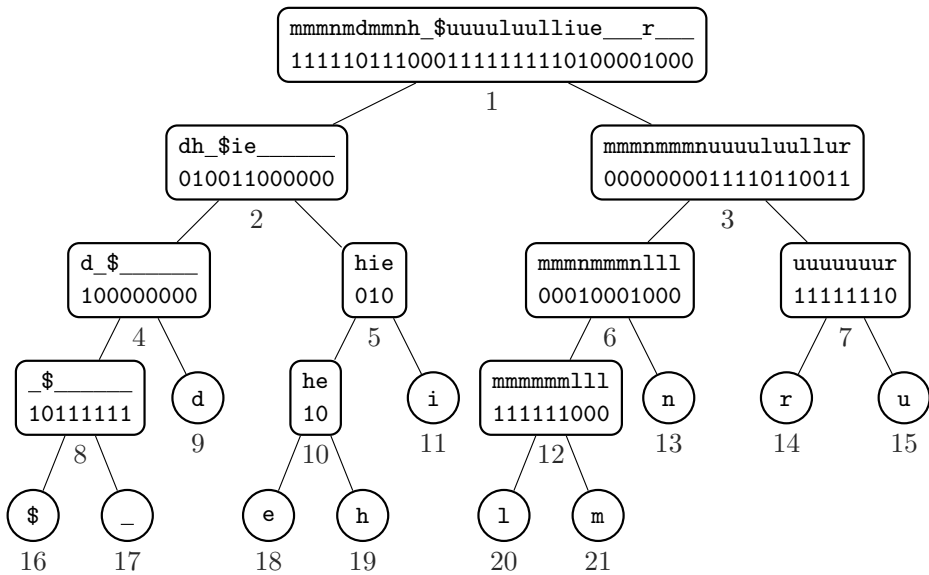


Figure 7.16: Conceptual illustration of the wavelet tree of the BWT of the string `in_ulm_um_ulm_und_um_ulm_herum$`. Only the bit vectors are stored; the strings are shown for clarity only.

Algorithm 7.13 For a character c and a position i , the procedure $\text{rank}_c(\text{BWT}, i)$ returns the number of occurrences of c in BWT up to (and including) i .

```

let  $s_I$  be the number of internal nodes in the wavelet tree
let  $A[1..s_I]$  be the array of bit vectors
let  $h = \lceil \log \sigma \rceil$  be the height of the wavelet tree
let  $B_{h-1}[1..2^{h-1}]$  keep the information about the nodes at level  $h - 1$ 
 $s_{h-1} \leftarrow 2^{h-1} - 1$ 
 $l \leftarrow 1$ 
 $r \leftarrow \sigma$ 
 $j \leftarrow 1$       /*  $j$  is root */
while  $j \leq s_{h-1}$  do      /* while  $j$  is not at level  $h - 1$  */
     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
    if  $c \leq \Sigma[m]$  then      /* go to left child */
         $i \leftarrow \text{rank}_0(A[j], i)$ 
         $j \leftarrow 2j$ 
         $r \leftarrow m$ 
    else      /* go to right child */
         $i \leftarrow \text{rank}_1(A[j], i)$ 
         $j \leftarrow 2j + 1$ 
         $l \leftarrow m + 1$ 
if  $l \neq r$  then      /* in this case  $l + 1 = r$  */
     $q \leftarrow \text{rank}_1(B_{h-1}, j - s_{h-1})$ 
    if  $c = \Sigma[l]$  then      /* go to left child */
         $i \leftarrow \text{rank}_0(A[s_{h-1} + q], i)$ 
    else      /* go to right child */
         $i \leftarrow \text{rank}_1(A[s_{h-1} + q], i)$ 
return  $i$ 

```

algorithm returns the current value of i . (Note that in this case there is no difference to the implementation on a perfect binary tree.) Otherwise, if $l \neq r$, then j is an internal node having two leaves as children. Clearly, if $c = \Sigma[l]$, then the algorithm must proceed with the left child of j ; otherwise $c = \Sigma[r]$ and the algorithm must proceed with the right child of j . To determine the correct value i , it must be able to find the bit vector of node j in the subarray $A[s_{h-1} + 1..s_I]$, where $s_{h-1} = 2^{h-1} - 1$ is the size of the tree up to level $h - 2$. To this end, the information about the nodes at level $h - 1$ is kept in a bit vector $B_{h-1}[1..2^{h-1}]$. To be precise, $B_{h-1}[j - s_{h-1}] = 1$ if and only if j is an internal node. Furthermore, the bit vector B_{h-1} is equipped with a rank data structure. Now, if $B_{h-1}[j - s_{h-1}] = 1$, then the bit vector corresponding to the internal node j can be found in constant time at index $s_{h-1} + \text{rank}_1(B_{h-1}, j - s_{h-1})$. Let us illustrate this with example

Algorithm 7.14 For a position i , this procedure returns $\text{BWT}[i]$.

```

let  $s_I$  be the number of internal nodes in the wavelet tree
let  $A[1..s_I]$  be the array of bit vectors
let  $h = \lceil \log \sigma \rceil$  be the height of the wavelet tree
let  $B_{h-1}[1..2^{h-1}]$  keep the information about the nodes at level  $h - 1$ 
 $s_{h-1} \leftarrow 2^{h-1} - 1$ 
 $j \leftarrow 1$  /*  $j$  is root */
while  $j \leq s_{h-1}$  do /* while  $j$  is not at level  $h - 1$  */
    if  $A[j][i] = 0$  then /* go to left child */
         $i \leftarrow \text{rank}_0(A[j], i)$ 
         $j \leftarrow 2j$ 
    else /* go to right child */
         $i \leftarrow \text{rank}_1(A[j], i)$ 
         $j \leftarrow 2j + 1$ 
 $k \leftarrow j - s_{h-1}$ 
 $q \leftarrow \text{rank}_1(B_{h-1}, k)$ 
if  $B_{h-1}[k] = 1$  and  $A[s_{h-1} + q][i] = 0$  then
    return  $\Sigma[k + q - 1]$ 
else
    return  $\Sigma[k + q]$ 

```

Figure 7.16. In this example, the bit vector B_{h-1} is 10101000. Algorithm 7.13 computes $\text{rank}_e(\text{BWT}, 25)$ by following the path from the root to node $j = 10$ as in Algorithm 7.10. At that node, the current values are $l = 4$, $r = 5$, and $i = 2$. The algorithm further calculates $q = \text{rank}_1(B_{h-1}, j - s_{h-1}) = \text{rank}_1(10101000, 10 - 7) = 2$ and returns $i = \text{rank}_0(A[7 + 2], 2) = 1$.

In order to compute $\text{BWT}[i]$, Algorithm 7.14 starts at the root and walks down the tree until a node j at level $h - 1$ is reached. Let $k = j - s_{h-1}$. If $B_{h-1}[k] = 0$, then j is a leaf: it corresponds to the character $\Sigma[k + q]$, where $q = \text{rank}_1(B_{h-1}, k)$. This is because every leaf (every 0-bit in B_{h-1}) corresponds to one character and every internal node at level $h - 1$ (every 1-bit in B_{h-1}) corresponds to two characters. Otherwise, if $B_{h-1}[k] = 1$, then j is an internal node. Its left child corresponds to the character $\Sigma[k + q - 1]$ and its right child corresponds to the character $\Sigma[k + q]$. So if $A[s_{h-1} + q][i] = 0$, then Algorithm 7.14 returns $\Sigma[k + q - 1]$; otherwise it returns $\Sigma[k + q]$. Again, we use the example of Figure 7.16 to illustrate this. Algorithm 7.14 computes $\text{BWT}[20]$ by following the path to node $j = 12$. At that node, the current values are $l = 7$, $r = 8$, and $i = 8$. The algorithm further calculates $k = j - s_{h-1} = 12 - 7 = 5$ and $q = \text{rank}_1(B_{h-1}, k) = \text{rank}_1(10101000, 5) = 3$. Since $B_{h-1}[k] = B_{h-1}[5] = 1$ and $A[s_{h-1} + q][i] = A[7 + 3][8] = A[10][8] = 0$, Algorithm 7.14 returns $\Sigma[k + q - 1] = \Sigma[5 + 3 - 1] = \Sigma[7] = l$.

Algorithm 7.15 For a character c and a number i , the procedure $select_c(\text{BWT}, i)$ returns the position of the i -th occurrence of c in BWT.

```

let  $c$  be the  $k$ -th character in  $\Sigma$ 
 $B_\Sigma[1..\sigma]$  keeps information about the level at which characters occur
 $s_{h-1} \leftarrow 2^{h-1} - 1$ 
 $q \leftarrow rank_1(B_\Sigma, k)$ 
 $j \leftarrow s_{h-1} + k - \lfloor \frac{q}{2} \rfloor$ 
if  $B_\Sigma[k] = 1$  then
    if  $q$  is odd then          /* left child */
         $b \leftarrow 0$ 
    else                      /*  $j$  is right child */
         $b \leftarrow 1$ 
     $i \leftarrow select_b(A[s_{h-1} + \lceil \frac{q}{2} \rceil], i)$ 
while  $j > 1$  do
    if  $j$  is even then        /*  $j$  is left child */
         $b \leftarrow 0$ 
    else                      /*  $j$  is right child */
         $b \leftarrow 1$ 
     $j \leftarrow \lfloor \frac{j}{2} \rfloor$     /* go to parent */
     $i \leftarrow select_b(A[j], i)$ 
return  $i$ 

```

To implement the *select* operation on the wavelet tree, we must be able to find the leaf corresponding to a character $c = \Sigma[k]$. For this reason, we introduce a bit vector $B_\Sigma[1..\sigma]$ defined by $B_\Sigma[k] = 0$ if the character $c = \Sigma[k]$ occurs at level $h - 1$, and $B_\Sigma[k] = 1$ if it occurs at level h . Note that B_Σ can be obtained from B_{h-1} by doubling every 1-bit in B_{h-1} . Algorithm 7.15 can determine the number $q = rank_1(B_\Sigma, k)$ of 1-bits in B_Σ up to and including position k in constant time provided that the bit vector B_Σ is equipped with a rank data structure. Furthermore, the algorithm sets $j = s_{h-1} + k - \lfloor \frac{q}{2} \rfloor$. If $B_\Sigma[k] = 0$, then character $c = \Sigma[k]$ occurs at node j . Otherwise, j is the parent node of the leaf c . More precisely, if q is odd, then the leaf c is the left child of j and j represents the string $\text{BWT}^{[k..k+1]}$. In this case, let $b = 0$, $l = k$, and $r = k + 1$. Otherwise, if q is even, then the leaf c is the right child of j and j represents the string $\text{BWT}^{[k-1..k]}$. In this case, let $b = 1$, $l = k - 1$, and $r = k$. Now, $i = select_b(A[s_{h-1} + \lceil \frac{q}{2} \rceil], i)$ is the position of the i -th occurrence of c in $\text{BWT}^{[l..r]}$. From that point on, the algorithm navigates as in the perfect binary tree. In the example of Figure 7.16, the bit vector B_Σ is 11011011000. Suppose we wish to compute $select_(\text{BWT}, 2)$, the position of the second occurrence of $_$ in BWT. Note that $_$ is the second character of Σ ; so $k = 2$. Algorithm 7.15 computes $q = rank_1(B_\Sigma, k) = rank_1(11011011000, 2) = 2$ and $j = s_{h-1} + k - \lfloor \frac{q}{2} \rfloor = 7 + 2 - 1 =$

8. Because $B_{\Sigma}[k] = B_{\Sigma}[2] = 1$ and $q = 2$ is even, the algorithm further calculates $i = \text{select}_1(A[s_{h-1} + \lceil \frac{q}{2} \rceil], 2) = \text{select}_1(A[8], 2) = 3$. From that point on, Algorithm 7.15 behaves exactly as Algorithm 7.12.

7.4.5 Other types of wavelet trees

We have barely scratched the surface of the potential of wavelet trees. The reader can find more information about the subject in recent publications, summarized e.g. in [237]. We confine ourselves to the following remarks.

In the previous section, we have seen that the wavelet tree essentially uses $n \log \sigma + o(n \log \sigma)$ bits to support the operations *rank*, *select*, and *access* in $O(\log \sigma)$ time (*access* provides access to a character $\text{BWT}[i]$). Golynski et al. [127] showed that within the same space bound the operations *rank* and *access* can be supported in $O(\log \log \sigma)$ time, and *select* in $O(1)$ time. Nevertheless, in subsequent analyses we will always use $O(\log \sigma)$ as an upper bound because we stick to wavelet trees.

It is possible to use the Huffman tree of the BWT as the basis of the wavelet tree. The resulting *Huffman shaped wavelet tree* is often the best in practice; see [124]. However, it has two disadvantages. First, it is not balanced, so its height is not bounded by $O(\log \sigma)$ but by $O(\sigma)$. To solve this problem, one can force the Huffman tree to be balanced after depth $(1+d) \log \sigma$, where d is some natural number; see [212]. Second, the leaves generally do not occur in alphabetic order. In some applications (e.g. bidirectional search), however, the alphabet order must be preserved. To remedy this problem, one can use the prefix-free code of Hu and Tucker [157] instead of the Huffman code.

Another variant is the *weight-balanced wavelet tree* devised by Hon et al. [155], in which the number of 0's is made almost equal to the number of 1's in the bit vector of each internal node. The depth of a weight-balanced wavelet tree can be $\log n$, so the operations *rank*, *select*, and *access* take $O(\log n)$ time in the worst-case.

7.5 Analyzing a string space efficiently

7.5.1 Construction of the LCP-array from the BWT

In this section, we introduce a LACA (LCP-array construction algorithm) that constructs the LCP-array directly from the BWT [33]. As usual, we assume that the string S (hence the BWT) contains every character from the alphabet Σ . The algorithm relies on a generalization of backward search: for an ω -interval $[i..j]$, the procedure *getIntervals*([$i..j$]) presented in Algorithm 7.16 returns the list of all $c\omega$ -intervals, where $c \in \Sigma \setminus \{\$$. More precisely, it starts with the ω -interval $[i..j]$ at the root and traverses the

Algorithm 7.16 For an ω -interval $[i..j]$, $getIntervals([i..j])$ returns the list of all $c\omega$ -intervals, where $c \in \Sigma \setminus \{\$ \}$.

```

getIntervals([i..j])
  list  $\leftarrow$  []
  getIntervals'([i..j], [1.. $\sigma$ ], list)
  return list

getIntervals'([i..j], [l..r], list)
  if  $l = r$  then
     $c \leftarrow \Sigma[l]$ 
    if  $c \neq \$$  then
      add(list, [ $C[c] + i..C[c] + j$ ])
  else
     $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, i - 1), rank_0(B^{[l..r]}, j))$ 
     $(a_1, b_1) \leftarrow (i - 1 - a_0, j - b_0)$ 
     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$ 
    if  $b_0 > a_0$  then
      getIntervals'([ $a_0 + 1..b_0$ ], [l..m], list)
    if  $b_1 > a_1$  then
      getIntervals'([ $a_1 + 1..b_1$ ], [ $m + 1..r$ ], list)

```

wavelet tree in a depth-first manner as follows: At the current node v , it uses constant time rank queries to obtain the number $b_0 - a_0$ of zeros in the bit vector of v within the current interval. If $b_0 > a_0$, then there are characters in $BWT[i..j]$ that belong to the left subtree of v , and the algorithm proceeds recursively with the left child v_L of v . Furthermore, if the number of ones is positive (i.e., if $b_1 > a_1$), then it proceeds with the right child v_R in an analogous fashion. (In Algorithm 7.16, the left child v_L of v corresponds to the string $BWT^{[l..m]}$ and the right child v_R corresponds to the string $BWT^{[m+1..r]}$, where $m = \lfloor \frac{l+r}{2} \rfloor$; cf. Section 7.4.) Clearly, if a leaf corresponding to character c is reached with current interval $[p..q]$, then $[C[c] + p .. C[c] + q]$ is the $c\omega$ interval. In this way, Algorithm 7.16 computes the list of all $c\omega$ -intervals, where $c \in \Sigma \setminus \{\$ \}$. This takes $O(k \log \sigma)$ time for a k -element list. Because the wavelet tree has less than 2σ nodes, $O(\sigma)$ is another upper bound for the wavelet tree traversal. Consequently, Algorithm 7.16 has a worst-case time complexity of $O(\min\{\sigma, k \log \sigma\})$, where k is the number of elements in the output list.

For example, if we apply Algorithm 7.16 to the i -interval $[2..5]$ in Figure 7.18, then it returns a list containing the mi -interval $[6..6]$, the pi -interval $[7..7]$, and the si -interval $[9..10]$. This works as follows. The procedure call $getIntervals([2..5])$ results in the procedure call $getIntervals'([2..5], [1..5], [])$. That is, the traversal of the wavelet tree in Figure 7.17 starts at the root

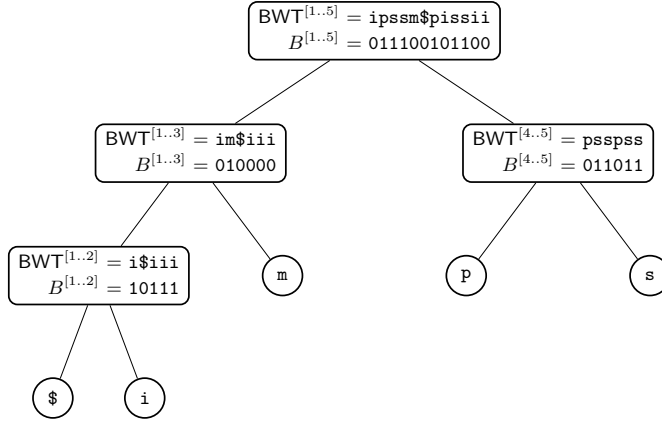


Figure 7.17: The wavelet tree of the BWT of the string *mississippi\$*.

and

$$\begin{aligned}(a_0, b_0) &= (\text{rank}_0(B^{[1..5]}, 2 - 1), \text{rank}_0(B^{[1..5]}, 5)) = (1, 2) \\ (a_1, b_1) &= (2 - 1 - a_0, 5 - b_0) = (0, 3)\end{aligned}$$

is calculated. Because $b_0 = 2 > 1 = a_0$, the algorithm proceeds recursively with the left child v_L of the root; i.e., $\text{getIntervals}'([2..2], [1..3], [])$ is invoked and

$$\begin{aligned}(a_0^L, b_0^L) &= (\text{rank}_0(B^{[1..3]}, 2 - 1), \text{rank}_0(B^{[1..3]}, 2)) = (1, 1) \\ (a_1^L, b_1^L) &= (2 - 1 - a_0^L, 2 - b_0^L) = (0, 1)\end{aligned}$$

is computed. Since $b_0^L = 1 = a_0^L$, the algorithm does not proceed with the left child of v_L . However, it proceeds recursively with the right child of v_L because $b_1^L = 1 > 0 = a_1^L$. When $\text{getIntervals}'([1..1], [3..3], [])$ is executed, the interval $[C[m] + 1..C[m] + 1] = [6..6]$ is added to $\text{list} = []$ because $l = 3 = r$ and $\Sigma[3] = m$. Then, the computation continues with the right child v_R of the root since $b_1 = 3 > 0 = a_1$; i.e., $\text{getIntervals}'([1..3], [4..5], [[6..6]])$ is invoked and

$$\begin{aligned}(a_0^R, b_0^R) &= (\text{rank}_0(B^{[4..5]}, 1 - 1), \text{rank}_0(B^{[1..3]}, 3)) = (0, 1) \\ (a_1^R, b_1^R) &= (1 - 1 - a_0^R, 3 - b_0^R) = (0, 2)\end{aligned}$$

is calculated. Because $b_0^R = 1 > 0 = a_0^R$ and $b_1^R = 2 > 0 = a_1^R$, the algorithm proceeds recursively with both children of v_R . During the execution of $\text{getIntervals}'([1..1], [4..4], [[6..6]])$, the interval $[C[p] + 1..C[p] + 1] = [7..7]$ is added to the list as $l = 4 = r$ and $\Sigma[4] = p$. Then, $\text{getIntervals}'([1..2], [5..5], [[6..6], [7..7]])$

i	LCP	BWT	$S_{SA[i]}$
1	-1	i	$\$$
2	0 \cancel{A}	p	$i\$$
3	\perp	s	$ippi\$$
4	\perp	s	$issippi\$$
5	\perp	m	$ississippi\$$
6	0 \cancel{A}	$\$$	$mississippi\$$
7	0 \cancel{A}	p	$pi\$$
8	\perp	i	$ppi\$$
9	0 \cancel{A}	s	$sippi\$$
10	\perp	s	$ssippi\$$
11	\perp	i	$ssippi\$$
12	\perp	i	$ssissippi\$$
13	-1		

Figure 7.18: BWT of $S = mississippi\$$ and its partially filled LCP-array.

is executed and the interval $[C[s] + 1..C[s] + 2] = [9..10]$ is added to the list because $l = 5 = r$ and $\Sigma[5] = s$. Consequently, $getIntervals([2..5])$ returns the list $[[6..6], [7..7], [9..10]]$.

Algorithm 7.17 shows how the LCP-array of a string S can be obtained solely based on the wavelet tree of the BWT of S . At the very beginning, $LCP[1]$ and $LCP[n + 1]$ are set to -1 . The algorithm maintains a queue Q , which initially contains the c -interval for each $c \in \Sigma$ (including $c = \$$). Moreover, the variable ℓ stores the current lcp-value (initially, $\ell = 0$) and $size$ memorizes how many elements there are in Q that correspond to the current lcp-value (initially, $size = \sigma$). Algorithm 7.17 computes lcp-values in increasing order (first the 0 entries, then the 1 entries, and so on) as follows: Whenever it dequeues an element from Q , say the ωa -interval $[lb..rb]$ where $|\omega| = \ell$ and $a \in \Sigma$, it tests whether $LCP[rb + 1] = \perp$. If so, it assigns ℓ to $LCP[rb + 1]$, generates all non-empty ωa -intervals and adds them to (the back of) Q . Otherwise, it does nothing.

Let us illustrate the algorithm by computing all 0 entries in the LCP-array of our example in Figure 7.18. Initially, $\ell = 0$ and the queue Q contains the $\$$ -interval $[1..1]$, the i -interval $[2..5]$, the m -interval $[6..6]$, the p -interval $[7..8]$ and the s -interval $[9..12]$. At that point, the $size$ of Q is 5; so five intervals correspond to the lcp-value $\ell = 0$. The first interval that is pulled from the queue is the $\$$ -interval $[1..1]$ and $size$ is decreased by one. Since $LCP[1 + 1] = \perp$, case 1 in Algorithm 7.17 applies. Thus, $LCP[2]$ is set to $\ell = 0$ and the $i\$$ -interval $[2..2]$, which is the only interval in the list returned by $getIntervals([1..1])$, is added to the queue. Next, the

Algorithm 7.17 Computation of the LCP-array.

```

initialize the array LCP[1..n + 1]      /* i.e., LCP[i] = ⊥ for all i */
LCP[1] ← −1
LCP[n + 1] ← −1
ℓ ← 0
initialize an empty queue Q
for each c in Σ do                  /* including c = $ */
    enqueue(Q, [C[c] + 1..C[c + 1]]) /* the c-interval */
size ← σ                               /* initial size of Q */
while Q is not empty do
    if size = 0 then
        ℓ ← ℓ + 1
        size ← |Q|                    /* current size of Q */
        [lb..rb] ← dequeue(Q)
        size ← size − 1
    if LCP[rb + 1] = ⊥ then             /* case 1 */
        LCP[rb + 1] ← ℓ
        list ← getIntervals([lb..rb])
        for each [i..j] in list do
            enqueue(Q, [i..j])
    else nothing to do                  /* case 2 */

```

i-interval [2..5] is dequeued (and *size* is decreased by one). Again, case 1 applies because LCP[5 + 1] = ⊥. So LCP[6] is set to ℓ = 0, *getIntervals*([2..5]) returns the list [[6..6], [7..7], [9..10]], and the intervals in the list are added to the queue *Q*. When the *m*-interval [6..6] is dequeued, *size* is decreased by one, LCP[7] is set to ℓ = 0, but no new interval is added to the queue (observe that *getIntervals* does not generate the \$*m*-interval because \$*m* is not a substring of *S*). Then the *p*-interval is dequeued, *size* is decreased by one, LCP[9] is set to 0, and the intervals [3..3] and [8..8] (the *ip*- and the *pp*-interval) are enqueued. Finally, the *s*-interval [9..12] is pulled from the queue. Again, *size* is decreased by one; it now has the value 0. Because LCP[12 + 1] = −1, case 2 in Algorithm 7.17 applies. In the next iteration of the while-loop, ℓ is increased by one and *size* is set to the current size 6 of *Q*. At that point in time the six elements in *Q* are intervals that corresponds to the lcp-value ℓ = 1 are

$$[2..2]_{i\$}, [6..6]_{mi}, [7..7]_{pi}, [9..10]_{si}, [3..3]_{ip}, [8..8]_{pp}$$

where the notation [*lb*..*rb*]_ω indicates that the interval [*lb*..*rb*] is the ω-interval. The reader is invited to compute all 1 entries in the LCP-array by executing the algorithm by hand.

Theorem 7.5.1 *Algorithm 7.17 correctly computes the LCP-array.*

Proof We proceed by induction on ℓ . In the base case, we have $\ell = 0$. Initially, the queue Q contains the c -interval $[lb..rb]$ for every character $c = \Sigma[k]$, where $lb = C[c] + 1$ and $rb = C[d]$ with $d = \Sigma[k + 1]$. The algorithm sets $LCP[rb + 1] = 0$ unless $rb = n$. This is certainly correct because the suffix $S_{SA[rb]}$ starts with the character c and the suffix $S_{SA[rb+1]}$ starts with the character d . Clearly, the LCP-array contains all entries with value 0. Let $\ell > 0$. By the inductive hypothesis, we may assume that Algorithm 7.17 has correctly computed all lcp-values $< \ell$ and the queue Q solely contains intervals that corresponds to the lcp-value ℓ . Let the ωa -interval $[lb..rb]$ be in Q , where $|\omega| = \ell$ and $a \in \Sigma$. If $LCP[rb + 1] = \perp$, then we know from the induction hypothesis that $LCP[rb + 1] \geq \ell$, i.e., ω is a common prefix of the suffixes $S_{SA[rb]}$ and $S_{SA[rb+1]}$. On the other hand, ωa is a prefix of $S_{SA[rb]}$ but not of $S_{SA[rb+1]}$. Consequently, ω is the longest common prefix of $S_{SA[rb]}$ and $S_{SA[rb+1]}$. Hence Algorithm 7.17 assigns the correct value ℓ to $LCP[rb + 1]$.

We still have to prove that all entries of the LCP-array with value ℓ are really set. So let k , $0 \leq k < n$, be an index with $LCP[k + 1] = \ell$. Since $\ell > 0$, the longest common prefix of $S_{SA[k]}$ and $S_{SA[k+1]}$ can be written as $c\omega$, where $c \in \Sigma$, $\omega \in \Sigma^*$, and $|\omega| = \ell - 1$. Let $c\omega a$ be the length $\ell + 1$ prefix of $S_{SA[k]}$ and $c\omega d$ be the length $\ell + 1$ prefix of $S_{SA[k+1]}$, where $a \neq d$. Because ωa is a prefix of $S_{SA[k+1]}$ and ωd is a prefix of $S_{SA[k+1]+1}$, it follows that ω is the longest common prefix of $S_{SA[k+1]}$ and $S_{SA[k+1]+1}$. Let $[i..j]$ be the ω -interval, p be the index with $SA[p] = SA[k] + 1$, and q be the index with $SA[q] = SA[k + 1] + 1$. Clearly, $i \leq p < q \leq j$. Let t , $p < t \leq q$, be the smallest index at which the corresponding suffix does not start with ωa ; see Figure 7.19. Consequently, $LCP[t] = |\omega| = \ell - 1$. According to the inductive hypothesis, Algorithm 7.17 assigns the value $\ell - 1$ to $LCP[t]$. Therefore, *getIntervals* is called with the ωa -interval $[s..t - 1]$. Since ωa is a prefix of $S_{SA[p]}$ and $BWT[p] = c$, it follows that the $c\omega a$ -interval, say $[lb..rb]$, is not empty. Moreover, $rb = k$ because $BWT[r] \neq c$ for all $p < r < q$. Thus, $[lb..k]$ is in the *list* returned by *getIntervals* $([s..t - 1])$. Hence it is added to the queue Q . At some point in time, $[lb..k]$ will be removed from Q and $LCP[k + 1]$ will be set to ℓ . \square

Theorem 7.5.2 *Algorithm 7.17 has a worst-case time complexity of $O(n \log \sigma)$.*

Proof We use an amortized analysis to prove that each of the cases 1 and 2 can occur at most n times. Case 1 occurs as often as an entry of the LCP-array is filled, and this happens exactly $n - 1$ times. It remains for us to analyze how often case 2 can occur. We claim that for a fixed position j , $1 \leq j \leq n$, there is at most one substring $\omega = S[i..j]$ ending at j for which the ω -interval $[lb..rb]$ belongs to case 2. If i is the largest position with $\omega = S[i..j]$ so that the ω -interval $[lb..rb]$ belongs to case 2, then none

i	$\text{LCP}[i]$	$\text{BWT}[i]$	$S_{\text{SA}[i]}$
\vdots	\vdots	\vdots	\vdots
p		c	$\omega a \dots$
\vdots			$\omega a \dots$
t	$\ell - 1$		$\omega b \dots$
\vdots			
q		c	$\omega d \dots$
\vdots			\vdots
k			$c \omega a \dots$
$k + 1$	ℓ		$c \omega d \dots$
\vdots	\vdots	\vdots	\vdots

Figure 7.19: Correctness of Algorithm 7.17.

of the left-extensions of ω is generated. More precisely, none of the ω' -intervals, where $\omega' = S[i'..j]$ with $1 \leq i' < i$, will be enqueued. This proves the claim. As there are only n possibilities for j , it follows that case 2 also occurs at most n times. In summary, the procedure *getIntervals* can create at most $2n$ intervals because every interval belongs to exactly one case. Each interval can be generated in $O(\log \sigma)$ time, so the runtime of Algorithm 7.17 is $O(n \log \sigma)$. \square

Algorithm 7.17 can be implemented space efficiently by writing LCP-entries to disk; see [33].

7.5.2 Bottom-up traversal of the lcp-interval tree

As we have seen in previous chapters, several sequence analysis problems can be solved by a bottom-up traversal of the lcp-interval tree. Now we have the tools to implement such a traversal space efficiently:

- construct the BWT directly (e.g. by the algorithm from Section 7.2.4),
- construct the LCP-array with Algorithm 7.17 (page 319), and
- apply Algorithm 4.6 (page 94).

Because Algorithm 4.6 accesses the LCP-array sequentially, the LCP-array can be streamed from disk. It follows as a consequence that the algorithms based on a bottom-up traversal of the lcp-interval tree can be implemented space efficiently.

7.5.3 Shortest unique substrings

Recall that a substring of S is unique if it occurs exactly once in S , and that the shortest unique substring problem is to find all shortest unique substrings of S . In Section 5.6.5 an algorithm was given that solves this problem. Here, we provide an alternative algorithm.

In the current context, S is terminated by the special symbol $\$$. It follows that every suffix of S is unique. Since we are not interested in these, we exclude them. In the following, we assume that—apart from $\$$ — S contains at least two different characters (if $S = a^{n-1}\$$, then a^{n-1} is the shortest unique substring because every other substring that is not a suffix of S occurs repeatedly).

In Section 5.6.5, it was shown that the length $\ell + 1$ prefix of $S_{\text{SA}[i]}$, where $\ell = \max\{\text{LCP}[i], \text{LCP}[i + 1]\}$, is the shortest unique substring of S that starts at position $\text{SA}[i]$. Using this observation, we can modify Algorithm 7.17 so that it computes a shortest unique substring of S . The resulting Algorithm 7.18 can easily be changed so that it computes all shortest unique substrings or even all unique substrings of S . We make use of the fact that Algorithm 7.17 computes lcp-values in ascending order. So when Algorithm 7.17 executes the statement $\text{LCP}[rb + 1] \leftarrow \ell$ and $\text{LCP}[rb]$ has been previously set, then $\max\{\text{LCP}[rb], \text{LCP}[rb + 1]\} = \ell$ and $S[\text{SA}[rb].. \text{SA}[rb] + \ell]$ is the shortest unique substring of S that starts at position $\text{SA}[rb]$. Analogously, if $\text{LCP}[rb + 2]$ has been previously set, then $\max\{\text{LCP}[rb + 1], \text{LCP}[rb + 2]\} = \ell$ and $S[\text{SA}[rb + 1].. \text{SA}[rb + 1] + \ell]$ is the shortest unique substring of S that starts at position $\text{SA}[rb + 1]$. Because the current value of ℓ is always available, all we have to know is whether or not $\text{LCP}[rb]$ ($\text{LCP}[rb + 2]$, respectively) has been previously computed. Consequently, we can replace the LCP-array with a bit vector B_{LCP} of length n , and $B_{\text{LCP}}[i]$ is set to one instead of assigning a value to $\text{LCP}[i]$. As mentioned earlier, we have to exclude suffixes of S in the above process. This can be done by keeping track of the suffix of length $\ell + 1$, where ℓ is the current length. To be precise, initially $\ell = 0$ and the suffix of length 1 is the character $\$$, which appears at index $idx = 1$. Every time ℓ is incremented, we obtain the index of the suffix of length $\ell + 1$ by the assignment $idx \leftarrow LF(idx)$ (recall that the wavelet allows us to compute $LF(i)$ in $O(\log \sigma)$ time). Consequently, a unique substring at index rb is output only if $rb \neq idx$.

Exercise 7.5.3 Analyze the worst-case time complexity of Algorithm 7.18.

Exercise 7.5.4 Let S be a string that is terminated by the sentinel $\$$ and let the alphabet Σ consist of the characters appearing in S . A string $\omega \in (\Sigma \setminus \{\$\})^+$ is called an *absent word* if it is not a substring of S . Devise an algorithm that takes S as input and computes all shortest absent words using the technique developed above.

Algorithm 7.18 Computation of a shortest unique substring.

```

initialize a bit array  $B_{\text{LCP}}[1..n+1]$       /* i.e.,  $B_{\text{LCP}}[i] = 0$  for all  $i$  */
 $B_{\text{LCP}}[1] \leftarrow 1$ 
 $B_{\text{LCP}}[n+1] \leftarrow 1$ 
 $\ell \leftarrow 0$ 
 $idx \leftarrow 1$ 
initialize an empty queue  $Q$ 
for each  $c$  in  $\Sigma$  do      /* including  $c = \$$  */
     $enqueue(Q, [C[c] + 1..C[c+1]])$       /* the  $c$ -interval */
 $size \leftarrow \sigma$       /* initial size of  $Q$  */
while  $Q$  is not empty do
    if  $size = 0$  then
         $\ell \leftarrow \ell + 1$ 
         $size \leftarrow |Q|$       /* current size of  $Q$  */
         $idx \leftarrow LF(idx)$ 
         $[lb..rb] \leftarrow dequeue(Q)$ 
         $size \leftarrow size - 1$ 
        if  $B_{\text{LCP}}[rb+1] = 0$  then
             $B_{\text{LCP}}[rb+1] \leftarrow 1$ 
             $list \leftarrow getIntervals([lb..rb])$ 
            for each  $[i..j]$  in  $list$  do
                 $enqueue(Q, [i..j])$ 
            if  $B_{\text{LCP}}[rb] = 1$  and  $rb \neq idx$  then
                return  $S[SA[rb]..SA[rb] + \ell]$       /* output as in Section 7.4.2 */
            if  $B_{\text{LCP}}[rb+2] = 1$  and  $rb+1 \neq idx$  then
                return  $S[SA[rb+1]..SA[rb+1] + \ell]$       /* output as in Section 7.4.2 */

```

7.5.4 Top-down traversal of the lcp-interval tree

It is possible to modify Algorithm 7.17 in such a way that all lcp-intervals are enumerated in increasing order of lcp-values [32]: Algorithm 7.19 first finds $0-[1..n]$ (the sole lcp-interval of lcp-value 0), then all lcp-intervals of lcp-value 1, etc. In the example depicted in Figure 7.20, it computes lcp-intervals in the order $0-[1..12]$, $1-[2..5]$, $1-[7..8]$, $1-[9..12]$, $2-[9..10]$, $3-[11..12]$, $4-[4..5]$. So Algorithm 7.19 traverses the lcp-interval tree in a top-down fashion. Note, however, that the traversal is neither a depth-first nor a breadth-first traversal.

The queue Q in Algorithm 7.17 is replaced with several queues in Algorithm 7.19: for each $c \in \Sigma$ there is a queue Q_c . This will ensure that all intervals corresponding to a certain value ℓ can be accessed in ascending order; see Exercise 7.5.5. Each queue Q_c initially contains the c -interval. To be able to place a $c\omega$ -interval into the correct queue Q_c , we

Algorithm 7.19 Top-down enumeration of lcp-intervals.

```

initialize the array LCP[1..n + 1]      /* i.e., LCP[i] = ⊥ for all i */
LCP[1] ← −1
LCP[n + 1] ← −1
for each  $c$  in  $\Sigma$  do
    initialize an empty queue  $Q_c$ 
    enqueue( $Q_c$ , [ $C[c] + 1..C[c + 1]$ ])    /* the  $c$ -interval */
 $\ell \leftarrow 0$ 
 $last_{lb} \leftarrow \perp$ 
 $last_{idx} \leftarrow \perp$ 
while there is a non-empty queue do
    for each  $c$  in  $\Sigma$  do
         $size[c] \leftarrow |Q_c|$       /* current size of the queue  $Q_c$  */
    for each  $c$  in  $\Sigma$  do      /* in increasing order */
        while  $size[c] > 0$  do
            [ $lb..rb$ ] ← dequeue( $Q_c$ )
             $size[c] \leftarrow size[c] - 1$ 
            if LCP[ $rb + 1$ ] = ⊥ then      /* case 1:  $rb + 1$  is an  $\ell$ -index */
                LCP[ $rb + 1$ ] ←  $\ell$ 
                if  $last_{lb} = \perp$  then
                     $last_{lb} \leftarrow lb$ 
                     $last_{idx} \leftarrow rb + 1$ 
                     $list \leftarrow getIntervals([lb..rb])$ 
                    for each  $(c, [i..j])$  in  $list$  do
                        enqueue( $Q_c$ , [ $i..j$ ])
                else if  $last_{idx} = lb$  then      /* case 2:  $last_{idx}$  is last  $\ell$ -index */
                    /* the lcp-interval  $\ell$ -[ $last_{lb}..rb$ ] has not been considered before */
                    process( $\langle \ell, last_{lb}, rb \rangle$ )
                     $last_{lb} \leftarrow \perp$ 
                     $last_{idx} \leftarrow \perp$ 
                     $list \leftarrow getIntervals([lb..rb])$ 
                    for each  $(c, [i..j])$  in  $list$  do
                        enqueue( $Q_c$ , [ $i..j$ ])
                else nothing to do      /* case 3 */
             $\ell \leftarrow \ell + 1$       /* last statement of the outer while-loop */

```

i	LCP	BWT	$S_{SA[i]}$	lcp-intervals		
1	-1	i	$\$$	0	1	4
2	0	p	$i\$$			
3	1	s	$ippi\$$			
4	1	s	$issippi\$$			
5	4	m	$issippi\$$		1	2
6	0	$\$$	$mississippi\$$			
7	0	p	$pi\$$		1	3
8	1	i	$ppi\$$			
9	0	s	$sippi\$$		1	3
10	2	s	$ssissippi\$$			
11	1	i	$ssippi\$$			
12	3	i	$ssissippi\$$			

Figure 7.20: LCP-array, BWT, and lcp-intervals of $S = mississippi\$$.

slightly modify the procedure *getIntervals* from Algorithm 7.16 (page 316): the statement $add(list, [C[c] + i..C[c] + j])$ is replaced with the statement $add(list, (c, [C[c] + i..C[c] + j]))$. In other words, an element of the list now contains the information to whichever queue the interval belongs. Furthermore, instead of the variable *size* in Algorithm 7.17, we now need an array *size* of length σ to keep track of the sizes of the queues Q_c . Algorithm 7.19 further maintains the following three variables:

- ℓ stores the current lcp-value,
- $last_{lb}$ memorizes the left boundary of the current lcp-interval,
- $last_{idx}$ contains the current ℓ -index.

Initially ℓ is set to 0 and the two variables $last_{lb}$ and $last_{idx}$ are set to the undefined value \perp .

For a fixed value of ℓ , Algorithm 7.19 computes the indices i_1, \dots, i_q with $LCP[i_k] = \ell$ in ascending order $i_1 < \dots < i_q$; see Exercise 7.5.5. When i_1 , the first of these indices, is detected, the variable $last_{lb}$ memorizes the left boundary of the ℓ -interval under consideration. If there are further ℓ -indices in this ℓ -interval, say i_2, \dots, i_p , then these are identified one after the other (by case 1) until the last ℓ -index is found (by case 2). Recall that $LCP[rb + 1] \neq \perp$ means that the index $rb + 1$ has an LCP-value that is strictly smaller than ℓ ; so rb is the right boundary of the lcp-interval that started at $last_{lb}$. Thus, the lcp-interval ℓ - $[last_{lb}..rb]$ is detected (and can be processed). It should be pointed out that all lcp-intervals of lcp-value ℓ are found in this way (if $i_p \neq i_q$, then i_{p+1} is the first ℓ -index of the next

ℓ	$Q_\$$	Q_i	Q_m	Q_p	Q_s
0	<u>$[1..1]_\\$</u>	<u>$[2..5]_i$</u>	<u>$[6..6]_m$</u>	<u>$[7..8]_p$</u>	<u>$[9..12]_s$</u>
1		<u>$[2..2]_{i\\$}$</u> , <u>$[3..3]_{ip}$</u> , <u>$[4..5]_{is}$</u>	<u>$[6..6]_{mi}$</u>	<u>$[7..7]_{pi}$</u> , <u>$[8..8]_{pp}$</u>	<u>$[9..10]_{si}$</u> , <u>$[11..12]_{ss}$</u>
2		<u>$[3..3]_{ipp}$</u> , <u>$[4..5]_{iss}$</u>	<u>$[6..6]_{mis}$</u>	<u>$[7..7]_{pi\\$}$</u> , <u>$[8..8]_{ppi}$</u>	<u>$[9..9]_{sip}$</u> , <u>$[10..10]_{sis}$</u> , <u>$[11..12]_{ssi}$</u>
3					<u>$[11..11]_{ssip}$</u> , <u>$[12..12]_{ssis}$</u>
4		<u>$[4..4]_{issip}$</u> , <u>$[5..5]_{issis}$</u>			
5			<u>$[6..6]_{missis}$</u>		<u>$[10..10]_{sissip}$</u>

Figure 7.21: Contents of the queues for increasing values of ℓ when Algorithm 7.19 is applied to the example of Figure 7.20. Intervals belonging to case 1 are wavy underlined, intervals belonging to case 2 are underlined, and intervals belonging to case 3 are not underlined.

lcp-interval of lcp-value ℓ , etc.). Since the algorithm proceeds in this way for increasing values of ℓ , it enumerates all lcp-intervals.

Let us illustrate Algorithm 7.19 with the example of Figure 7.20. After the initialization phase, $Q_\$$ contains the $\$$ -interval $[1..1]$, Q_i contains the i -interval $[2..5]$, and so on; see row $\ell = 0$ in Figure 7.21. In the outer while-loop of Algorithm 7.19, $size[c]$ is set to the current size of queue Q_c for each $c \in \Sigma$; in the first iteration, we have $size[c] = 1$ for each $c \in \Sigma$. Then, the algorithm accesses the queues in increasing order. In our example, it first removes the interval $[lb..rb] = [1..1]$ from $Q_\$$. Since $LCP[rb + 1] = LCP[2] = \perp$ (case 1), the algorithm has detected the first index $i_1 = 2$ with $LCP[i_1] = \ell = 0$; hence the assignment $LCP[2] \leftarrow 0$. Furthermore, it sets $last_{lb} = lb = 1$ and $last_{idx} = rb + 1 = 2$. The procedure call $getInterval([1..1])$ returns a list that contains just the $i\$$ -interval $[2..2]$, which is added to the queue Q_i ; see Figure 7.21. The intervals $[2..5]$, $[6..6]$, and $[7..8]$ are processed similarly (in this order), and the algorithm sets $LCP[i_k] = \ell = 0$ for the indices $i_2 = 6$, $i_3 = 7$, and $i_4 = 9$. Afterwards $last_{idx} = 9$ holds. Finally, when the interval $[lb..rb] = [9..12]$ is processed, we have $LCP[12 + 1] = -1$ (recall that $LCP[n + 1] = -1$) and $last_{idx} = lb$. So case 2 applies, and the algorithm has found an lcp-interval, namely the interval $[last_{lb}..rb] = [1..12]$ of lcp-value $\ell = 0$. This is because $LCP[1] = -1 < 0$, $LCP[2] = LCP[6] = LCP[7] = LCP[9] = 0$, and $LCP[13] = -1$. Note that for $k \in \{3, 4, 5, 8, 10, 11, 12\}$ the inequality $LCP[k] > 0$ must hold because $LCP[k] = \perp$. The generic procedure *process* “processes” the lcp-interval, the variables $last_{lb}$ and $last_{idx}$ are reset to \perp , and—as in case 1—new intervals are generated and added to the queues. In the last statement of the outer while-loop, ℓ is incremented by one. The contents of the queues at this point in time is depicted in row $\ell = 1$ of Figure 7.21.

The reader is invited to compute the first lcp-interval [2..5] of lcp-value 1 by executing the algorithm with the intervals in the queue Q_i .

It may happen, however, that the procedure *getIntervals* generates intervals that do not lead to a new value in the LCP-array. In our example, the *mi*-interval [6..6] is such an interval; see row $\ell = 1$ of Figure 7.21. Immediately before Algorithm 7.19 processes this interval, the lcp-interval [2..5] of lcp-value 1 was detected, and $last_{lb}$ and $last_{idx}$ were reset to \perp . For the *mi*-interval $[lb..rb] = [6..6]$, we have $LCP[rb + 1] = LCP[7] = 0$ and $last_{idx} = \perp \neq lb$, so none of the cases 1 or 2 applies. Hence case 3 applies, and the algorithm does nothing.

Exercise 7.5.5 By a similar argument as in the proof of Theorem 7.5.1, show that Algorithm 7.19 correctly fills the LCP-array with increasing values of ℓ , but this time all indices i_1, \dots, i_q with $LCP[i_k] = \ell$ are found in ascending order $i_1 < \dots < i_q$.

Hint: Prove that Algorithm 7.19 maintains the following invariant for each ℓ : the intervals corresponding to ℓ appear in ascending order in the lexicographically ordered queues. (If the intervals $[lb..rb]$ and $[lb'..rb']$ belong to different queues Q_c and $Q_{c'}$, then $rb < lb'$ if and only if $c < c'$. Thus, it must be shown that if $[lb..rb]$ and $[lb'..rb']$ are in the same queue and $[lb..rb]$ occurs before $[lb'..rb']$, then $rb < lb'$).

Theorem 7.5.6 Algorithm 7.19 enumerates all lcp-intervals.

Proof We show by induction on ℓ that all lcp-intervals of lcp-value ℓ are found (and processed) in ascending order. In the base case $\ell = 0$, each queue Q_c initially contains the c -interval. When the $\$$ -interval [1..1] is pulled from $Q_\$$, we have $last_{lb} = \perp$ and $last_{idx} = \perp$. Thus, the algorithm sets $LCP[2] = 0$ (i.e., index $i_1 = 2$ is the first index that satisfies $LCP[i_1] = 0$), $last_{lb} = 1$, and $last_{idx} = 2$. Then, it computes the other indices i_2, \dots, i_p with $LCP[i_k] = 0$ in ascending order because the queues are accessed in alphabetical order. Let z be the largest character in the alphabet Σ . When the z -interval $[lb..n]$ is considered, case 2 applies because $LCP[n + 1] = -1$ and $last_{idx} = i_p = lb$. Consequently, *process* is invoked with the lcp-interval $0\text{--}[1..n]$, and $last_{lb}$ as well as $last_{idx}$ are reset to \perp .

Let $\ell > 0$. By the inductive hypothesis, we may assume that Algorithm 7.19 has correctly computed all lcp-intervals of lcp-value $\ell - 1$. Let $[lb_1..rb_1], \dots, [lb_q..rb_q]$ be the lcp-intervals of lcp-value ℓ in ascending order. We use finite induction on k ($1 \leq k \leq q$) to show that the lcp-interval $[lb_k..rb_k]$ will be detected. Because $last_{lb}$ and $last_{idx}$ are reset whenever an lcp-interval is found, we have $last_{lb} = \perp$ and $last_{idx} = \perp$. The algorithm computes the ℓ -indices i_1, \dots, i_p of $[lb_k..rb_k]$ in ascending order. Let cwa_1, \dots, cwa_p , where $|\omega| = \ell - 1$, be the substrings of S so that the cwa_j -interval leads to the assignment $LCP[i_j] \leftarrow \ell$. Immediately after the assignment $LCP[i_1] \leftarrow \ell$, $last_{lb}$ is set to the left boundary of the cwa_1 -interval,

which equals lb_k . Moreover, after the assignment $LCP[i_p] \leftarrow \ell$, $last_{idx}$ is set to i_p . Let cwa_{p+1} be the length $\ell + 1$ prefix of the suffix $S_{SA[i_p]}$. Observe that the left boundary of the cwa_{p+1} -interval is equal to i_p and its right boundary must be equal to rb_k . So the ℓ -interval $[lb_k..rb_k]$ will be detected provided that the cwa_{p+1} -interval is in the queue Q_c . We will show that this is indeed the case. Clearly, the ω -interval is an lcp-interval of lcp-value $\ell - 1$. According to the inductive hypothesis, Algorithm 7.19 has computed it and its $(\ell - 1)$ -indices. We further distinguish two cases: first, whether there is a character b so that ωb is a substring of S and $a_{p+1} < b$, and second, whether there is no such character. If there is such a character b , then the ωa_{p+1} -interval $[lb..rb]$ must have been considered in case 1 of Algorithm 7.19 and $LCP[rb + 1]$ was set to $\ell - 1$. If there is no such character, then the ωa_{p+1} -interval $[lb..rb]$ must have been considered in case 2 of Algorithm 7.19 because lb was the last $(\ell - 1)$ -index of the ω -interval. In both cases, *getIntervals* must have been applied to $[lb..rb]$. In other words, the cwa_{p+1} -interval entered the queue Q_c . \square

Theorem 7.5.7 *Algorithm 7.19 has a worst-case time complexity of $O(n \log \sigma)$.*

Proof We use an amortized analysis similar to that of Theorem 7.5.2. We prove that each of the cases 1, 2, and 3 can occur at most n times. Case 1 is verbatim the same as in Theorem 7.5.2: it occurs as often as an entry of the LCP-array is filled, and this happens exactly $n - 1$ times. By contrast, case 2 has no analogon in Algorithm 7.17. Whenever case 2 occurs, the algorithm processes a different lcp-interval. As there are at most $n - 1$ lcp-intervals, this happens at most $n - 1$ times. Case 3 in Algorithm 7.19 corresponds to case 2 in Algorithm 7.17, and it was shown in the proof of Theorem 7.5.2 that this case can occur at most n times. In summary, the procedure *getIntervals* can create at most $3n$ intervals because every interval belongs to exactly one case. Each interval can be generated in $O(\log \sigma)$ time, so the runtime of Algorithm 7.19 is $O(n \log \sigma)$. \square

In some applications, it suffices to traverse the (virtual) lcp-interval tree and there is no explicit need for the LCP-array (see, for instance, Section 7.5.5). In this case one can save space by replacing the LCP-array with a bit array B_{LCP} of length $n + 1$ (initially all entries of B_{LCP} are set to zero). Then during the computation, $B_{LCP}[i]$ is set to one if and only if a value is (or can be) assigned to $LCP[i]$; cf. Section 7.5.3.

7.5.5 Finding repeats

In this section, we will use the approach of Section 7.5.4 to find all maximal and supermaximal repeats in a string S . As a matter of fact, this is

quite easy in the case of maximal repeats: we just need to implement the procedure *process* in Algorithm 7.19 appropriately.

Recall from Lemma 5.3.15 that a substring ω of S is a maximal repeat if and only if the ω -interval $[i..j]$ is an lcp-interval of lcp-value $\ell = |\omega|$, and the characters $\text{BWT}[i], \text{BWT}[i+1], \dots, \text{BWT}[j]$ are not all the same. Because Algorithm 7.19 enumerates all lcp-intervals, it can be used to search for all maximal repeats: whenever the procedure *process* is called with an lcp-interval $\ell\text{-}[i..j]$, then it must be tested whether the characters in $\text{BWT}[i..j]$ are not all the same. Using an idea of [191], this test can be done in constant time with a bit vector $B_{\text{BWT}}[1..n]$, which initially contains a series of zeros. In a linear scan of the BWT, set $B_{\text{BWT}}[i] = 1$ if $\text{BWT}[i] \neq \text{BWT}[i-1]$. Then, the bit vector is preprocessed so that rank queries can be answered in constant time. A rank query $\text{rank}_b(B_{\text{BWT}}, i)$ returns the number of occurrences of bit b in $B_{\text{BWT}}[1..i]$. It is not difficult to verify that the characters in $\text{BWT}[i..j]$ are not all the same if and only if $\text{rank}_1(B_{\text{BWT}}, j) - \text{rank}_1(B_{\text{BWT}}, i) > 0$.

The computation of all supermaximal repeats is a bit trickier. According to Lemma 5.3.11, an lcp-interval $\ell\text{-}[i..j]$ induces a supermaximal repeat if and only if

- (a) $\text{LCP}[k] = \ell$ for all $i+1 \leq k \leq j$ (i.e., $[i..j]$ is a local maximum), and
- (b) the characters $\text{BWT}[i], \text{BWT}[i+1], \dots, \text{BWT}[j]$ are pairwise distinct.

To cope with (a), we modify Algorithm 7.19 a little bit. Because the algorithm successively considers the intervals $[i..i_1-1], [i_1..i_2-1], \dots, [i_p..j]$, where i_1, i_2, \dots, i_p are the ℓ -indices of $[i..j]$, the interval $[i..j]$ is a local maximum if and only if each of these intervals is a singleton interval. Algorithm 7.20 incorporates the tests for singleton intervals: the statements that deal with the Boolean variable *locMax* make sure that when the procedure *process* is called with the parameters ℓ, i, j and the Boolean parameter *locMax*, we have *locMax* = *true* if and only if the lcp-interval $\ell\text{-}[i..j]$ is a local maximum. Note that the LCP-array is replaced with a bit array B_{LCP} , and $B_{\text{LCP}}[i] = 1$ in Algorithm 7.20 if and only if $\text{LCP}[i] \neq \perp$ in Algorithm 7.19.

The procedure *superMax* detailed in Algorithm 7.21 solves problem (b). It is very similar to Algorithm 5.14 (page 147). The algorithm uses a global bit array B of size σ (which initially contains a series of zeros) and a local Boolean variable *pd* (which initially is set to *true*; *pd* stands for “pairwise distinct”). For each character c encountered in a left-to-right scan of $\text{BWT}[i..j]$:

- if $B[c] = 0$ (c has not been seen before), the algorithm sets $B[c]$ to 1;
- if $B[c] = 1$ (c has been seen before), it assigns *false* to *pd* and stops the scan.

Algorithm 7.20 Space efficient computation of supermaximal repeats.

```

initialize a bit vector  $B_{\text{LCP}}[1..n+1]$       /* i.e.,  $B_{\text{LCP}}[i] = 0$  for all  $i$  */
 $B_{\text{LCP}}[1] \leftarrow 1$ 
 $B_{\text{LCP}}[n+1] \leftarrow 1$ 
for each  $c$  in  $\Sigma$  do
    initialize an empty queue  $Q_c$ 
    enqueue( $Q_c, [C[c] + 1..C[c+1]]$ )      /* the  $c$ -interval */
 $\ell \leftarrow 0$ 
 $last_{lb} \leftarrow \perp$ 
 $last_{idx} \leftarrow \perp$ 
 $locMax \leftarrow true$ 
while there is a non-empty queue do
    for each  $c$  in  $\Sigma$  do
         $size[c] \leftarrow |Q_c|$       /* current size of the queue  $Q_c$  */
    for each  $c$  in  $\Sigma$  do      /* in increasing order */
        while  $size[c] > 0$  do
             $[lb..rb] \leftarrow dequeue(Q_c)$ 
             $size[c] \leftarrow size[c] - 1$ 
            if  $B_{\text{LCP}}[rb+1] = 0$  then      /* case 1:  $rb+1$  is an  $\ell$ -index */
                 $B_{\text{LCP}}[rb+1] \leftarrow 1$ 
                if  $lb \neq rb$  then
                     $locMax \leftarrow false$ 
                if  $last_{lb} = \perp$  then
                     $last_{lb} \leftarrow lb$ 
                 $last_{idx} \leftarrow rb+1$ 
                 $list \leftarrow getIntervals([lb..rb])$ 
                for each  $(c, [i..j])$  in  $list$  do
                    enqueue( $Q_c, [i..j]$ )
            else if  $last_{idx} = lb$  then      /* case 2:  $last_{idx}$  is last  $\ell$ -index */
                /* the lcp-interval  $\ell$ - $[last_{lb}..rb]$  has not been considered before */
                if  $lb \neq rb$  then
                     $locMax \leftarrow false$ 
                process( $\langle \ell, last_{lb}, rb, locMax \rangle$ )
                 $last_{lb} \leftarrow \perp$ 
                 $last_{idx} \leftarrow \perp$ 
                 $locMax \leftarrow true$ 
                 $list \leftarrow getIntervals([lb..rb])$ 
                for each  $(c, [i..j])$  in  $list$  do
                    enqueue( $Q_c, [i..j]$ )
            else nothing to do      /* case 3 */
         $\ell \leftarrow \ell + 1$ 

```

Algorithm 7.21 Procedure *superMax*($\langle \ell, i, j \rangle$) tests whether the lcp-interval $\ell\text{-}[i..j]$ induces a supermaximal repeat. It uses a global bit array B of size σ initially containing a series of zeros.

```

pd  $\leftarrow$  true
k  $\leftarrow$  i
while k  $\leq$  j and pd = true do
    c  $\leftarrow$  BWT[k]
    if  $B[c] = 0$  then
         $B[c] \leftarrow 1$ 
        k  $\leftarrow$  k + 1
    else
        pd  $\leftarrow$  false
repeat /* reset the bits to zero */
    k  $\leftarrow$  k - 1
    c  $\leftarrow$  BWT[k]
     $B[c] \leftarrow 0$ 
until k = i
if pd = true then
    report that  $\ell\text{-}[i..j]$  induces a supermaximal repeat

```

After that, the algorithm rescans the portion of BWT[$i..j$] that has been scanned and resets the 1-bits to zero (Section 5.3.2 explains why this is done). Finally, if $pd = true$, it reports that $\ell\text{-}[i..j]$ induces a supermaximal repeat.

Exercise 7.5.8 asks you to give an alternative solution to problem (b).

Exercise 7.5.8 Modify the procedure *getIntervals*($[i..j]$) from Algorithm 7.16 (page 316) so that it tests in $O(\log \sigma)$ time whether or not the characters in BWT[$i..j$] are pairwise distinct.

It is possible to compute maximal and supermaximal repeats simultaneously by the combination of Algorithms 7.20 and 7.22. Before we go into the details, let us briefly consider constraints on the output. It is always useful to restrict the output to repeats that have a certain minimum length ml (which usually can be defined by the user) because short repeats are somewhat meaningless. If desired, one can also restrict the output to repeats that occur at least min times and at most max times in the string S . Algorithm 7.22 first tests whether the lcp-interval $\ell\text{-}[i..j]$ induces repeats that exceed the minimum length ml (this is the case if $\ell \geq ml$). If desired, the condition $min \leq j - i + 1 \leq max$ can be added. If $\ell \geq ml$, the algorithm checks whether $[i..j]$ induces a maximal repeat (this is the case if $rank_1(B_{\text{BWT}}, j) - rank_1(B_{\text{BWT}}, i) > 0$). If this is also the case, it further tests whether the induced maximal repeat is even a supermaximal

Algorithm 7.22 *process*($\langle \ell, i, j, locMax \rangle$) tests whether the lcp-interval ℓ - $[i..j]$ induces a supermaximal or a maximal repeat of length at least ml .

```

if  $\ell \geq ml$  and  $rank_1(B_{BWT}, j) - rank_1(B_{BWT}, i) > 0$  then
  if  $locMax = true$  and  $superMax([i..j])$  then /* short-circuit evaluation */
    report that  $\ell$ - $[i..j]$  induces a supermaximal repeat
  else
    report that  $\ell$ - $[i..j]$  induces a maximal repeat

```

repeat. Depending on the outcome of this test, it reports a supermaximal or a maximal repeat.

Exercise 7.5.9 Prove that the simultaneous computation of maximal and supermaximal repeats by the combination of Algorithms 7.20 and 7.22 runs in $O(n \log \sigma)$ time.

Exercise 7.5.10 Show that an lcp-interval ℓ - $[i..j]$ induces a supermaximal repeat only if $rank_1(B_{BWT}, j) - rank_1(B_{BWT}, i) = j - i$. (This condition can be used to avoid unnecessary calls to the procedure *superMax* in Algorithm 7.22.)

7.5.6 Lempel-Ziv factorization

The LZ-factorization of a string S was treated in Section 5.2. There, the factorization was based on the LPS-array, i.e., the length of a longest previous substring was computed for every position in S , regardless of whether an LZ-factor actually begins there or not. A quote from Chen et al. [56]:

Ideally we would like the output to contain only information about the positions where factors start; the difficulty is that it is hard to tell in advance where the factors will begin.

In this section, we resume the discussion on LZ-factorization under this aspect. Another aspect is the space-efficiency of the algorithms.

Let us start with the LZ-factorization algorithm CPS2 of Chen et al. [56]. To explain its idea, suppose that we have already computed the first $k - 1$ factors of a string S , say $S[1..j] = \omega_1 \cdots \omega_{k-1}$ and we want to compute the next factor ω_k , which is a prefix of S_{j+1} . To this end, we determine the $S[j + 1]$ -interval $[lb_1..rb_1]$ in SA and use a range minimum query to find the index $i_1 = \text{RMQ}_{SA}(lb_1, rb_1)$ so that $\text{SA}[i_1]$ is the minimum value in $\text{SA}[lb_1..rb_1]$. Clearly, $S[j + 1]$ has no previous occurrence in S if and only if $\text{SA}[i_1] = j + 1$ (this is because $j + 1$ is an element of $\text{SA}[lb_1..rb_1]$). If $\text{SA}[i_1] < j + 1$, then $\text{SA}[i_1]$ is a previous occurrence and we further proceed as follows: We determine the $S[j + 1..j + 2]$ -interval $[lb_2..rb_2]$, where $lb_1 \leq lb_2 \leq rb_2 \leq rb_1$. Now there are two possibilities:

i	$SA[i]$	$ISA[i]$	$S_{SA[i]}$	$PSV[i]$	$NSV[i]$
0	0		ε		
1	3	3	<i>aaacatat</i>	0	3
2	4	7	<i>aacatat</i>	1	3
3	1	1	<i>acaaacatat</i>	0	11
4	5	2	<i>acatat</i>	3	7
5	9	4	<i>at</i>	4	6
6	7	8	<i>atat</i>	4	7
7	2	6	<i>caaacatat</i>	3	11
8	6	10	<i>catat</i>	7	11
9	10	5	<i>t</i>	8	10
10	8	9	<i>tat</i>	8	11
11	0		ε		

Figure 7.22: The suffix array of the string $S = acaaacatat$ and the PSV- and NSV-arrays.

- (a) If i_1 also belongs to the interval $[lb_2..rb_2]$ (i.e., $lb_2 \leq i_1 \leq rb_2$), then we infer that $i_1 = \text{RMQ}_{SA}(lb_2, rb_2)$ and thus $SA[i_1] < j + 1$.
- (b) If i_1 lies outside the interval $[lb_2..rb_2]$ another range minimum query yields the index $i_2 = \text{RMQ}_{SA}(lb_2, rb_2)$ so that $SA[i_2]$ is the minimum value in $SA[lb_2..rb_2]$.

In case (b), we check whether $SA[i_2] < j + 1$. If the test is negative, we output $\omega_k = S[j + 1]$ and $SA[i_1]$ as a previous occurrence. Otherwise, we iterate the process described above. This is also done in case (a).

Let us illustrate the algorithm by the example of Figure 7.22. Suppose that we have already computed the first two factors $\omega_1 = a$ and $\omega_2 = c$ of the LZ-factorization of the string $S = acaaacatat$, and we want to compute the next factor ω_3 , a prefix of $S_3 = aaacatat$. We determine the a -interval $[1..6]$ in SA and calculate $i_1 = \text{RMQ}_{SA}(1, 6) = 3$. Since $SA[3] = 1 < 3$, the aa -interval in SA is computed. Clearly, $i_1 = 3$ does not lie inside the aa -interval $[1..2]$. So we compute $i_2 = \text{RMQ}_{SA}(1, 2) = 1$ and check whether $SA[1] < 3$. Because this is not the case, the next factor $\omega_3 = a$ and the starting position 1 of a previous occurrence of a in S is output.

A variant of the preceding algorithm uses PSV_{SA} - and NSV_{SA} -values of the suffix array (see Definition 5.2.4) instead of range minimum queries on the suffix array. As in Section 5.2, we often write PSV and NSV instead of PSV_{SA} and NSV_{SA} . The algorithm works as follows. To find the next factor ω_k of S , which must be a prefix of S_{j+1} , we determine the $S[j + 1]$ -interval $[lb_1..rb_1]$ in SA and the rank $i = ISA[j + 1]$ of suffix S_{j+1} . Obviously, $S[j + 1]$ has a previous occurrence in S if and only if $PSV[i]$ or $NSV[i]$ (or both) lie inside

the $S[j+1]$ -interval $[lb_1..rb_1]$. If this is the case, $SA[PSV[i]]$ or $SA[NSV[i]]$ is a previous occurrence of $S[j+1]$. Then we iterate this process: we determine the $S[j+1..j+2]$ -interval $[lb_2..rb_2]$ and check whether $PSV[i]$ or $NSV[i]$ lie in $[lb_2..rb_2]$, and so on. The maximum ℓ for which the $S[j+1..j+\ell]$ -interval $[lb_\ell..rb_\ell]$ contains $PSV[i]$ or $NSV[i]$ is the length of ω_k . Moreover, if $PSV[i]$ ($NSV[i]$, respectively) lies in $[lb_\ell..rb_\ell]$, then $SA[PSV[i]]$ ($SA[NSV[i]]$, respectively) is the start position of a previous occurrence of $S[j+1..j+\ell]$.

Let us use the same example as above to exemplify the algorithm. The fourth factor ω_4 of $S = acaaacatat$ must be a prefix of $S_4 = aacatat$. We determine the a -interval $[1..6]$ in SA and the rank $2 = ISA[4]$ of suffix S_4 . Character a has previous occurrences at positions $SA[PSV[2]] = 3$ and $SA[NSV[2]] = 1$ in S because both $PSV[2] = 1$ and $NSV[2] = 3$ lie inside the a -interval $[1..6]$. Therefore, the aa -interval $[1..2]$ is computed. Again, $PSV[2] = 1$ is within the interval, so we turn to the aac -interval $[2..2]$. Since neither $PSV[2] = 1$ nor $NSV[2] = 3$ lie inside this interval, it follows that $\omega_4 = aa$ and a previous occurrence of aa starts at position $SA[PSV[2]] = 3$ in S .

Both algorithms must repeatedly find intervals of LZ-factors. For a factor ω_k , this can be done space efficiently in $O(|\omega_k| \log \sigma)$ time by the technique described in Section 6.3.3 or, without using any extra space, in $O(|\omega_k| \log n)$ time by binary search; see Section 5.1.3. The first algorithm further relies on RMQ_{SA} , while the second relies on PSV_{SA} and NSV_{SA} . If one uses the balanced parentheses sequence BPS_{SA} of the suffix array, then RMQ_{SA} , PSV_{SA} , and NSV_{SA} can be found in constant time using only a little extra space. Although the BPS was introduced in Section 6.3 w.r.t. the LCP-array, the same concept can of course be applied to the suffix array. As a matter of fact, the computation of BPS_{SA} , RMQ_{SA} , PSV_{SA} , and NSV_{SA} becomes even simpler because a suffix array does not have equal entries (so one can get rid of the additional bit vector B).

In contrast to the first algorithm, which solely uses the suffix array, the second algorithm also requires the inverse suffix array. A method that avoids the inverse suffix array uses backward search on the string $T = S^{rev}\$,$ where S^{rev} is the reverse of S and $\$$ is the sentinel character. However, since we work with $T = S^{rev}\$,$ instead of S , we have to replace previous/next smaller values PSV and NSV with previous/next *greater* values PGV and NGV . In the remainder of this section, SA denotes the suffix array of T . Furthermore, to deal with boundary cases, we set $SA[0] = \infty = SA[n+1]$.

Definition 7.5.11 For any index $1 \leq i \leq n$, we define

$$\begin{aligned} PGV(i) &= \max\{j : 0 \leq j < i \text{ and } SA[j] > SA[i]\} \\ NGV(i) &= \min\{j : i < j \leq n+1 \text{ and } SA[j] > SA[i]\} \end{aligned}$$

The values $PGV(i)$ and $NGV(i)$ can be computed in constant time on the balanced parentheses sequence BPS'_{SA} , which can be constructed by Algorithm 7.23 (pay attention to the condition $SA[i] > SA[top()]$ in its while-loop), as follows:

Algorithm 7.23 Construction of the balanced parentheses sequence BPS'_{SA} .

```

push(0)      /* SA[0] = ∞ */
write '('
for  $i \leftarrow 1$  to  $n + 1$  do
    while  $\text{SA}[i] > \text{SA}[\text{top}()]$  do
        pop() and write ')'
        push( $i$ ) and write '('
write ')'
```

/* for $\text{SA}[0] = \infty$ and $\text{SA}[n + 1] = \infty$ */

- $\text{PGV}(i) = \text{rank}_\ell(\text{enclose}(\text{select}_\ell(i)))$; Lemma 6.3.3 applies with a grain of salt because a suffix array does not have equal entries.
- $\text{NGV}(i) = \text{rank}_\ell(\text{findclose}(\text{select}_\ell(i))) + 1$; see Lemma 6.3.1.

The balanced parentheses sequence BPS'_{SA} requires only $2n$ bits, and we have seen in Section 6.1 that $o(n)$ additional bits suffice to support the operations *rank*, *select*, *findclose*, and *enclose* in constant time. All in all, $\text{PGV}(i)$ and $\text{NGV}(i)$ can be computed in constant time, using only $2n + o(n)$ bits.

Now we have all the ingredients for the space-efficient LZ-factorization algorithm 7.24, which runs in $O(n \log \sigma)$ time. Each execution of its outer while-loop computes the next factor of the right-to-left LZ-factorization of S^{rev} (starting at position i'), which coincides with the next factor of the left-to-right LZ-factorization of S (starting at $n + 1 - i'$). Note that we do not need the inverse suffix array because the rank of the current suffix can be determined with the help of the LF -mapping. We still need the (sparse) suffix array to determine the previous occurrence but only once for each factor.

Exercise 7.5.12 Give pseudo-code of an LZ-factorization algorithm that uses backward search and range maximum queries.

We would like to point out that the idea of replacing the binary search in the algorithm of [56] with the backward search in the reverse string [250] was also used by Krefl and Navarro [188] for an alternative Lempel-Ziv parsing (called LZ-End). Furthermore, there is another LZ-factorization algorithm that uses succinct data structures: the online algorithm developed by Okanohara and Sadakane [253]. With the aid of *rank/select* operations and range minimum queries, it dynamically maintains succinct representations of the suffix array, the LCP-array, and the BWT. Its worst-case time complexity is $O(n \log^3 n)$.

Algorithm 7.24 LZ-factorization of S using backward search on $T = S^{rev}\$$.

```

 $i \leftarrow n$           /*  $|S^{rev}| = n$  */
 $j \leftarrow 1$        /*  $\$$  appears at index 1 in SA */
while  $i > 1$  do
     $i' \leftarrow i$ 
     $[sp..ep] \leftarrow \text{backwardSearch}(T[i], [1..n + 1])$ 
    while  $\text{NGV}[LF(j)] \leq ep$  or  $\text{PGV}[LF(j)] \geq sp$  do
         $j \leftarrow LF(j)$ 
         $[lb..rb] \leftarrow [sp..ep]$ 
         $i \leftarrow i - 1$ 
         $[sp..ep] \leftarrow \text{backwardSearch}(T[i], [sp..ep])$ 
    LPS  $\leftarrow i' - i$ 
    if LPS = 0 then
        PrevOcc  $\leftarrow T[i]$ 
         $i \leftarrow i - 1$ 
    else if  $\text{NGV}[j] \leq rb$  then
        PrevOcc  $\leftarrow n + 1 - \text{SA}[\text{NGV}[j]] - (\text{LPS} - 1)$ 
    else          /*  $\text{PGV}[j] \geq lb$  */
        PrevOcc  $\leftarrow n + 1 - \text{SA}[\text{PGV}[j]] - (\text{LPS} - 1)$ 
    output (PrevOcc, LPS)

```

7.6 Space-efficient comparison of two strings

In this section, we revisit two major problems in string comparisons: computing matching statistics and maximal exact matches [251]. Furthermore, we address the problem of merging BWTs.

7.6.1 Matching statistics

In the following, let S^1 and S^2 be strings of length n_1 and n_2 , where S^1 but not S^2 has the sentinel character at the end. Let us recall the definition of matching statistics from Definition 5.5.10.

The *matching statistics* of S^2 w.r.t. S^1 is an array ms so that for every entry $ms[p_2] = (q, [lb..rb])$, $1 \leq p_2 \leq n_2$, the following holds:

1. $S^2[p_2..p_2 + q - 1]$ is the longest prefix of $S^2_{p_2}$ that is a substring of S^1 .
2. $[lb..rb]$ is the $S^2[p_2..p_2 + q - 1]$ -interval in the suffix array of S^1 .

In Section 5.5.4, we computed the matching statistics of S^2 w.r.t. S^1 by matching S^2 in a *forward* direction against the suffix tree or the ESA of S^1 . This takes $O(n_2 \log \sigma)$ time if suffix links are used as shortcuts; cf. Sections 5.5.4, 6.3.3, and 6.3.5. By contrast, here we match S^2 in a *backward*

Algorithm 7.25 Computing matching statistics by backward search.

```

 $p_2 \leftarrow n_2$ 
 $(q, [i..j]) \leftarrow (0, [1..n_1])$ 
while  $p_2 \geq 1$  do
   $[lb..rb] \leftarrow \text{backwardSearch}(S^2[p_2], [i..j])$ 
  if  $[lb..rb] \neq \perp$  then
     $q \leftarrow q + 1$ 
     $ms[p_2] \leftarrow (q, [lb..rb])$ 
     $[i..j] \leftarrow [lb..rb]$ 
     $p_2 \leftarrow p_2 - 1$ 
  else if  $[i..j] = [1..n_1]$  then
     $ms[p_2] \leftarrow (0, [1..n_1])$ 
     $p_2 \leftarrow p_2 - 1$ 
  else
     $q-[i..j] \leftarrow \text{parent}([i..j])$ 

```

direction against a compressed full text index of S^1 . The algorithm does not rely on suffix links but on the ability to determine parent intervals of lcp-intervals. Using only $3n + o(n)$ bits, a parent interval can be identified in constant time with the enhanced balanced parentheses sequence BPS; cf. Section 6.3.1. Alternatively, one could use e.g. Sadakane's [273] or Fischer et al.'s [110] method.

Algorithm 7.25 shows pseudo-code for the computation of matching statistics. It matches S^2 backwards against a compressed full-text index of S^1 . Let ω be the current matching substring of length q (initially, $\omega = \varepsilon$), $[i..j]$ be the ω -interval, p_2 be the current position (initially, $p_2 = n_2$), and $c = S^2[p_2]$. The algorithm determines the $c\omega$ -interval $[lb..rb]$. If $lb \leq rb$, then $c\omega$ matches a substring of S^1 . Therefore, the algorithm increments q by one, decrements p_2 by one, and sets $ms[p_2] \leftarrow (q, [lb..rb])$. Otherwise, $c\omega$ is not a substring of S^1 . In that case, the algorithm determines the longest proper prefix of ω that is a substring of S^1 . Because this is the string represented by the parent interval of $[i..j]$, the algorithm continues with this parent interval.

To exemplify Algorithm 7.25, we match the string $S^2 = caaca$ backwards against the compressed full-text index of $S^1 = acaaacatat\$$; see Figure 7.23. Starting with the last character of S^2 and the ε -interval $[1..n_1]$, backward search returns the a -interval $[2..7]$, and Algorithm 7.25 sets $ms[5] = (1, [2..7])$. Similarly, it determines $ms[4] = (2, [8..9])$, $ms[3] = (3, [4..5])$, and $ms[2] = (4, [3..3])$. The procedure call $\text{backwardSearch}(c, [3..3])$ returns \perp , indicating that $S^2[1..5] = caaca$ is not a substring of S^1 . In this case—if a mismatch occurs—the algorithm determines the parent interval of the current interval, which in our example is the aa -interval $2-[2..3]$. The

i	SA	LCP	BWT	$S_{SA[i]}$	lcp-intervals		
1	11	-1	t	$\$$	0	1	2
2	3	0	c	$aaacatat\$$			
3	4	2	a	$aacatat\$$			
4	1	1	$\$$	$acaaacatat\$$			3
5	5	3	a	$acatat\$$			
6	9	1	t	$at\$$		2	2
7	7	2	c	$atat\$$			
8	2	0	a	$caaacatat\$$			
9	6	2	a	$catat\$$		2	1
10	10	0	a	$t\$$			
11	8	1	a	$tat\$$			

Figure 7.23: The BWT of the string $S = acaaacatat\$$.

subsequent procedure call $backwardSearch(c, [2..3])$ returns the caa -interval $[8..8]$, and $ms[1]$ is set to $(3, [8..8])$.

Lemma 7.6.1 *Algorithm 7.25 correctly computes the matching statistics of S^2 w.r.t. S^1 .*

Proof We prove the correctness of Algorithm 7.25 by finite induction on the length $n_2 - p_2 + 1$ of the suffix $S^2_{p_2}$ of S^2 . If the length equals 1 (i.e., $p_2 = n_2$), then there are two possibilities. The character $c = S^2[n_2]$ either (a) occurs in S^1 or (b) it does not. In case (a), Algorithm 7.25 sets $ms[n_2] = (1, [lb..rb])$, where $[lb..rb]$ is the c -interval. This is certainly correct. In case (b), Algorithm 7.25 sets $ms[n_2] = (0, [1..n_1])$, where $[1..n_1]$ is the ε -interval. This is also correct. We assume, as induction hypothesis, that for some fixed position $p_2 + 1$ with $1 \leq p_2 < n_2$, Algorithm 7.25 correctly computed the matching statistic $ms[p_2 + 1] = (q, [i..j])$, i.e.,

1. $\omega = S^2[p_2 + 1..p_2 + q]$ is the longest prefix of $S^2_{p_2+1}$ that occurs as a substring of S^1 .
2. $[i..j]$ is the ω -interval in the suffix array of S^1 .

In the inductive step, we must show that Algorithm 7.25 correctly computes $ms[p_2]$. Let $c = S^2[p_2]$. Then $backwardSearch(c, [i..j])$ yields the $c\omega$ -interval $[lb..rb]$ in the suffix array of S^1 provided that $c\omega$ is a substring of S^1 . It is readily verified that $c\omega = S^2[p_2..p_2 + q]$ is the longest prefix of $S^2_{p_2}$ that occurs as a substring of S^1 . Consequently, $ms[p_2] = (q + 1, [lb..rb])$. Otherwise, if $c\omega$ is not a substring of S^1 , then $backwardSearch(c, [i..j])$ returns \perp . We consider the two subcases (a) $[i..j] = [1..n_1]$ and (b) $[i..j] \neq [1..n_1]$.

(a) If $[i..j] = [1..n_1]$, i.e., $\omega = \varepsilon$, then the character c does not occur in S^1 . This means that the longest prefix of $S^2_{p_2}$ that occurs as a substring of S^1 is the empty string ε and $ms[p_2] = (0, [1..n_1])$.

(b) If $[i..j] \neq [1..n_1]$, then $\omega \neq \varepsilon$. Because $c\omega$ is not a substring of S^1 , we must search for the longest prefix u' of ω so that cu' is a substring of S^1 . Let $[i'..j']$ be the parent lcp-interval of $[i..j]$. The lcp-interval $[i'..j']$ is the u -interval of a proper prefix u of ω . Suppose that b is the character immediately following u in ω , i.e., $\omega = ubv$ for some string v . Because the u -interval $[i'..j']$ is the parent lcp-interval of the ω -interval $[i..j]$, every substring ω' of S^1 that has ub as a prefix must also have ω as a prefix. We claim that the string cub cannot occur in S^1 . To prove the claim, suppose to the contrary that cub is a substring of S^1 . Because every substring ω' of S^1 that has ub as a prefix must also have ω as a prefix, it follows that $c\omega$ must be a substring of S^1 . This contradicts the fact that $c\omega$ is not a substring of S^1 and thus proves the claim that the string cub cannot occur in S^1 . Consequently, u is the longest prefix of ω so that cu is a possible substring of S^1 . Observe that the algorithm checks in the next iteration of the while-loop whether or not cu is indeed a substring of S^1 . If so, then u is the longest prefix of ω so that cu is a substring of S^1 . If not, the algorithm continues with the parent interval of the u -interval $[i'..j']$, and so on, until either backward search succeeds or the interval $[1..n_1]$ is found. In both cases $ms[p_2]$ is correctly assigned. \square

Lemma 7.6.2 *Given the wavelet tree of the BWT of S^1 and the enhanced BPS, Algorithm 7.25 runs in $O(n_2 \log \sigma)$ time.*

Proof We use an amortized analysis to derive the worst-case time complexity of Algorithm 7.25. Each statement in the while-loop takes only constant time, except for the backward search step, which takes $O(\log \sigma)$ time on the wavelet tree of the BWT of S^1 . We claim that the number of iterations of the while-loop over the entire algorithm is bounded by $2n_2$. In each iteration of the while-loop, either the position p_2 in S^2 is decreased by one or the search interval $[i..j]$ is replaced with its parent interval. Clearly, p_2 is decreased n_2 times and we claim that at most n_2 many search intervals can be replaced with its parent interval. To see this, let the search interval $[i..j]$ be the ω -interval and let $[i'..j']$ denote its parent interval. The lcp-interval $[i'..j']$ is the u -interval of a proper prefix u of ω . Consequently, each time a search interval is replaced with its parent interval, the length of the search string ω is shortened by at least one. Since the overall length increase of all search strings is bounded by n_2 , the claim follows. Thus, under the assumption that the wavelet tree and the enhanced BPS have already been constructed, Algorithm 7.25 has a worst-case time complexity of $O(n_2 \log \sigma)$. \square

7.6.2 Maximal exact matches

The starting point for any comparison of large genomes (e.g. mammalian or plant genomes) is the computation of exact matches between their DNA sequences S^1 and S^2 , and maximal exact matches (cf. Definition 5.4.5) can be used for this task. In genome comparisons, one is merely interested in MEMs (q, p_1, p_2) that exceed a user-defined length threshold ℓ , i.e., $q \geq \ell$. In the software-tools MUMmer 3.0 [196] and CoCoNUT [3], maximal exact matches between S^1 and S^2 are computed by matching S^2 in a *forward* direction against the suffix tree or the enhanced suffix array of S^1 . The bottleneck in large-scale applications like genome comparisons is often the space requirement of the software-tool. If the index structure (e.g. an enhanced suffix array) does not fit into main memory, then it is worthwhile to use a compressed index structure instead. Algorithm 7.26 computes maximal exact matches by matching S^2 in a backward direction against a compressed full-text index of S^1 .

Algorithm 7.26 is similar to Algorithm 7.25: for each position p_2 in S^2 , it computes the longest match of $S^2_{p_2}$ with a substring of S^1 , say of length q , and the $S^2[p_2..q-1]$ -interval $[lb..rb]$. This time, however, it keeps track of the longest matching path. To be precise, each matching statistic $ms[p_2] = (q, [lb..rb])$ satisfying $q \geq \ell$ is stored as a triple $(q, [lb..rb], p_2)$ in a list called *path* until a mismatch occurs (i.e., the until backward search returns \perp). Then, the algorithm computes MEMs from the triples in the list *path* (in its outer for-loop). If all elements of the list *path* have been processed, it computes the next longest matching path, and so on.

By construction (or more precisely, by the correctness of Algorithm 7.25), if the triple $(q, [lb..rb], p_2)$ occurs in some matching path, then $ms[p_2] = (q, [lb..rb])$ and $q \geq \ell$. (Note that for each position p_2 in S^2 at most one triple $(q, [lb..rb], p_2)$ appears in the matching paths.) Clearly, this implies that each $(q, SA[k], p_2)$ is a longest right maximal exact match at position p_2 in S^2 , where $lb \leq k \leq rb$. Now Algorithm 7.26 tests left maximality by $BWT[k] \neq S^2[p_2-1]$. If $(q, SA[k], p_2)$ is left maximal, then it is a maximal exact match between S^1 and S^2 with $q \geq \ell$, and the algorithm outputs it. After that, it considers the parent lcp-interval of $[lb..rb]$. Let us denote this parent interval by $q' \text{--}[lb'..rb']$. For each k with $lb' \leq k < lb$ or $rb < k \leq rb'$, the triple $(q', SA[k], p_2)$ is a right maximal exact match because $S^1[SA[k]..SA[k] + q' - 1] = S^2[p_2..p_2 + q' - 1]$ and $S^1[SA[k] + q'] \neq S^2[p_2 + q']$. So if $q' \geq \ell$ and $BWT[k] \neq S^2[p_2-1]$, the algorithm outputs $(q', SA[k], p_2)$. Then it considers the parent lcp-interval of $[lb'..rb']$ and so on. To sum up, Algorithm 7.26 checks every right maximal exact match exceeding the length threshold ℓ for left maximality. It follows as a consequence that it detects every maximal exact match of length $\geq \ell$.

Algorithm 7.26 Computing MEMs of length $\geq \ell$ by backward search.

```

 $p_2 \leftarrow n_2$ 
 $(q, [i..j]) \leftarrow (0, [1..n_1])$ 
while  $p_2 \geq 1$  do
     $path \leftarrow []$ 
     $[lb..rb] \leftarrow backwardSearch(S^2[p_2], [i..j])$ 
    while  $[lb..rb] \neq \perp$  and  $p_2 \geq 1$  do
         $q \leftarrow q + 1$ 
        if  $q \geq \ell$  then
             $add(path, (q, [lb..rb], p_2))$ 
             $[i..j] \leftarrow [lb..rb]$ 
             $p_2 \leftarrow p_2 - 1$ 
             $[lb..rb] \leftarrow backwardSearch(S^2[p_2], [i..j])$ 
        for each  $(q', [lb'..rb'], p'_2)$  in  $path$  do
             $[lb..rb] \leftarrow \perp$ 
            while  $q' \geq \ell$  do
                for each  $k \in [lb'..rb'] \setminus [lb..rb]$  do
                    if  $p'_2 = 1$  or  $BWT[k] \neq S^2[p'_2 - 1]$  then
                        output  $(q', SA[k], p'_2)$ 
                         $[lb..rb] \leftarrow [lb'..rb']$ 
                         $q' - [lb'..rb'] \leftarrow parent([lb'..rb'])$ 
            if  $[i..j] = [1..n_1]$  then
                 $p_2 \leftarrow p_2 - 1$ 
            else
                 $q - [i..j] \leftarrow parent([i..j])$ 
    
```

We exemplify the algorithm by matching the string $S^2 = caaca$ backwards against the compressed full-text index of $S^1 = acaaacatat\$$. For the length threshold $\ell = 2$, the first matching path is $(2, [8..9], 4)$, $(3, [4..5], 3)$, $(4, [3..3], 2)$. The triple $(2, [8..9], 4)$ yields no output, but for the triple $(3, [4..5], 3)$, the algorithm outputs the MEM $(3, 1, 3)$. (Note that the parent intervals of $[8..9]$ and $[4..5]$ are not considered because their lcp-value is smaller than $\ell = 2$.) The triple $(4, [3..3], 2)$ yields the output $(4, 4, 2)$, and when its parent interval $2-[2..3]$ is considered, the algorithm does not output the right maximal exact match $(2, 3, 2)$ because it is not left maximal. Now all triples in the matching path have been considered, and the algorithm computes the next longest matching path starting at position $p_2 = 1$ and the parent interval $2-[2..3]$ of $[i..j] = [3..3]$. This new path consists of the triple $(3, [8..8], 1)$ resulting in the output $(3, 2, 1)$ and $(2, 6, 1)$.

Let us analyze the worst-case time complexity of Algorithm 7.26. If the outer for-loop was not there, it would run in $O(n_2 \log \sigma)$ time; see the run-time analysis of Algorithm 7.25. In each execution of the while-loop within

the outer for-loop, Algorithm 7.26 tests a right maximal exact match of length $\geq \ell$ for left maximality by $\text{BWT}[k] \neq S^2[p'_2 - 1]$. This test requires only constant time if BWT is kept in main memory. However, the test can be performed solely based on the wavelet tree. This is because the wavelet tree allows determining $\text{BWT}[k]$ in $O(\log \sigma)$ time; see Section 7.4. Alternatively, the test $\text{BWT}[k] \neq c$ in Algorithm 7.26 can be replaced with the test $LF(k) \notin [i..j]$, where $[i..j]$ is the c -interval, because $\text{BWT}[k] \neq c$ if and only if $LF(k) \notin [i..j]$. So, under the assumption that the wavelet tree of S^1 and the enhanced BPS have already been constructed, the algorithm runs in $O(n_2 \log \sigma + z \log \sigma + \text{occ} \cdot s)$ time, where occ (z , respectively) is the number of maximal (right maximal, respectively) exact matches of length $\geq \ell$ between the strings S^1 and S^2 , and s is the sampling parameter of the sparse suffix array.

7.6.3 Merging Burrows-Wheeler transformed strings

In Section 5.5.5, we saw that the common suffix array of two strings S^1 and S^2 can be obtained by merging their suffix arrays SA_1 and SA_2 in linear time. Thus, given the Burrows-Wheeler transformed strings BWT_1 and BWT_2 of S^1 (terminated by the character $\#$) and S^2 (terminated by the character $\$$), the Burrows-Wheeler transformed string BWT of the concatenated string $S = S^1 S^2$ can be obtained as follows:

- Derive SA_1 from BWT_1 and SA_2 from BWT_2 ; see Exercise 7.2.6.
- Merge SA_1 and SA_2 into the common suffix array SA.
- Derive BWT from SA.

However, it is crystal-clear that this method is not space efficient. Sirén [293] and Ferragina et al. [99] proposed methods to directly merge BWTs. Here is the technique:

1. Construct a data structure that supports $\text{rank}_c(\text{BWT}_1, i)$ queries (e.g. the wavelet tree of BWT_1).
2. Construct a data structure that supports access to $LF_2(i)$ (e.g. the wavelet tree of BWT_2).
3. Use Algorithm 7.27 to compute a bit vector B with $B[k] = 1$ if and only if the k -th lexicographically smallest suffix of S belongs to S^2 .
4. Merge BWT_1 and BWT_2 with the help of the bit vector B .

We exemplify Algorithm 7.27 by the toy example of Figure 7.24. In the first iteration of the repeat-loop, $\text{BWT}_2[1] = t$ is the next-to-last character of S^2 and $LF_2(1) = 4$. Thus, the suffix $t\$$ is the fourth lexicographically

Algorithm 7.27 Given the wavelet trees of BWT_1 and BWT_2 , this procedure computes the bit vector $B[1..n_1 + n_2]$.

```

initialize a bit vector  $B$ , i.e.,  $B[1..n_1 + n_2] \leftarrow [0, \dots, 0]$ 
 $B[2] \leftarrow 1$       /*  $\# < \$$  and both are smaller than all other characters */
 $i \leftarrow 1$ 
 $j \leftarrow 1$ 
repeat
     $c \leftarrow \text{BWT}_2[j]$ 
     $j \leftarrow \text{LF}_2(j)$       /* or equivalently  $j \leftarrow C_2[c] + \text{rank}_c(\text{BWT}_2, j)$  */
     $i \leftarrow C_1[c] + \text{rank}_c(\text{BWT}_1, i)$ 
     $B[i + j] \leftarrow 1$ 
until  $c = \$$ 
return  $B[1..n_1 + n_2]$ 
    
```

i	BWT_1	$S^1_{\text{SA}_1[i]}$	i	LF_2	BWT_2	$S^2_{\text{SA}_2[i]}$	i	B	BWT	$S_{\text{SA}[i]}$
1	t	$\#$	1	4	t	$\$$	1	0	t	$\#$
2	t	$at\#$	2	1	$\$$	$aat\$$	2	1	t	$\$$
3	$\#$	$atat\#$	3	2	a	$at\$$	3	1	$\$$	$aat\$$
4	a	$t\#$	4	3	a	$t\$$	4	0	t	$at\#$
5	a	$tat\#$					5	1	a	$at\$$
							6	0	$\#$	$atat\#$
							7	0	a	$t\#$
							8	1	a	$t\$$
							9	0	a	$tat\#$

Figure 7.24: Algorithm 7.27 applied to the strings $S^1 = atat\#$ and $S^2 = aat\$$ yields the bit vector B .

smallest suffix of S^2 (in fact, it is the lexicographically largest suffix of S^2). Since $C_1[t] + \text{rank}_t(\text{BWT}_1, 1) = 3 + 1 = 4$, it follows that four suffixes of S^1 are lexicographically smaller than $t\$$. Consequently, $t\$$ appears at index $4 + 4 = 8$ in the common suffix array of S^1 and S^2 . Hence $B[8] = 1$. In the second iteration of the repeat-loop, $\text{BWT}_2[4] = a$ and $\text{LF}_2(4) = 3$ imply that $at\$$ is the third lexicographically smallest suffix of S^2 . Furthermore, we derive from $C_1[a] + \text{rank}_a(\text{BWT}_1, 4) = 1 + 1 = 2$ that two suffixes of S^1 are lexicographically smaller than $at\$$. So $at\$$ appears at index $3 + 2 = 5$ in the common suffix array of S^1 and S^2 and we set $B[5] = 1$. Similarly, we have $B[3] = 1$ because $aat\$$ occurs at index $2 + 1 = 3$ in the common suffix array of S^1 and S^2 .

Algorithm 7.28 Given BWT_1 , BWT_2 , and the bit vector B , this procedure computes the Burrows-Wheeler transformed string BWT of S^1S^2 .

initialize an array $\text{BWT}[1..n_1 + n_2]$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow 1$ **to** $n_1 + n_2$ **do**

if $B[k] = 0$ **then**

$\text{BWT}[k] \leftarrow \text{BWT}_1[i]$

$i \leftarrow i + 1$

else

$\text{BWT}[k] \leftarrow \text{BWT}_2[j]$

$j \leftarrow j + 1$

return $\text{BWT}[1..n_1 + n_2]$

Lemma 7.6.3 *Algorithm 7.27 maintains the following invariant: $S_{\text{SA}_1[i]}^1$ is the lexicographically largest suffix of S^1 that is smaller than $S_{\text{SA}_2[j]}^2$.*

Proof Initially, $i = 1$ and $j = 1$. Recall that $\text{SA}_1[1] = n_1$ and $S^1[n_1] = \#$, as well as $\text{SA}_2[1] = n_2$ and $S^2[n_2] = \$$. Since $\#$ and $\$$ are the smallest characters in Σ and $\# < \$$, the invariant holds before the repeat-loop is entered for the first time. Suppose that after k iterations of the repeat-loop, $S_{\text{SA}_1[i]}^1$ is the lexicographically largest suffix of S^1 that is smaller than $S_{\text{SA}_2[j]}^2$. Hence $S_{\text{SA}_1[i]}^1 < S_{\text{SA}_2[j]}^2 < S_{\text{SA}_1[i+1]}^1$. Note that $S_{\text{SA}_2[j]}^2 = S_{n_2-k}^2$ because in each iteration, j is updated by $j \leftarrow LF_2(j)$. In iteration $k + 1$, the assignments $c \leftarrow \text{BWT}_2[j]$ and $j \leftarrow LF_2(j)$ imply that now $S_{\text{SA}_2[j]}^2 = S_{n_2-k-1}^2 = cS_{n_2-k}^2$ is the j -th lexicographically smallest suffix of S^2 .

Let $i_1 < i_2 < \dots < i_m$ be all the indices with $\text{BWT}_1[i_q] = c$, $1 \leq q \leq m$. Because the suffixes in SA_1 are ordered lexicographically, we have $S_{\text{SA}_1[i_1]}^1 < S_{\text{SA}_1[i_2]}^1 < \dots < S_{\text{SA}_1[i_m]}^1$. Clearly, this implies $cS_{\text{SA}_1[i_1]}^1 < cS_{\text{SA}_1[i_2]}^1 < \dots < cS_{\text{SA}_1[i_m]}^1$. Let i_q be the largest index among i_1, \dots, i_m with $i_q \leq i$. Thus, $S_{\text{SA}_1[i_q]}^1 \leq S_{\text{SA}_1[i]}^1 < S_{\text{SA}_1[i_{q+1}]}^1$. This altogether yields

$$S_{\text{SA}_1[i_q]}^1 \leq S_{\text{SA}_1[i]}^1 < S_{n_2-k}^2 < S_{\text{SA}_1[i+1]}^1 \leq S_{\text{SA}_1[i_{q+1}]}^1$$

and has

$$cS_{\text{SA}_1[i_q]}^1 < cS_{n_2-k}^2 < cS_{\text{SA}_1[i_{q+1}]}^1$$

as a consequence. Because the q -th occurrence of c can be found at index i_q in BWT_1 , the suffix $cS_{\text{SA}_1[i_q]}^1$ can be found at index $C_1[c] + \text{rank}_c(\text{BWT}_1, i)$ in SA_1 . \square

Once the bit vector B is known, it is easy to compute the Burrows-Wheeler transformed string BWT of $S = S^1S^2$; see Algorithm 7.28. There is

one subtle difference, however, that should not go unnoticed. In BWT_1 the character $\#$ occurs at the index of the suffix S^1 and in BWT_2 the character $\$$ occurs at the index of the suffix S^2 . In BWT, however, $\$$ occurs at the index of the suffix S^1S^2 and $\#$ occurs at the index of the suffix S^2 . So the roles of $\#$ and $\$$ are exchanged (i.e., $\$$ separates S^1 and S^2 , and $\#$ is the last character of S).

Given the wavelet trees of BWT_1 and BWT_2 , Algorithm 7.27 has a time complexity of $O(n_2 \log \sigma)$, while Algorithm 7.28 takes $O(n_1 + n_2)$ time.

7.7 Space-efficient comparison of multiple strings

In this section, we review the most important algorithms that compare multiple strings from a different perspective. Recall that we solved several problems by means of the generalized suffix array of m strings S^1, S^2, \dots, S^m . This was constructed in linear time by sorting the suffixes of the concatenated string $S^1\#_1S^2\#_2\dots S^m\#_m$, where $\#_1, \#_2, \dots, \#_m$ are pairwise distinct separator symbols that do not occur in any of the strings. In some applications, the use of these m separator symbols is acceptable, but in others it is disadvantageous because it “blows up” the alphabet. In this section, we avoid different separator symbols: we use just one separator symbol $\#$ in addition to the sentinel $\$$, which marks the end of the string. For reasons that will become clear in Section 7.7.3, we prepend the symbol $\#$ to each of the m strings S^1, S^2, \dots, S^m and work with their concatenation $S = \#S^1\#S^2\dots\#S^m\$$. By definition, the $\#$ symbol left to S^j belongs to string j , and the sentinel $\$$ belongs to string 0. In the following, we assume that $\$ < \#$.

7.7.1 Document array, LCP-array, and correction terms

The first problem that should be addressed is the construction of the D -array, the document array. If the suffix array of the string $S = \#S^1\#S^2\dots\#S^m\$$ is available on disk, then the D -array can be computed space efficiently by Algorithm 7.29. It uses a bit vector $B_D[1..n]$ defined by $B_D[i] = 1$ if and only if $S[i] = \#$, which is prepared for constant time rank_1 queries (the idea stems from Sadakane [274]). As an example, we use the string $S = \#gaaa\#aac\#aag\#ga\#aaa\$$ and its suffix array SA depicted in Figure 7.25. The corresponding bit vector is $B_D = 100001000100010010000$. With that bit vector, one can determine in constant time that the suffix at index i belongs to the string $\text{rank}_1(B_D, \text{SA}[i])$. It should be stressed that both arrays, the suffix array and the D -array are accessed sequentially in Algorithm 7.29. So the suffix array can be streamed from disk and the D -array can be written to disk.

i	SA	LF	BWT	$S_{SA[i]}$	D
1	21	7	a	$\$$	0
2	17	8	a	$\#aaa\$$	5
3	6	9	a	$\#aac\#aag\#ga\#aaa\$$	2
4	10	18	c	$\#aag\#ga\#aaa\$$	3
5	14	19	g	$\#ga\#aaa\$$	4
6	1	1	$\$$	$\#gaaa\#aac\#aag\#ga\#aaa\$$	1
7	20	10	a	$a\$$	5
8	16	20	g	$a\#aaa\$$	4
9	5	11	a	$a\#aac\#aag\#ga\#aaa\$$	1
10	19	12	a	$aa\$$	5
11	4	13	a	$aa\#aac\#aag\#ga\#aaa\$$	1
12	18	2	$\#$	$aaa\$$	5
13	3	21	g	$aaa\#aac\#aag\#ga\#aaa\$$	1
14	7	3	$\#$	$aac\#aag\#ga\#aaa\$$	2
15	11	4	$\#$	$aag\#ga\#aaa\$$	3
16	8	14	a	$ac\#aag\#ga\#aaa\$$	2
17	12	15	a	$ag\#ga\#aaa\$$	3
18	9	16	a	$c\#aag\#ga\#aaa\$$	2
19	13	17	a	$g\#ga\#aaa\$$	3
20	15	5	$\#$	$ga\#aaa\$$	4
21	2	6	$\#$	$gaaa\#aac\#aag\#ga\#aaa\$$	1

Figure 7.25: The BWT of the string $S = \#gaaa\#aac\#aag\#ga\#aaa\$$.

Algorithm 7.29 Space-efficient computation of the D -array with SA.

```

 $D[1] \leftarrow 0$           /* suffix $ */
for  $i \leftarrow 2$  to  $n$  do
     $D[i] \leftarrow \text{rank}_1(B_D, \text{SA}[i])$ 
  
```

Algorithm 7.30 Computation of the D -array with LF .

```

 $D[1] \leftarrow 0$           /* suffix $ */
 $i \leftarrow LF(1)$ 
for  $j \leftarrow m$  downto 1 do
    for  $k \leftarrow 1$  to  $n_j + 1$  do
         $D[i] \leftarrow j$ 
         $i \leftarrow LF(i)$ 
  
```

If the wavelet tree of the BWT of the string $S = \#S^1\#S^2\ldots\#S^m\$$ is available, then the D -array can be obtained in almost the same fashion as the string S was obtained from BWT and LF ; cf. Algorithm 7.3 (page 286). Using the lengths n_1, n_2, \dots, n_m of S^1, S^2, \dots, S^m (so $\#S^j$ has length $n_j + 1$), Algorithm 7.30 computes the D -array solely based on LF . Note, however, that the D -array is accessed non-sequentially.

Exercise 7.7.1 Suppose that the lexicographic order of the strings S^1, S^2, \dots, S^m is known. Explain how the first $m + 1$ entries of the D -array can be derived from this knowledge. Conclude that the D -array can be computed in parallel.

The use of just one separator symbol $\#$ rises a new problem: the longest common prefix of two suffixes of S may contain $\#$ so that the LCP-array of S is not the intended one (i.e., it does not coincide with that of the generalized suffix array of S^1, S^2, \dots, S^m). For example, $\text{LCP}[9] = 4$ in Figure 7.25, but it should be 1. However, this problem can easily be fixed by a slight modification of Algorithm 7.17 (page 319).⁵ Recall that its queue Q initially contains the c -interval for each character of Σ . Instead of the $\#$ -interval $[2..m + 1]$, m singleton intervals $[2..2], [3..3], \dots, [m + 1..m + 1]$ are added to Q in the modified algorithm. In other words, the m occurrences of $\#$ are treated as if they were pairwise distinct. This modification of Algorithm 7.17 computes the intended LCP-array.

We have seen that several applications can benefit from the correction terms. These are calculated as described in Section 5.6.3, but the algorithm that computes the CT' -array (Algorithm 5.32 on page 220) is implemented with the balanced parentheses sequence BPS of the LCP-array.

⁵The procedure *getIntervals* from Algorithm 7.16 must also be changed so that it does not generate $\#\omega$ -intervals.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$CT'[i]$	0	1	1	1	0	2	1	0	0	3	1	0	1	0
$CT''[i]$	0	1	2	3	3	5	6	6	6	9	10	10	11	11

Figure 7.26: The arrays CT' and CT'' .

For this purpose, the BPS is preprocessed so that range minimum queries on the LCP-array can be answered in constant time; see Algorithm 6.10 (page 275). Furthermore, it is advantageous to use a succinct representation of the CT'' -array. This representation (suggested by Sadakane [274]) is a bit vector $B_{CT''}$, which we construct as follows: $B_{CT''}$ is initially empty and, for $i = 1, 2, \dots, n$, we successively append $CT'[i]$ many zero bits followed by a one bit to $B_{CT''}$. We use the example of Figure 7.26 to illustrate this. The bit vector corresponding to the CT'' -array is

$$B_{CT''} = 1\ 01\ 01\ 01\ 1\ 001\ 01\ 1\ 1\ 0001\ 01\ 1\ 01\ 1$$

Since Algorithm 5.32 (page 220) increments entries of the CT' -array at most n times, the bit vector $B_{CT''}$ occupies at most $2n$ bits (at most n zero bits and exactly n one bits). It is an immediate consequence of the definition $CT''[q] = \sum_{i=1}^q CT'[i]$ that $CT''[q]$ coincides with the number of zeros in $B_{CT''}[1..k]$, where k is the position of the q -th one in the bit vector. Thus, we preprocess $B_{CT''}$ so that $select_1$ queries can be answered in constant time. The value $CT''[q]$ can then be determined in constant time by $select_1(B_{CT''}, q) - q$, using only $2n + o(n)$ bits. Continuing our example, $select_1(B_{CT''}, 6) - 6 = 11 - 6 = 5$ yields the value of $CT''[6]$; cf. Figure 7.26.

Although the final implementation of the CT'' -array (the bit vector $B_{CT''}$) requires only $2n + o(n)$ bits, its computation uses the CT' -array, which requires $n \log n$ bits. It is possible to avoid this “memory peak” at the expense of slower preprocessing time. The CT'' -array can be implemented by a dynamic succinct data structure for *searchable partial sums* [111]; see e.g. [239] for such a data structure that supports the needed operations like *insert* in $O(\log n)$ time. Another possibility is to keep only parts of the CT' -array in main memory.

7.7.2 Document retrieval with wavelet trees

In this section, we shed some light on the usefulness of the wavelet tree of the D -array in document retrieval. For instance, Figure 7.27 shows the wavelet tree of the D -array of Figure 7.25. And so for the first time we will use the wavelet tree for a different purpose than string matching.

Let us revisit the document listing problem introduced in Section 5.6.4: given a database (or library) \mathcal{D} of strings (or documents) S^1, \dots, S^m on the

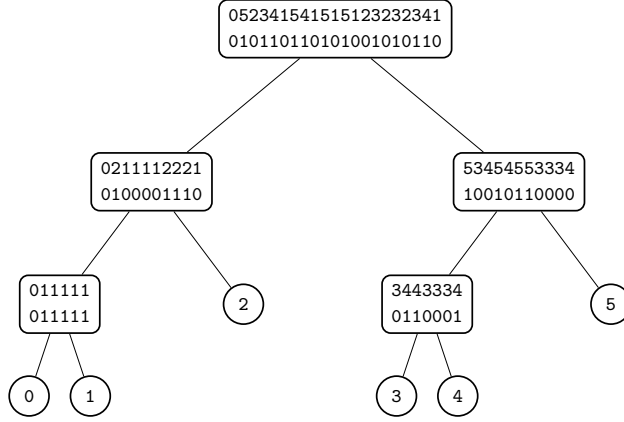


Figure 7.27: The wavelet tree of the document array D of Figure 7.25.

alphabet Σ , we wish to compute the set $dl(\phi)$ of documents in \mathcal{D} that contain the string ϕ . Algorithm 7.31 solves this problem with the help of the wavelet tree of the D -array. For a string ϕ , it computes the ϕ -interval $[p..q]$ by a backward search and then applies Algorithm 7.32 to the interval $[p..q]$. This algorithm returns a list $[\langle d_1, occ_1 \rangle, \dots, \langle d_k, occ_k \rangle]$ satisfying (i) $dl(\phi) = \{d_1, \dots, d_k\}$ and (ii) for all i the string ϕ occurs occ_i times in document S^{d_i} . Observe that Algorithm 7.32 is almost verbatim the same as Algorithm 7.16 (page 316); the sole differences are the statements after the conditional statement “**if** $l = r$ **then**” and that they use different wavelet trees. For this reason, an analysis of Algorithm 7.32 is left to the reader. We remark, however, that Algorithm 7.32 takes $O(\min(m, k \log m))$ time to return a k -element list.

Algorithm 7.31 does not only compute the set $dl(\phi) = \{d_1, \dots, d_k\}$ of all documents that contain ϕ but for each document number d_i it also returns the number $occ_i = occ_\phi(S^{d_i})$ of occurrences of ϕ in S^{d_i} . Because $df(\phi) = |dl(\phi)|$, it enables us to compute the TF-IDF score

$$occ_\phi(S^j) \log \frac{m}{df(\phi)}$$

of the string ϕ with respect to each document S^j (note that the score is zero when $j \notin dl(\phi)$); cf. Section 5.7.5.

As a matter of fact, with an approach similar to Algorithm 7.31, we can solve more general problems like multi-pattern document listing, document listing with forbidden patterns, or combinations thereof.

Definition 7.7.2 The *two-patterns document listing problem* is to preprocess a database \mathcal{D} of string documents S^1, \dots, S^m of total length n so that

Algorithm 7.31 Solving the document listing problem with the wavelet tree of the BWT and the wavelet tree of the D -array.

Let \mathcal{D} be a database of string documents S^1, \dots, S^m of total length n .

1. Preprocessing phase:
 - (a) Construct the wavelet tree of the Burrows-Wheeler transformed string of $S = \#S^1\#S^2\dots\#S^m\$$ in $O(n \log \sigma)$ time.
 - (b) Construct the wavelet tree of the D -array in $O(n \log m)$ time.
 2. Computation of $dl(\phi)$ for a string ϕ :
 - (a) Determine the ϕ -interval $[p..q]$ by a backward search in $O(|\phi| \log \sigma)$ time.
 - (b) Use Algorithm 7.32 to obtain a list $[\langle d_1, occ_1 \rangle, \dots, \langle d_k, occ_k \rangle]$ of pairs so that $dl(\phi) = \{d_1, \dots, d_k\}$ and ϕ occurs occ_i times in document S^{d_i} for all i with $1 \leq i \leq k$.
-

the following queries can be answered quickly: “Which documents contain both string patterns ϕ^1 and ϕ^2 ?”

In the *document listing problem with a forbidden pattern*, the query-type is: “Which documents contain the pattern ϕ^1 but *not* the (forbidden) pattern ϕ^2 ?”

The solutions to these two problems use exactly the same preprocessing phase as Algorithm 7.31. In phase 2a, they determine the ϕ^1 and ϕ^2 intervals $[p_1..q_1]$ and $[p_2..q_2]$ by a backward search. In phase 2b, the algorithms compute $dl(\phi^1)$ and for each $d \in dl(\phi^1)$ they test in $O(\log m)$ time if $rank_d(D, q_2) - rank_d(D, p_2 - 1) > 0$. Clearly, ϕ^2 occurs in document S^d if and only if this difference is strictly positive. Consequently, the “two-patterns” algorithm outputs d if this is the case, whereas the “forbidden-pattern” algorithm outputs d if this is not the case. Both algorithms need the wavelet trees of the BWT and the D -array. The preprocessing phase requires $O(n(\log \sigma + \log m))$ time, phase 2a takes $O((|\phi^1| + |\phi^2|) \log \sigma)$ time, and phase 2b runs in $O(df(\phi^1) \log m)$ time.

Let us illustrate the algorithms. If we are interested in all documents that contain both patterns ga and aa in the example of Figure 7.25 (page 346), then the “two-patterns” algorithm computes the ga -interval $[20..21]$ and the aa -interval $[10..15]$ by backward search (using the wavelet tree of the BWT) and then $dl(ga) = \{4, 1\}$ by the procedure call *getDocs*($[20..21]$) (using the wavelet tree of the document array D ; see Figure 7.27). After that, it computes $rank_4(D, 15) - rank_4(D, 9) = 2 - 2 = 0$ and $rank_1(D, 15) - rank_5(D, 9) = 4 - 2 = 2$ and outputs 1. If we were interested in all documents

Algorithm 7.32 For the ϕ -interval $[p..q]$, the function call $getDocs([p..q])$ returns a list $[\langle d_1, occ_1 \rangle, \dots, \langle d_k, occ_k \rangle]$ so that $dl(\phi) = \{d_1, \dots, d_k\}$ and ϕ occurs occ_i times in document S^{d_i} for all i with $1 \leq i \leq k$.

```

getDocs([p..q])
    list  $\leftarrow []$ 
    getDocs'([p..q], [1..m], list)
    return list

getDocs'([p..q], [l..r], list)
    if  $l = r$  then
        if  $p \leq q$  then
            add(list,  $\langle l, q - p + 1 \rangle$ )
        else
             $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, p - 1), rank_0(B^{[l..r]}, q))$ 
             $(a_1, b_1) \leftarrow (p - 1 - a_0, q - b_0)$ 
             $mid = \lfloor \frac{l+r}{2} \rfloor$ 
            if  $b_0 > a_0$  then
                getDocs'([a0 + 1..b0], [l..mid], list)
            if  $b_1 > a_1$  then
                getDocs'([a1 + 1..b1], [mid + 1..r], list)

```

that contain ga but not aa , then the “forbidden-pattern” algorithm would perform the same computations, but it would output 4.

The factor $df(\phi^1)$ in the worst-case time complexity of phase 2b of the “two-patterns” algorithm can be improved to $\min(df(\phi^1), df(\phi^2))$ as follows: Compute $df(\phi^1) = |dl(\phi^1)|$ and $df(\phi^2) = |dl(\phi^2)|$ in constant time with the aid of the correction terms as explained in Section 7.7.1 (keep in mind that this requires either more space or more time in the preprocessing phase). If $df(\phi^1) \leq df(\phi^2)$, then proceed as above; otherwise swap ϕ^1 and ϕ^2 in phase 2b.

Exercise 7.7.3 The *multi-pattern document listing problem with forbidden patterns* is to preprocess a database \mathcal{D} of string documents S^1, \dots, S^m of total length n so that queries of the following form can be answered quickly: “Which documents contain at least i of the r string patterns ϕ^1, \dots, ϕ^r and at most j of the s forbidden string patterns $\omega^1, \dots, \omega^s$?” Give an algorithm that solves this problem and analyze its worst-case time and space complexities.

Let us briefly comment on the history of the document listing problem. As we have seen in Section 5.6.4, Muthukrishnan [233] first showed that this problem can be solved in $O(|\phi| + k)$ time, where k is the number of distinct documents in which the string ϕ occurs. Given the ϕ -interval, his

Algorithm 7.33 Matching S^k backwards against the suffix array of S , where $S = \#S^1\#S^2\ldots\#S^m\$$.

```

initialize an empty queue  $Q$ 
 $lb \leftarrow 1$ 
 $rb \leftarrow n$           /*  $n = |S|$  */
 $p \leftarrow n_k$          /*  $n_k = |S^k|$  */
while  $p \geq 1$  do      /*  $S^k$  is a substring of  $S$ , so we always have  $lb \leq rb$  */
     $c \leftarrow S^k[p]$ 
     $lb \leftarrow C[c] + rank_c(\text{BWT}, lb - 1) + 1$ 
     $rb \leftarrow C[c] + rank_c(\text{BWT}, rb)$ 
     $i \leftarrow rank_{\#}(\text{BWT}, lb - 1) + 1$ 
     $j \leftarrow rank_{\#}(\text{BWT}, rb)$ 
    if  $i \leq j$  then
        enqueue( $Q, (n_k - p + 1, [i..j])$ )
     $p \leftarrow p - 1$ 
return  $Q$ 

```

technique used range minimum queries on the Prev-array to return all k documents that contain ϕ in optimal $O(k)$ time. Välimäki and Mäkinen [317] gave a more space efficient solution based on the wavelet tree of the D -array and an RMQ data structure on the Prev-array, but their solution uses $O(k \log m)$ time. Gagie et al. [118] observed that the wavelet tree of the D -array alone is sufficient to solve the problem in the same $O(k \log m)$ time bound, and Culpepper et al. [73] improved on it.

7.7.3 All-pairs suffix-prefix matching

From Section 5.6.7, we recall the *all-pairs suffix-prefix matching problem*: Given m strings S^1, S^2, \dots, S^m find, for all $k \neq l$ with $1 \leq k \leq m$ and $1 \leq l \leq m$, the *longest* suffix of S^k that is a prefix of S^l .

Simpson and Durbin [291] showed how an assembly string graph can be efficiently constructed using a backward search. Here, we solve the all-pairs suffix-prefix matching problem with the help of their search algorithm. It is convenient to work with the string $S = \#S^1\#S^2\ldots\#S^m\$$ because one of the strings S^1, S^2, \dots, S^m is a prefix of the suffix $S_{\text{SA}[k]}$ at index k if and only if $\text{BWT}[k] = \#$. To find out which one this is, we will use the array $A[1..m]$, which reflects the lexicographic order of S^1, S^2, \dots, S^m . More precisely, with the definition $A[1..m] = D[2..m+1]$, we have $S^{A[1]} \leq S^{A[2]} \leq \dots \leq S^{A[m]}$. Now, if $\text{BWT}[k] = \#$ is the d -th occurrence of $\#$ in the string BWT, then $S^{A[d]}$ is a prefix of the suffix $S_{\text{SA}[k]}$ at index k .

Algorithm 7.33 shows pseudo-code of Simpson and Durbin's [291] algorithm for finding overlaps between suffixes of string S^k and prefixes

i	BWT	$S_{SA[i]}$
1	a	$\$$
2	a	$\#aaa\$$
3	a	$\#aac\#aag\#ga\#aaa\$$
4	c	$\#aag\#ga\#aaa\$$
5	g	$\#ga\#aaa\$$
6	$\$$	$\#gaaa\#aac\#aag\#ga\#aaa\$$
7	a	$a\$$
8	g	$a\#aaa\$$
9	a	$a\#aac\#aag\#ga\#aaa\$$
10	a	$aa\$$
11	a	$aa\#aac\#aag\#ga\#aaa\$$
12	$\#$	$aaa\$$
13	g	$aaa\#aac\#aag\#ga\#aaa\$$
14	$\#$	$aac\#aag\#ga\#aaa\$$
15	$\#$	$aag\#ga\#aaa\$$
16	a	$ac\#aag\#ga\#aaa\$$
17	a	$ag\#ga\#aaa\$$
18	a	$c\#aag\#ga\#aaa\$$
19	a	$g\#ga\#aaa\$$
20	$\#$	$ga\#aaa\$$
21	$\#$	$gaaa\#aac\#aag\#ga\#aaa\$$

Figure 7.28: BWT of $S = \#gaaa\#aac\#aag\#ga\#aaa\$$, where $A = [5, 2, 3, 4, 1]$.

of the other strings. The algorithm uses backward search to determine the ω -interval $[lb..rb]$ of a suffix ω of S^k (it considers suffixes of S^k in increasing order of length). Then, it determines the interval $[i..j]$ by $i = \text{rank}_{\#}(\text{BWT}, lb - 1) + 1$ and $j = \text{rank}_{\#}(\text{BWT}, rb)$. As explained above, ω is a prefix of all strings in $A[i..j]$ (in particular, if $[i..j]$ is empty, then ω is not a prefix of any of the strings S^1, S^2, \dots, S^m). The algorithm stores the pair $([\omega], [i..j])$ in a queue Q provided that $i \leq j$.

To illustrate how Algorithm 7.33 works, we apply it to the example from Figure 7.28, i.e., we match $S^1 = gaaa$ backwards against the suffix array of the concatenated string S . In the first iteration of the while-loop, the algorithm determines the a -interval $[7..17]$ because a is the last character of S^1 . In this iteration, it computes $i = 1$ and $j = 3$, and adds the pair $(1, [1..3])$ to the initially empty queue Q . This means that the strings S^5, S^2 , and S^3 have the length 1 suffix of S^1 as a prefix because $A[1..3] = [5, 2, 3]$. In the second iteration of the while-loop, the algorithm searches for the

aa-interval and finds $[10..15]$. Afterwards, it computes $i = 1$ and $j = 3$, and adds the pair $(2, [1..3])$ to Q . In the third iteration, it determines the *aaa*-interval $[12..13]$, computes $i = 1$ and $j = 1$, and adds $(3, [1..1])$ to Q . Finally, it computes the *aaaa*-interval $[21..21]$, adds the pair $(4, [5..5])$ to Q , and returns the queue

$$Q = [(1, [1..3]), (2, [1..3]), (3, [1..1]), (4, [5..5])]$$

In general, Algorithm 7.33 delivers a queue of pairs

$$[(\ell_1, [i_1..j_1]), (\ell_2, [i_2..j_2]), \dots, (\ell_q, [i_q..j_q])]$$

of all matchings of suffixes of S^k with prefixes of S^1, S^2, \dots, S^m (in increasing order of their first component).

It remains to solve the problem of finding all *longest* suffix-prefix matchings. For any k , we would like to find all longest overlaps of suffixes of S^k with prefixes of the other strings. Given the queue Q returned by Algorithm 7.33, we can find the length of the longest suffix-prefix match of S^k with $S^{A[l]}$ as follows: determine the largest subscript p ($1 \leq p \leq q$) so that l is contained in the interval $[i_p..j_p]$ and output ℓ_p . Exercise 7.7.4 allows us to restate this task into a problem that can be solved with a line-sweep algorithm (this type of algorithm is a central concept in computational geometry).

Exercise 7.7.4 Let $(\ell_r, [i_r..j_r])$ and $(\ell_s, [i_s..j_s])$ be two elements in the queue Q with $\ell_r < \ell_s$. Show that either

- $[i_s..j_s]$ is a subinterval of $[i_r..j_r]$, i.e., $i_r \leq i_s \leq j_s \leq j_r$, or
- $[i_r..j_r]$ and $[i_s..j_s]$ are disjoint, i.e., $j_r < i_s$ or $j_s < i_r$.

The idea behind line-sweep algorithms is to imagine that a line (often a vertical line) is swept or moved across the plane, stopping at some points. Here, we have a one-dimensional problem: our algorithm moves a vertical line along a horizontal line (the x-axis) and stops whenever it finds the left- or right boundary of an interval of the queue Q . To ease the detection of these points, we slightly modify Algorithm 7.33. Instead of using one queue, we use m queues Q_1, \dots, Q_m , which are initially empty. Furthermore, instead of storing a pair $(\ell, [i..j])$ in the queue Q , we store the pair (ℓ, j) in the queue Q_i . The key feature of this approach is that now the intervals are sorted by their left boundary and the contents of a queue Q_i is sorted by the first component.

Algorithm 7.34 implements the line-sweep algorithm. It maintains a stack, which is initially empty. Elements on the stack are triples $\langle \ell, lb, rb \rangle$, where ℓ is the number of matching characters, lb is the left boundary of the interval, and rb is its right boundary. In its for-loop, the algorithm scans

Algorithm 7.34 Computing all longest suffix-prefix matches between S^k and S^1, \dots, S^m .

```

initialize an empty stack
for  $i \leftarrow 1$  to  $m$  do
    while  $Q_i$  is not empty do
         $(\ell, j) \leftarrow \text{dequeue}(Q_i)$ 
         $\text{push}(\langle \ell, i, j \rangle)$ 
    if stack is not empty then
        output longest suffix-prefix match of  $S^k$  and  $S^{A[i]}$  has length  $\text{top}().\ell$ 
        while  $\text{top}().rb = i$  do
             $\text{pop}()$ 

```

the region $1, \dots, m$ from left to right. For each value of the loop variable i , it proceeds as follows: First, if the queue Q_i is not empty, it dequeues the front element (ℓ, j) from Q_i and pushes the triple $\langle \ell, i, j \rangle$ onto the stack.⁶ This process is repeated until Q_i is empty. Second, if the stack is not empty, then the algorithm reports that the longest suffix-prefix match of S^k and $S^{A[i]}$ has length ℓ , where ℓ is the first component of the topmost element of the stack. Third, as long as $\text{top}().rb = i$, it pops the topmost element from the stack.

To illustrate how the algorithm works, we continue the example of Figure 7.28. The modified version of Algorithm 7.33 applied to S^1 returns the queues $Q_1 = [(1, 3), (2, 3), (3, 1)]$ and $Q_5 = [(4, 5)]$ (the other queues are empty). Now we apply Algorithm 7.34. For $i = 1$, it first pushes $\langle 1, 1, 3 \rangle$, then $\langle 2, 1, 3 \rangle$, and eventually $\langle 3, 1, 1 \rangle$ onto the stack. After that, it reports that the longest suffix-prefix match of S^1 and $S^{A[1]} = S^5$ has length 3. Since $i = 1 = \text{top}().rb$, it pops $\langle 3, 1, 1 \rangle$ from the stack. For $i = 2$, the algorithm merely outputs 2 as the length of the longest suffix-prefix match of S^1 and $S^{A[2]} = S^2$ because Q_2 is empty and $\text{top}().rb = 3$. For $i = 3$, the algorithm also outputs 2 as the length of the longest suffix-prefix match of S^1 and $S^{A[3]} = S^3$. Furthermore, it pops two elements from the stack because $\text{top}().rb = 3$ holds twice; the stack is now empty. For $i = 4$, there is nothing to do, but for $i = 5$ the queue $Q_5 = [(4, 5)]$ is not empty. Therefore, the algorithm pushes $\langle 4, 5, 5 \rangle$ onto the stack, reports that the longest suffix-prefix match of S^1 and $S^{A[5]} = S^1$ has length 4, and pops $\langle 4, 5, 5 \rangle$ from the stack.

Theorem 7.7.5 Algorithms 7.33 and 7.34 correctly solve the problem of finding all longest suffix-prefix overlaps.

Proof Algorithm 7.33 successively computes pairs $(\ell, [i..j])$, for ℓ increasing from 1 to n_k , so that the length ℓ suffix ω of S^k is a prefix of S^d if

⁶In fact, i need not to be stored, but this makes the correctness proof easier.

and only if $d \in A[i..j]$. As discussed above, the modified version of this algorithm stores a pair $(\ell, [i..j])$ by adding (ℓ, j) to the queue Q_i . The algorithm returns queues Q_1, \dots, Q_m (some of which may be empty). The contents of a non-empty queue Q_i is sorted by the first component, i.e., if $Q_i = [(\ell'_1, j'_1), \dots, (\ell'_{q_i}, j'_{q_i})]$, then $\ell'_1 < \dots < \ell'_{q_i}$. According to Exercise 7.7.4, if $\ell'_r < \ell'_s$, then $[i..j'_s]$ is a subinterval of $[i..j'_r]$ (i.e., $j'_s \leq j'_r$). We conclude that $j'_{q_i} \leq \dots \leq j'_1$.

We show that Algorithm 7.34 outputs the lengths of all longest suffix-prefix overlaps of S^k and S^1, \dots, S^m . Moreover, we prove that the algorithm maintains the following invariant: before iteration i , if the stack contains the triples $\langle \ell_1, i_1, j_1 \rangle, \dots, \langle \ell_t, i_t, j_t \rangle$ (from bottom to top), then

- $1 \leq \ell_1 < \dots < \ell_t$,
- $1 \leq i_1 \leq \dots \leq i_t < i$,
- $i \leq j_t \leq \dots \leq j_1$.

Clearly, the invariant holds before the first iteration of the for-loop. For an inductive proof, suppose the invariant holds before the i -th iteration. If the queue Q_i is not empty, say $Q_i = [(\ell'_1, j'_1), \dots, (\ell'_{q_i}, j'_{q_i})]$, then the algorithm dequeues the front element (ℓ'_1, j'_1) from Q_i and pushes $\langle \ell'_1, i, j'_1 \rangle$ onto the stack. We show that $\ell_t < \ell'_1$ and $j'_1 \leq j_t$, where $\langle \ell_t, i_t, j_t \rangle$ is the topmost element of the stack. For an indirect proof, suppose that $\ell'_1 < \ell_t$ (note that $\ell_t = \ell'_1$ is impossible). According to Exercise 7.7.4, $[i..j'_1]$ must be a subinterval of $[i..j'_t]$ (clearly, the intervals cannot be disjoint). That is, $i \leq i_t \leq j_t \leq j'_1$. This, however, contradicts $i_t < i$. Thus, $\ell_t < \ell'_1$. It follows from Exercise 7.7.4 that $[i..j'_1]$ is a subinterval of $[i..j_t]$. Hence $j'_1 \leq j_t$.

After all elements have been removed from Q_i and pushed onto the stack, we have

- $1 \leq \ell_1 < \dots < \ell_t < \ell'_1 < \dots < \ell'_{q_i}$,
- $1 \leq i_1 \leq \dots \leq i_t < i \leq \dots \leq i$,
- $i \leq j'_{q_i} \leq \dots \leq j'_1 \leq j_t \leq \dots \leq j_1$.

At that point in time, the contents of the stack represents all pairs $\langle \ell', [i'..j'] \rangle$ with $i' \leq i \leq j'$ and the topmost element contains the length of the longest suffix-prefix match of S^k and $S^{A[i]}$ unless the stack is empty. Therefore, the output of Algorithm 7.34 is correct.

Furthermore, the invariant holds before iteration $i + 1$ because at the end of the i -th iteration the algorithm pops elements from the stack while $top().rb = i$. \square

Let us analyze the time complexity of this solution to the longest suffix-prefix matching problem. It is readily verified that Algorithm 7.33 has a

time complexity of $O(n_k \log \sigma)$ if the wavelet tree of the BWT of S is used to support backward search. In Algorithm 7.34, at most n_k elements enter and leave the stack because there are at most n_k elements in the queues Q_1, \dots, Q_m . Moreover, exactly m suffix-prefix matches are output. Thus, Algorithm 7.34 takes $O(n_k + m)$ time. It follows as a consequence that the overall time complexity to compute all longest suffix-prefix matches is $O(n \log \sigma + m^2)$, where $n = |S| = m + \sum_{k=1}^m n_k$.

Exercise 7.7.6 Modify Algorithm 7.34 so that it does not report the longest suffix-prefix match of S^k and itself.

7.8 Bidirectional search

As explained in Chapter 1, RNA molecules play an active role in gene expression, but they also catalyze biological reactions and communicate responses to cellular signals. Most biologically active RNAs, including mRNA, tRNA, and rRNA contain self-complementary sequences that allow parts of the RNA to fold and pair with itself; see Figure 7.29. In RNA molecules, Watson-Crick base pairs (G–C and A–U) and non-Watson-Crick interactions (e.g. G–U) allow RNAs to fold into a vast range of specific structures. RNAs are highly conserved in the course of evolutionary time, not on the sequence level, but as secondary structures. Thus, the task of finding the genes coding for a certain RNA in a genome is to find all regions in the genomic DNA sequence that match its structural pattern. Because the structural pattern often consists of a hairpin loop and a stem (which may also have bulges), the most efficient algorithms first search for candidate regions matching the loop and then try to extend both ends by searching for complementary base pairs A–U, G–C, or G–U that form the stem. For example, in Figure 7.29 the loop is the sequence UCGCCGU, so one must search for all occurrences of the pattern TCGCCGT in the genomic DNA sequence. The loop is then extended by one of the four nucleotides to the left or to the right, say by A to the left. Consequently, all regions in the DNA sequence matching ATCGCCGT are searched for. That is, given the TCGCCGT-interval, one determines the ATCGCCGT-interval by a backward search. If the A to the left of TCGCCGT is part of the stem, then it must pair with T (one must search for the pairs A–T, G–C, or G–T in the DNA sequence because T is replaced with U in the transcription from DNA to RNA). In other words, in the next step one searches for all regions in the DNA sequence matching ATCGCCGTT. More precisely, given the ATCGCCGT-interval, one determines the ATCGCCGTT-interval by a forward search. Such a search strategy can be pursued only if bidirectional search is possible; see [217, 302].

In Section 7.8.4, we will present a data structure that supports bidirectional searching [284]. It consists of the wavelet tree of the Burrows-

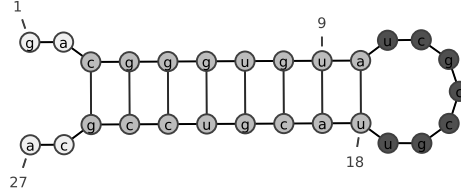


Figure 7.29: A hairpin loop.

Wheeler transformed string of S (supporting backward search) and the wavelet tree of the Burrows-Wheeler transformed string BWT^{rev} of S^{rev} (supporting forward search).

The Burrows-Wheeler transform BWT of a string S can be computed by sorting all suffixes of S (hence the suffix array of S is known). Of course, the Burrows-Wheeler transform BWT^{rev} of the reverse string S^{rev} can be obtained in the same fashion. However, because of the strong relationship between a string and its reverse, it is quite natural to ask whether BWT^{rev} can be directly derived from BWT *without* sorting the suffixes of S^{rev} . In Section 7.8.1, we prove that this is indeed the case. More precisely, we give an algorithm for this task that has $O(n \log \sigma)$ worst-case time complexity. The same algorithm computes the arrays SA^{rev} and LCP^{rev} , but only partially. In Sections 7.8.2 and 7.8.3, we show how these arrays can be completed. Furthermore, in Section 7.8.3 it is shown that LCP^{rev} is a permutation of LCP. These results originate from [246].

7.8.1 Burrows-Wheeler transform of the reverse string

If we reverse the order of the characters in a string, we obtain its *reverse string*. For technical reasons, however, we assume that the sentinel symbol $\$$ occurs at the end of each string under consideration. For this reason, the reverse string S^{rev} of a string S that is terminated by $\$$ is obtained by deleting $\$$ from S , reversing the order of the characters, and appending $\$$. For example, the reverse string of $S = \text{ctaataatg}\$$ is $S^{\text{rev}} = \text{gtaataatc}\$$ (and not $\text{\$gtaataatc}$).

Algorithm 7.35 recursively computes the Burrows-Wheeler transformed string BWT^{rev} of S^{rev} by the procedure call $\text{bwtrev}(1, [1..n], 0)$. We stress that the procedure getIntervals , on which Algorithm 7.35 relies, must also generate $\$$ -intervals; this can be easily achieved by omitting the condition “**if** $c \neq \$$ **then**” in Algorithm 7.16 (page 316). The string BWT^{rev} is computed in a left-to-right fashion: first $\text{BWT}^{\text{rev}}[1]$, then $\text{BWT}^{\text{rev}}[2]$, etc. Suppose that the algorithm has already calculated the first $k - 1$ characters of BWT^{rev} . When it tries to determine $\text{BWT}^{\text{rev}}[k]$, it is known that $[k..k + rb - lb]$ is the

Algorithm 7.35 Procedure $bwtrev(k, [i..j], \ell)$ uses the wavelet tree of the BWT, the suffix array SA, and S . The call $bwtrev(1, [1..n], 0)$ computes BWT^{rev} .

```

bwtrev( $k, [i..j], \ell$ )
   $list \leftarrow getIntervals([i..j])$  /* intervals in increasing lexicographic order */
  while  $list$  not empty do
     $[lb..rb] \leftarrow head(list)$ 
    if  $lb = rb$  or  $S[SA[lb] + \ell + 1] = S[SA[rb] + \ell + 1]$  then
       $pos \leftarrow SA[lb] + \ell + 1$ 
      if  $lb = rb$  then
        if  $pos > n$  then
           $pos \leftarrow pos - n$ 
           $SA^{rev}[k] \leftarrow n - pos + 1$  /* this will be explained in Section 7.8.2 */
           $c \leftarrow S[pos]$ 
          for  $q \leftarrow k$  to  $k + rb - lb$  do
             $BWT^{rev}[q] \leftarrow c$ 
             $count[c] \leftarrow count[c] + 1$  /* this will be explained in Section 7.8.2 */
             $LF^{rev}[q] \leftarrow count[c]$  /* this will be explained in Section 7.8.2 */
        else
           $bwtrev(k, [lb..rb], \ell + 1)$ 
       $k \leftarrow k + rb - lb + 1$ 
    if  $list$  not empty then
       $LCP^{rev}[k] \leftarrow \ell$  /* this will be explained in Section 7.8.3 */

```

$\omega^{rev}c$ -interval in SA^{rev} because the $c\omega$ -interval $[lb..rb]$ in SA has been identified by a backward search (with the help of the wavelet tree of BWT). At that point, the algorithm proceeds by case analysis (below, $\ell + 1$ is the length of $c\omega$):

- If each occurrence of $c\omega$ in S is followed by the same character a (in particular, this is true whenever $lb = rb$), then each occurrence of $\omega^{rev}c$ is preceded by a in S^{rev} . Thus, $BWT^{rev}[q] = a$ for every q with $k \leq q \leq k + rb - lb$.
- If not all the characters in $BWT^{rev}[k..k + rb - lb]$ are the same, then the algorithm recursively determines the lexicographic order of the suffixes in the $\omega^{rev}c$ -interval $[k..k + rb - lb]$ as far as it is needed.

We exemplify Algorithm 7.35 by applying it to $S = ctaataatg\$$. The procedure call $getIntervals([1..10])$ returns the list $[[1..1], [2..5], [6..6], [7..7], [8..10]]$, where $[1..1]$ is the $\$$ -interval, $[2..5]$ is the a -interval, and so on; cf. Figure 7.30. Then, the first interval $[lb..rb] = [1..1]$ is taken from the list (note that $head(list)$ removes the first element of $list$ and returns it). Because $lb = rb$, the algorithm computes $pos = SA[1] + 0 + 1 = 11$. Furthermore, since

i	SA	BWT	$S_{SA[i]}$	i	SA	BWT	$S_{SA[i]}^{rev}$	LCP	LF
1	10	g	$\$$	1	<u>10</u>	c	$\$$	<u>-1</u>	6
2	3	t	$aataatg\$$	2	<u>3</u>	t	$aataatc\$$	<u>0</u>	8
3	6	t	$aatg\$$	3	6	t	$aatc\$$	3	9
4	4	a	$ataatg\$$	4	4	a	$ataatc\$$	<u>1</u>	2
5	7	a	$atg\$$	5	7	a	$atc\$$	2	3
6	1	$\$$	$ctaataatg\$$	6	<u>9</u>	t	$c\$$	<u>0</u>	10
7	9	t	$g\$$	7	<u>1</u>	$\$$	$gtaataatc\$$	<u>0</u>	1
8	2	c	$taataatg\$$	8	<u>2</u>	g	$taataatc\$$	<u>0</u>	7
9	5	a	$taatg\$$	9	<u>5</u>	a	$taatc\$$	<u>4</u>	4
10	8	a	$tg\$$	10	<u>8</u>	a	$tc\$$	<u>1</u>	5

Figure 7.30: Left-hand side: Suffix array SA and BWT of the string $S = ctaataatg\$$ (the input). Right-hand side: Burrows-Wheeler transform of $S^{rev} = gtaataatc\$$ (the output). The computation of the suffix array and the lcp-array of S^{rev} will be explained in Section 7.8.2 and Section 7.8.3, respectively.

$pos > n = 10$, it assigns the character $S[11 - 10] = S[1] = c$ to $BWT^{rev}[1]$ and increments k (so the new value of k is 2). Now the interval $[lb..rb] = [2..5]$ is taken from the list. Because $S[SA[lb] + \ell + 1] = S[SA[2] + 0 + 1] = S[4] = a \neq t = S[8] = S[SA[5] + 0 + 1] = S[SA[rb] + \ell + 1]$, Algorithm 7.35 recursively calls $bwtrv(2, [2..5], 1)$. The procedure call $getIntervals([2..5])$ returns the list $[[2..3], [8..9]]$, where $[2..3]$ is the aa -interval and $[8..9]$ is the ta -interval. Then, the first interval $[lb..rb] = [2..3]$ is taken from the list. In this case, $S[SA[lb] + \ell + 1] = S[SA[2] + 1 + 1] = S[5] = t = t = S[8] = S[SA[3] + 1 + 1] = S[SA[rb] + \ell + 1]$. Thus, t is assigned to both $BWT^{rev}[2]$ and $BWT^{rev}[3]$. Now the algorithm continues with the new value $k = 2 + 3 - 2 + 1 = 4$. Figure 7.31 shows the recursion tree of Algorithm 7.35.

In essence, the correctness of Algorithm 7.35 is a consequence of the following lemma.

Lemma 7.8.1 *Let $[i..j]$ be the ω -interval for some substring ω of S , and let k be the left boundary of the ω^{rev} -interval in SA^{rev} . If $[lb_1..rb_1], \dots, [lb_m..rb_m]$ are the intervals in $list = getIntervals([i..j])$ corresponding to the strings $c_1\omega, \dots, c_m\omega$, where $c_1 < \dots < c_m$, then the intervals $[s_1..e_1], \dots, [s_m..e_m]$ in SA^{rev} corresponding to the strings $\omega^{rev}c_1, \dots, \omega^{rev}c_m$ satisfy $s_q = k + \sum_{p=1}^{q-1} (rb_p - lb_p + 1)$ and $e_q = s_q + (rb_q - lb_q)$, where $1 \leq q \leq m$.*

Proof We prove the lemma by finite induction on q . In the base case $q = 1$. Because c_1 is the smallest character in Σ for which the $c_1\omega$ -interval is non-empty, the suffixes of S^{rev} that have $\omega^{rev}c_1$ as a prefix are lexicographically

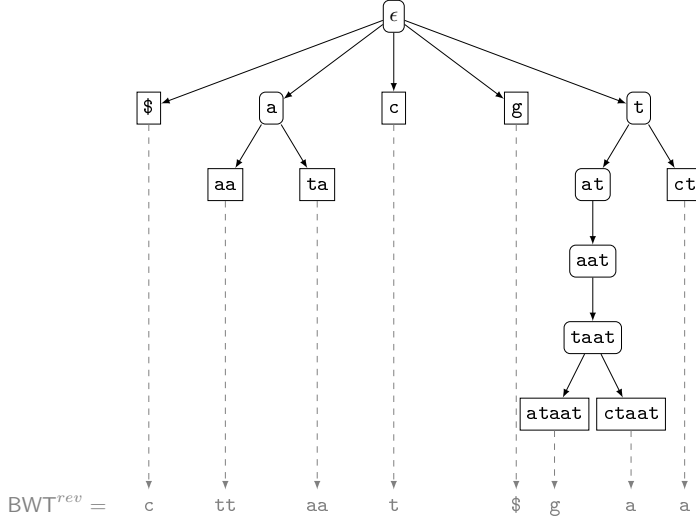


Figure 7.31: The recursion tree of Algorithm 7.35 applied to the BWT of $S = ctaataatg\$$: for each internal node there is a recursive call. For example, the recursive call with the aat -interval as a parameter determines the characters of BWT^{rev} in the taa -interval of SA^{rev} .

smaller than those suffixes of S^{rev} that have $\omega^{rev}c_p$, $2 \leq p \leq m$, as a prefix. Hence $s_1 = k$. Moreover, it follows from the fact that the $\omega^{rev}c_1$ -interval has size $rb_1 - lb_1 + 1$ that the $\omega^{rev}c_1$ -interval is the interval $[k..k + (rb_1 - lb_1)]$. For the inductive step suppose that the $\omega^{rev}c_{q-1}$ -interval $[s_{q-1}..e_{q-1}]$ satisfies $s_{q-1} = k + \sum_{p=1}^{q-2} (rb_p - lb_p + 1)$ and $e_{q-1} = s_{q-1} + (rb_{q-1} - lb_{q-1})$. Because c_q is the q -th smallest character in Σ for which the $c_q\omega$ -interval is non-empty, the suffixes of S^{rev} that have $\omega^{rev}c_q$ as a prefix are lexicographically larger than the suffixes of S^{rev} that have $\omega^{rev}c_p$ as a prefix, where $1 \leq p \leq q-1$. Since the suffixes of S^{rev} that have $\omega^{rev}c_q$ as a prefix are lexicographically smaller than the suffixes of S^{rev} that have $\omega^{rev}c_p$ as a prefix, where $q+1 \leq p \leq m$, it follows that the $\omega^{rev}c_q$ -interval $[s_q..e_q]$ satisfies $s_q = e_{q-1} + 1 = k + \sum_{p=1}^{q-1} (rb_p - lb_p + 1)$ and $e_q = s_q + (rb_q - lb_q)$. \square

Theorem 7.8.2 Algorithm 7.35 correctly computes BWT^{rev} .

Proof We prove the theorem by induction on k . Suppose Algorithm 7.35 is applied to the ω -interval $[i..j]$ for some ℓ -length substring ω of S , and let the $c\omega$ -interval $[lb..rb]$ be the interval dealt with in the current execution of the while-loop. According to the inductive hypothesis, $BWT^{rev}[1..k-1]$ has

been computed correctly. Moreover, by Lemma 7.8.1, $[k..k + rb - lb]$ is the $\omega^{rev}c$ -interval in SA^{rev} . We proceed by case analysis.

- If $lb = rb$, then $c\omega$ occurs exactly once in S and is the length $\ell + 1$ prefix of suffix $S_{SA[lb]}$. In this case, the suffix of S^{rev} that has $\omega^{rev}c$ as a prefix is the k -th lexicographically smallest suffix of S^{rev} . The character $BWT^{rev}[k]$ is $S[SA[lb] + \ell + 1]$ because this is the character that immediately follows the prefix $S[SA[lb]..SA[lb] + \ell] = c\omega$ of suffix $S_{SA[lb]}$.
- If $lb \neq rb$ and $S[SA[lb] + \ell + 1] = S[SA[rb] + \ell + 1]$, then each occurrence of $c\omega$ in S is followed by the same character $a = S[SA[lb] + \ell + 1]$. Thus, each suffix of S^{rev} in the $\omega^{rev}c$ -interval $[k..k + rb - lb]$ is preceded by a . Consequently, $BWT^{rev}[q] = a$ for every q with $k \leq q \leq k + rb - lb$. So in this case, it is not necessary to know the exact lexicographic order of the suffixes in the $\omega^{rev}c$ -interval.
- If $lb \neq rb$ and $S[SA[lb] + \ell + 1] \neq S[SA[rb] + \ell + 1]$, then not all the characters in $BWT^{rev}[k..k + rb - lb]$ are the same and thus the recursive call $bwtrev(k, [lb..rb], \ell + 1)$ determines the lexicographic order of the suffixes in the $\omega^{rev}c$ -interval $[k..k + rb - lb]$ as far as it is needed.

□

Next, we analyze the worst-case time complexity of Algorithm 7.35.

Lemma 7.8.3 *The procedure $bwtrev$ is executed with the parameters $(k, [i..j], \ell)$, where $[i..j]$ is the ω -interval for some substring ω of S , if and only if $\bar{\omega}$ is an internal node in the suffix tree ST of S .*

Proof We use induction on ℓ . There is just one procedure call with $\ell = 0$, namely $bwtrev(1, [1..n], 0)$. The interval $[1..n]$ is the ε -interval, where ε denotes the empty string. Clearly, the node $\bar{\varepsilon}$ is the root node of the suffix tree ST , and the root is an internal node. According to the inductive hypothesis, the procedure $bwtrev$ is executed with the parameters $(k, [i..j], \ell)$, where $[i..j]$ is the ω -interval for some substring ω of S , if and only if $\bar{\omega}$ is an internal node in the suffix tree ST of S . For the inductive step, assume that $[lb..rb]$ is one of the intervals returned by the procedure call $getIntervals([i..j])$, say the $c\omega$ -interval. We prove that there is a recursive procedure call $bwtrev(k', [lb..rb], \ell + 1)$ if and only if $\overline{c\omega}$ is an internal node of ST . It is clear that $\overline{c\omega}$ is an internal node of ST if and only if the $c\omega$ -interval contains two different suffixes of S , one having $c\omega a$ as a prefix and one having $c\omega b$ as a prefix, where a and b are different characters from Σ . Again, we proceed by case analysis:

- If $lb = rb$, then $c\omega$ occurs exactly once in S . Hence $\overline{c\omega}$ is not an internal node of ST . Note that Algorithm 7.35 does not invoke $bwtrev$.

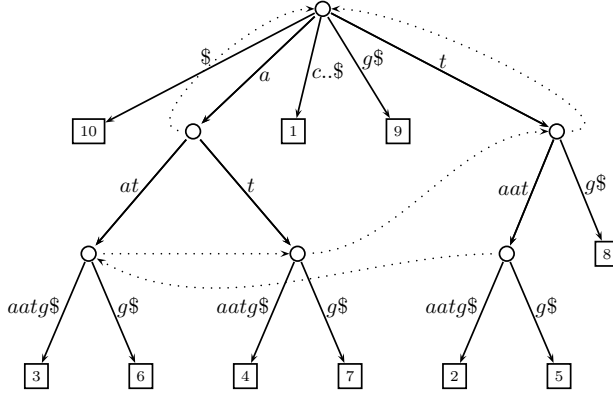


Figure 7.32: The suffix tree for $S = ctaataatg\$$. Suffix links are depicted by dotted arrows.

- If $S[SA[lb] + \ell + 1] = S[SA[rb] + \ell + 1]$, then each occurrence of $c\omega$ in S is followed by the same character. Again, $\overline{c\omega}$ is not an internal node of ST and Algorithm 7.35 does not invoke *bwtrev*.
- If $a = S[SA[lb] + \ell + 1] \neq S[SA[rb] + \ell + 1] = b$, then $\overline{c\omega}$ is an internal node of ST and Algorithm 7.35 invokes *bwtrev*(k' , $[lb..rb]$, $\ell + 1$).

□

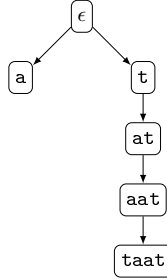
Lemma 7.8.3 implies that the recursion tree of Algorithm 7.35 coincides with the suffix link tree SLT of S , defined as follows.

Definition 7.8.4 The suffix link tree SLT of a suffix tree ST has a node $\underline{\omega}$ for each internal node $\overline{\omega}$ of ST. For each suffix link $slink(\overline{c\omega})$ from $\overline{c\omega}$ to $\overline{\omega}$ in ST, there is an edge $\underline{\omega} \rightarrow \underline{c\omega}$ in SLT.

SLT is indeed a tree: this follows from the fact that each internal node in ST has exactly one suffix link; cf. [93, 123]. Figure 7.32 shows the suffix tree of the string $S = ctaataatg\$$ and Figure 7.33 shows the corresponding suffix link tree.

That the recursion tree of Algorithm 7.35 coincides with the suffix link tree SLT of S can be seen as follows. If the execution of *bwtrev*(k , $[i..j]$, ℓ) invokes the recursive call *bwtrev*(k' , $[lb..rb]$, $\ell + 1$), where $[i..j]$ is the ω -interval and $[lb..rb]$ is the $c\omega$ -interval, then there is a suffix link from node $\overline{c\omega}$ to node $\overline{\omega}$ because both are internal nodes in the suffix tree of S .

Theorem 7.8.5 Algorithm 7.35 has a worst-case time complexity of $O(n \log \sigma)$.

Figure 7.33: The suffix link tree for $S = ctaataatg\$$.

Proof According to Lemma 7.8.3, there are as many recursive calls to the procedure *bwtrev* as there are internal nodes in the suffix tree ST of S . Because ST has n leaves and each internal node in ST is branching, the number of internal nodes is at most $n - 1$. We use an amortized analysis to show that the overall number of intervals returned by calls to the procedure *getIntervals* is bounded by $2n - 1$. Let L denote the concatenation of all lists returned by procedure calls to *getIntervals*. For each element $[lb..rb]$ of L , either

- at least one entry of BWT^{rev} is filled in, or
- there is a recursive call to the procedure *bwtrev*.

It follows that L has at most $2n - 1$ elements because BWT^{rev} has n entries and there are at most $n - 1$ recursive calls to the procedure *bwtrev*. It is a consequence of this amortized analysis that the overall time taken by all procedure calls to *getIntervals* is $O(n \log \sigma)$ because a procedure call to *getIntervals* that returns a k -element list takes $O(k \log \sigma)$ time. Clearly, the theorem follows from this fact. \square

Exercise 7.8.6 Given the Burrows-Wheeler transform of a DNA sequence, sketch an algorithm that calculates the Burrows-Wheeler transform of the reverse complement of the DNA sequence.

7.8.2 The suffix array of the reverse string

Algorithm 7.35 recursively computes the *whole* Burrows-Wheeler transformed string BWT^{rev} of S^{rev} , but it cannot be used to calculate the *whole* suffix array SA^{rev} . This is because an S^{rev} -value can be assigned in only one of the two base cases of the recursion.

- In the base case $lb = rb$, the character $c = S[pos]$ is assigned to $\text{BWT}^{rev}[k]$; see Algorithm 7.35. If $c \neq \$$, then this occurrence of c

Algorithm 7.36 Given a partial suffix array, these procedures compute the whole suffix array SA.

<pre> for $i \leftarrow 1$ to n do if $SA[i] \neq \perp$ then $j \leftarrow SA[i]$ $k \leftarrow LF[i]$ while $SA[k] = \perp$ do $j \leftarrow j - 1$ $SA[k] \leftarrow j$ $k \leftarrow LF[k]$ </pre>	<pre> initialize an empty stack for $i \leftarrow 1$ to n do $k \leftarrow i$ while $SA[k] = \perp$ do $push(k)$ $k \leftarrow LF[k]$ $j \leftarrow SA[k]$ while stack is not empty do $j \leftarrow j + 1$ $SA[pop()] \leftarrow j$ </pre>
--	---

appears at position $n - pos$ in S^{rev} . Thus, $SA^{rev}[k] = n - pos + 1$. If $c = \$$, then $pos = n$ and $\$$ also appears at position n in S^{rev} . Again, $SA^{rev}[k] = n - pos + 1$.

- If $lb \neq rb$ and $S[SA[lb] + \ell + 1] = S[SA[rb] + \ell + 1]$, then all the characters in $BWT^{rev}[k..k + rb - lb]$ are the same and the algorithm does not determine the lexicographic order of the suffixes. In this case, the values in $SA^{rev}[k..k + rb - lb]$ remain unknown.

It follows as a consequence that Algorithm 7.35 fills the suffix array SA^{rev} only partially; in Figure 7.30 (page 360) the computed entries are underlined. Nevertheless, partial information is better than no information.

Completing a partially filled suffix array

Next, we explain how a partially filled suffix array SA of a string⁷ can be completed. It should be clear from Section 7.2.2 that the LF -mapping can not only be used to recover the original string from the BWT, but also its suffix array; see Exercise 7.2.6. In essence, this is a consequence of the equation

$$SA[i] = SA[LF[i]] + 1$$

(The equation was proven in Lemma 7.2.8.) The LF -mapping, in turn, can be obtained by Algorithm 7.35 provided that the *count* array is initialized by $count[c] \leftarrow C[c]$ for each $c \in \Sigma$.

Algorithm 7.36 shows two alternative ways of completing a partially filled suffix array SA. In a left-to-right scan of the SA array, the pseudo-code on the left-hand side initiates a new computation whenever it detects a defined SA-entry (an entry that has already been computed); it follows

⁷In our context, the string under consideration is S^{rev} .

LF -pointers and fills in SA-entries that have not been computed yet until another defined SA-entry is reached. The pseudo-code on the right-hand side also scans the SA array from left to right, but this time it ignores defined entries. Instead, whenever it finds an undefined entry $SA[i]$, it follows LF -pointers until an index k is reached with $SA[k] \neq \perp$, and it stores the sequence $i, LF(i), \dots, LF^q(i)$ on a stack, where $k = LF^{q+1}(i)$. Clearly, if $SA[k] = j$, then the SA-value at index $LF^q(i)$ —the topmost element of the stack—is $SA[LF^q(i)] = j + 1$. After $LF^q(i)$ has been popped from the stack, the subsequent values $SA[LF^{q-1}(i)], \dots, SA[LF(i)], SA[i]$ are similarly obtained.

7.8.3 The lcp-array of the reverse string

In fact, Algorithm 7.35 can also be used to compute LCP^{rev} . This can be seen as follows. Suppose Algorithm 7.35 is applied to the ω -interval $[i..j]$ for some ℓ -length substring ω of S , and let k be the left boundary of the ω^{rev} -interval in SA^{rev} . It is a consequence of Lemma 7.8.1 that the procedure call $bwtrev(k, [i..j], \ell)$ correctly computes the boundaries $[s_q..e_q]$ of the $\omega^{rev}c_q$ -intervals in SA^{rev} , where c_1, \dots, c_m are the characters for which $c_q\omega$ is a substring of S ($1 \leq q \leq m$). By the conditional statement “**if** *list* not empty **then** $LCP^{rev}[k] \leftarrow \ell$ ”, Algorithm 7.35 assigns the value ℓ at each index s_2, \dots, s_m (but not at index s_1). This is correct, i.e., $LCP^{rev}[s_q] = \ell$ for $2 \leq q \leq m$, because $\omega^{rev}c_{q-1}$ is a prefix of the suffix at index e_{q-1} and $\omega^{rev}c_q$ is a prefix of the suffix at index $s_q = e_{q-1} + 1$.

Thus, whenever Algorithm 7.35 fills an entry in the lcp-array LCP^{rev} , it assigns the correct value. However, the algorithm does not fill LCP^{rev} completely; in Figure 7.30 (page 360), the computed entries are underlined. This is because whenever Algorithm 7.35 detects a $c\omega$ -interval $[lb..rb]$ in the list returned by $getIntervals([i..j])$ with $lb \neq rb$ and $S[SA[lb] + \ell + 1] = S[SA[rb] + \ell + 1]$, then it does *not* determine the lexicographic order of the suffixes in the $\omega^{rev}c$ -interval $[s..e]$. Instead, it fills $BWT^{rev}[s..e]$ with a 's because each occurrence of $c\omega$ in S is followed by the same character $a = S[SA[lb] + \ell + 1]$. Consequently, if an entry $LCP^{rev}[p]$ is not filled by Algorithm 7.35, then $BWT^{rev}[p - 1] = BWT^{rev}[p]$.

Completing a partially filled LCP-array

Now we explain how a partially filled LCP-array of a string⁸ can be completed. By “partially filled” we mean that the array contains all entries $LCP[i]$ with $BWT[i] \neq BWT[i - 1]$.

Definition 7.8.7 Suppose $2 \leq i \leq n$. The value $LCP[i]$ is called *reducible* if $BWT[i] = BWT[i - 1]$; otherwise it is *irreducible*.

⁸In our context, the string under consideration is S^{rev} .

idx	LCP	BWT	$S_{SA[idx]}$
$i - 1$		c	$S_{SA[i-1]}$
i	ℓ	c	$S_{SA[i]}$
$LF(i - 1)$			$cS_{SA[i-1]}$
$LF(i)$	$\ell + 1$		$cS_{SA[i]}$

Figure 7.34: If $BWT[i] = c = BWT[i - 1]$, then $LF(i) = LF(i - 1) + 1$. It follows as a consequence that $LCP[LF(i)] = LCP[i] + 1 = \ell + 1$.

So given an array that contains all irreducible LCP-values (and possibly some reducible values), we wish to compute the whole LCP-array. The solution is based on the following lemma.

Lemma 7.8.8 *A reducible value $LCP[i]$ can be computed by the equation*

$$LCP[i] = LCP[LF(i)] - 1$$

Proof By definition, $LCP[i] = |\text{lcp}(S_{SA[i-1]}, S_{SA[i]})|$. Moreover, $BWT[i - 1] = BWT[i]$ because $LCP[i]$ is reducible. This implies that the suffixes $cS_{SA[i-1]}$ and $cS_{SA[i]}$, where $c = BWT[i]$, must occur consecutively in the suffix array, namely at the indices $LF(i - 1)$ and $LF(i)$; see Figure 7.34. Hence $LF(i) = LF(i - 1) + 1$. Consequently,

$$\begin{aligned}
 LCP[LF(i)] &= |\text{lcp}(S_{SA[LF(i)-1]}, S_{SA[LF(i)]})| \\
 &= |\text{lcp}(S_{SA[LF(i-1)]}, S_{SA[LF(i)]})| \\
 &= 1 + |\text{lcp}(S_{SA[i-1]}, S_{SA[i]})| \\
 &= 1 + LCP[i]
 \end{aligned}$$

This proves the lemma. \square

Algorithm 7.37 shows pseudo-code for the completion of a partially filled LCP-array. Because it is very similar to the pseudo-code on the right-hand side of Algorithm 7.36, we need not explain it in detail.

Exercise 7.8.9 Algorithm 7.36 contains two alternative ways of completing a partially filled suffix array SA. Is it possible to give an alternative to Algorithm 7.37 in a similar fashion? If not, provide a counterexample.

Exercise 7.8.10 Is it possible to complete the partial suffix array and the partial lcp-array (delivered by Algorithm 7.35) simultaneously, starting only from undefined lcp-values (as in Algorithm 7.37)?

Algorithm 7.37 Given a partial LCP-array that contains all irreducible LCP-values, this procedure computes the whole LCP-array.

```

initialize an empty stack
for  $i \leftarrow 1$  to  $n$  do
     $k \leftarrow i$ 
    while  $\text{LCP}[k] = \perp$  do
         $\text{push}(k)$ 
         $k \leftarrow \text{LF}[k]$ 
     $\ell \leftarrow \text{LCP}[k]$ 
    while stack is not empty do
         $\ell \leftarrow \ell - 1$ 
         $\text{LCP}[\text{pop}()] \leftarrow \ell$ 

```

LCP^{rev} is a permutation of LCP

In the remainder of this section, we prove the following strong relationship between LCP and LCP^{rev} .

Lemma 7.8.11 *The longest common prefix array LCP^{rev} of S^{rev} is a permutation of the longest common prefix array LCP of S .*

Proof We show that each lcp-value occurs as often in LCP as in LCP^{rev} . Since $\text{LCP}[n+1] = -1 = \text{LCP}^{rev}[n+1]$, the boundary value -1 at index $n+1$ can be neglected. Let $\ell \in \{1, \dots, n\}$ and define the set M_ℓ by $M_\ell = \{\omega \mid \omega \text{ is an } \ell\text{-length substring but not a suffix of } S\}$. We count how many entries in the array LCP are smaller than ℓ . There are ℓ proper suffixes of S having a length $\leq \ell$. For each such suffix $S_{\text{SA}[k]}$ we have $\text{LCP}[k] < \ell$. Any other suffix has a length greater than ℓ and hence its ℓ -length prefix belongs to M_ℓ . Let $\omega \in M_\ell$ and let $[i..j]$ be the ω -interval. Clearly, for all k with $i < k \leq j$, we have $\text{LCP}[k] \geq \ell$ because the suffixes $S_{\text{SA}[k-1]}$ and $S_{\text{SA}[k]}$ share the prefix ω . By contrast, $\text{LCP}[i] < \ell$ because ω is not a prefix of $S_{\text{SA}[i-1]}$. Thus, there are $|M_\ell|$ many entries in the array LCP satisfying $\text{LCP}[k] < \ell$ and $|S_{\text{SA}[k]}| > \ell$. In total, the array LCP has $|M_\ell| + \ell$ many entries that are smaller than ℓ . Analogously, there are $|M_{\ell+1}| + \ell + 1$ many entries in the array LCP that are smaller than $\ell + 1$, where $\ell \in \{1, \dots, n-1\}$. Consequently, the lcp-value ℓ occurs $(|M_{\ell+1}| + \ell + 1) - (|M_\ell| + \ell) = |M_{\ell+1}| - |M_\ell| + 1$ times in the LCP-array. By the same argument, the lcp-value ℓ occurs $|M_{\ell+1}^{rev}| - |M_\ell^{rev}| + 1$ many times in the array LCP^{rev} , where $M_\ell^{rev} = \{\omega \mid \omega \text{ is an } \ell\text{-length substring but not a suffix of } S^{rev}\}$. Now the lemma follows from the equality $|M_\ell| = |M_\ell^{rev}|$, which is true because $\omega \in M_\ell$ if and only if $\omega^{rev} \in M_\ell^{rev}$. \square

Figure 7.35 illustrates the proof of Lemma 7.8.11. The proper suffixes of S with length ≤ 2 occur at the indices 1 and 4 in the (conceptual) suffix array, so $\text{LCP}[1]$ and $\text{LCP}[4]$ are smaller than 2. Furthermore,

i	LCP	$S_{SA[i]}^r$
1	-1	\$
2	0	atc\$
3	2	atgcatc\$
4	0	c\$
5	1	catc\$
6	3	catgcatc\$
7	0	gcatc\$
8	4	gcatgcatc\$
9	0	tc\$
10	1	tgcac\$

i	LCP^{rev}	$S_{SA[i]}^{rev}$
1	-1	\$
2	0	acg\$
3	3	acgtacg\$
4	0	cg\$
5	2	cgtacg\$
6	1	ctacgtacg\$
7	0	g\$
8	1	gtacg\$
9	0	tacg\$
10	4	tacgtacg\$

Figure 7.35: The lcp-arrays of $S = gcatgcatc\$$ and $S^{rev} = ctacgtacg\$$.

we have $M_2 = \{at, ca, gc, tc, tg\}$ and the corresponding entries in the LCP-array at the indices 2, 5, 7, 9, and 10 are also smaller than 2. So there are $|M_2| + 2 = 7$ entries of the LCP-array that are smaller than 2. Since $M_3 = \{atc, atg, cat, gca, tgc\}$, there are $|M_3| + 3 = 8$ entries of the LCP-array that are smaller than 3. We conclude that the value 2 occurs $8 - 7 = 1$ times in the LCP-array. By the same argument, it occurs only once in LCP^{rev} . Note that $M_2^{rev} = \{ac, cg, ct, gt, ta\}$ and $M_3^{rev} = \{acg, cgt, cta, gta, tac\}$.

7.8.4 The bidirectional search algorithm

This section presents a data structure that supports bidirectional search.

Definition 7.8.12 The *bidirectional wavelet index* of a string S consists of

- the *backward index*, supporting backward search based on the wavelet tree of the Burrows-Wheeler transformed string BWT of S , and
- the *forward index*, supporting backward search on the reverse string S^{rev} of S (hence forward search on S) based on the wavelet tree of the Burrows-Wheeler transformed string BWT^{rev} of S^{rev} .

The difficult part is to synchronize the search on both indexes. To see this, suppose we know the ω -interval $[i..j]$ in the backward index as well as the ω^{rev} -interval $[i^{rev}..j^{rev}]$ in the forward index, where ω is some substring of S . Given $[i..j]$ and a character c , *backwardSearch*($c, [i..j]$) returns the $c\omega$ -interval in the backward index (cf. Algorithm 7.7 on page 302), but it is unclear how the corresponding interval, the interval of the string $(c\omega)^{rev} = \omega^{rev}c$, can be found in the forward index. Conversely, given $[i^{rev}..j^{rev}]$ and

i	$S_{SA}[i]$	i	$S_{SA}^{rev}[i]$
1	n	1	e
2	l	2	l
3	e	3	a
4	-	4	n
5	p	5	n
6	l	6	l
7	\$	7	p
8	n	8	-
9	n	9	n
10	l	10	l
11	l	11	l
12	e	12	e
13	e	13	-
14	e	14	e
15	-	15	e
16	e	16	e
17	a	17	e
18	a	18	\$
19	e	19	a

Figure 7.36: Bidirectional wavelet index of $S = \text{el_anele_lepanelen}\$,$ consisting of the backward index (left) and the forward index (right).

a character c , backward search returns the $c\omega^{rev}$ -interval in the forward index, but it is unclear how the corresponding ωc -interval can be found in the backward index. Because both cases are symmetric, we will only deal with the first case. So given the ω^{rev} -interval, we have to find the $\omega^{rev}c$ -interval in the forward index.

As an example, consider the bidirectional wavelet index of the string $S = \text{el_anele_lepanelen}\$$ in Figure 7.36, and the substring $\omega = \text{e} = \omega^{rev}$. The e -interval in both indexes is $[6..11]$. The le -interval in the backward index is determined by $\text{backwardSearch}(1, [6..11]) = [13..15]$ and the task is to identify the el -interval in the forward index.

All we know is that the suffixes of S^{rev} are lexicographically ordered in the forward index. In other words, the $\omega^{rev}c$ -interval $[lb^{rev}..rb^{rev}]$ is a subinterval of $[i^{rev}..j^{rev}]$ so that (note that $|\omega^{rev}| = |\omega|$)

- $S^{rev}[\text{SA}^{rev}[k] + |\omega|] < c$ for all k with $i^{rev} \leq k < lb^{rev}$,
- $S^{rev}[\text{SA}^{rev}[k] + |\omega|] = c$ for all k with $lb^{rev} \leq k \leq rb^{rev}$,
- $S^{rev}[\text{SA}^{rev}[k] + |\omega|] > c$ for all k with $rb^{rev} < k \leq j^{rev}$.

In the example of Figure 7.36,

- $S^{rev}[\text{SA}^{rev}[k] + 1] = \$ < \text{l}$ for $k = 6$,

Algorithm 7.38 Given a BWT-interval $[i..j]$ and $c \in \Sigma$, $getBounds([i..j], c)$ returns the triple $(i', j', smaller)$, where $[C[c] + i'..C[c] + j']$ is the new BWT-interval after a backward search step for c , and $smaller$ is the number of occurrences of characters in $BWT[i..j]$ that are strictly smaller than c .

```

getBounds([i..j], c)
    return getBounds'([i..j], c, [1..σ], 0)

getBounds'([i..j], c, [l..r], smaller)
    if  $l = r$  then return  $(i, j, smaller)$ 
    else
         $(a_0, b_0) \leftarrow (rank_0(B^{[l..r]}, i - 1), rank_0(B^{[l..r]}, j))$ 
         $(a_1, b_1) \leftarrow (i - 1 - a_0, j - b_0)$ 
         $m = \lfloor \frac{l+r}{2} \rfloor$ 
        if  $c \leq \Sigma[m]$  then
            return  $getBounds'([a_0 + 1..b_0], c, [l..m], smaller)$ 
        else
            return  $getBounds'([a_1 + 1..b_1], c, [m + 1..r], smaller + b_0 - a_0)$ 

```

- $S^{rev}[SA^{rev}[k] + 1] = 1$ for all k with $7 \leq k \leq 9$,
- $S^{rev}[SA^{rev}[k] + 1] = n > 1$ for all k with $9 < k \leq 11$.

Unfortunately, we do not know these characters, but if we would know the number $smaller$ of all occurrences of characters at these positions that precede c in the alphabet, together with the size of the new $c\omega$ -interval $[lb..rb]$ in the backward index, then we could identify the unknown $\omega^{rev}c$ -interval $[lb^{rev}..rb^{rev}]$ by $lb^{rev} = i^{rev} + smaller$ and $rb^{rev} = lb^{rev} + (rb - lb)$. In our example, the knowledge of $smaller = 1$ and $[lb..rb] = [13..15]$ would yield the $e1$ -interval $[6 + 1..(6 + 1) + (15 - 13)] = [7..9]$. The *key observation* is that the multiset of characters $\{S^{rev}[SA^{rev}[k] + |\omega|] : i^{rev} \leq k \leq j^{rev}\}$ coincides with the multiset $\{BWT[k] : i \leq k \leq j\}$. In the example of Figure 7.36, $\{S^{rev}[SA^{rev}[k] + 1] : 6 \leq k \leq 11\} = \{\$, 1, 1, 1, n, n\} = \{BWT[k] : 6 \leq k \leq 11\}$. In other words, it suffices to determine the number $smaller$ of all occurrences of characters in the string $BWT[i..j]$ that precede character c in the alphabet Σ , and the new interval $[lb..rb]$ in the backward index.

Given an ω -interval $[i..j]$ and a character c , Algorithm 7.38 traverses the wavelet tree of BWT in a top-down fashion and computes the three values $i' = rank_c(i - 1) + 1$, $j' = rank_c(j)$, and $smaller$. Note that the $c\omega$ -interval in the backward index can directly be determined by $[lb..rb] = [C[c] + i'..C[c] + j']$. As discussed above, the $\omega^{rev}c$ -interval $[lb^{rev}..rb^{rev}]$ in the forward index can then be computed by $lb^{rev} = i^{rev} + smaller$ and $rb^{rev} = lb^{rev} + (rb - lb)$.

We compute the values of $i' = rank_1(5) + 1$, $j' = rank_1(11)$ and $smaller$ for the interval $[6..11]$ and the character 1 by invoking $getBounds([6..11], 1)$. This

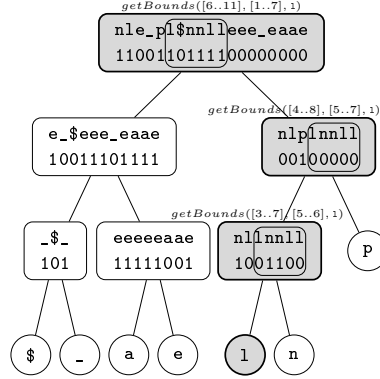


Figure 7.37: $\text{getBounds}([6..11], 1)$ returns the triple $(2, 4, 1)$.

example is illustrated in Figure 7.37. Because 1 belongs to the second half $\Sigma[5..7]$ of the ordered alphabet Σ , the occurrences of 1 correspond to ones in the bit vector at the root of the wavelet tree, and they go to the right child, say node v_1 , of the root. In order to compute the number of occurrences of characters in the interval $[6..11]$ that belong to $\Sigma[1..4]$ and hence are smaller than 1, we compute

$$(a_0, b_0) = (\text{rank}_0(B^{[1..7]}, 6 - 1), \text{rank}_0(B^{[1..7]}, 11)) = (2, 3)$$

and the number we are searching for is $b_0 - a_0 = 3 - 2 = 1$, so we add it to a variable *smaller*, which was initialized to 0 at the beginning. Then we descend to the right child v_1 and have to compute the boundaries of the search interval in the bit vector $B^{[5..7]}$ that corresponds to the search interval $[6..11]$ in the bit vector $B^{[1..7]}$. These boundaries are $a_1 + 1$ and b_1 , where

$$(a_1, b_1) = (\text{rank}_1(B^{[1..7]}, 6 - 1), \text{rank}_1(B^{[1..7]}, 11)) = (3, 8)$$

Proceeding recursively, we find that 1 belongs to the third quarter $\Sigma[5..6]$ of Σ , so the occurrences of 1 correspond to zeros in the bit vector at v_1 , and they go to the left child, say node v_2 , of v_1 . Again, we compute

$$\begin{aligned} (a'_0, b'_0) &= (\text{rank}_0(B^{[5..7]}, 4 - 1), \text{rank}_0(B^{[5..7]}, 8)) = (2, 7) \\ (a'_1, b'_1) &= (\text{rank}_1(B^{[5..7]}, 4 - 1), \text{rank}_1(B^{[5..7]}, 8)) = (1, 1) \end{aligned}$$

The number of occurrences of characters in the string $\text{BWT}^{[5..7]}[4..8]$ that belong to $\Sigma[7] = \text{p}$ is $b'_1 - a'_1 = 1 - 1 = 0$ and the new search interval in the bit vector $B^{[5..6]}$ is $[a'_0 + 1..b'_0] = [3..7]$. In the third step, we compute

$$\begin{aligned} (a''_0, b''_0) &= (\text{rank}_0(B^{[5..6]}, 3 - 1), \text{rank}_0(B^{[5..6]}, 7)) = (1, 4) \\ (a''_1, b''_1) &= (\text{rank}_1(B^{[5..6]}, 3 - 1), \text{rank}_1(B^{[5..6]}, 7)) = (1, 3) \end{aligned}$$

Algorithm 7.39 Given the ω -interval $[i..j]$ in the backward index and the ω^{rev} -interval $[i^{rev}..j^{rev}]$ in the forward index, the procedure call *backwardSearch*($c, [i..j], [i^{rev}..j^{rev}]$) returns the pair $([lb..rb], [lb^{rev}..rb^{rev}])$, where $[lb..rb]$ is the $c\omega$ -interval in the backward index and $[lb^{rev}..rb^{rev}]$ is the $\omega^{rev}c$ -interval in the forward index (if $c\omega$ does not occur in S , it returns \perp).

```

backwardSearch( $c, [i..j], [i^{rev}..j^{rev}]$ )
  ( $i', j', smaller$ )  $\leftarrow$  getBounds( $[i..j], c$ )
  if  $i' \leq j'$  then
     $lb \leftarrow C[c] + i'$ 
     $rb \leftarrow C[c] + j'$ 
     $lb^{rev} \leftarrow i^{rev} + smaller$ 
     $rb^{rev} \leftarrow lb^{rev} + (rb - lb)$ 
    return  $([lb..rb], [lb^{rev}..rb^{rev}])$ 
  else
    return  $\perp$ 

```

and find that there are $b_1'' - a_1'' = 2$ occurrences of the character n and $b_0'' - a_0'' = 3$ occurrences of the character l . In the last step, the search interval in the bit vector $B^{[5..5]}$ is $[a_0'' + 1..b_0''] = [2..4]$. We have reached a leaf of the wavelet tree, and $BWT^{[5..5]} = 1111$. So obviously, $a_0'' + 1 = 2$ is exactly the value of $rank_1(5) + 1$, and $b_0'' = 4$ is the value of $rank_1(11)$. The variable $smaller = 1$ contains the sum of all occurrences of characters in $BWT[6..11]$ that are smaller than l . All in all, Algorithm 7.38 applied to the interval $[6..11]$ and the character l returns the triple $(2, 4, 1)$. Algorithm 7.39 uses this triple to compute the le -interval in the backward index by $[lb..rb] = [C[l] + 2..C[l] + 4] = [11 + 2..11 + 4] = [13..15]$ and the $e1$ -interval $[lb^{rev}..rb^{rev}]$ in the forward index by $lb^{rev} = i^{rev} + smaller = 6 + 1 = 7$ and $rb^{rev} = lb^{rev} + (rb - lb) = 7 + (9 - 7) = 9$.

The bidirectional search algorithm presented in this section appeared in [284]. Independently and contemporaneously, [197] presented a similar data structure, which they call *bi-directional BWT*. Their main motivation was short read alignment, so they use bidirectional search to find approximate matches of relatively short DNA sequences within a whole genome; see Section 7.9.2. It turns out that the two approaches use the same basic idea, but a closer look reveals that none is superior to the other. On the one hand, one search step with the bidirectional wavelet index takes $O(\log \sigma)$ time while it takes $O(\sigma)$ time with the bi-directional BWT. On the other hand, the bi-directional BWT uses less space than the bidirectional wavelet index.

We should not sweep under the rug the fact that bidirectional search has a longer history. To the best of our knowledge, research on data structures supporting bidirectional search in a string started in 1995 with

Stoye's diploma thesis on affix trees (the English translation appeared in [300]), and Maaß [208] showed that affix trees can be constructed on-line in linear time. Basically, the affix tree of a string S comprises both the suffix tree of S (supporting forward search) and the suffix tree of the reverse string S^{rev} (supporting backward search). Strothmann [302] showed that affix arrays have the same functionality as affix trees, but they require less than half the space. An affix array combines the suffix arrays of S and S^{rev} , but it is a complex data structure because the interplay between the two suffix arrays is rather difficult to implement. A reimplementation of affix arrays is described in [221].

7.9 Approximate string matching

Approximate string matching is the technique of finding substrings of a long string S (or a collection of strings) that match a pattern P approximately (rather than exactly). Approximate search algorithms are abundant and there is a vast literature on the topic. We shall not discuss this field in detail, but instead refer to the overview article [236]. Here, we consider solely the case in which S is fixed and many on-line queries of the form “Where are all approximate matches of P in S ?” must be answered efficiently. A prime example in bioinformatics is short read mapping. High-throughput sequencing (or next-generation sequencing) technologies produce billions of bases in a single run. In their short read mapping primer [312], Trapnell and Salzberg write:

One of the challenges presented by the new sequencing technology is the so-called ‘read mapping’ problem. Sequencing machines made by Illumina of San Diego, Applied Biosystems (ABI) of Carlsbad, California, and Helicos of Cambridge, Massachusetts, produce short sequences of 25–100 base pairs (bp), called ‘reads’, which are sequence fragments read from a longer DNA molecule present in the sample that is fed into the machine. In contrast to whole-genome assembly, in which these reads are assembled together to reconstruct a previously unknown genome, many of the next-generation sequencing projects begin with a known, or so-called ‘reference’, genome. In this case, to make sense of the reads, their positions within the reference sequence must be determined. This process is known as aligning or ‘mapping’ the read to the reference.

Short-read mappers are, among others, Bowtie [198], BWA [202], SOAP2 [203], and 2BWT [197]; see e.g. [113, 312] for overview articles. In the following, we discuss the basic algorithms used in BWA, Bowtie, and 2BWT.

The short read mapping problem is exacerbated by sequencing errors and variations between the sequenced chromosomes and the reference genome.⁹ First, we consider the scenario in which only mismatches are allowed (Hamming distance) and subsequently address the problem of also allowing insertions and deletions (edit distance).

7.9.1 Using backward search

Definition 7.9.1 The *Hamming distance* between two strings S^1 and S^2 of equal length is the number of positions at which the corresponding characters are different:

$$hdist(S^1, S^2) = |\{i \mid S^1[i] \neq S^2[i]\}|$$

To put it another way, the Hamming distance measures the minimum number of substitutions required to change S^1 into S^2 (or vice versa).

Definition 7.9.2 Let P and S be strings with $m = |P| < |S| = n$, and let k be a natural number with $k < m$. An m -length substring $S[i..i+m-1]$ is called a k -mismatch of P in S if $hdist(P, S[i..i+m-1]) \leq k$. The k -mismatch problem is to find all positions in S at which a k -mismatch of P in S starts.

When k and Σ are small, one can solve the k -mismatch problem by the following approach. First, generate the so-called Hamming sphere \mathcal{P} of radius k at center P (defined below). Second, use the Aho-Corasick algorithm from Section 2.5 to find all positions in S at which a pattern from \mathcal{P} starts. The Hamming sphere of radius k at center P is the set

$$\mathcal{P} = \{P' \mid hdist(P, P') \leq k\}$$

The number of strings in the Hamming sphere \mathcal{P} is

$$\sum_{i=0}^k \binom{m}{i} (|\Sigma| - 1)^i \in O(m^k |\Sigma|^k)$$

As an example, consider the pattern $P = tact$ on the alphabet $\Sigma = \{a, c, g, t\}$. The Hamming sphere of radius $k = 1$ at center P is the set

$$\{tact, aact, cact, gact, tcct, tgct, ttct, taat, tagt, tatt, taca, tacc, tacg\}$$

Li and Durbin [202] suggested a different solution to the problem. Their algorithm uses an FM-index to simultaneously find different occurrences of subpatterns, and it prunes the search space using a lower bound on the distance.

⁹Li and Durbin [202] suggest the following number k of differences (mismatches or gaps) that should be tolerated: for 15-37 bp reads, k equals 2; for 38-63 bp, $k = 3$; for 64-92 bp, $k = 4$; for 93-123 bp, $k = 5$; and for 124-156 bp reads, $k = 6$.

Algorithm 7.40 The procedure *k-mismatch*.

```

procedure k-mismatch( $P, j, d, [lb..rb]$ )
  if  $d < 0$  then
    return  $\emptyset$ 
  if  $j = 0$  then      /*  $k$ -mismatches detected */
    return  $\{[lb..rb]\}$ 
   $\mathcal{I} \leftarrow \emptyset$ 
   $list \leftarrow getIntervals([lb..rb])$ 
  for each  $(c, [lb..rb])$  in  $list$  do
    if  $P[j] = c$  then
       $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatch}(P, j - 1, d, [lb..rb])$ 
    else      /* substitution of  $P[j]$  with  $c$  */
       $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatch}(P, j - 1, d - 1, [lb..rb])$ 
  return  $\mathcal{I}$ 

```

Algorithm 7.40 implements the approach, but it does not prune the search space. The procedure call $k\text{-mismatch}(P, m, k, [1..n])$ returns all k -mismatches of P in S . Let us illustrate the algorithm for $k = 1$. For each position j in P , it uses backward search to find the $P[j + 1..m]$ -interval, generates all $bP[j + 1..m]$ -intervals (where b can be any character except $\$$), and for each such interval it continues the backward search to find the $P[1..j - 1]bP[j + 1..m]$ -interval. As an example, consider the pattern $P = tact$ and the full-text index of the string $S = ctaataatg\$$ shown on the left-hand side of Figure 7.38. For the position $j = 4$, the algorithm generates the b -interval of every character $b \in \{a, c, g, t\}$, and with each interval it continues the backward search to find the $tacb$ -interval. In other words, it searches for $taca$, $tacc$, $tacg$, and $tact$. Only the recursive search for $tact$ still allows for one mismatch; in the other cases the algorithm searches for the exact phrases $taca$, $tacc$, and $tacg$ (in all three cases, the backward search stops after one step because neither ca nor cc nor cg are substrings of S). In case $j = 3$, the algorithm generates the at -interval $[4..5]$ and the ct -interval $[6..6]$. The latter results in the recursive call $k\text{-mismatch}(tact, 2, 1, [6..6])$; since $getIntervals([6..6])$ returns an empty list, the search stops here. The former results in the recursive call $k\text{-mismatch}(tact, 2, 0, [4..5])$, which leads to the output $\mathcal{I} = \{[8..9]\}$. This means that there are two 1-mismatches of P in S , namely starting at the positions $SA[8] = 2$ and $SA[9] = 5$.

Algorithm 7.40 can be extended in such a way that it can deal with insertions and deletions. To be precise, the modified algorithm solves the k -differences problem, which we formally define below.

i	SA	BWT	$S_{SA[i]}$
1	10	g	$\$$
2	3	t	$aataatg\$$
3	6	t	$aatg\$$
4	4	a	$ataatg\$$
5	7	a	$atg\$$
6	1	$\$$	$ctaataatg\$$
7	9	t	$g\$$
8	2	c	$taataatg\$$
9	5	a	$taatg\$$
10	8	a	$tg\$$

i	BWT^{rev}	$S_{SA^{rev}[i]}^{rev}$
1	c	$\$$
2	t	$aataatc\$$
3	t	$aatc\$$
4	a	$ataatc\$$
5	a	$atc\$$
6	t	$c\$$
7	$\$$	$gtaataatc\$$
8	g	$taataatc\$$
9	a	$taatc\$$
10	a	$tc\$$

Figure 7.38: Left-hand side: suffix array SA and BWT of the string $S = ctaataatg\$$ (the backward index). Right-hand side: Burrows-Wheeler transform BWT^{rev} of $S^{rev} = gtaataatc\$$ (the forward index).

Definition 7.9.3 Given $S^1, S^2 \in \Sigma^*$, we write $S^1 \rightarrow S^2$ if

- S^2 can be obtained from S^1 by replacing one occurrence of $x \in \Sigma$ by $y \in \Sigma$, i.e., $S^1 = uxv$ and $S^2 = uyv$ (substitution),
- S^2 can be obtained from S^1 by inserting one occurrence of $y \in \Sigma$, i.e., $S^1 = uv$ and $S^2 = uylv$ (insertion),
- S^2 can be obtained from S^1 by deleting one occurrence of $x \in \Sigma$, i.e., $S^1 = uxv$ and $S^2 = uv$ (deletion).

In what follows, the term *indel* is used to mean an insertion or a deletion; substitutions and indels are collectively referred to as *edit operations*.

Furthermore, we write $S^1 \rightarrow^k S^2$ if S^1 can be transformed into S^2 by a sequence of $k \in \mathbb{N}$ edit operations.

Definition 7.9.4 The *edit distance* (or *Levenshtein distance*) between two strings S^1 and S^2 is the minimum number of edit operations needed to transform S^1 into S^2 . Formally,

$$edist(S^1, S^2) = \min\{k \mid S^1 \rightarrow^k S^2\}$$

Definition 7.9.5 Let P and S be strings with $m = |P| < |S| = n$, and let k be a natural number with $k < m$. A substring $S[i..j]$ is called an

Algorithm 7.41 The procedure call $k\text{-differences}(P, m, k, [1..n])$ finds all approximate occurrences of P in S , using the array M_{lr} .

```

procedure  $k\text{-differences}(P, j, d, [lb..rb])$ 
  if  $d < M_{lr}[j]$  then /*  $M_{lr}[j]$  is a lower bound on the remaining differences */
    return  $\emptyset$ 
  if  $j = 0$  then /* approximate occurrences of  $P$  in  $S$  detected */
    return  $\{[lb..rb]\}$ 
   $\mathcal{I} \leftarrow \emptyset$ 
   $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-differences}(P, j - 1, d - 1, [lb..rb])$  /* deletion of  $P[j]$  */
   $list \leftarrow getIntervals([lb..rb])$ 
  for each  $(c, [lb..rb])$  in  $list$  do
     $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-differences}(P, j, d - 1, [lb..rb])$  /* insertion of  $c$  */
    if  $P[j] = c$  then
       $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-differences}(P, j - 1, d, [lb..rb])$ 
    else /* substitution of  $P[j]$  with  $c$  */
       $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-differences}(P, j - 1, d - 1, [lb..rb])$ 
  return  $\mathcal{I}$ 

```

approximate occurrence of P in S if $edist(P, S[i..j]) \leq k$. The $k\text{-differences}$ problem is to find all positions in S at which an approximate occurrence of P in S starts.

Algorithm 7.41 solves the $k\text{-differences}$ problem. In the first if-then statement, it uses a lower bound on the edit distance that can be used to prune the search space. To derive the lower bound, we use the following definition, which is motivated by Ehrenfeucht and Haussler's [84] notion of *compatible markings*. As a side remark: Ukkonen [314] as well as Chang and Lawler [54] used this technique in fast approximate string matching algorithms.

Definition 7.9.6 For a string S of length n and a pattern P of length m , there is a unique *left-to-right partition* $P = w_1c_1w_2c_2 \dots w_kc_kw_{k+1}$ of P w.r.t. S so that each w_i is a substring of S but w_ic_i is not. The characters c_1, \dots, c_k are the *marked characters* and the *left-to-right marking*

$$M_{lr}(P, S) = \{p_i \mid p_i = \sum_{j=1}^i |\omega_j c_j|\}$$

is the set of positions at which the marked characters appear in P .

As an example, consider the pattern $P = ttaatt$ and the string $S = ctaataatg\$$; see Figure 7.38 (page 377). The left-to-right partition consists of $w_1 = t$, $c_1 = t$, $w_2 = aat$, $c_2 = t$, and $w_3 = \varepsilon$. Therefore, $M_{lr}(P, S) = \{2, 6\}$.

Lemma 7.9.7 *If $|M_{lr}(P, S)| = k$, then no substring of S matches P with less than k differences.*

Proof Let $P = w_1c_1w_2c_2 \dots w_kc_kw_{k+1}$ be the left-to-right partition of P w.r.t. S . We show by finite induction on d , $1 \leq d \leq k$, that no substring of S matches the prefix $w_1c_1 \dots w_dc_d$ of P with less than d differences. In the base case $d = 1$, we know that the longest substring of S that matches a prefix of P is w_1 . Thus, no substring of S exactly matches the prefix w_1c_1 of P . In the inductive step, consider d with $2 \leq d \leq k$. According to the inductive hypothesis, no substring of S matches the prefix $w_1c_1 \dots w_{d-1}c_{d-1}$ of P with less than $d - 1$ differences. The longest substring of S that matches a prefix of $w_dc_d \dots w_kc_kw_{k+1}$ has length $|w_d|$. Therefore, a substring of S may match $w_1c_1 \dots w_{d-1}c_{d-1}w_d$ with $d - 1$ differences, but no substring of S can match $w_1c_1 \dots w_{d-1}c_{d-1}w_dc_d$ with $d - 1$ differences (an exact match of a substring of S with a suffix of P that starts at or before position p_{d-1} must end before position p_d in P). This proves the lemma. \square

Definition 7.9.8 Given S and P , let the array M_{lr} of size m be defined by

$$M_{lr}[j] = |\{p_i \leq j \mid p_i \in M_{lr}(P, S)\}|$$

for all j with $1 \leq j \leq m$.

Continuing our example from above, we have $M_{lr} = [0, 1, 1, 1, 1, 2]$.

Corollary 7.9.9 *If $M_{lr}[j] = d$, then no substring of S matches $P[1..j]$ with less than d differences.*

Proof Since $M_{lr}[j] = d$, the left-to-right partition $P = w_1c_1 \dots w_kc_kw_{k+1}$ of P w.r.t. S restricted to the first j characters is $P[1..j] = w_1c_1 \dots w_dc_dw$, where w is a prefix of w_{d+1} . It is readily verified that the left-to-right partition of $P[1..j]$ w.r.t. S coincides with $w_1c_1 \dots w_dc_dw$. Thus, the corollary immediately follows from Lemma 7.9.7. \square

According to the preceding corollary, the backward search in Algorithm 7.41 can be stopped when $M_{lr}[j]$ is larger than the number of tolerated mismatches. This criterion effectively prunes the search space without sacrificing the correctness of the algorithm. Of course, we still have to find a way to compute M_{lr} efficiently. Li and Durbin [202] used the forward index for this purpose; see Exercise 7.9.10. An alternative is to use matching statistics, which can be computed space efficiently with the balanced parentheses sequence of the LCP-array; see Exercise 7.9.11.

Exercise 7.9.10 Give pseudo-code of an algorithm that computes the M_{lr} -array based on BWT^{rev} , the Burrows-Wheeler transform of S^{rev} . Analyze the run-time of the algorithm.

Exercise 7.9.11 Prove that Algorithm 7.42 correctly calculates the array M_{lr} and analyze its worst-case time complexity.

Algorithm 7.42 The procedure $\text{calc}M_{lr}(P)$ computes the M_{lr} -array based on the BWT of S .

```

compute the matching statistics  $ms$  of  $P$  w.r.t.  $S$  by Alg. 7.25 (page 337)
 $m \leftarrow |P|$ 
 $k \leftarrow 0$ 
 $j \leftarrow 1$ 
 $flag \leftarrow true$ 
while  $j \leq m$  do
  if  $flag = true$  then
    for  $i = j$  to  $j + ms[j] - 1$  do
       $M_{lr}[i] \leftarrow k$ 
     $j \leftarrow j + ms[j]$ 
     $flag \leftarrow false$ 
  else
     $k \leftarrow k + 1$ 
     $M_{lr}[j] \leftarrow k$ 
     $j \leftarrow j + 1$ 
     $flag \leftarrow true$ 

```

7.9.2 Using bidirectional search

During the development of the tool Bowtie, Langmead et al. [198] observed that a similar approach as in Algorithm 7.40 suffered from excessive backtracking. They write:

Backtracking scenarios play out within the context of a stack structure that grows when a new substitution is introduced and shrinks when the aligner rejects all candidate alignments for the substitutions currently on the stack.

At first glance, there is no stack in Algorithm 7.40, but it is implicitly there: when the procedure is called, the program's runtime environment keeps track of the various instances of the procedure using a call stack. Bowtie mitigates excessive backtracking using two indexes that support backward and forward search (but without synchronization). We use the 1-mismatch problem to convey the flavor of the method. The mismatch (if there is one at all) either occurs (a) in the first half or (b) in the second half of the pattern. Let $s = \lfloor \frac{m}{2} \rfloor$.

- (a) In this case, the second half $P[s + 1..m]$ of the pattern must match exactly, and the procedure call $\text{backwardSearch}(P[s + 1..m])$ returns the $P[s + 1..m]$ -interval $[lb..rb]$ (if it exists). It then tries to extend this exact match to the left, allowing for one mismatch. This is exactly what the procedure $k\text{-mismatch}(P, s, 1, [lb..rb])$ does.

- (b) In this case, the first half $P[1..s]$ of the pattern must match exactly, and the $P[1..s]$ -interval is computed by a forward search. This time, it tries to extend the exact match to the right, allowing for one mismatch.

Let us compare the approach with Algorithm 7.40. In case $k = 1$, Algorithm 7.40 proceeds as follows: For each position j in P , it uses a backward search to find the $P[j + 1..m]$ -interval, generates all $bP[j + 1..m]$ -intervals (where b can be any character except \$), and for each such interval it continues the backward search to find the $P[1..j - 1]bP[j + 1..m]$ -interval. If there is no 1-mismatch of P in S , all searches eventually end up with a dead end. Of course, the same is true for the method described above but starting with a rather long exact match often speeds up the search. This is because the $P[s + 1..m]$ -interval ($P[1..s]$ -interval, respectively) is usually small (or even empty), and there is a fair chance that the search will stop after a few more steps.

However, a generalization of the method to more than one mismatch requires the ability to search bidirectionally for a pattern. Let us consider the case $k = 2$ to see why this is so. We split the pattern into three parts of (almost) equal size. There are six different ways to distribute two mismatches:

$$(a) 200 \quad (b) 110 \quad (c) 020 \quad (d) 011 \quad (e) 002 \quad (f) 101$$

Cases (a)–(c) can be handled by a backward search, where at least the last part of the pattern must match exactly, and cases (d)–(e) can be handled by a forward search, where at least the first part of the pattern must match exactly. By contrast, if one starts with the middle of the pattern in case (f), then one must be able to search bidirectionally as explained in Section 7.8.4. And this is exactly what the software tool 2BWT [197] does.

In the explanation below, the pattern P is divided into the three parts $P[1..s_1]$, $P[s_1 + 1..s_2]$, and $P[s_2 + 1..m]$, where $s_1 = \lfloor \frac{m}{3} \rfloor$ and $s_2 = m - s_1$.

- (a)–(c) The algorithm first determines the $P[s_2 + 1..m]$ -interval $[lb..rb]$ by backward search. Then, the procedure call $k\text{-mismatch}(P, s_2, 2, [lb..rb])$ delivers the set \mathcal{I} of all intervals $[p..q]$ with the property: for every r with $p \leq r \leq q$, a 2-mismatch \bar{P} of P in S starts at position $\text{SA}[r]$, and $\bar{P}[s_2 + 1..m] = P[s_2 + 1..m]$.
- (d) First, the algorithm determines the $P[1..s_1]$ -interval by a forward search. Second, it tries to extend this exact match to the middle of the pattern, allowing for one mismatch. Third, for every 1-mismatch \bar{P} of $P[1..s_2]$ in S obtained in this way, it tries to extend the match to the last part, again allowing for one mismatch.

- (e) The algorithm determines the $P[1..s_2]$ -interval $[lb..rb]$ by a forward search and tries to extend this exact match to the last part of P , allowing for two mismatches.
- (f) The algorithm determines the $P[s_1+1..s_2]$ -interval by a forward search. Continuing the forward search, it tries to extend this exact match to the last part of P , allowing for one mismatch. At that point in time, for every 1-mismatch \bar{P} of $P[s_1+1..m]$ in S obtained in this way, the forward search delivers the \bar{P}^{rev} -interval $[lb^{rev}..rb^{rev}]$ in the forward index and the \bar{P} -interval $[lb..rb]$ in the backward index. Then, the procedure call $k\text{-mismatch}(P, s_1, 1, [lb..rb])$ completes the job.

As an example, we solve the 2-mismatch problem for the pattern $P = \text{ttaatt}$ and the string $S = \text{ctaataatg\$}$; see Figure 7.38 (page 377). The pattern is divided into the three parts tt , aa , and tt . The algorithm starts with the last part and searches backwards for tt . Because there is no exact match of tt with a substring of S , there is no 2-mismatch of the types (a)–(c). Then, the algorithm searches for the first part of P in forward direction. Again, this search fails and we conclude that there is no 2-mismatch of the types (d)–(e). When the algorithm searches for the middle part aa of P , it finds the aa -interval $[2..3]$ (both in the forward and backward index). Continuing the forward search, it detects the 1-mismatches $aata$ and $aatg$ of $aatt$ in S . Backward search with the $aata$ -interval $[2..2]$ yields the 2-mismatch $ctaata$ of P in S , which starts at position $\text{SA}[6] = 1$. Similarly, starting with the $aatg$ -interval $[3..3]$, it finds the 2-mismatch $ataatg$, which begins at position $\text{SA}[4] = 4$ in S .

Pseudo-code of the algorithm described above can be found in Algorithm 7.43. Apart from the procedure *backwardSearch* (Algorithm 7.8 on page 303), it uses the following procedures:

- $\text{forwardSearch}(c, [lb..rb], [lb^{rev}..rb^{rev}])$: cf. Algorithm 7.39 (page 373).
 Input: character c , the ω -interval $[lb..rb]$ in the backward index, and the ω^{rev} -interval $[lb^{rev}..rb^{rev}]$ in the forward index.
 Output: the ωc -interval in the backward index and the $c\omega^{rev}$ -interval in the forward index.
- $\text{forwardSearch}(P[i..j], [lb..rb], [lb^{rev}..rb^{rev}])$
 Input: string $P[i..j]$, the ω -interval $[lb..rb]$ in the backward index, and the ω^{rev} -interval $[lb^{rev}..rb^{rev}]$ in the forward index.
 Output: the $\omega P[i..j]$ -interval in the backward index as well as the $P[i..j]^{rev}\omega^{rev}$ -interval in the forward index.
- $k\text{-mismatch}(P, j, d, [lb..rb])$: cf. Algorithm 7.40 (page 376).
 Input: pattern P , a position j in P , number d of allowed mismatches, and the ω -interval $[lb..rb]$.
 Output: the set \mathcal{I} of all intervals $[p..q]$ so that $S[\text{SA}[r].. \text{SA}[r] + j - 1]$ is a

Algorithm 7.43 This procedure finds all 2-mismatches of P in S .

```

procedure 2-mismatch( $P$ )
   $s_1 \leftarrow \lfloor \frac{m}{3} \rfloor$ 
   $s_2 \leftarrow m - s_1$ 
   $\mathcal{I} \leftarrow \emptyset$ 
  /* Cases (a)–(c) */
   $[lb..rb] \leftarrow \text{backwardSearch}(P[s_2 + 1..m])$ 
  if  $[lb..rb] \neq \perp$  then
     $\mathcal{I} \leftarrow k\text{-mismatch}(P, s_2, 2, [lb..rb])$ 
  /* Case (d) */
   $([lb..rb], [lb^{rev}..rb^{rev}]) \leftarrow \text{forwardSearch}(P[1..s_1], [1..n], [1..n])$ 
  if  $([lb..rb], [lb^{rev}..rb^{rev}]) \neq \perp$  then
     $\mathcal{F} \leftarrow k\text{-mismatchF}(P, s_1 + 1, s_2, 1, [lb..rb], [lb^{rev}..rb^{rev}])$ 
    for each  $([lb..rb], [lb^{rev}..rb^{rev}]) \in \mathcal{F}$  do
       $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatchF}(P, s_2 + 1, m, 1, [lb..rb], [lb^{rev}..rb^{rev}])$ 
  /* Case (e), note that  $[lb..rb]$  is still the  $P[1..s_1]$ -interval */
   $([lb..rb], [lb^{rev}..rb^{rev}]) \leftarrow \text{forwardSearch}(P[s_1 + 1..s_2], [lb..rb], [lb^{rev}..rb^{rev}])$ 
  if  $([lb..rb], [lb^{rev}..rb^{rev}]) \neq \perp$  then
     $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatchF}(P, s_2 + 1, m, 2, [lb..rb], [lb^{rev}..rb^{rev}])$ 
  /* Case (f) */
   $([lb..rb], [lb^{rev}..rb^{rev}]) \leftarrow \text{forwardSearch}(P[s_1 + 1..s_2], [1..n], [1..n])$ 
  if  $([lb..rb], [lb^{rev}..rb^{rev}]) \neq \perp$  then
     $\mathcal{F} \leftarrow k\text{-mismatchF}(P, s_2 + 1, m, 1, [lb..rb], [lb^{rev}..rb^{rev}])$ 
    for each  $([lb..rb], [lb^{rev}..rb^{rev}]) \in \mathcal{F}$  do
       $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatch}(P, s_1, 1, [lb..rb])$ 
return  $\mathcal{I}$ 

```

d -mismatch of $P[1..j]$ and $S[\text{SA}[r] + j..\text{SA}[r] + j + |\omega| - 1] = \omega$ for every r with $p \leq r \leq q$.

- $k\text{-mismatchF}(P, i, j, d, [lb..rb], [lb^{rev}..rb^{rev}])$: cf. Algorithm 7.44.
 Input: pattern P , positions i and j in P with $i \leq j$, number d of allowed mismatches, the ω -interval $[lb..rb]$ in the backward index, and the ω^{rev} -interval $[lb^{rev}..rb^{rev}]$ in the forward index.
 Output: the set \mathcal{I} of all pairs $([p..q], [p^{rev}..q^{rev}])$ of intervals so that $S[\text{SA}[r]..\text{SA}[r] + |\omega| - 1] = \omega$ and $S[\text{SA}[r] + |\omega|..\text{SA}[r] + |\omega| + (j - i)]$ is a d -mismatch of $P[i..j]$ for every r with $p \leq r \leq q$; the interval $[p^{rev}..q^{rev}]$ has an analogous property.

Exercise 7.9.12 For each $c \in \Sigma \setminus \{\$ \}$, the for-loop in Algorithm 7.44 performs a forward search step. Give pseudo-code of a procedure that carries out these steps simultaneously by one top-down traversal of the wavelet tree; cf. procedure *getIntervals* presented in Algorithm 7.16 (page 316).

Algorithm 7.44 The procedure $k\text{-mismatch}F$.

```

procedure  $k\text{-mismatch}F(P, i, j, d, [lb..rb], [lb^{rev}..rb^{rev}])$ 
  if  $d < 0$  then
    return  $\emptyset$ 
  if  $i = j + 1$  then
    return  $\{[lb..rb]\}$ 
   $\mathcal{I} \leftarrow \emptyset$ 
  for each  $c \in \Sigma \setminus \{\$ \}$  do
     $([lb..rb], [lb^{rev}..rb^{rev}]) \leftarrow \text{forwardSearch}(c, [lb..rb], [lb^{rev}..rb^{rev}])$ 
    if  $([lb..rb], [lb^{rev}..rb^{rev}]) \neq \perp$  then      /*  $lb \leq rb$  */
      if  $c = P[i]$  then
         $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatch}F(P, i + 1, j, d, [lb..rb], [lb^{rev}..rb^{rev}])$ 
      else
         $\mathcal{I} \leftarrow \mathcal{I} \cup k\text{-mismatch}F(P, i + 1, j, d - 1, [lb..rb], [lb^{rev}..rb^{rev}])$ 
  return  $\mathcal{I}$ 

```

The above methodology can be generalized to handle three or more mismatches as well as indels; see [197].

The preceding algorithm can be viewed as an instance of the seed-and-extend paradigm. A seed-and-extend algorithm reduces an approximate matching problem (like the k -mismatches problem, the k -differences problem, or the problem of finding degenerate repeats [195]) to an exact matching problem. It involves two steps:

- identifying exact matches (or repeats) of a certain minimum length—these are the seeds;
- finding inexact matches (or repeats) by extending these seeds.

In the k -differences problem, for example, one can partition the pattern P into consecutive regions of length

$$\left\lfloor \frac{m}{k+1} \right\rfloor$$

and use these regions as seeds; see [23, 139].

Sequence Alignment

In this chapter, we discuss methods to measure how similar biological sequences are. In modern molecular biology, this is important because of the, as Dan Gusfield [139] calls it, “first fact of biological sequence analysis:”

In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.

To understand why this is so, we need to know the basic principles of evolution. Evolution is broadly described as the theory that all life on earth is descended from a single common ancestor. Genes in two species are *homologous* if the same gene was present in their last common ancestor. The term *homolog* may apply to the relationship between genes separated (a) by the event of speciation or (b) by the event of duplication. In case (a), the genes are *orthologs*. Normally, orthologs retain the same function in the course of evolution. The identification of orthologs is critical for a reliable prediction of gene function in newly sequenced genomes. In case (b), the genes are *paralogs*. Paralogs evolve new functions, usually related to the original one. This is possible because after duplication there are two copies of the same gene, and mutations altering the product of one copy are not harmful to the organism as long as the other copy functions properly (this paradigm of protein evolution is often called “duplication with modification”).

Sequence alignment allows us to measure sequence similarity. David Mount [227] introduces it as follows:

Sequence alignment is the procedure of comparing two (pair-wise alignment) or more (multiple alignment) sequences by searching for a series of individual characters or character patterns that are in the same order in the sequences. Two sequences are aligned by writing them across a page in rows.

Identical or similar characters are placed in the same column, and nonidentical characters can either be placed in the same column as a mismatch or opposite a gap in the other sequence. In an optimal alignment, nonidentical characters and gaps are placed to bring as many identical or similar characters as possible into vertical register. Sequences that can readily be aligned in this manner are said to be similar.

Sequence alignment is central in bioinformatics. As mentioned above, one distinguishes between pairwise and multiple alignment. The former will be discussed in Section 8.1 and the latter is dealt with in Section 8.2. Moreover, there are two types of sequence alignment: global and local. Global alignments form the basis of phylogenetic inference (see Chapter 10) and comparative genomics, whereas similarity detected in a local alignment is generally interpreted as structural/functional closeness. In this book, we will focus on global alignment methods.

8.1 Pairwise alignment

In the following, let S^1 and S^2 be strings on the alphabet Σ of length n_1 and n_2 , respectively. The formal definition of an alignment between S^1 and S^2 reads as follows.

Definition 8.1.1 A *global alignment* between S^1 and S^2 is a $(2 \times n)$ matrix A so that:

1. $A(i, j) \in \Sigma \cup \{-\}$, where $-$ is a special gap symbol not occurring in Σ .
2. After removal of all gap symbols the first row of A equals S^1 and the second row of A equals S^2 .
3. No column of A consists solely of gap symbols.

Note that condition (2) implies that $n \geq \max\{n_1, n_2\}$, while condition (3) has $n \leq n_1 + n_2$ as a consequence. In fact, for $n_1 \geq n_2$ a shortest possible alignment between S^1 and S^2 is the alignment

$$\begin{pmatrix} S^1[1] & S^1[2] & \dots & S^1[n_2] & S^1[n_2 + 1] & \dots & S^1[n_1] \\ S^2[1] & S^2[2] & \dots & S^2[n_2] & - & \dots & - \end{pmatrix}$$

of length $n = n_1 = \max\{n_1, n_2\}$, while a longest possible alignment between S^1 and S^2 is the following alignment of length $n = n_1 + n_2$:

$$\begin{pmatrix} S^1[1] & S^1[2] & \dots & S^1[n_1] & - & - & \dots & - \\ - & - & \dots & - & S^2[1] & S^2[2] & \dots & S^2[n_2] \end{pmatrix}$$

$$\begin{pmatrix} a & g & g & c & t & g & a \\ a & g & g & g & g & a & a \end{pmatrix} \qquad \begin{pmatrix} a & g & g & c & t & g & a & - \\ a & g & g & - & g & g & a & a \end{pmatrix}$$

Figure 8.1: Two alignments of the strings $S^1 = aggc tga$ and $S^2 = aggggaa$.

In what follows, we will denote the upper row of an alignment A of length n by x_1, x_2, \dots, x_n and the lower row by y_1, y_2, \dots, y_n . That is,

$$A = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ y_1 & y_2 & y_3 & \dots & y_{n-1} & y_n \end{pmatrix}$$

The j -th column of A is called a

- *match* (identity) if $x_j, y_j \in \Sigma$ with $x_j = y_j$,
- *substitution* (replacement) if $x_j, y_j \in \Sigma$ with $x_j \neq y_j$,
- *insertion* if $x_j = -$ and $y_j \in \Sigma$,
- *deletion* if $x_j \in \Sigma$ and $y_j = -$.

We will use the term *indels* as shorthand for insertions and deletions.

As an example, consider the two alignments of the strings $S^1 = aggc tga$ and $S^2 = aggggaa$ in Figure 8.1. Which one is better? To answer this question, we need a measurement that allows us to assess the quality of an alignment. Usually, one of the following two methods is used for this purpose:

1. Minimum distance method: given a *cost* (distance) function δ , find an alignment of minimum cost (distance).
2. Maximum similarity method: given a *similarity* function σ , find an alignment of maximum similarity score.

Of course, the functions have to satisfy certain properties. We shall see later that there is a duality between the two methods (for global alignments, but not for local alignments).

8.1.1 Distance methods

We start our considerations with functions that assign the same cost r to each substitution and assign the same cost d to each indel. This makes sense for DNA sequences but hardly for amino acid sequences.

Definition 8.1.2 An operation-weighted *cost function*

$$\delta : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \setminus \{(-, -)\} \rightarrow \mathbb{R}_{\geq 0}$$

assigns to each pair (x, y) the following value:

- $\delta(x, y) = 0$ if $x = y$ (match),
- $\delta(x, y) = r > 0$ if $x, y \in \Sigma$ with $x \neq y$ (substitution),
- $\delta(x, y) = d > 0$ if $x = -$ and $y \in \Sigma$ or $x \in \Sigma$ and $y = -$ (indel).

We will also use the notation $\delta(\text{match}) = 0$, $\delta(\text{sub}) = r$, and $\delta(\text{indel}) = d$.

Definition 8.1.3 The *cost of an alignment*

$$A = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ y_1 & y_2 & y_3 & \dots & y_{n-1} & y_n \end{pmatrix}$$

of two strings S^1 and S^2 for the cost function δ is

$$\delta(A) = \sum_{i=1}^n \delta(x_i, y_i)$$

An alignment A^{opt} of S^1 and S^2 is *optimal* for the cost function δ if

$$\delta(A^{\text{opt}}) = \min\{\delta(A) \mid A \text{ is an alignment of } S^1 \text{ and } S^2\}$$

We remark that a cost function δ as defined in Definition 8.1.2 satisfies the following metric axioms:

- $\delta(x, y) = 0$ if and only if $x = y$ (identity of indiscernibles)
- $\delta(x, y) = \delta(y, x)$ (symmetry)

Therefore, δ would be a *distance function* if it would satisfy the triangle inequality $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$. However, in general, this need not be the case. For instance, if $2d < r$, then for $x, y \in \Sigma$ with $x \neq y$, we have

$$\delta(x, y) = r > 2d = \delta(x, -) + \delta(-, y)$$

In other words, in an alignment the deletion of x followed by the insertion of y is cheaper than the substitution of x by y ; thus, no optimal alignment can have a substitution. To avoid such cases, we will tacitly assume that δ is a distance function, i.e., it also satisfies the triangle inequality. Exercise 8.1.4 asks you to prove that this is the case if and only if $r \leq 2d$.

Exercise 8.1.4 Let δ be a cost function with $\delta(\text{match}) = 0$, $\delta(\text{sub}) = r > 0$, and $\delta(\text{indel}) = d > 0$. Show that δ is a distance function if and only if $r \leq 2d$.

For example, the function δ that assigns the same cost 1 to each substitution and indel (so $\delta(\text{match}) = 0$, $\delta(\text{sub}) = 1$, and $\delta(\text{indel}) = 1$) is a distance function. It will henceforth be called *Levenshtein costs*.

Both alignments in Figure 8.1 have cost 3 for Levenshtein costs.

The next lemma states that a part of an optimal alignment between two strings S^1 and S^2 is itself an optimal alignment of certain substrings of S^1 and S^2 .

Lemma 8.1.5 *Let A^{opt} be an optimal alignment between the two strings S^1 and S^2 . Let \bar{A} be a part of A^{opt} that aligns a substring $S^1[i..j]$ of S^1 with a substring $S^2[k..l]$ of S^2 . Then \bar{A} is an optimal alignment of $S^1[i..j]$ and $S^2[k..l]$.*

Proof For a proof by contradiction, suppose that \bar{A} is not an optimal alignment of $S^1[i..j]$ and $S^2[k..l]$. This means that $\delta(\bar{A}) > \delta(\bar{A}^{\text{opt}})$, where \bar{A}^{opt} is an optimal alignment of $S^1[i..j]$ and $S^2[k..l]$. However, \bar{A}^{opt} can be substituted for \bar{A} in A^{opt} , decreasing the cost of A^{opt} . Thus, A^{opt} is not an optimal alignment of S^1 and S^2 , a contradiction. \square

We shall see that an optimal alignment between S^1 and S^2 can be obtained by successively computing the minimum costs of aligning prefixes of S^1 and S^2 , respectively.

Definition 8.1.6 Given a cost function δ and two strings S^1 and S^2 of length n_1 and n_2 , respectively, we define for $0 \leq i \leq n_1$ and $0 \leq j \leq n_2$:

$$D(i, j) = \min\{\delta(A) \mid A \text{ is an alignment of } S^1[1..i] \text{ and } S^2[1..j]\}$$

In the special case $i = 0$, the alignment of $S^1[1..i] = \varepsilon$ and $S^2[1..j]$ consists of j insertions. Hence $D(0, j) = j \cdot d$. Analogously, we obtain $D(i, 0) = i \cdot d$. Theorem 8.1.7 states how $D(i, j)$ can be calculated for $i \neq 0$ and $j \neq 0$.

Theorem 8.1.7 *For a cost function δ with $\delta(\text{match}) = 0$, $\delta(\text{sub}) = r > 0$, and $\delta(\text{indel}) = d > 0$, we have*

$$\begin{aligned} D(0, 0) &= 0 \\ D(i, 0) &= i \cdot d \\ D(0, j) &= j \cdot d \\ D(i, j) &= \min \left\{ \begin{array}{l} D(i-1, j) + d \\ D(i, j-1) + d \\ D(i-1, j-1) + \delta(S^1[i], S^2[j]) \end{array} \right\} \end{aligned}$$

for each $1 < i \leq n_1$ and $1 < j \leq n_2$, where $\delta(S^1[i], S^2[j]) = r$ if $S^1[i] \neq S^2[j]$ and $\delta(S^1[i], S^2[j]) = 0$ if $S^1[i] = S^2[j]$.

Proof The cases $i = 0$ or $j = 0$ (or both) are obvious. So suppose that $1 < i \leq n_1$ and $1 < j \leq n_2$. Let

$$A^{opt} = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ y_1 & y_2 & y_3 & \dots & y_{n-1} & y_n \end{pmatrix}$$

be an optimal alignment between the two strings $S^1[1..i]$ and $S^2[1..j]$. By Definition 8.1.6, $D(i, j) = \delta(A^{opt})$. Let

$$A = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} \\ y_1 & y_2 & y_3 & \dots & y_{n-1} \end{pmatrix}$$

- If A^{opt} ends with an insertion, i.e., $x_n = -$ and $y_n = S^2[j]$, then A is an optimal alignment between the two strings $S^1[1..i]$ and $S^2[1..j-1]$ by Lemma 8.1.5. That is, $D(i, j-1) = \delta(A)$. Therefore, $D(i, j) = \delta(A^{opt}) = \delta(A) + d = D(i, j-1) + d$.
- If A^{opt} ends with a deletion, i.e., $x_n = S^1[i]$ and $y_n = -$, then A is an optimal alignment between the strings $S^1[1..i-1]$ and $S^2[1..j]$. Thus, $D(i, j) = \delta(A^{opt}) = \delta(A) + d = D(i-1, j) + d$.
- If A^{opt} ends with a match or a substitution, i.e., $x_n = S^1[i]$ and $y_n = S^2[j]$, then A is an optimal alignment between the two strings $S^1[1..i-1]$ and $S^2[1..j-1]$. Consequently, $D(i, j) = \delta(A^{opt}) = \delta(A) + \delta(S^1[i], S^2[j]) = D(i-1, j-1) + \delta(S^1[i], S^2[j])$.

Of course we do not know an optimal alignment, but such an alignment must end either with an insertion or a deletion or a substitution, and we know the cost in each of these cases. Because an optimal alignment has the least cost of these three possibilities, the theorem is proven. \square

The recurrence relation of Theorem 8.1.7 can be solved by a dynamic programming algorithm that tabulates the values $D(i, j)$ in a matrix D . The algorithm starts with solutions to the simplest subproblems $i = 0$ or $j = 0$ (the first row and the first column of D). Then, it fills the matrix row by row (or column by column). Figure 8.2 depicts an example. In the bioinformatics community, the dynamic programming algorithm is usually referred to as “the Needleman and Wunsch” algorithm because Needleman and Wunsch [240] were the first to devise such an algorithm (although their algorithm has some limitations). In the computer science community this algorithm is often attributed to Wagner and Fisher [323]. For more details on dynamic programming algorithms in computational biology and their history, the reader is referred to [326]. An algebraic style of dynamic programming over sequence data is described in [122, 281].

In general, *dynamic programming* is a method of solving complex problems (often optimization problems) by breaking them down into simpler steps. It is applicable to problems that exhibit the properties of optimal substructure and overlapping subproblems. A problem has

		c	g	a	c	a	c	
D		0	1	2	3	4	5	6
	0	0	1	2	3	4	5	6
g	1	1	1	1	2	3	4	5
t	2	2	2	2	2	3	4	5
a	3	3	3	3	2	3	3	4
t	4	4	4	4	3	3	4	4
c	5	5	4	5	4	3	4	4

Figure 8.2: The D -matrix of the strings $S^1 = gtatc$ and $S^2 = cgacac$ for Levenshtein costs.

- *optimal substructure* (or satisfies Richard Bellman's Principle of Optimality [34]) if an optimal solution can be constructed efficiently from optimal solutions to its subproblems,
- *overlapping subproblems* if the problem can be broken down into subproblems that are reused several times.

The minimum cost of aligning S^1 and S^2 can be found at entry $D(n_1, n_2)$ in the D -matrix. For example, an optimal global alignment between the strings $S^1 = gtatc$ and $S^2 = cgacac$ has cost 4; see Figure 8.2. Given the matrix D , an optimal alignment can be obtained by a traceback from the entry $D(n_1, n_2)$ to the entry $D(0, 0)$. It is instructive to use a graph-theoretical formulation of the problem.

Definition 8.1.8 The *alignment graph* $G = (V, E)$ of two strings S^1 and S^2 is an edge labeled directed graph with $V = \{(i, j) \mid 0 \leq i \leq n_1, 0 \leq j \leq n_2\}$ and E consists of the following edges:

- For every i and j with $1 \leq i \leq n_1, 0 \leq j \leq n_2$, there is a deletion edge

$$(i-1, j) \xrightarrow{(S^1[i], -)} (i, j)$$

The deletion edge is optimal if $D(i, j) = D(i-1, j) + d$.

- For every i and j with $0 \leq i \leq n_1, 1 \leq j \leq n_2$, there is an insertion edge

$$(i, j-1) \xrightarrow{(-, S^2[j])} (i, j)$$

The insertion edge is optimal if $D(i, j) = D(i, j-1) + d$.

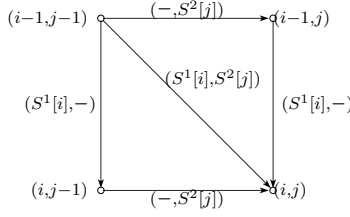
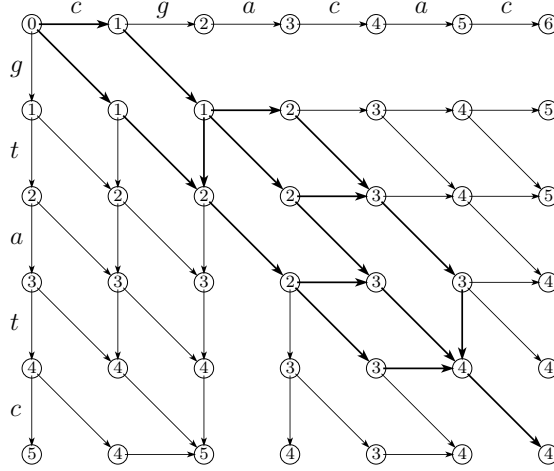


Figure 8.3: Alignment graph (cut-out).

Figure 8.4: Optimal edges in the alignment graph of $S^1 = gtatc$ and $S^2 = cgacac$. Following thick edges from node $(0, 0)$ to node (n_1, n_2) yields an optimal path.

- For every i and j with $1 \leq i \leq n_1, 1 \leq j \leq n_2$, there is a substitution edge

$$(i-1, j-1) \xrightarrow{(S^1[i], S^2[j])} (i, j)$$

The substitution edge is optimal if $D(i, j) = D(i-1, j-1) + \delta(S^1[i], S^2[j])$.

An *optimal path* in the alignment graph is a path from node $(0, 0)$ to node (n_1, n_2) that solely consists of optimal edges.

Definition 8.1.8 is illustrated in Figures 8.3 and 8.4.

Each path in the alignment graph from node $(0, 0)$ to node (n_1, n_2) corresponds to an alignment of S^1 and S^2 . To be precise, the concatenation of the edge labels along the path yields the alignment. It is not difficult to see that an optimal path in the alignment graph corresponds to an optimal alignment of S^1 and S^2 , and vice versa. For example, the optimal paths in the alignment graph of Figure 8.4 correspond to the optimal alignments of Figure 8.5.

- g t a t - c	g t a t - c	- g t - a t c	- g - t a t c	- g t a - t c	g t a - t c	- g t a t c
c g - a c a c	c g a c a c	c g a c a - c	c g a c a - c	c g - a c a c	c g a c a c	c g a c a c

Figure 8.5: Optimal alignments between $S^1 = gtatc$ and $S^2 = cgacac$.

Finding an optimal path amounts to tracing back along optimal edges from node (n_1, n_2) to node $(0, 0)$. If the D -matrix is available, then we can decide at node (i, j) which of its incoming edges are optimal by simply testing whether $D(i, j) = D(i - 1, j) + d$, $D(i, j) = D(i, j - 1) + d$, and $D(i, j) = D(i - 1, j - 1) + \delta(S^1[i], S^2[j])$. Each test that returns *true* corresponds to an optimal edge that is part of an optimal path. However, this approach requires the storage of the entire D -matrix. We observe that the computation of the value $D(n_1, n_2)$ can be done in $O(n_2)$ space ($O(n_1)$ space, respectively) because the computation of one row (column, respectively) of D is essentially based on the previous row (column, respectively). To still be able to find an optimal path we use three bits at each node (i, j) to mark incoming edges (these bits are set during the computation of the value $D(n_1, n_2)$):

- The first bit is set if and only if $D(i, j) = D(i - 1, j) + d$.
- The second bit is set if and only if $D(i, j) = D(i, j - 1) + d$.
- The third bit is set if and only if $D(i, j) = D(i - 1, j - 1) + \delta(S^1[i], S^2[j])$.

Then, at node (i, j) an optimal path can be continued backwards with

- the incoming deletion edge if the first bit of (i, j) is set,
- the incoming insertion edge if the second bit of (i, j) is set,
- the incoming substitution edge if the third bit of (i, j) is set.

Exercise 8.1.9 Give pseudo-code for the algorithms described above, and analyze their time and space complexities.

8.1.2 Computing an optimal alignment in linear space

In this section, we shall see that an optimal alignment between two strings can be computed in linear space [152]. For ease of presentation, we assume throughout the section that n_1 (the length of S^1) is a power of 2. To illustrate the idea, suppose for a moment that we know an optimal alignment A between the strings S^1 and S^2 . Then we can split A into two subalignments A^1 and A^2 so that A^1 contains the first half of S^1 and A^2

	0	1	2	3	4	5	6	7	8	9
0	x									
1		x								
2			x							
3				x						
4					x	x				
5							x			
6								x		
7									x	
8										x

Figure 8.6: An optimal path is split into two subpaths by row $n_1/2 = 4$.

contains the second half. More precisely, A^1 is the alignment between $S^1[1..n_1/2]$ and some prefix $S^2[1..k]$ of S^2 and A^2 is the alignment between $S^1[n_1/2 + 1..n_1]$ and $S^2[k + 1..n_2]$. Both alignments A^1 and A^2 are optimal by Lemma 8.1.5. Proceeding recursively, we can split the emerging alignments into two “halves” until a base case is reached. Conversely, if we wish to compute an optimal alignment between S^1 and S^2 and we have an oracle that tells us what k is, then we can use the divide-and-conquer approach to actually compute an optimal alignment:

- Compute an optimal alignment A^1 between the strings $S^1[1..n_1/2]$ and $S^2[1..k]$.
- Compute an optimal alignment A^2 between the strings $S^1[n_1/2 + 1..n_1]$ and $S^2[k + 1..n_2]$.
- The concatenation A of the alignments A^1 and A^2 is an optimal alignment between S^1 and S^2 .

This is a fairly simple idea, but of course we do not have such an oracle to assist us, so we must find an algorithmic way to determine k . To put the problem differently, we have to find a column k at which an optimal path in the (unknown) alignment graph of S^1 and S^2 “leaves” the row $n_1/2$. In the example of Figure 8.6, we have $k = 5$.

In order to identify node $(n_1/2, k)$, we try all combinations of a forward path from node $(0, 0)$ to node $(n_1/2, j)$, $1 \leq j \leq n_2$, with a backward path from node (n_1, n_2) to this node $(n_1/2, j)$, and add up the costs of these two paths. Then, $(n_1/2, k)$ can be found by comparing these costs, as we shall see.

Definition 8.1.10 Let S_{rev} denote the reverse of the string S . Given a cost function δ and two strings S^1 and S^2 of length n_1 and n_2 , respectively, we

define for $0 \leq i \leq n_1$ and $0 \leq j \leq n_2$:

$$D^{rev}(i, j) = \min\{\delta(A) \mid A \text{ is an alignment of } S_{rev}^1[1..i] \text{ and } S_{rev}^2[1..j]\}.$$

Exercise 8.1.11 asks you to show that $D^{rev}(i, j)$ is the cost of an optimal alignment of the last i characters of S^1 and the last j characters of S^2 .

Exercise 8.1.11 Prove that

$$D^{rev}(i, j) = \min\{\delta(A) \mid A \text{ is alignment of } S^1[n_1 - i + 1..n_1] \text{ and } S^2[n_2 - j + 1..n_2]\}$$

Lemma 8.1.12 *The following equality holds:*

$$D(n_1, n_2) = \min_{0 \leq j \leq n_2} \{D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)\}$$

Proof We show (a) $D(n_1, n_2) \geq \min_{0 \leq j \leq n_2} \{D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)\}$ and (b) $D(n_1, n_2) \leq \min_{0 \leq j \leq n_2} \{D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)\}$.

(a) Let A^{opt} be an optimal alignment of S^1 and S^2 , i.e., $\delta(A^{opt}) = D(n_1, n_2)$. Split A^{opt} immediately after the column that contains the character $S^1[n_1/2]$ into two subalignments A^1 and A^2 . By Lemma 8.1.5, A^1 is an optimal alignment of $S^1[1..n_1/2]$ and $S^2[1..k]$, where k is the rightmost position in S^2 that is aligned in A^{opt} with a character at or before position $n_1/2$ in S^1 . Analogously, A^2 is an optimal alignment of $S^1[n_1/2 + 1..n_1]$ and $S^2[k + 1..n_2]$. Clearly, $\delta(A^{opt}) = \delta(A^1) + \delta(A^2)$ and $\delta(A^1) = D(n_1/2, k)$. Furthermore, $\delta(A^2) = D^{rev}(n_1/2, n_2 - k)$ according to Exercise 8.1.11. To sum up, we have

$$D(n_1, n_2) = D(n_1/2, k) + D^{rev}(n_1/2, n_2 - k) \geq \min_{0 \leq j \leq n_2} \{D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)\}$$

(b) Suppose $k \in \arg \min_{0 \leq j \leq n_2} \{D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)\}$. Let A^1 be an optimal alignment of $S^1[1..n_1/2]$ and $S^2[1..k]$, i.e., $\delta(A^1) = D(n_1/2, k)$. Furthermore, let A^2 be an optimal alignment of $S^1[n_1/2 + 1..n_1]$ and $S^2[k + 1..n_2]$, i.e., $\delta(A^2) = D^{rev}(n_1/2, n_2 - k)$ by Exercise 8.1.11. Obviously, the concatenation of A^1 and A^2 yields an alignment of S^1 and S^2 with cost $D(n_1/2, k) + D^{rev}(n_1/2, n_2 - k)$. Therefore,

$$\min_{0 \leq j \leq n_2} \{D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)\} = D(n_1/2, k) + D^{rev}(n_1/2, n_2 - k) \geq D(n_1, n_2)$$

□

So the number k we are searching for is a position j in S^2 that minimizes $D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)$. To determine this number, the dynamic programming matrices D and D^{rev} are calculated up to row $n_1/2$; see Figure 8.7. In the example of Figure 8.7, there are three position that minimize this value: 3, 4, and 5.

Pseudo-code of Hirschberg's divide and conquer algorithm is given in Algorithm 8.1. The computation of an optimal alignment of S^1 and S^2

D		a	t	a	a	a	a	t	g	g	
	0	1	2	3	4	5	6	7	8	9	
a	1	0	1	2	3	4	5	6	7	8	
g	2	1	1	2	3	4	5	6	6	7	
t	3	2	1	2	3	4	5	5	6	7	
a	4	3	2	1	2	3	4	5	6	7	
Σ	9	7	5	3	3	3	5	7	9	11	
	5	4	3	2	1	0	1	2	3	4	a
	6	5	4	3	2	1	0	1	2	3	t
	7	6	5	4	3	2	1	0	1	2	g
	8	7	6	5	4	3	2	1	0	1	g
	9	8	7	6	5	4	3	2	1	0	
	a	t	a	a	a	a	t	g	g		D^{rev}

Figure 8.7: The dynamic programming matrices D and D^{rev} are computed up to row $n_1/2$, using Levenshtein costs. In the figure, the calculation of D starts at the top row and proceeds from left to right, whereas the calculation of D^{rev} starts at the bottom row and proceeds from right to left. For each j with $0 \leq j \leq n_2$, we have $\Sigma[j] = D(n_1/2, j) + D^{rev}(n_1/2, n_2 - j)$, where Σ is the array depicted in the middle of the figure.

starts with the procedure call $Hb(1, n_1, 1, n_2)$. In the first base case, the algorithm outputs an optimal alignment of $S^1[l_1..r_1]$ and ε , which consists of $r_1 - l_1 + 1$ deletions. In the second base case $l_1 = r_1$, the algorithm outputs an optimal alignment of $S^1[l_1]$ and $S^2[l_2..r_2]$. If the character $c = S^1[l_1]$ occurs in $S^2[l_2..r_2]$, then the optimal alignment—for an operation-weighted distance function—consists of a match and $r_2 - l_2$ insertions. Otherwise, the optimal alignment consists of a mismatch and $r_2 - l_2$ insertions.

We claim that Hirschberg's algorithm requires only $O(n_2)$ space. Row $n_1/2$ of the dynamic programming matrix D can be calculated in $O(n_2)$ space. Of course, the same is true for D^{rev} . Therefore, the number k can be computed in $O(n_2)$ space. Once k is known, the remaining two rows of the dynamic programming matrices are obsolete. The recursive calls $Hb(l_1, m, l_2, k)$ and $Hb(m + 1, r_1, k + 1, r_2)$ use $O(k)$ space and $O(n_2 - k)$ space, respectively. So $O(n_2)$ space is used in the recursion. Hence the claim follows.

Let $T_{\min}(n_1, n_2)$ denote the overall number of minimum calculations when Hirschberg's algorithm is applied to two strings of lengths n_1 and n_2 . We will show that $T_{\min}(n_1, n_2)$ is in $O(n_1 n_2)$. It is not difficult to see that the total running time is proportional to $T_{\min}(n_1, n_2)$, so it is also in $O(n_1 n_2)$. In the computation of row $n_1/2$ in the dynamic programming matrix D , we have to calculate $n_1 n_2 / 2$ times the minimum of three numbers. The

Algorithm 8.1 Hirschberg's algorithm

```

Hb( $l_1, r_1, l_2, r_2$ )
  if  $l_2 > r_2$  then          /* base case */
    output an optimal alignment of  $S^1[l_1..r_1]$  and  $\varepsilon$ 
  else if  $l_1 = r_1$  then      /* base case */
    output an optimal alignment of  $S^1[l_1]$  and  $S^2[l_2..r_2]$ 
  else
     $m \leftarrow \lfloor \frac{r_1 - l_1}{2} \rfloor$ 
     $m' \leftarrow \lceil \frac{r_1 - l_1}{2} \rceil$ 
    compute the  $m$ -th row of the DP-matrix  $D$  of  $S^1[l_1..r_1]$  and  $S^2[l_2..r_2]$ 
    compute the  $m'$ -th row of the DP-matrix  $D^{rev}$  of  $S_{rev}^1[l_1..r_1]$  and  $S_{rev}^2[l_2..r_2]$ 
     $r_{len} \leftarrow r_2 - l_2 + 1$  /* current row length */
    determine  $k \in \arg \min_{0 \leq j \leq r_{len}} \{D(m, j) + D^{rev}(m', r_{len} - j)\}$ 
    Hb( $l_1, m, l_2, k$ )
    Hb( $m + 1, r_1, k + 1, r_2$ )

```

same is true for the computation of row $n_1/2$ in the dynamic programming matrix D^{rev} . To determine k we must take n_2 times the minimum of two numbers. This gives the following recurrence:

$$T_{\min}(n_1, n_2) = n_1 n_2 + n_2 + T_{\min}(\frac{n_1}{2}, k) + T_{\min}(\frac{n_1}{2}, n_2 - k)$$

Lemma 8.1.13

$$T_{\min}(n_1, n_2) \leq 2n_1 n_2 + n_2 \log_2 n_1$$

Proof By induction on n_1 . In the base case $n_1 = 1$, the lemma holds. For $n_1 > 1$, we have

$$\begin{aligned}
T_{\min}(n_1, n_2) &= n_1 n_2 + n_2 + T_{\min}(\frac{n_1}{2}, k) + T_{\min}(\frac{n_1}{2}, n_2 - k) \\
&\stackrel{I.H.}{\leq} n_1 n_2 + n_2 + 2\frac{n_1}{2}k + k \log_2 \frac{n_1}{2} + 2\frac{n_1}{2}(n_2 - k) + (n_2 - k) \log_2 \frac{n_1}{2} \\
&= n_1 n_2 + n_2 + n_1(k + n_2 - k) + (k + n_2 - k) \log_2 \frac{n_1}{2} \\
&= 2n_1 n_2 + n_2 + n_2(\log_2 n_1 - \log_2 2) \\
&= 2n_1 n_2 + n_2 + n_2(\log_2 n_1 - 1) \\
&= 2n_1 n_2 + n_2 \log_2 n_1
\end{aligned}$$

□

Theorem 8.1.14 An optimal alignment of two strings of lengths n_1 and n_2 can be computed in $O(n_1 n_2)$ time and $O(\min\{n_1, n_2\})$ space.

Proof This is the bottom line of the considerations in this section. □

8.1.3 Edit distance

Next, we show that the edit distance between two strings S^1 and S^2 coincides with the Levenshtein costs of an optimal alignment between S^1 and S^2 . Although this fact is almost obvious, it requires a proof.

We recall from Definition 7.9.4 that the edit distance between two strings S^1 and S^2 is the minimum number of edit operations (substitutions, insertions, and deletions) needed to transform S^1 into S^2 . Formally,

$$\text{edist}(S^1, S^2) = \min\{k \mid S^1 \rightarrow^k S^2\}$$

where $S^1 \rightarrow^k S^2$ means that S^1 can be transformed into S^2 by a sequence of $k \in \mathbb{N}$ edit operations.

Theorem 8.1.15 *We have*

$$\text{edist}(S^1, S^2) = \delta(A^{\text{opt}})$$

where A^{opt} is an optimal alignment of the two strings S^1 and S^2 for the Levenshtein cost function δ .

Proof We show (a) $\text{edist}(S^1, S^2) \leq \delta(A^{\text{opt}})$ and (b) $\text{edist}(S^1, S^2) \geq \delta(A^{\text{opt}})$.

(a) In an optimal alignment

$$A^{\text{opt}} = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ y_1 & y_2 & y_3 & \dots & y_{n-1} & y_n \end{pmatrix}$$

let $i_1 < i_2 < \dots < i_m$ be the indices of all columns so that

$$\begin{pmatrix} x_{i_j} \\ y_{i_j} \end{pmatrix}$$

is not a match. Obviously, these columns can be viewed as a sequence of edit operations transforming S^1 into S^2 :

- If $x_{i_j}, y_{i_j} \in \Sigma$, then the j -th edit operation is a substitution.
- If $x_{i_j} = -$ and $y_{i_j} \in \Sigma$, then the j -th edit operation is an insertion.
- If $x_{i_j} \in \Sigma$ and $y_{i_j} = -$, then the j -th edit operation is a deletion.

So there is a sequence of m edit operations that transforms S^1 into S^2 . Since $\text{edist}(S^1, S^2)$ is the minimum number of edit operations needed to transform S^1 into S^2 , it follows that $\text{edist}(S^1, S^2) \leq m = \delta(A^{\text{opt}})$.

(b) We show by induction on k that for every sequence of k edit operations transforming S^1 into S^2 there is an alignment A of S^1 and S^2 with cost $\delta(A) \leq k$. The base case $k = 0$ obviously holds (because in this case $S^1 = S^2$). Under the inductive hypothesis that the claim is true for k , it

must be proven for $k+1$. We show the inductive step for the case in which the last edit operation in the sequence is a substitution. The other two cases can be proven analogously. Let $S^1 \rightarrow^k S \rightarrow S^2$ for some string $S \in \Sigma^*$ so that S^2 is obtained from S by replacing one occurrence of character $a \in \Sigma$ by character $b \in \Sigma$, where $a \neq b$. We must show that there is an alignment A of S^1 and S^2 with cost $\delta(A) \leq k+1$. By induction hypothesis, there is an alignment

$$A' = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ y_1 & y_2 & y_3 & \dots & y_{n-1} & y_n \end{pmatrix}$$

of S^1 and S so that $\delta(A') \leq k$. In that alignment, suppose that the character a , which is replaced by b in the last edit operation, occurs in the j -th column, i.e.,

$$\begin{pmatrix} x_j \\ y_j \end{pmatrix} = \begin{pmatrix} x_j \\ a \end{pmatrix}$$

Let A be the alignment obtained from A' by replacing the j -th column with

$$\begin{pmatrix} x_j \\ b \end{pmatrix}$$

Clearly, A is an alignment of S^1 and S^2 . If $x_j = a$, then $\delta(A) = \delta(A') + 1 \leq k+1$. If $x_j \neq a$, then $\delta(A) \leq \delta(A') \leq k$. In both cases, $\delta(A) \leq k+1$. \square

8.1.4 Similarity methods

The basic definitions needed for similarity methods are analogous to those for distance methods.

Definition 8.1.16 An operation-weighted *similarity function* σ assigns similarity scores to each pair $(x, y) \in (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \setminus \{(-, -)\}$ as follows:

- $\sigma(x, y) = \alpha$ if $x = y$ (match),
- $\sigma(x, y) = -\beta$ if $x, y \in \Sigma$ with $x \neq y$ (substitution),
- $\sigma(x, y) = -\gamma$ if $x = -$ and $y \in \Sigma$ or $x \in \Sigma$ and $y = -$ (indel),

where α , β , and γ are positive constants.

Consequently, a similarity function σ rewards matches and penalizes mismatches and indels. For nucleotide sequences, the similarity function with $\alpha = 1$, $\beta = 1$, and $\gamma = 2$ is used very often.

$$\begin{pmatrix} g & g & g & a & a & t & t & - & - \\ - & - & - & a & a & t & t & c & c \end{pmatrix} \qquad \begin{pmatrix} g & g & g & a & a & t & t \\ a & a & t & t & c & c & c \end{pmatrix}$$

Figure 8.8: Two alignments of the strings $S^1 = g g g a a t t$ and $S^2 = a a t t c c c$.

Definition 8.1.17 The *similarity score* of an alignment

$$A = \begin{pmatrix} x_1 & x_2 & x_3 & \dots & x_{n-1} & x_n \\ y_1 & y_2 & y_3 & \dots & y_{n-1} & y_n \end{pmatrix}$$

of two strings S^1 and S^2 for a similarity function σ is

$$\sigma(A) = \sum_{i=1}^n \sigma(x_i, y_i)$$

An alignment A^{opt} of S^1 and S^2 is *optimal* for a similarity function σ if

$$\sigma(A^{opt}) = \max\{\sigma(A) \mid A \text{ is an alignment of } S^1 \text{ and } S^2\}$$

For example, the similarity score of the first alignment in Figure 8.8 for the similarity function defined by $\sigma(match) = 1$, $\sigma(sub) = -1$, and $\sigma(indel) = -2$ is -8 , while the similarity score of the second alignment is -7 .

Definition 8.1.18 Given a similarity function σ and two strings S^1 and S^2 of length n_1 and n_2 , respectively, we define for $0 \leq i \leq n_1$ and $0 \leq j \leq n_2$:

$$D(i, j) = \max\{\sigma(A) \mid A \text{ is an alignment of } S^1[1..i] \text{ and } S^2[1..j]\}$$

Theorem 8.1.19 For a similarity function σ with $\sigma(match) = \alpha$, $\sigma(sub) = -\beta$, and $\sigma(indel) = -\gamma$, we have

$$\begin{aligned} D(0, 0) &= 0 \\ D(i, 0) &= -i \cdot \gamma \\ D(0, j) &= -j \cdot \gamma \\ D(i, j) &= \max \left\{ \begin{array}{l} D(i-1, j) - \gamma \\ D(i, j-1) - \gamma \\ D(i-1, j-1) + \sigma(S^1[i], S^2[j]) \end{array} \right\} \end{aligned}$$

for each $1 < i \leq n_1$ and $1 < j \leq n_2$, where $\sigma(S^1[i], S^2[j]) = \alpha$ if $S^1[i] = S^2[j]$ and $\sigma(S^1[i], S^2[j]) = -\beta$ if $S^1[i] \neq S^2[j]$.

Proof The proof is verbatim the same as that of Theorem 8.1.7. \square

It is quite natural to ask whether an alignment that is optimal for the similarity function defined by $\sigma(match) = 1$, $\sigma(sub) = -1$, and $\sigma(indel) = -2$ is also optimal for Levenshtein costs (and vice versa). The example in Figure 8.8 refutes this; see Exercise 8.1.20.

Exercise 8.1.20 Let δ be the Levenshtein cost function and let the similarity function σ be defined by $\sigma(\text{match}) = 1$, $\sigma(\text{sub}) = -1$, and $\sigma(\text{indel}) = -2$. Show that the first alignment in Figure 8.8 is optimal for δ but not for σ , and that the second alignment in Figure 8.8 is optimal for σ but not for δ .

8.1.5 Distance vs. similarity

The result in this section goes back to the work of Smith et al. [294]; see also [289]. It holds for global alignments (but not for local alignments).

Lemma 8.1.21 Suppose that the alignment A of S^1 and S^2 consists of m matches, r substitutions, and d indels. Then

$$n_1 + n_2 = 2m + 2r + d$$

Furthermore, the cost of A for an operation-weighted cost function δ is

$$\delta(A) = r \cdot \delta(\text{sub}) + d \cdot \delta(\text{indel})$$

and the score of A for an operation-weighted similarity function σ is

$$\sigma(A) = m \cdot \sigma(\text{match}) + r \cdot \sigma(\text{sub}) + d \cdot \sigma(\text{indel})$$

Proof Straightforward. □

Definition 8.1.22 Let δ be an operation-weighted cost function. A corresponding similarity function σ satisfies

$$\begin{aligned} \sigma(\text{match}) &= \alpha \\ \sigma(\text{sub}) &= \alpha - k \cdot \delta(\text{sub}) \\ \sigma(\text{indel}) &= \frac{\alpha}{2} - k \cdot \delta(\text{indel}) \end{aligned}$$

where $\alpha > 0$ and $k > \frac{\alpha}{\delta(\text{sub})}$ are constants.

As an example, let δ be the Levenshtein cost function, i.e., $\delta(\text{match}) = 0$, $\delta(\text{sub}) = 1$, and $\delta(\text{indel}) = 1$. For $\alpha = 1$ and $k = 2$, the corresponding similarity function is defined by $\sigma(\text{match}) = 1$, $\sigma(\text{sub}) = -1$, and $\sigma(\text{indel}) = -1.5$.

Theorem 8.1.23 Let δ be a cost function and σ be a corresponding similarity function. An alignment has minimum cost for δ if and only if it has maximum score for σ .

Proof For any alignment A , we have

$$\begin{aligned} \sigma(A) &= m \cdot \sigma(\text{match}) + r \cdot \sigma(\text{sub}) + d \cdot \sigma(\text{indel}) \\ &= m \cdot \alpha + r \cdot (\alpha - k \cdot \delta(\text{sub})) + d \cdot \left(\frac{\alpha}{2} - k \cdot \delta(\text{indel})\right) \\ &= \left(m + r + \frac{d}{2}\right) \cdot \alpha - k \cdot (r \cdot \delta(\text{sub}) + d \cdot \delta(\text{indel})) \\ &= \frac{n_1 + n_2}{2} \cdot \alpha - k \cdot \delta(A) \end{aligned}$$

σ	A	C	G	T
A	1	-1	-0.5	-1
C	-1	1	-1	-0.5
G	-0.5	-1	1	-1
T	-1	-0.5	-1	1

Figure 8.9: Transversion/transition similarity function.

For two alignments A and A' of the same strings, it then follows

$$\begin{aligned}
 \sigma(A) \geq \sigma(A') &\Leftrightarrow \frac{n_1+n_2}{2} \cdot \alpha - k \cdot \delta(A) \geq \frac{n_1+n_2}{2} \cdot \alpha - k \cdot \delta(A') \\
 &\Leftrightarrow -k \cdot \delta(A) \geq -k \cdot \delta(A') \\
 &\Leftrightarrow \delta(A) \leq \delta(A')
 \end{aligned}$$

because $-k < 0$. In other words, an alignment A has maximum score for σ if and only if it has minimum cost for δ . \square

Exercise 8.1.24 Let σ be a similarity function with $\sigma(\text{match}) = \alpha > 0$, $\sigma(\text{sub}) = -\beta < 0$, and $\sigma(\text{indel}) = -\gamma < 0$. Define a corresponding cost function δ by $\delta(\text{match}) = 0$, $\delta(\text{sub}) = c(\alpha + \beta)$, and $\delta(\text{indel}) = c(\frac{1}{2}\alpha + \gamma)$, where $c > 0$. (For example, let $\alpha = 1$, $\beta = 1$, and $\gamma = 2$. For $c = \frac{1}{2}$ the corresponding cost function δ is defined by $\delta(\text{match}) = 0$, $\delta(\text{sub}) = 1$, and $\delta(\text{indel}) = 1.25$.) Show that an alignment has maximum score for σ if and only if it has minimum cost for δ .

8.1.6 General similarity functions and gap penalties

Up to now, a similarity function scored each substitution equally, i.e., $\sigma(x, y) = -\beta$ for each $x, y \in \Sigma$ and $x \neq y$. For DNA sequences, this means that aligning A (adenine) with G (guanine) is just as bad as aligning A with T (thymine). However, adenine and guanine are purines, whereas cytosine and thymine are pyrimidines, and studies of mutations in homologous genes indicate that transition mutations (i.e., purine/purine or pyrimidine/pyrimidine substitutions) occur approximately twice as frequently as transversions (purine/pyrimidine substitutions). In the comparison of genes, this fact can be taken into account by a similarity function that penalizes transitions less than transversions; see Figure 8.9.

In sequences of amino acid residues (the primary structure of proteins), a substitution is more likely to occur between amino acids with similar biochemical properties, and a reasonable similarity function should reflect this. For example, the replacement of the hydrophobic amino acid isoleucine (I) with the hydrophobic amino acid valine (V) should get a positive score, while its replacement with the hydrophilic amino acid cystine (C) should get a negative score. Thus, a similarity function is based on

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	2																			
R	-2	6																		
N	0	0	2																	
D	0	-1	2	4																
C	-2	-4	-4	-5	12															
Q	0	1	1	2	-5	4														
E	0	-1	1	3	-5	2	4													
G	1	-3	0	1	-3	-1	0	5												
H	-1	2	2	1	-3	3	1	-2	6											
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5										
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6									
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5								
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6							
F	-4	-4	-4	-6	-4	-5	-5	-5	-2	1	2	-5	0	9						
P	1	0	-1	-1	-3	0	-1	-1	0	-2	-3	-1	-2	-5	6					
S	1	0	1	0	0	-1	0	1	-1	-3	0	-2	-3	1	0	2				
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3			
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17		
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4

Figure 8.10: PAM250 matrix.

an estimation of how well two residues of given types would match if they were aligned in a sequence alignment. Even identities (matches) in an alignment of amino acid sequences should be scored differently. For example, the amino acid tryptophan (W)—a relatively rare amino acid—has bulky side groups and cannot be inserted easily into any site in a peptide. Thus, a replacement of tryptophan with another amino acid is relatively rare, and a similarity function should assign a high positive score to a tryptophan identity. By contrast, an amino acid like alanine (A) can often be replaced with another biochemically similar amino acid, so an alanine identity should get a low positive score.

There have been extensive studies examining the frequencies in which amino acids substituted for each other during evolution. The studies involved carefully aligning all of the proteins in several families of proteins and then constructing phylogenetic trees for each family. Each phylogenetic tree can then be examined for the substitutions found on each branch. This can be used to produce tables (substitution matrices, also called scoring matrices) of the relative frequencies with which amino acids replace each other over a short evolutionary period. Thus a substitution matrix describes the likelihood that two residue types would mutate to each other in evolutionary time. Prime examples are the PAM matrices (PAM is an acronym of Point Accepted Mutations) devised by Dayhoff et al. [75] (Figure 8.10 shows the PAM250 matrix) and the BLOSUM matrices (BLOSUM stands for BLOcks SUBstitution Matrix) developed by Henikoff and Henikoff [149] (Figure 8.11 shows the BLOSUM62 matrix).

A second issue relates to the scoring of gaps. A gap is a maximal consecutive run of spaces in a single row of a given alignment. It corresponds to an atomic insertion or deletion of a substring. Up to now, we have scored a gap of length k as

$$g(k) = -k\gamma$$

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	4																			
R	-1	5																		
N	-2	0	6																	
D	-2	-2	1	6																
C	0	-3	-3	-3	9															
Q	-1	1	0	0	-3	5														
E	-1	0	0	2	-4	2	5													
G	0	-2	0	-1	-3	-2	-2	6												
H	-2	0	1	-1	-3	0	0	-2	8											
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4										
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4									
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5								
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5							
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6						
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7					
S	1	-1	1	0	-1	0	0	-1	-2	-2	-2	0	-1	-2	-1	4				
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-2	-1	1	5				
W	-3	-3	-4	-4	-2	-2	-3	-2	-3	-2	-3	-1	1	-4	-3	-2	11			
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4

Figure 8.11: BLOSUM62 matrix.

where γ is a positive constant. It follows as a consequence that a gap of length k in an alignment causes the same penalty as k interspersed indels. However, in many biological applications, it does not make sense to penalize gaps in this manner. To cope with this problem, one can use more general gap penalty functions than the linear function above.

Definition 8.1.25 A function $g : \mathbb{N} \rightarrow \mathbb{R}$ with $g(0) = 0$ and $g(k) \geq 0$ is called *gap penalty function* if it is subadditive, i.e., for all $k, l \in \mathbb{N}$

$$g(k + l) \leq g(k) + g(l)$$

Theorem 8.1.26 An optimal alignment between the two strings S^1 and S^2 for a similarity function σ and a gap penalty function g can be obtained by the recurrences

$$\begin{aligned}
 D(0, 0) &= 0 \\
 D(i, 0) &= -g(i) \\
 D(0, j) &= -g(j) \\
 D(i, j) &= \max \left\{ \begin{array}{l} \max_{1 \leq k \leq i} \{D(i - k, j) - g(k)\} \\ \max_{1 \leq k \leq j} \{D(i, j - k) - g(k)\} \\ D(i - 1, j - 1) + \sigma(S^1[i], S^2[j]) \end{array} \right\}
 \end{aligned}$$

and traceback.

Proof See Waterman et al. [327]. □

The previous dynamic programming algorithms had a time complexity of $O(n_1 n_2)$, i.e., the run time is quadratic if $n_1 = n_2$. In sharp contrast, the algorithm incorporating arbitrary gap penalties has a worst-case time complexity of $O(n_1 n_2 (n_1 + n_2))$, i.e., the run time is cubic if $n_1 = n_2$. For the important class of affine gap penalties, however, an $O(n_1 n_2)$ time algorithm exists, as we shall see next.

Definition 8.1.27 An affine gap penalty function g satisfies $g(0) = 0$ and

$$g(k) = a + b(k - 1)$$

for $k > 0$, where a and b are constants with $0 \leq b \leq a$. These constants a and b are called *gap-open penalty* and *gap-extension penalty*, respectively.

Note that an affine gap penalty is indeed subadditive:

$$g(k + l) = a + b(k + l - 1) = g(k) + bl = g(k) + b + b(l - 1) \leq g(k) + g(l)$$

because $b \leq a$.

Exercise 8.1.28 provides an alternative way to define affine gap penalty functions, one that is used by many authors.

Exercise 8.1.28 Show that the definition $g'(0) = 0$ and $g'(k) = a' + b'k$ for $k > 0$ and constants $0 \leq b' \leq a'$ yields an affine gap penalty function g' .

Corollary 8.1.29 The dynamic programming recurrences to compute an optimal alignment for a similarity function σ and an affine gap penalty function g read as follows:

$$\begin{aligned} D(0, 0) &= 0 \\ D(0, j) &= -a - b(j - 1) \text{ for } j \geq 1 \\ D(i, 0) &= -a - b(i - 1) \text{ for } i \geq 1 \\ D(i, j) &= \max \left\{ \begin{array}{l} \max_{1 \leq k \leq i} \{D(i - k, j) - (a + b(k - 1))\} \\ \max_{1 \leq k \leq j} \{D(i, j - k) - (a + b(k - 1))\} \\ D(i - 1, j - 1) + \sigma(S^1[i], S^2[j]) \end{array} \right\} \end{aligned}$$

Proof Immediate consequence of Theorem 8.1.26. □

The main idea of Gotoh's [130] $O(n_1 n_2)$ time dynamic programming algorithm is to use three matrices instead of a single one.

Theorem 8.1.30 Let the matrix D be defined as in Corollary 8.1.29. Then, for all $1 \leq i \leq n_1$ and $1 \leq j \leq n_2$, we have

$$D(i, j) = \max\{E(i, j), F(i, j), D(i - 1, j - 1) + \sigma(S^1[i], S^2[j])\}$$

where

$$\begin{aligned} E(0, j) &= -\infty \\ E(i, j) &= \max\{D(i - 1, j) - a, E(i - 1, j) - b\} \end{aligned}$$

and

$$\begin{aligned} F(i, 0) &= -\infty \\ F(i, j) &= \max\{D(i, j - 1) - a, F(i, j - 1) - b\} \end{aligned}$$

Proof Define

$$E(i, j) = \max_{1 \leq k \leq i} \{D(i - k, j) - (a + b(k - 1))\}$$

and

$$F(i, j) = \max_{1 \leq k \leq j} \{D(i, j - k) - (a + b(k - 1))\}$$

According to Corollary 8.1.29, it suffices to show that

$$E(i, j) = \max\{D(i - 1, j) - a, E(i - 1, j) - b\}$$

and

$$F(i, j) = \max\{D(i, j - 1) - a, F(i, j - 1) - b\}$$

We prove the second equality by finite induction on j . The first equality can be proven analogously by finite induction on i . The base case $j = 1$ yields

$$\begin{aligned} F(i, 1) &= \max_{1 \leq k \leq 1} \{D(i, 1 - k) - (a + b(k - 1))\} \\ &= \max\{D(i, 0) - a\} \\ &= \max\{D(i, 0) - a, -\infty\} \\ &= \max\{D(i, 0) - a, F(i, 0) - b\} \end{aligned}$$

For $j \geq 2$, we have

$$\begin{aligned} F(i, j) &= \max_{1 \leq k \leq j} \{D(i, j - k) - (a + b(k - 1))\} \\ &= \max\{D(i, j - 1) - a, \max_{2 \leq k \leq j} \{D(i, j - k) - (a + b(k - 1))\}\} \\ &= \max\{D(i, j - 1) - a, \max_{1 \leq k \leq j-1} \{D(i, j - k - 1) - (a + b(k))\}\} \\ &= \max\{D(i, j - 1) - a, \max_{1 \leq k \leq j-1} \{D(i, j - k - 1) - (a + b(k - 1))\} - b\} \\ &= \max\{D(i, j - 1) - a, F(i, j - 1) - b\} \end{aligned}$$

where the last equality is a consequence of the inductive hypothesis. \square

Because an entry in the D matrix (E or F matrix, respectively) is the maximum of three (two, respectively) numbers that have already been computed, each entry can be computed in constant time. Thus, Gotoh's dynamic programming algorithm has a time complexity of only $O(n_1 n_2)$.

8.2 Multiple alignment

A multiple sequence alignment (MSA) is a sequence alignment of three or more biological sequences (amino acid-, DNA-, or RNA-sequences). In many cases, the sequences under consideration are assumed to have an

Sequence	Species	Common name	Acronym
$S^1 = AGTAATGG$	<i>Solea vulgaris</i>	Common sole	SoV
$S^2 = TTTAATGA$	<i>Solea lascaris</i>	Sand sole	SoL
$S^3 = AAGAAATGG$	<i>Monochirus hispidus</i>	Whiskered sole	MoH
$S^4 = ATAAAATGG$	<i>Microchirus ocellatus</i>	Foureyed sole	MiO

Table 8.1: Subsequences from a mtDNA gene (coding for 16S rRNA) of four Mediterranean sole species.

evolutionary relationship by which they share a lineage and are descended from a common ancestor. From the resulting multiple sequence alignment, sequence homology can be inferred and phylogenetic analysis can be conducted to assess the sequences' shared evolutionary origins.

Again, we focus on global alignments.

Definition 8.2.1 A global *multiple alignment* of m strings S^1, \dots, S^m (on the alphabet Σ) is an $(m \times N)$ matrix A so that:

1. $A(i, j) \in \Sigma \cup \{-\}$, where $-$ is a special gap symbol not occurring in Σ .
2. After removal of all gap symbols the k -th row of A equals S^k .
3. No column of A consists solely of gap symbols.

This definition imposes restrictions on the length N of A . Condition (2) implies that $\max_{1 \leq k \leq m} \{n_k\} \leq N$, while condition (3) has $N \leq \sum_{k=1}^m n_k$ as a consequence.

As an example, we consider the following small part of a multiple alignment of the DNA sequences of the 16S rRNA genes of the mitochondrial genomes of four Mediterranean sole species; see Table 8.1 and Tinti et al. [310] for more details.

$$A^{sole} = \begin{pmatrix} A & - & G & T & A & A & T & G & G \\ T & T & - & T & A & A & T & G & A \\ A & A & G & A & A & A & T & G & G \\ A & T & A & A & A & A & T & G & G \end{pmatrix}$$

To assess the quality of a multiple alignment, we need to *score* the alignment. This score is usually based on pairwise alignment scoring schemes like Levenshtein costs (a dissimilarity scoring scheme used e.g. for DNA sequences) or PAM and BLOSUM substitution matrices (a similarity scoring scheme for amino acid sequences). Here we confine ourselves to the so-called sum-of-pairs score. In what follows, let $\pi_{i,j}(A)$ be the projection to the i -th and j -th row of the alignment A . For example,

$$\pi_{1,2}(A^{sole}) = \begin{pmatrix} A & - & G & T & A & A & T & G & G \\ T & T & - & T & A & A & T & G & A \end{pmatrix}$$

Definition 8.2.2 Let A be a multiple alignment of the strings S^1, \dots, S^m . Given a pairwise alignment scoring scheme $score$, the *sum-of-pairs score* of A is defined by

$$score_{SP}(A) = \sum_{1 \leq i < j \leq m} score(\pi_{i,j}(A))$$

where $score(-, -) = 0$.

In our example, we use the Levenshtein cost function δ as a pairwise alignment scoring scheme and obtain $\delta(\pi_{1,2}(A^{sole})) = 4$, $\delta(\pi_{1,3}(A^{sole})) = 2$, $\delta(\pi_{1,4}(A^{sole})) = 3$, $\delta(\pi_{2,3}(A^{sole})) = 5$, $\delta(\pi_{2,4}(A^{sole})) = 4$, and $\delta(\pi_{3,4}(A^{sole})) = 2$. Thus, the sum-of-pairs score of A^{sole} is 20.

Exercise 8.2.3 Compute the sum-of-pairs score of the following multiple alignment of the amino acid sequences $S^1 = \text{NFLS}$, $S^2 = \text{NFS}$, $S^3 = \text{NKYLS}$, and $S^4 = \text{NYLS}$. Use the PAM250 matrix and score each indel with -8 .

$$A^{prot} = \begin{pmatrix} \text{N} & - & \text{F} & \text{L} & \text{S} \\ \text{N} & - & \text{F} & - & \text{S} \\ \text{N} & \text{K} & \text{Y} & \text{L} & \text{S} \\ \text{N} & - & \text{Y} & \text{L} & \text{S} \end{pmatrix}$$

Definition 8.2.4 Given strings S^1, \dots, S^m and a cost function δ (similarity function σ , respectively), the *global multiple alignment problem* is to compute a global multiple alignment of S^1, \dots, S^m that has minimum (maximum, respectively) *sum-of-pairs score* for δ (σ , respectively).

In what follows, we only use cost functions as scoring schemes, and the notation δ_{SP} and δ (instead of $score_{SP}$ and $score$) will emphasize this.

It is possible to generalize the dynamic programming recurrences for pairwise alignments to multiple alignments. For $m = 3$ sequences and $i_1 \neq 0$, $i_2 \neq 0$, $i_3 \neq 0$, they have the following form:

$$D(i_1, i_2, i_3) = \min \begin{cases} D(i_1 - 1, i_2 - 1, i_3 - 1) + \delta_{SP}(S^1[i_1], S^2[i_2], S^3[i_3]) \\ D(i_1 - 1, i_2 - 1, i_3) + \delta_{SP}(S^1[i_1], S^2[i_2], -) \\ D(i_1, i_2 - 1, i_3 - 1) + \delta_{SP}(-, S^2[i_2], S^3[i_3]) \\ D(i_1 - 1, i_2, i_3 - 1) + \delta_{SP}(S^1[i_1], -, S^3[i_3]) \\ D(i_1 - 1, i_2, i_3) + \delta_{SP}(S^1[i_1], -, -) \\ D(i_1, i_2 - 1, i_3) + \delta_{SP}(-, S^2[i_2], -) \\ D(i_1, i_2, i_3 - 1) + \delta_{SP}(-, -, S^3[i_3]) \end{cases}$$

It is possible to simplify the notation by introducing Δ_k , which is 0 or 1, and defining

$$\Delta_k \circ c = \begin{cases} c & \text{if } \Delta_k = 1 \\ - & \text{if } \Delta_k = 0 \end{cases}$$

Then, $D(i_1, i_2, i_3)$ can be computed by

$$\min_{\Delta_1 + \Delta_2 + \Delta_3 > 0} D(i_1 - \Delta_1, i_2 - \Delta_2, i_3 - \Delta_3) + \delta_{SP}(\Delta_1 \circ S^1[i_1], \Delta_2 \circ S^2[i_2], \Delta_3 \circ S^3[i_3])$$

For an arbitrary number m of strings, the recurrence relation is

$$D(i_1, \dots, i_m) = \min_{\Delta_1 + \dots + \Delta_m > 0} D(i_1 - \Delta_1, \dots, i_m - \Delta_m) + \delta_{SP}(\Delta_1 \circ S^1[i_1], \dots, \Delta_m \circ S^m[i_m])$$

Although the multiple alignment problem can be solved exactly via dynamic programming and traceback, this solution is not practical. This can be seen by the following complexity analysis. The algorithm must compute an $n_1 \times n_2 \times \dots \times n_m$ matrix, so the space complexity is $O(n_1 n_2 \dots n_m)$, which is $O(n^m)$ if n denotes the maximum sequence length. Each of the $O(n^m)$ entries in the matrix is computed by taking a maximum of $2^m - 1$ values. Moreover, the computation of each of the $2^m - 1$ values requires $O(m^2)$ time (one has to compute the sum-of-pairs score of an m -dimensional vector). So the overall time complexity is $O(m^2 \cdot 2^m \cdot n^m)$.

Even worse, the multiple alignment problem has been proven to be NP-complete [171, 325]. There are three possible ways out of this trap:

1. Try to find methods that prune the search space without sacrificing an optimal solution. In this way, larger instances of the problem can be exactly solved in reasonable time.
2. Try to devise an efficient approximation algorithm that computes an approximate solution that is optimal up to a small constant factor.
3. Try to develop heuristics that find reasonably good solutions reasonably fast, without guaranteeing optimality.

In Sections 8.2.1, 8.2.2, and 8.2.3 we describe a representative of each of these methods. It will be convenient to use the following notation:

$$dist_\delta(S^1, S^2) = \min\{\delta(A) \mid A \text{ is an alignment of } S^1 \text{ and } S^2\}$$

That is, given a cost function δ , $dist_\delta(S^1, S^2)$ denotes the cost of an optimal alignment between S^1 and S^2 (the “distance” between S^1 and S^2 w.r.t. δ).

8.2.1 Pruning the search space

In the following, let A^{opt} be an optimal alignment of the strings S^1, \dots, S^m , and let A^{heur} be an alignment obtained by a fast heuristic algorithm (for example by one of the alignment methods given in subsequent sections).

We have

$$\begin{aligned}
\delta(A^{heur}) &\geq \delta(A^{opt}) \\
&= \sum_{k < l} \delta(\pi_{k,l}(A^{opt})) \\
&= \delta(\pi_{p,q}(A^{opt})) + \sum_{k < l, (k,l) \neq (p,q)} \delta(\pi_{k,l}(A^{opt})) \\
&\geq \delta(\pi_{p,q}(A^{opt})) + \sum_{k < l, (k,l) \neq (p,q)} dist_{\delta}(S^k, S^l)
\end{aligned}$$

for every pair (p, q) , $1 \leq p < q \leq m$.

Consequently,

$$U_{p,q} = \delta(A^{heur}) - \sum_{k < l, (k,l) \neq (p,q)} dist_{\delta}(S^k, S^l)$$

is an upper bound for $\delta(\pi_{p,q}(A^{opt}))$, i.e., $\delta(\pi_{p,q}(A^{opt})) \leq U_{p,q}$.

For every pair (p, q) , define for $1 \leq i \leq n_p$ and $1 \leq j \leq n_q$

$$B_{p,q}(i, j) = dist_{\delta}(S^p[1..i], S^q[1..j]) + dist_{\delta}(S^p[i+1..n_p], S^q[j+1..n_q])$$

Thus, $B_{p,q}(i, j)$ is the minimum cost of all paths in the alignment graph of S^p and S^q that pass through the node (i, j) . Note that $B_{p,q}(i, j)$ can be computed in $O(n_p n_q)$ time for all $1 \leq i \leq n_p$ and $1 \leq j \leq n_q$ by computing the dynamic programming matrix $D_{p,q}$, which computes the distance of S^p and S^q in the standard “forward” direction, and the dynamic programming matrix $D_{p,q}^{rev}$, which computes the distance of S^p and S^q in the “backward” direction; see Section 8.1.2.

The method of Carillo and Lipman [51] uses the notions defined above to restrict the search space as follows: The m -dimensional dynamic programming algorithm computes values only for those nodes (i_1, i_2, \dots, i_m) in the m -dimensional alignment graph of S^1, \dots, S^m that satisfy $B_{p,q}(i_p, i_q) \leq U_{p,q}$ for all pairs (p, q) ; the corresponding entries in the m -dimensional dynamic programming matrix are marked in Figure 8.12.

Why can the other nodes safely be skipped? To understand this, suppose that the path in the alignment graph that corresponds to a multiple alignment A of S^1, \dots, S^m passes through a node (i_1, i_2, \dots, i_m) for which there is a pair (k, l) so that $B_{k,l}(i_k, i_l) > U_{k,l}$. It then follows from $\delta(\pi_{k,l}(A)) \geq B_{k,l}(i_k, i_l)$ that $\delta(\pi_{k,l}(A)) > U_{k,l}$. According to the preceding discussion, this means that the alignment A cannot be optimal.

One still has to find a path of minimum cost from node $(0, \dots, 0)$ to node (n_1, \dots, n_m) in the reduced m -dimensional alignment graph. This can e.g. be done by using Dijkstra's algorithm [61, 80] (a standard shortest path algorithm) or the so-called A^* -algorithm [200].

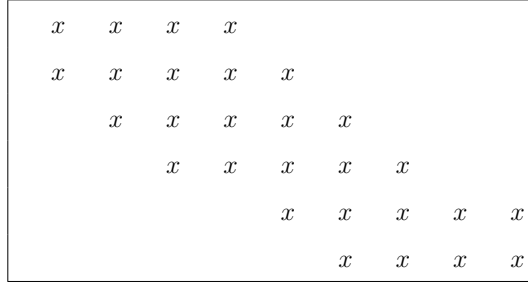


Figure 8.12: A 2-dimensional projection of the restricted m -dimensional dynamic programming matrix.

Carillo and Lipman's method is implemented in the program MSA. According to [205], MSA "is able to align in reasonable time as many as eight sequences the length of an average protein." Another implementation by Gupta et al. [137] uses a heuristic to further reduce the search space. They added an additional parameter $\epsilon_{p,q}$ to the program, and a node in the alignment graph is considered irrelevant when $B_{p,q}(i_p, i_q) \leq U_{p,q} - \epsilon_{p,q}$. Note that this heuristic does not guarantee an optimal alignment.

8.2.2 A 2-approximation algorithm

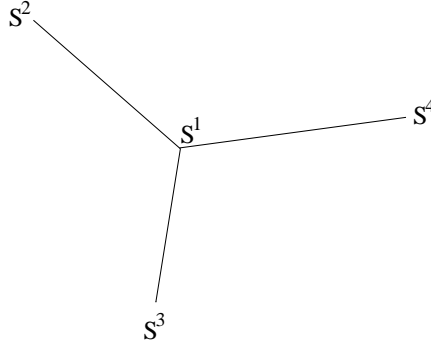
If the cost function δ is a distance (i.e., it satisfies the metric axioms), then the so-called *center star method* [138, 139] yields a simple 2-approximation algorithm for the multiple alignment problem as we shall now see. Given strings S^1, \dots, S^m , a *center string* S^c is one of the strings S^1, \dots, S^m that minimizes

$$\sum_{i=1}^m \text{dist}_{\delta}(S^c, S^i)$$

That is to say, S^c is a string whose distance to all the rest is minimum; see Figure 8.13. Then, the center star alignment A^c is constructed as a combination of the pairwise optimal alignments of the center string with the other strings.

To determine the center string S^c in our *sole* example, we compute all pairwise edit distances (i.e., we use Levenshtein costs).

dist_{δ}	S^1	S^2	S^3	S^4	\sum
S^1	0	3	2	3	8
S^2	3	0	5	4	12
S^3	2	5	0	2	9
S^4	3	4	2	0	9

Figure 8.13: S^1 is the center string of S^1, \dots, S^4 .

Taking the minimum of the sums of each row, we infer that $S^c = S^1$ is the center string. Optimal pairwise alignments of S^1 with the other sequences are:

$$A^{1,2} = \begin{pmatrix} A & G & T & A & A & T & G & G \\ T & T & T & A & A & T & G & A \end{pmatrix}$$

$$A^{1,3} = \begin{pmatrix} - & A & G & T & A & A & T & G & G \\ A & A & G & A & A & A & T & G & G \end{pmatrix}$$

$$A^{1,4} = \begin{pmatrix} A & - & G & T & A & A & T & G & G \\ A & T & A & A & A & A & T & G & G \end{pmatrix}$$

Then, the center star alignment A^c is constructed as a combination of the pairwise optimal alignments of the center string with the other strings. First, by combining $A^{1,2}$ and $A^{1,3}$, we obtain the multiple alignment

$$A^{1,2,3} = \begin{pmatrix} - & A & G & T & A & A & T & G & G \\ - & T & T & T & A & A & T & G & A \\ A & A & G & A & A & A & T & G & G \end{pmatrix}$$

Second, the combination of the multiple alignment $A^{1,2,3}$ with the pairwise alignment $A^{1,4}$ yields the center star alignment A^c . Note that entire columns must be shifted to incorporate the gap of the alignment $A^{1,4}$.

$$A^c = \begin{pmatrix} - & A & - & G & T & A & A & T & G & G \\ - & T & - & T & T & A & A & T & G & A \\ A & A & - & G & A & A & A & T & G & G \\ - & A & T & A & A & A & A & T & G & G \end{pmatrix}$$

The sum-of-pairs score of the center star alignment is $\delta(A^c) = 21$. This is not optimal because the sum-of-pairs score of the alignment A^{sole} is $\delta(A^{sole}) = 20$.

This example shows that the center star method does—in general—not yield an optimal alignment. However, Gusfield [138, 139] showed that

$$\delta_{SP}(A^c) \leq \frac{2(m-1)}{m} \delta_{SP}(A^{opt})$$

That is, the center star method is a $(2 - \frac{2}{m})$ -approximation algorithm for the multiple sequence alignment problem. In particular, even for large m , the sum-of-pairs score of the center star alignment A^c is at most twice the sum-of-pairs score of an optimal alignment.

In order to derive this guaranteed error bound, we use the next lemma; cf. [299].

Lemma 8.2.5 *Let S^c be a center string and A^{opt} be an optimal alignment of the strings S^1, \dots, S^m . Then*

$$\frac{m}{2} \sum_{i=1}^m \text{dist}_\delta(S^i, S^c) \leq \delta_{SP}(A^{opt})$$

Proof

$$\begin{aligned} \frac{m}{2} \sum_{i=1}^m \text{dist}_\delta(S^i, S^c) &= \frac{1}{2} \underbrace{\left(\sum_{i=1}^m \text{dist}_\delta(S^i, S^c) + \dots + \sum_{i=1}^m \text{dist}_\delta(S^i, S^c) \right)}_{m\text{-times}} \\ &\leq \frac{1}{2} \left(\sum_{i=1}^m \text{dist}_\delta(S^i, S^1) + \dots + \sum_{i=1}^m \text{dist}_\delta(S^i, S^m) \right) \\ &= \frac{1}{2} \sum_{j=1}^m \sum_{i=1}^m \text{dist}_\delta(S^i, S^j) \\ &= \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \text{dist}_\delta(S^i, S^j) \\ &\leq \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} \delta(\pi_{i,j}(A^{opt})) \\ &= \sum_{1 \leq i < j \leq m} \delta(\pi_{i,j}(A^{opt})) \\ &= \delta_{SP}(A^{opt}) \end{aligned}$$

□

Now we are in a position to prove the error bound of the center star method.

Theorem 8.2.6 Let A^{opt} be an optimal alignment of S^1, \dots, S^m . Furthermore, let S^c be a center string and A^c the corresponding alignment. Then

$$\delta_{SP}(A^c) \leq \frac{2(m-1)}{m} \delta_{SP}(A^{opt})$$

Proof

$$\begin{aligned}
\delta_{SP}(A^c) &= \sum_{1 \leq i < j \leq m} \delta(\pi_{i,j}(A^c)) \\
&= \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} \delta(\pi_{i,j}(A^c)) \\
&\leq \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} (\delta(\pi_{i,c}(A^c)) + \delta(\pi_{c,j}(A^c))) && \text{(triangle inequality)} \\
&= \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} (\delta(\pi_{i,c}(A^c)) + \delta(\pi_{j,c}(A^c))) && \text{(symmetry)} \\
&= \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} \delta(\pi_{i,c}(A^c)) + \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} \delta(\pi_{j,c}(A^c)) \\
&= \frac{1}{2} \sum_{i=1}^m \sum_{j \neq i} \delta(\pi_{i,c}(A^c)) + \frac{1}{2} \sum_{j=1}^m \sum_{i \neq j} \delta(\pi_{j,c}(A^c)) \\
&= \frac{1}{2} (m-1) \sum_{i=1}^m \delta(\pi_{i,c}(A^c)) + \frac{1}{2} (m-1) \sum_{j=1}^m \delta(\pi_{j,c}(A^c)) \\
&= (m-1) \sum_{i=1}^m \delta(\pi_{i,c}(A^c)) \\
&= (m-1) \sum_{i=1}^m \text{dist}_\delta(S^i, S^c) && \text{(center star align.)} \\
&\leq \frac{2(m-1)}{m} \delta_{SP}(A^{opt}) && \text{(Lemma 8.2.5)}
\end{aligned}$$

□

In the complexity analysis, we assume for ease of presentation that the sequences are roughly of the same length n . The computation of all $\binom{m}{2}$ pairwise distances takes $O(m^2 n^2)$ time and $O(m^2 + n)$ space (provided all $O(m^2)$ pairwise distances are stored). Once the center string S^c is identified, the $m-1$ pairwise alignments of S^c with the other strings can be computed in $O(mn^2)$ time and $O(mn)$ space by Hirschberg's algorithm (cf. Section 8.1.2) from the $m-1$ pairwise distances of S^c with the other strings. The combination of these pairwise alignments into a multiple alignment can be done in time proportional to the size of the multiple alignment,

which is $O(mn)$. To sum up, the overall time complexity is $O(m^2n^2)$, while the overall space complexity is $O(mn)$.

8.2.3 Progressive alignment

A progressive alignment method builds up a final multiple sequence alignment by combining pairwise alignments beginning with the most similar pair and progressing to the most distantly related. All progressive alignment methods require three stages:

1. For each pair of sequences S^i and S^j , $1 \leq i < j \leq m$, compute an “approximate” evolutionary distance.
2. Based on these pairwise distances, compute an “approximate” phylogenetic tree (guide tree).
3. Successively align the sequences in the order induced by the guide tree.

Feng and Doolittle were the first who coupled progressive alignment with the rule: “once a gap, always a gap.” In their seminal paper [98], they wrote:

It seems to us folly that a gap should be discarded in an alignment of two closely related sequences merely because an alignment with some distantly related sequence might be improved.

The three phases of progressive alignment can, for example, be implemented as follows:

- (1) For DNA sequences, one can use dissimilarity measures like a “normalized” edit distance or more sophisticated measures like the Jukes-Cantor distance [170] or Kimura’s two-parameter distance [181]; see e.g. [242].

For amino acid sequences, however, one usually uses a similarity measure. Feng and Doolittle [98] proposed the following conversion method: For each pair of sequences i and j , $1 \leq i < j \leq m$, compute

$$d_{i,j} = -\log \frac{S_{pair} - S_{rand}}{S_{aver} - S_{rand}}$$

where S_{pair} is the similarity score of the two sequences S^i and S^j , and S_{aver} is the average of the similarity scores obtained by aligning S^i with itself and S^j with itself. Furthermore, S_{rand} is the expected similarity score of two random sequences of the same length and residue composition (S_{rand} can be obtained by a simulation or by an approximate calculation).

- (2) Based on these pairwise distances $d_{i,j}$, Feng and Doolittle's method [98] computes a guide tree with the clustering algorithm of Fitch and Margoliash [112]. However, nowadays one would probably use the neighbor-joining method of Saitou and Nei [276] as a standard distance-based method for reconstructing phylogenetic trees; see Section 10.5.2. An application of Saitou and Nei's neighbor-joining algorithm to our *sole* example yields a tree in which SoV and SoL (sequences S^1 and S^2) as well as MoH and MiO (sequences S^3 and S^4) are neighbors; cf. [310].
- (3) In the third phase, one successively aligns the sequences in the order induced by the guide tree. In our example, we must first align S^1 with S^2 yielding the alignment $A^{1,2}$ from Section 8.2.2, and S^3 with S^4 yielding the alignment

$$A^{3,4} = \begin{pmatrix} A & A & G & A & A & A & T & G & G \\ A & T & A & A & A & A & T & G & G \end{pmatrix}$$

Then these two alignments must be combined into the final multiple alignment. Thus, it is a prerequisite of the third phase to have a method for aligning groups of sequences (alignment of alignments). According to Durbin et al. [83], this can be done as follows:

Sequence-sequence alignments are done with the usual pairwise dynamic programming algorithm. A sequence is added to an existing group by aligning it pairwise to each sequence in the group in turn. The best scoring pairwise alignment determines how the sequence will be aligned to the group. For aligning a group to a group, all sequence pairs between the two groups are tried; the best pairwise sequence alignment determines the alignment of the two groups.

To continue our example, the lowest distance between members of the two groups is $\text{dist}_\delta(S^1, S^3) = 2$ and an optimal alignment of S^1 and S^3 is e.g.

$$A^{1,3} = \begin{pmatrix} - & A & G & T & A & A & T & G & G \\ A & A & G & A & A & A & T & G & G \end{pmatrix}$$

The combined alignment of $A^{1,2}$ and $A^{3,4}$ by way of $A^{1,3}$ is

$$\begin{pmatrix} - & A & G & T & A & A & T & G & G \\ - & T & T & T & A & A & T & G & A \\ A & A & G & A & A & A & T & G & G \\ A & T & A & A & A & A & T & G & G \end{pmatrix}$$

Its sum-of-pairs score is 20.

One of the most prominent progressive multiple alignment programs is CLUSTALW [309]. It uses so-called *profiles* to represent intermediate alignments. More on profile alignment and the heuristics of CLUSTALW can be found e.g. in [83].

8.3 Whole genome alignment

Nowadays, it is quite common for a project to sequence the genome of an organism that is very closely related to another completed genome. For example, over 60 complete genomic sequences of *Escherichia* and *Shigella* species are available today. Most strains of these bacteria are harmless, but some are pathogenic. A global alignment of the (circular) chromosomes may help, for example, in understanding why a strain is pathogenic or resistant to antibiotics while another is not. In general, such a whole chromosome alignment makes sense only if the organisms under consideration are closely related, i.e., if no or only a few genome rearrangements have occurred; see Chapter 9. For diverged genomic sequences, however, a global alignment strategy is likely predestined to failure for having to align non-colinear and unrelated regions.

Several comparative sequence approaches are based on software-tools for aligning two or multiple genomic DNA sequences; see e.g. [223]. We will focus on comparing two genomes, but the method described in this section can be extended to the comparison of multiple genomes. To cope with the sheer volume of data, most of the software-tools use an anchor-based method that is composed of three phases:

1. computation of fragments (segments in the genomes that are similar),
2. computation of a highest-scoring global chain of colinear non-overlapping fragments: the anchors that form the basis of the alignment,
3. alignment of the regions between the anchors.

See e.g. [53, 313] for reviews about tools using this strategy.

Here, we discuss algorithms for solving the combinatorial chaining problem of the second phase: finding a highest-scoring global chain of colinear non-overlapping fragments. Roughly speaking, two fragments are *colinear*

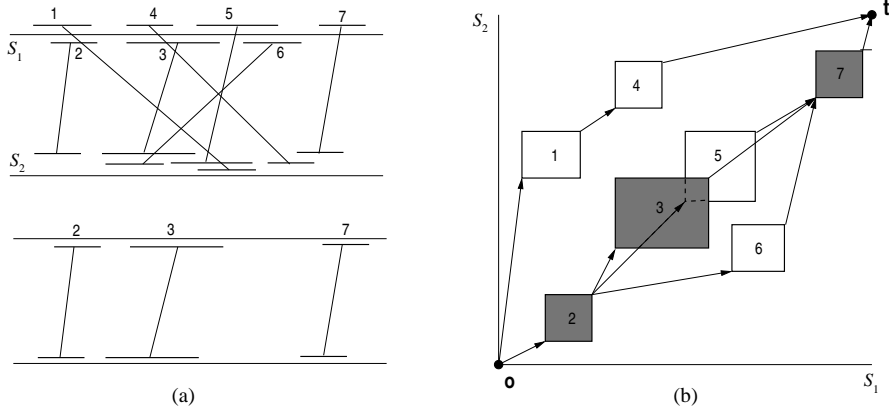


Figure 8.14: Given a set of fragments (upper left figure), an optimal global chain of colinear non-overlapping fragments (lower left figure) can be computed e.g. by computing an optimal path in the graph in (b) (in which not all edges are shown).

if the order of their respective segments is the same in both genomes. In the pictorial representation of Figure 8.14(a), two fragments are colinear if the lines connecting their segments are non-crossing (in Figure 8.14, for example, the fragments 2 and 3 are colinear, while 1 and 6 are not). Two fragments *overlap* if their segments overlap in one of the genomes (in Figure 8.14, for example, the fragments 1 and 2 are overlapping, while 2 and 3 are non-overlapping).

This section is organized as follows. Basic definitions and concepts are given in Section 8.3.1. In Section 8.3.2, we explain a global chaining algorithm that neglects gap costs, i.e., gaps between the fragments are not penalized. A special case is the algorithm presented in Section 8.3.4 that solves the heaviest increasing subsequence problem. The material presented here originates from [5], and we refer to [245] for

- incorporating gap costs into the algorithm,
- higher-dimensional chaining algorithms,
- variations of the algorithm (including local chaining), and
- a discussion of related work.

8.3.1 Basic definitions and concepts

In the following, S^1 and S^2 are two genomic sequences of lengths n_1 and n_2 . A *fragment* f consists of two pairs $beg(f) = (l_1, l_2)$ and $end(f) = (h_1, h_2)$ so that

the segments (substrings) $S^1[l_1 \dots h_1]$ and $S^2[l_2 \dots h_2]$ are “similar.” If the segments are exact matches, i.e., $S^1[l_1 \dots h_1] = S^2[l_2 \dots h_2]$, then we speak of *exact fragments*. Examples of exact fragments are maximal unique matches as used in MUMmer [77, 78], maximal exact matches as used in MGA [153], AVID [42], and CoCoNUT [3], rare maximal exact matches [252], or exact k -mers as used in GLASS [29]. In general, however, one may also allow substitutions (yielding fragments as in DIALIGN [225] and LAGAN [43]) or even insertions and deletions (as the BLASTZ-hits [287] that are used in PipMaker [288]). Each fragment f has a positive weight (denoted by $f.weight$) that can, for example, be the length of the fragment (in case of exact fragments) or its similarity score.

A fragment f can be represented by a rectangle in \mathbb{R}^2 with the lower left corner $beg(f)$ and the upper right corner $end(f)$, where each coordinate of the corner points is a non-negative integer. To fix notation, we recall the following concepts: For any point $p \in \mathbb{R}^2$, let $p.x_1$ and $p.x_2$ denote its coordinates. A rectangle, whose sides are parallel to the axes, is the Cartesian product of two intervals $[l_1 \dots h_1]$ and $[l_2 \dots h_2]$ on distinct coordinate axes, where $l_i < h_i$ for $1 \leq i \leq 2$. A rectangle $[l_1 \dots h_1] \times [l_2 \dots h_2]$ will also be denoted by $R(p, q)$, where $p = (l_1, l_2)$ and $q = (h_1, h_2)$ are the lower left and the upper right corner, respectively.

In what follows, we will often identify the point $beg(f)$ or $end(f)$ with the fragment f . This is possible because we assume that all fragments are known from the first phase of the anchor-based approach described earlier (so that every point can be annotated with a tag that identifies the fragment it stems from). For example, if we speak about the score of a point $beg(f)$ or $end(f)$, we mean the score of the fragment f . For ease of presentation, we consider the origin $0 = (0, 0)$ and the terminus $t = (n_1 + 1, n_2 + 1)$ as fragments with weight 0. For these fragments, we define $beg(0) = \perp$, $end(0) = 0$, $beg(t) = t$, and $end(t) = \perp$, where \perp stands for an undefined value.

Definition 8.3.1 We define a binary relation \ll on the set of fragments by $f \ll f'$ if and only if $end(f).x_i < beg(f').x_i$ for all i with $1 \leq i \leq 2$. If $f \ll f'$, then we say that f *precedes* f' .

Note that $0 \ll f \ll t$ for every fragment f with $f \neq 0$ and $f \neq t$.

Definition 8.3.2 A chain of colinear non-overlapping fragments (“chain” for short) is a sequence of fragments f_1, f_2, \dots, f_ℓ so that $f_i \ll f_{i+1}$ for all i with $1 \leq i < \ell$. The *score* of C is $score(C) = \sum_{i=1}^{\ell} f_i.weight - \sum_{i=1}^{\ell-1} g(f_{i+1}, f_i)$, where $g(f_{i+1}, f_i)$ is the cost (penalty) of connecting fragment f_i to f_{i+1} in the chain. We will call this cost *gap cost*.

Definition 8.3.3 Given m weighted fragments and a gap cost function, the *global fragment-chaining problem* is to determine a chain of highest

score (called *optimal global chain* in the following) starting at the origin 0 and ending at terminus t.

The global fragment-chaining problem was previously called the fragment alignment problem [90, 331]. A direct solution to this problem is to construct a weighted directed acyclic graph $G = (V, E)$, where the set V of vertices consists of all fragments (including 0 and t) and the set of edges E is characterized as follows: there is an edge $f \rightarrow f'$ with weight $f'.weight - g(f', f)$ if $f \ll f'$; see Figure 8.14(b). An optimal global chain of fragments corresponds to a path of maximum score from vertex 0 to vertex t in the graph. Because the graph is acyclic, such a path can be computed as follows: Let $f'.score$ be defined as the maximum score of all chains starting at 0 and ending at f' . $f'.score$ can be expressed by the recurrence: $0.score = 0$ and

$$f'.score = f'.weight + \max\{f.score - g(f', f) : f \ll f'\} \quad (8.1)$$

A dynamic programming algorithm based on this recurrence takes $O(|V| + |E|)$ time provided that gap costs can be computed in constant time. Because $|V| + |E| \in O(m^2)$, computing an optimal global chain takes quadratic time and linear space; see [61, 199]. This graph-based solution works for any number of genomes and for any kind of gap cost. It has been proposed as a practical approach for aligning biological sequences, first for two sequences by Wilbur and Lipman [331] and for multiple sequences by Sobel and Martinez [297]. However, the $O(m^2)$ time bound can be improved by considering the geometric nature of the problem.

8.3.2 A global chaining algorithm

Because our algorithm is based on orthogonal range-searching for a maximum, we have to recall this notion: Given a set S of points in \mathbb{R}^2 with associated score, a *range maximum query* $RMaxQ(p, q)$ asks for a point of maximum score in $R(p, q)$.

Lemma 8.3.4 *If the gap cost function is the constant function 0 and $RMaxQ(0, beg(f') - \vec{1})$ (where $\vec{1}$ denotes the vector $(1, 1)$) returns the end point of fragment f , then we have $f'.score = f'.weight + f.score$.*

Proof This follows immediately from recurrence (8.1). □

We will further use the line-sweep paradigm to construct an optimal chain. Suppose that the start and end points of the fragments are sorted according to their x_1 coordinate. Then, processing the points in the ascending order of their x_1 coordinate simulates a vertical line that sweeps the plane from left to right. If a point has already been scanned by the

Algorithm 8.2 2-dimensional chaining of m fragments

Sort all start and end points of the m fragments in ascending order w.r.t. their x_1 coordinate and store them in the array `points`; because we include the end point of the origin and the start point of the terminus, there are $2m + 2$ points. Store all end points of the fragments (ignoring their x_1 coordinate) as inactive (in the 1-dimensional) data structure D .

for $i \leftarrow 1$ **to** $2m + 2$

if `points` $[i]$ is the start point of fragment f' **then**

$q \leftarrow \text{RMaxQ}(0, \text{points}[i].x_2 - 1)$

 determine the fragment f with $\text{end}(f) = q$

$f'.\text{prec} \leftarrow f$

$f'.\text{score} \leftarrow f'.\text{weight} + f.\text{score}$

else $/\star$ `points` $[i]$ is end point of a fragment f' $\star/$

 activate `points` $[i].x_2$ in D $/\star$ activate with score $f'.\text{score}$ $\star/$

sweeping line, it is said to be *active*; otherwise it is said to be *inactive*. During the sweeping process, the x_1 coordinates of the active points are smaller than the x_1 coordinate of the currently scanned point s . According to Lemma 8.3.4, if s is the start point of fragment f' , then an optimal chain ending at f' can be found by RMaxQ over the set of active end points of fragments. Since $p.x_1 < s.x_1$ for every active end point p (without loss of generality, start points are handled before end points; hence the case $p.x_1 = s.x_1$ cannot occur), the RMaxQ need not take the first coordinate into account. In other words, the RMaxQ is confined to the range $R(0, s.x_2 - 1)$, so that the dimension of the problem is reduced by one. To manipulate the point set during the sweeping process, we need a data structure D that stores the end points of fragments and efficiently supports the following two operations: (1) activation and (2) RMaxQ over the set of active points. Algorithm 8.2 is based on such a data structure D , which will be defined later. In the algorithm, $f'.\text{prec}$ denotes the preceding fragment of f' in a chain. It is an immediate consequence of Lemma 8.3.4 that Algorithm 8.2 finds an optimal chain. One can output this chain by tracing back the *prec* pointers from the terminus to the origin. The complexity of the algorithm depends, of course, on how the data structure D is implemented.

The data structure D can be implemented by McCreight's [219] *priority search tree*; cf. [76]. Let S be a set of m one dimensional points. For ease of presentation, assume that no two points have the same coordinate ([219] shows how to proceed if this is not the case). The priority search tree of S is a minimum-height binary search tree T with m leaves, whose i -th leftmost leaf stores the point in S with the i -th smallest coordinate. Let $v.L$ and $v.R$ denote the left and right child, respectively, of an internal node v . To each internal node v of T , we associate a canonical subset

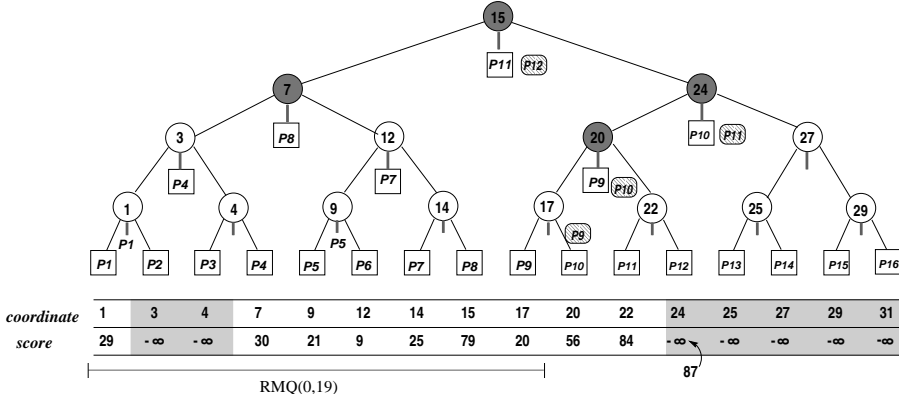


Figure 8.15: Priority search tree: The points with score $-\infty$ are inactive; the others are active. The value $h_{v.L}$ in every internal node v is the coordinate that separates the points in its left subtree $v.L$ from those occurring in its right subtree $v.R$. The p_v value of v is depicted as a “child” between $v.L$ and $v.R$. If it is missing, then p_v is undefined. The colored nodes are visited in answering $\text{RMaxQ}(0, 19)$. The hatched boxes contain the modified p_v values when point p_{12} is activated with score 87.

$C_v \subseteq S$ containing the points stored at the leaves of the subtree rooted at v . Furthermore, v stores the values h_v and p_v , where h_v denotes the largest coordinate of any point in C_v , and p_v denotes the point of highest score (priority) in C_v that has not been stored at a shallower depth in T . If such a point does not exist, p_v is undefined; see Figure 8.15.

In Algorithm 8.2, storing all end points of the fragments as inactive boils down to constructing the priority search tree of the end points, where each point stored at a leaf has score $-\infty$ and p_v is undefined for each internal node v . It is straightforward to derive a recursive algorithm for building the priority search tree in $O(m \log m)$ time and $O(m)$ space.

In a priority search tree T , a point q can be activated with priority *score* in $O(\log m)$ time as follows: First, update the value of $q.\text{score}$ by $q.\text{score} \leftarrow \text{score}$. Second, update the p_v values by a traversal of T that starts at the root. Suppose node v is visited in the traversal. If v is a leaf, there is nothing to do. If v is an internal node with $p_v.\text{score} \leq q.\text{score}$, then swap q and p_v (more precisely, exchange the values of the variables q and p_v); otherwise retain q and p_v unchanged. Then determine where the procedure is to be continued. If $q \leq h_{v.L}$, proceed recursively with the left child $v.L$ of v . Otherwise, proceed recursively with the right child $v.R$ of v .

A range maximum query $\text{RMaxQ}(0, q)$ can be answered in $O(\log m)$ time as follows: We traverse the priority search tree starting at the root. Dur-

Algorithm 8.3 Implementation of the operation *activate* in the data structure D

```

if ( $q.score > predecessor(q).score$ ) then
     $insert(q)$ 
    while ( $q.score > successor(q).score$ )
         $delete(successor(q))$ 

```

ing the traversal, we maintain a variable max_point that stores the point of highest score in the range $R(0, q)$ seen so far (initially, max_point is undefined). Suppose node v is visited in the traversal. If v is a leaf storing the point p_v , we proceed as in case (1) below. If v is an internal node, we distinguish the cases (1) $p_v \leq q$ and (2) $p_v \not\leq q$. In case (1), if $p_v.score \geq max_point.score$, we update max_point by $max_point \leftarrow p_v$. In case (2), if $q \leq h_{v,L}$, we recursively proceed with the left child $v.L$ of v ; otherwise, we recursively proceed with both children of v .

If Algorithm 8.2 is implemented with a priority search tree, then it has a worst-case time complexity of $O(m \log m)$ because both activating a point and $RMaxQ$ over the set of active points take $O(\log m)$ time.

8.3.3 Alternative data structures

The priority search tree is a semi-dynamic data structure, in the sense that points are not really inserted. The advantage of using such a data structure in Algorithm 8.2 is that the approach can be naturally extended to the higher-dimensional case. Here, however, we can also use any other one dimensional *dynamic* data structure D that supports the following operations:¹

- $insert(q)$: if q is not in D , then put q into D ; otherwise update its satellite information, i.e., the fragment corresponding to q
- $delete(q)$: remove q from D
- $predecessor(q)$: gives the largest element $\leq q$ in D
- $successor(q)$: gives the smallest element $> q$ in D

To answer $RMaxQ(0, q)$ boils down to computing $predecessor(q)$ in D , and Algorithm 8.3 shows how to activate a point q in D . Note that the operations $predecessor(q)$ and $successor(q)$ are always well-defined if we initialize the data structure D with the origin and the terminus point.

Many data structures supporting the aforementioned operations are known. For example, the priority queues devised by van Emde Boas

¹In the dynamic case, the first sentence of Algorithm 8.2 must be deleted.

[318, 319] and Johnson's improvement [168] support the operations in time $O(\log \log N)$ and space $O(N)$, provided that every q satisfies $1 \leq q \leq N$. The space requirement can be reduced to $O(n)$, where n denotes the number of elements stored in the priority queue; see [220]. Recall that a fragment corresponds to segments of the strings S^1 and S^2 that are similar. Without loss of generality, we may assume that $n_1 = |S^1| \leq |S^2| = n_2$ (otherwise, we swap the sequences). In Algorithm 8.2, using counting sort (see e.g. [61]) to sort all start and end points of the m fragments in ascending order w.r.t. their x_1 coordinate takes $O(n_1)$ time. Since Algorithm 8.2 employs at most $O(m)$ priority queue operations, each of which takes time $O(\log \log n_1)$, the overall time complexity of this implementation is $O(n_1 + m \log \log n_1)$. If the fragments are already ordered as in the heaviest increasing subsequence problem (see below), the worst-case time complexity reduces to $O(m \log \log n_1)$. Using Johnson's data structure [168], Eppstein et al. [90] showed that their sparse dynamic programming algorithm solves the problem in $O(n_1 + n_2 + m \log \log \min(m, n_1 n_2 / m))$ time. However, as noted by Chao and Miller [55], the data structure employed to obtain this theoretical efficiency is unusable in practice. With a practical data structure, the complexity becomes $O(m \log m)$; see also [139, 169]. Moreover, in most applications m is relatively small compared to n_1 , so that it is advantageous to sort the start and end points of the m fragments in $O(m \log m)$ time. Then the usage of AVL trees (see e.g. [10]), red-black trees (see e.g. [61]), or any other practical data structure that supports the above-mentioned operations in $O(\log m)$ time, gives an $O(m \log m)$ time and $O(m)$ space implementation of Algorithm 8.2.

8.3.4 Longest/heaviest increasing subsequence

It will be shown below that the problems of finding longest or heaviest increasing subsequences are special cases of the 2-dimensional fragment-chaining problem.

Definition 8.3.5 Given a sequence $A = a_1, a_2, \dots, a_m$ of positive integers,² the *longest increasing subsequence* (LIS) problem is to find a longest subsequence of A that is strictly increasing.

If every element a_i of A has a weight, then the *heaviest increasing subsequence* (HIS) problem is to find a strictly increasing subsequence of A so that the sum of the weights of its elements is maximum (among all strictly increasing subsequences).

In the LIS problem, one searches for a longest subsequence $a_{i_1}, a_{i_2}, \dots, a_{i_k}$ (where $1 \leq i_1 < i_2 < \dots < i_k \leq m$) with $a_{i_1} < a_{i_2} < \dots < a_{i_k}$. Note that this

²One can use any other totally ordered set instead of the positive integers.

Algorithm 8.4 This procedure computes a heaviest increasing subsequence of the sequence $A = a_1, \dots, a_m$.

Initialize D with the origin and the terminus point, i.e., add 0 and ∞ with $0.idx = 0$, $0.score = 0$, $\infty.idx = \infty$, and $\infty.score = \infty$ to D .
Initialize an array $Prec[1..m]$.
for $i \leftarrow 1$ **to** m
 $q \leftarrow predecessor(a_i - 1)$
 $Prec[i] \leftarrow q.idx$
 $a_i.idx \leftarrow i$
 $a_i.score \leftarrow a_i.weight + q.score$
 if $(a_i.score > predecessor(a_i).score)$ **then**
 $insert(a_i)$
 while $(a_i.score > successor(a_i).score)$ **do**
 $delete(successor(a_i))$
 $q \leftarrow predecessor(\infty)$
 $idx \leftarrow q.idx$
 while $idx \neq 0$ **do**
 output a_{idx}
 $idx \leftarrow Prec[idx]$

subsequence is not necessarily contiguous or unique. As an example, consider the sequence

$$4, 1, 6, 2, 5, 3$$

The increasing subsequences with at least two elements are 4, 6 and 1, 2, 3 and 1, 2, 5. Obviously, the last two are longest increasing subsequences. If all elements have their integral values as weights, then 4, 6 is the heaviest increasing subsequence.

Clearly, the LIS problem is a special case of the HIS problem in which each element a_i of the sequence A has weight 1. There is an abundance of papers on the LIS problem and the closely related longest common subsequence (LCS) problem; we refer the interested reader to [139] for references. For the HIS problem, Jacobson and Vo [164] devised an $O(m \log m)$ time algorithm.

If we write the sequence A in the form $(1, a_1), (2, a_2), \dots, (m, a_m)$ and view the pair (i, a_i) as the i -th fragment, then it becomes apparent that the HIS problem is a special case of the 2-dimensional fragment-chaining problem in which the fragments are just points (i.e., the start point of a fragment coincides with its end point). Algorithm 8.4, a variant of Algorithm 8.2, solves the HIS problem. In the algorithm, an element q of the sequence A carries the following data:

- $q.idx$ is the index so that $q = a_{idx}$, and

4	1	6	2	5	3
(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
(4, 1, 1)	(1, 2, 1)	(1, 2, 1)	(1, 2, 1)	(1, 2, 1)	(1, 2, 1)
(∞ , ∞ , ∞)	(4, 1, 1)	(4, 1, 1)	(2, 4, 2)	(2, 4, 2)	(2, 4, 2)
	(∞ , ∞ , ∞)	(6, 3, 2)	(6, 3, 2)	(5, 5, 3)	(3, 6, 3)
		(∞ , ∞ , ∞)	(∞ , ∞ , ∞)	(∞ , ∞ , ∞)	(5, 5, 3)
					(∞ , ∞ , ∞)

Figure 8.16: Progression of the data structure D as elements of the sequence are processed. In each triple, the first component is the element q in D , the second component is $q.idx$, and the third component is $q.score$.

i	1	2	3	4	5	6
$Prec[i]$	0	0	1	2	4	4

Figure 8.17: The $Prec$ -array.

- $q.score$ is the score of the heaviest increasing subsequence ending at (and including) q .

As mentioned in Section 8.3.3, the data structure D is initialized with the origin and the terminus to ensure that the operations $predecessor(q)$ and $successor(q)$ are always well-defined. Since the elements of A are positive integers, we can use 0 as the origin (with $0.idx = 0$ and $0.score = 0$). If we choose ∞ as terminus (with $\infty.idx = \infty$ and $\infty.score = \infty$), then we obtain an *on-line* algorithm that processes the input sequence A incrementally from left to right, and at each stage it has a heaviest increasing subsequence for the part of A that has so far been seen. The first four statements of the for-loop of Algorithm 8.4 correspond to the processing of start points, while the if-then-statement processes (activates) end points. Note that the Algorithm 8.4 maintains the following invariant: If $q_1 < q_2 < \dots < q_\ell$ are the entries in the data structure D , then $q_1.score \leq q_2.score \leq \dots \leq q_\ell.score$. A heaviest increasing subsequence is output in *reverse order* in the last while-loop of the algorithm.

We exemplify Algorithm 8.4 by computing a longest increasing subsequence of the sequence $A = a_1, a_2, a_3, a_4, a_5, a_6 = 4, 1, 6, 2, 5, 3$. (So we compute a heaviest increasing subsequence of A under the assumption that each element a_i of A has weight 1.) Figure 8.16 shows the contents of the data structure D during the execution of Algorithm 8.4, while Figure 8.17 shows the contents of the $Prec$ -array. When all elements of the sequence A have been processed in the for-loop of Algorithm 8.4, a longest

increasing subsequence is output as follows. According to the invariant, $q = \text{predecessor}(\infty)$ has maximum score (apart from ∞). It follows as a consequence that a longest increasing subsequence ends at q . Such a subsequence can be reconstructed with the help of the *Prec*-array. In our example, the algorithm first outputs 5. Then it sets $\text{idx} = \text{Prec}[5] = 4$ and outputs $a_{\text{idx}} = a_4 = 2$. Finally, it sets $\text{idx} = \text{Prec}[4] = 2$ and outputs $a_{\text{idx}} = a_2 = 1$. The reversed sequence 1, 2, 5 is a longest increasing subsequence of A .

Exercise 8.3.6 Use Algorithm 8.4 to compute a heaviest increasing subsequence of the sequence $A = 4, 1, 6, 2, 5, 3$ in which all elements have their integral values as weights.

Exercise 8.3.7 Modify Algorithm 8.4 in such a way that it also determines whether or not the heaviest increasing subsequence is unique. Analyze the worst-case time complexity of the algorithm.

Sorting by Reversals

9.1 Introduction

During evolution, genomes are subject to both small-scale and large-scale mutations. Small-scale mutations (point mutations) consist of the substitution, insertion or deletion of single nucleotides, while large-scale mutations (genome rearrangements) alter the order and orientation (strandedness) of genes on the chromosomes. In the single chromosome case (e.g. bacterial or mitochondrial DNA), the most common rearrangements are *inversions* (where a section of the genome is excised, reversed in orientation, and reinserted) and *transpositions* (where a section of the genome is excised and reinserted at a new position in the genome; this may or may not also involve an inversion). As is usually done in bioinformatics, we will use the term “reversal” as synonym for “inversion.” Further large-scale mutations are duplications, deletions (gene loss), and insertions (horizontal gene transfer). In genomes with multiple chromosomes, important genome rearrangements are reciprocal translocations (where two non-homologous chromosomes break and exchange fragments), but also fusions (where two chromosomes fuse) and fissions (where a chromosome breaks into two parts) occur. Genome rearrangements are rare compared to point mutations, and they can give us valuable information about ancient events in the evolutionary history of organisms. For this reason, one is interested in the most plausible genome rearrangement scenario between species. Work on genome rearrangements dates back to the 1930’s, when Dobzhansky and Sturtevant studied the geographical and temporal variation of chromosomal arrangements in *Drosophila pseudoobscura* and its relatives; see e.g. [81]. The comparison of genomes based on their gene arrangements was pioneered by Sankoff; see e.g. [278].

As an example, we consider mitochondrial DNA (mtDNA). A eukaryotic cell contains several thousand mitochondria. Mitochondria descended from free-living bacteria that became symbiotic with eukaryotic cells. In

Gene	Gene product
COI, COII, COIII	Cytochrome c oxidase subunits I, II, and III
Cytb	Cytochrome b apoenzyme
ND1-6, 4L	NADH dehydrogenase subunits 1 to 6 and 4L
ATP6, ATP8	ATP synthase subunits 6 and 8
srRNA	small ribosomal subunit RNA (12S rRNA)
lrRNA	large ribosomal subunit RNA (16S rRNA)
L1 and L2	two leucine tRNAs
S1 and S2	two serine tRNAs
tRNAs	18 amino acid-specific transfer RNAs

Figure 9.1: Mitochondrial DNA contains 37 genes: 13 code for proteins, two for rRNA, and 22 code for tRNA (identified by the one-letter amino acid code; cf. Figure 1.6 on page 6). The following synonyms are often used: *cox1*, *cox2*, *cox3* for COI, COII, COIII; *cob* for Cytb; *nad1-6* and *nad4L* for ND1-6 and ND4L; *rns* for srRNA and *rnl* for lrRNA.

1 COI	2 -S2	3 D	4 COII	5 K	6 ATP8	7 ATP6	8 COIII	9 G	10 ND3
11 R	12 ND4L	13 ND4	14 H	15 S1	16 L1	17 ND5	18 -ND6	19 -E	20 Cytb
21 T	22 -P	23 F	24 srRNA	25 V	26 lrRNA	27 L2	28 ND1	29 I	30 -Q
31 M	32 ND2	33 W	34 -A	35 -N	36 -C	37 -Y			

Figure 9.2: Mitochondrial gene arrangement of *Homo sapiens*; see [14,40]. A minus sign (-) indicates that the gene lies on the light strand of the mtDNA.

other words, the mtDNA is derived from the circular genomes of the bacteria that were engulfed by the early ancestors of today's eukaryotic cells. The circular chromosome of mtDNA is quite small: the human mtDNA, for example, contains about 16568 bp. The mitochondrial DNA molecule contains 37 genes; see Figure 9.1. In humans, one strand (called the heavy strand because it is heavier than the other strand) carries 28 genes and the other (the light strand) carries only 9 genes; see Figure 9.2.

Let us consider the gene arrangement in the circular mitochondrial chromosome of two different species: the fruit fly *Drosophila melanogaster* and the mosquito *Anopheles quadrimaculatus*; see Figures 9.3 and 9.4. The mitochondrial gene arrangements are the same except for three tRNA differences: R and A have switched positions and S1 was inverted. It is

1	2	3	4	5	6	7	8	9	10
COI	L2	COII	K	D	ATP8	ATP6	COIII	G	ND3
11	12	13	14	15	16	17	18	19	20
A	R	N	S1	E	-F	-ND5	-H	-ND4	-ND4L
21	22	23	24	25	26	27	28	29	30
T	-P	ND6	Cytb	S2	-ND1	-L1	-lrRNA	-V	-srRNA
31	32	33	34	35	36	37			
I	-Q	M	ND2	W	-C	-Y			

Figure 9.3: Mitochondrial gene arrangement of the fruit fly *Drosophila melanogaster*; see [119] and e.g. [290, Additional file 5].

1	2	3	4	5	6	7	8	9	10
COI	L2	COII	K	D	ATP8	ATP6	COIII	G	ND3
11	12	13	14	15	16	17	18	19	20
R	A	N	-S1	E	-F	-ND5	-H	-ND4	-ND4L
21	22	23	24	25	26	27	28	29	30
T	-P	ND6	Cytb	S2	-ND1	-L1	-lrRNA	-V	-srRNA
31	32	33	34	35	36	37			
I	-Q	M	ND2	W	-C	-Y			

Figure 9.4: Mitochondrial gene arrangement of the mosquito *Anopheles quadrimaculatus*; see [224].

rather plausible that one of the genes A or R was translocated, i.e., one reversal and one transposition have occurred during evolution.

From an evolutionary perspective, it would be interesting to know which of the two genome rearrangements appeared on which evolutionary path in Figure 9.5. For instance, the inversion could have happened in the fruit fly lineage and the transposition in the mosquito lineage or vice versa. It is also possible that both of them occurred in the same lineage. However, to answer such a question requires more than two genomes and is difficult to ascertain. In fact, the problem is NP-hard; see [21, 50, 257]. So given two genomes, one wants to find a most parsimonious¹ sequence of genome rearrangements that transforms one into the other. (The number of rearrangements gives a kind of “evolutionary distance” between the two species.)

Here, we confine ourselves to the problem of finding a *shortest* sequence of *reversals* that transforms one chromosome into another. (the interested reader can find many related problems in the book by Fertin et al. [101]). For example, we wish to solve the problem for the mtDNAs of *Drosophila melanogaster* and *Homo sapiens*. To tackle the problem systematically, we number the 37 genes in the human mtDNAs consecutively (gene 37 is

¹In general, parsimony is the principle that the simplest explanation that can explain the data is to be preferred.

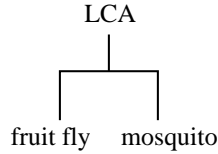


Figure 9.5: Evolutionary tree: LCA denotes the last common ancestor of the fruit fly *Drosophila melanogaster* and the mosquito *Anopheles quadrimaculatus*.

followed by gene 1 because the chromosome is circular). If gene k occurs on the heavy strand, we say that it has positive orientation and denote this by \overrightarrow{k} . Otherwise, if gene k has negative orientation, it is denoted by \overleftarrow{k} . This gives the first circular sequence:

$\overrightarrow{1}, \overleftarrow{2}, \overrightarrow{3}, \overleftarrow{4}, \overrightarrow{5}, \overleftarrow{6}, \overrightarrow{7}, \overleftarrow{8}, \overrightarrow{9}, \overrightarrow{10}, \overrightarrow{11}, \overrightarrow{12}, \overrightarrow{13}, \overrightarrow{14}, \overrightarrow{15}, \overrightarrow{16}, \overrightarrow{17}, \overrightarrow{18}, \overrightarrow{19}, \overrightarrow{20}, \overrightarrow{21}, \overrightarrow{22}, \overrightarrow{23}, \overrightarrow{24}, \overrightarrow{25}, \overrightarrow{26}, \overrightarrow{27}, \overrightarrow{28}, \overrightarrow{29}, \overrightarrow{30}, \overrightarrow{31}, \overrightarrow{32}, \overrightarrow{33}, \overrightarrow{34}, \overrightarrow{35}, \overrightarrow{36}, \overrightarrow{37}$

Using the numbers of the genes, we similarly obtain the following second circular sequence (corresponding to the fruit fly):

$\overrightarrow{1}, \overrightarrow{27}, \overrightarrow{4}, \overrightarrow{3}, \overrightarrow{5}, \overrightarrow{6}, \overrightarrow{7}, \overrightarrow{8}, \overrightarrow{9}, \overrightarrow{10}, \overrightarrow{34}, \overrightarrow{11}, \overrightarrow{35}, \overrightarrow{15}, \overrightarrow{19}, \overrightarrow{23}, \overrightarrow{17}, \overrightarrow{14}, \overrightarrow{13}, \overrightarrow{12}, \overrightarrow{21}, \overrightarrow{22}, \overrightarrow{18}, \overrightarrow{20}, \overrightarrow{2}, \overrightarrow{28}, \overrightarrow{16}, \overrightarrow{26}, \overrightarrow{25}, \overrightarrow{24}, \overrightarrow{29}, \overrightarrow{30}, \overrightarrow{31}, \overrightarrow{32}, \overrightarrow{33}, \overrightarrow{36}, \overrightarrow{37}$

Now the task is to find a shortest sequence of reversals that transforms one sequence into the other. It is possible to “standardize” the problem as follows: if gene k has negative orientation in the first sequence, then flip its orientation in both sequences. For example, $\overleftarrow{2}$ is flipped to $\overrightarrow{2}$ in the first sequence and $\overrightarrow{2}$ is flipped to $\overleftarrow{2}$ in the second sequence. Analogously, $\overleftarrow{22}$ is flipped to $\overrightarrow{22}$ in the first sequence and $\overrightarrow{22}$ is flipped to $\overleftarrow{22}$ in the second sequence. This gives the two sequences shown in Figure 9.6. After “standardization” the first sequence is the identity permutation $id = (\overrightarrow{1}, \overrightarrow{2}, \dots, \overrightarrow{n})$ (here $n = 37$) and the problem is to find a shortest sequence of reversals that transforms the second sequence into the identity permutation. That is why the problem is known under the name *sorting by reversals*. For our example, Figure 9.7 shows a solution to this problem.

$\overrightarrow{1}, \overrightarrow{27}, \overrightarrow{4}, \overrightarrow{3}, \overrightarrow{5}, \overrightarrow{6}, \overrightarrow{7}, \overrightarrow{8}, \overrightarrow{9}, \overrightarrow{10}, \overrightarrow{34}, \overrightarrow{11}, \overrightarrow{35}, \overrightarrow{15}, \overrightarrow{19}, \overrightarrow{23}, \overrightarrow{17}, \overrightarrow{14}, \overrightarrow{13}, \overrightarrow{12}, \overrightarrow{21}, \overrightarrow{22}, \overrightarrow{18}, \overrightarrow{20}, \overrightarrow{2}, \overrightarrow{28}, \overrightarrow{16}, \overrightarrow{26}, \overrightarrow{25}, \overrightarrow{24}, \overrightarrow{29}, \overrightarrow{30}, \overrightarrow{31}, \overrightarrow{32}, \overrightarrow{33}, \overrightarrow{36}, \overrightarrow{37}$
 $\overrightarrow{1}, \overrightarrow{2}, \overrightarrow{3}, \overrightarrow{4}, \overrightarrow{5}, \overrightarrow{6}, \overrightarrow{7}, \overrightarrow{8}, \overrightarrow{9}, \overrightarrow{10}, \overrightarrow{11}, \overrightarrow{12}, \overrightarrow{13}, \overrightarrow{14}, \overrightarrow{15}, \overrightarrow{16}, \overrightarrow{17}, \overrightarrow{18}, \overrightarrow{19}, \overrightarrow{20}, \overrightarrow{21}, \overrightarrow{22}, \overrightarrow{23}, \overrightarrow{24}, \overrightarrow{25}, \overrightarrow{26}, \overrightarrow{27}, \overrightarrow{28}, \overrightarrow{29}, \overrightarrow{30}, \overrightarrow{31}, \overrightarrow{32}, \overrightarrow{33}, \overrightarrow{34}, \overrightarrow{35}, \overrightarrow{36}, \overrightarrow{37}$

Figure 9.6: The standardized problem: The upper sequence (fruit fly) must be transformed into the lower (human) sequence by reversals.

In this chapter, we show how one linear chromosome can be sorted by reversals (the solution for a circular chromosome is very similar). The seminal paper on the topic is that by Hannenhalli and Pevzner [145], who showed that the problem of sorting by reversals can be solved in polynomial time; see also [258]. The overall presentation in this chapter follows the work of Setubal and Meidanis [289], which in turn is based on [24, 145, 178]. However, they presented an $O(n^5)$ time algorithm, while we develop a quadratic time algorithm. An alternative algorithm can be found in [38]. We stress that the problem can in fact be solved in $O(n^{\frac{3}{2}})$ time; see [143, 305].

Before we tackle the sorting by reversals problem, let us give some more background on the problem. First of all, the reader should be aware of the limitations of the model:

1. only reversals are taken into account,
2. both genomes consist of a single chromosome,
3. the order and orientation of the genes in both genomes is known,
4. both genomes contain a single copy of each gene,
5. both genomes share the same set of genes.

A few comments on these restrictions might be helpful.

(1) The problem of sorting by reversals *and* transpositions has also been extensively studied. Articles on that topic include [22, 91, 136, 147, 204, 324]. However, only approximation algorithms are known. This is not surprising because Bulteau et al. [46] have recently solved a long-standing open problem: sorting by transpositions is NP-hard. That is why one cannot hope for a polynomial time algorithm that solves the problem exactly. Bafna and Pevzner [25] proposed the first 1.5-approximation algorithm for the sorting by transpositions problem, and the algorithm with the currently best approximation ratio 1.375 is that of [86].

(2) Hannenhalli and Pevzner [144] have shown that their theory can be extended to multi-chromosomal genomes, where the rearrangement operations are reversals, reciprocal translocations, fusions, and fission; see also [256, 308].

(3-5) These issues are problematic. Genome sequencing is routine but gene prediction still remains a challenge. Moreover, there can be several copies of a gene in a genome and the assumption that both genomes share the same set of genes is in many cases an oversimplification.

To overcome the obstacles (3-5), researches started to use so-called syntenic blocks instead of genes. This notion is closely related to the notion of conserved segments, which was introduced in a seminal paper [234] by Nadeau and Taylor. *Conserved segments* are segments of chromosomes

in two species in which both gene content and gene order are preserved. Nadeau and Taylor [234] estimated that there are roughly 180 conserved segments in human and mouse. A human-mouse comparative map with the locations of conserved segments in the genomes can be found e.g. in [161, Figure 46]. A citation from this article:

The largest apparently contiguous conserved segment in the human genome is on chromosome 4, including roughly 90.5 Mb of human DNA that is orthologous to mouse chromosome 5.

Pevzner and Tesler [259] argued that *synteny blocks* are more suitable than conserved segments for reconstructing genome rearrangements:

However, these estimates suffer from low resolution of comparative maps in certain genomic areas. Current genomic sequences provide evidence that the human and mouse genomes are significantly more rearranged than previously thought. Moreover, they indicate that a large proportion of previously identified conserved segments are not really conserved since there is evidence of multiple micro-rearrangements in many of them (...). These micro-rearrangements were not visible in the comparative genetic maps that were used for defining 180 conserved segments in the past. We study synteny blocks instead of conserved segments. Intuitively, the synteny blocks are segments that can be converted into conserved segments by micro-rearrangements; see the GRIMM-Synteny algorithm below for a formal definition.

It is beyond the scope of this book to discuss methods for finding synteny blocks. The GRIMM-Synteny algorithm is described in [259] and alternative methods exist; see e.g. [4]. We would like to point out the relationship between these methods and the anchor-based method for whole genome alignment discussed in Section 8.3; see [3].

Pevzner and Tesler [259] identified 281 synteny blocks in a comparison of the genomes of the house mouse *Mus musculus* and *Homo sapiens*.² With the algorithm of [144], a most parsimonious human-mouse rearrangement scenario was obtained, consisting of 149 reversals, 93 reciprocal translocations, and 3 fissions. As part of the comparison, they identified 11 synteny blocks between the mouse and human X chromosomes. In a similar comparison, the CoCoNUT system [3] identified 12 synteny blocks; see Figure 9.8. To be consistent with [259], we consider only 11 synteny blocks. The arrangement of these blocks in the human X

²The mouse nuclear genome is contained in 20 chromosome pairs, whereas 23 chromosome pairs are normally present in humans, including the pair XX (females) or XY (males) which determines the sex. The last common ancestor of mice and humans is estimated to have lived approximately 75 million years ago; see e.g. [57].

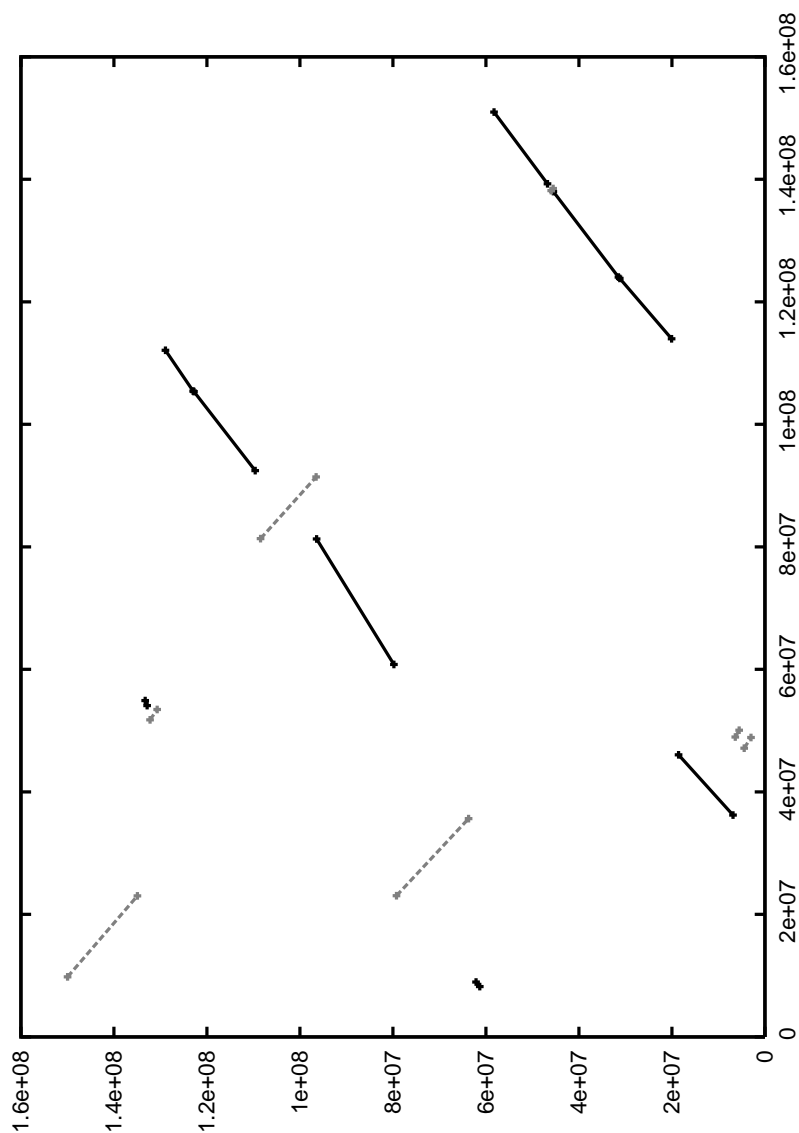


Figure 9.8: Synteny blocks obtained with the CoCoNUT system [3]: the human X chromosome (more than 153 million base pairs) is plotted on the x-axis and the mouse X chromosome is plotted on the y-axis. The leftmost block (near the tick mark $6e+07$ on the y-axis) is deleted by the GRIMM-Synteny algorithm.

chromosome corresponds to the identity permutation (because we number them consecutively and each of them occurs on the same strand), while the arrangement in the mouse X chromosome is

$$\overleftarrow{4}, \overleftarrow{5}, \overrightarrow{3}, \overrightarrow{11}, \overleftarrow{2}, \overrightarrow{8}, \overleftarrow{9}, \overrightarrow{10}, \overleftarrow{6}, \overrightarrow{7}, \overleftarrow{1} \quad (9.1)$$

To see this, consider for example the anti-diagonal in the left upper corner of Figure 9.8. It corresponds to a syntenic block that is first in the human X chromosome (so it is the element $\overrightarrow{1}$ in the human permutation) and last in the mouse X chromosome. Its orientation in the mouse X chromosome is different from its orientation in the human X chromosome because anti-diagonals indicate different orientations (by contrast, diagonals indicate the same orientations). Consequently, $\overleftarrow{1}$ is the last element in the mouse permutation.

A shortest sequence of reversals that transforms the permutation (9.1) into the identity permutation has length eight, so one might be tempted to conclude that a most parsimonious sequence that transforms the mouse X chromosome into the human X chromosome consists of eight reversals. However, there is one subtlety: the syntenic blocks were obtained by a comparison of the forward strands of the X chromosomes. Recall that the two strands of a DNA molecule carry the same information by complementary base pairing (A-T, G-C). Thus, genome projects usually provide only one strand of a chromosome, which is designated the *forward strand* and the other the *reverse strand*. (Sometimes the terms “plus strand” and “minus strand” are used instead.) This designation is arbitrary. Consequently, it might be the case that the transformation of the reverse strand of the mouse X chromosome into the forward strand of the human X chromosome requires less than eight reversals. Therefore, we must also find a shortest sequence of reversals that transforms the “reflection”

$$\overrightarrow{1}, \overleftarrow{7}, \overleftarrow{6}, \overleftarrow{10}, \overrightarrow{9}, \overleftarrow{8}, \overrightarrow{2}, \overleftarrow{11}, \overleftarrow{3}, \overrightarrow{5}, \overrightarrow{4}$$

of the permutation (9.1) into the identity permutation. Such a sequence is depicted in Figure 9.9: it contains only seven reversals. Thus, it is a most parsimonious sequence of reversals that transforms the mouse X chromosome into the human X chromosome.

9.2 Basic definitions

Definition 9.2.1 An *oriented permutation* $\pi = (\pi_1, \dots, \pi_n)$ is a permutation of the set $\{1, \dots, n\}$, in which each element (syntenic block) has an orientation.³ If a syntenic block $k \in \{1, \dots, n\}$ is on the forward strand of the DNA

³An oriented permutation is also termed *signed permutation* in the literature.



Figure 9.9: Sorting the permutation obtained from synteny blocks between the reverse strand of the mouse X chromosome and the forward strand of the human X chromosome; see [259]. In each step, the underlined segment is inverted.

double-strand, then it has positive orientation, denoted by \overrightarrow{k} . If k is on the reverse strand, then it has negative orientation, denoted by \overleftarrow{k} .

In the following, we will use the term *permutation* as short hand for *oriented permutation*. Moreover, we will tacitly assume that each permutation consists of n pairwise distinct synteny blocks (numbered from 1 to n), unless stated otherwise.

Definition 9.2.2 A *segment* π_i, \dots, π_j (where $1 \leq i \leq j \leq n$) of a permutation $\pi = (\pi_1, \dots, \pi_n)$ is a consecutive sequence of elements in π , with π_i being the first element and π_j being the last element.

A *reversal* $\rho(i, j)$ is an operation that inverts the order of the elements of the segment π_i, \dots, π_j in the permutation $\pi = (\pi_1, \dots, \pi_n)$. Additionally, the orientation of each element in the segment is flipped.

We use postfix notation $\pi\rho(i, j)$ to denote the application of $\rho(i, j)$ to π . For example, if $\pi = (\overleftarrow{2}, \overleftarrow{1}, \overleftarrow{3}, \overleftarrow{5}, \overleftarrow{6}, \overleftarrow{4})$, then $\pi\rho(2, 3) = (\overleftarrow{2}, \overleftarrow{3}, \overleftarrow{1}, \overleftarrow{5}, \overleftarrow{6}, \overleftarrow{4})$. It should be stressed that in the biological literature reversal are called *inversions*.

Definition 9.2.3 The *reversal distance* $d(\pi, \pi')$ between two permutations π and π' is the minimum number of reversals required to transform π into π' . Given a permutation π , the *reversal distance problem* is to find the

$$\begin{aligned}
\pi\rho(1,2)\rho(5,6)\rho(4,5) &= (\overrightarrow{0}, \overleftarrow{2}, \overleftarrow{1}, \overrightarrow{3}, \overleftarrow{5}, \overleftarrow{6}, \overrightarrow{4}, \overrightarrow{7})\rho(1,2)\rho(5,6)\rho(4,5) \\
&= (\overrightarrow{0}, \overrightarrow{1}, \overrightarrow{2}, \overrightarrow{3}, \overleftarrow{5}, \overleftarrow{6}, \overrightarrow{4}, \overrightarrow{7})\rho(5,6)\rho(4,5) \\
&= (\overrightarrow{0}, \overrightarrow{1}, \overrightarrow{2}, \overrightarrow{3}, \overleftarrow{5}, \overrightarrow{4}, \overrightarrow{6}, \overrightarrow{7})\rho(4,5) \\
&= (\overrightarrow{0}, \overrightarrow{1}, \overrightarrow{2}, \overrightarrow{3}, \overrightarrow{4}, \overrightarrow{5}, \overrightarrow{6}, \overrightarrow{7})
\end{aligned}$$

Figure 9.10: A sorting sequence of reversals.

minimum number $d_\pi = d(\pi, id)$ of reversals required to transform π into the *identity permutation* $id = (\overrightarrow{1}, \overrightarrow{2}, \dots, \overrightarrow{n})$. In the *sorting by reversals problem*, one additionally asks for a sequence $\rho_1, \rho_2, \dots, \rho_{d_\pi}$ that actually sorts π , i.e., $\pi\rho_1\rho_2\dots\rho_{d_\pi} = id$.

Exercise 9.2.4 Show that the reversal distance on the set of permutations satisfies the metric axioms.

Because we deal with linear permutations (and not cyclic permutations), we augment every permutation with boundary elements $\overrightarrow{0}$ and $\overleftarrow{n+1}$ at the beginning and at the end. More precisely, from now on we will use the extended permutation

$$\pi = (\pi_0, \pi_1, \dots, \pi_n, \pi_{n+1}) = (\overrightarrow{0}, \pi_1, \dots, \pi_n, \overleftarrow{n+1})$$

instead of the original permutation (π_1, \dots, π_n) . We stress that these boundary elements cannot be moved by a reversal.

Figure 9.10 shows a sorting sequence of the (extended) permutation $\pi = (\overrightarrow{0}, \overleftarrow{2}, \overleftarrow{1}, \overrightarrow{3}, \overleftarrow{5}, \overleftarrow{6}, \overrightarrow{4}, \overrightarrow{7})$. It will become clear below that this is a shortest sorting sequence.

Definition 9.2.5 A pair π_i, π_{i+1} of consecutive elements in a permutation $\pi = (\pi_0, \pi_1, \dots, \pi_n, \pi_{n+1})$ is called *adjacency* if it has the form $\overrightarrow{k}, \overleftarrow{k+1}$ or $\overleftarrow{k+1}, \overrightarrow{k}$ for some $k \in \{0, 1, \dots, n\}$. (In this case, we also use the phrase “the oriented elements k and $k+1$ are adjacent in π .”) Otherwise, we speak of a *breakpoint*.

In what follows, a_π and b_π denote the number of adjacencies and breakpoints in π , respectively. Obviously, a_π and b_π are related as follows:

$$b_\pi = (n+1) - a_\pi$$

Furthermore, $\pi = id$ if and only if $b_\pi = 0$ (or equivalently, $a_\pi = n+1$). In our example permutation $\pi = (\overrightarrow{0}, \overleftarrow{2}, \overleftarrow{1}, \overrightarrow{3}, \overleftarrow{5}, \overleftarrow{6}, \overrightarrow{4}, \overrightarrow{7})$ there is only one adjacency, viz, $\overleftarrow{2}, \overleftarrow{1}$. Thus, $b_\pi = 7 - 1 = 6$.

Lemma 9.2.6 *There is the following lower bound for d_π :*

$$d_\pi \geq \frac{1}{2}b_\pi$$

Proof Obviously, a reversal can remove at most two breakpoints. Therefore, at least $\frac{1}{2}b_\pi$ reversals are required to remove all b_π breakpoints. \square

The length 3 sorting sequence in Figure 9.10 is a shortest possible because Lemma 9.2.6 implies that any sequence that sorts π has length at least $\frac{1}{2}b_\pi = 3$.

9.3 The reality-desire diagram

The main tool for studying the sorting by reversals problem (and related problems) is the breakpoint graph introduced by Bafna and Pevzner [24]. As suggested by Setubal and Meidanis [289], we use the term “reality-desire diagram” instead.

In the forthcoming discussion, it is handy to split every element in the permutation π into two signed elements. An element \overrightarrow{k} with positive orientation is replaced with $-k, +k$, whereas \overleftarrow{k} is replaced with $+k, -k$. So $+$ stand for the arrow-head and $-$ stands for the shaft of the arrow that indicates the orientation of an element. Because the boundary elements always have positive orientation, it suffices to replace $\overrightarrow{0}$ with $+0$ and $\overrightarrow{n+1}$ with $-(n+1)$. In this new notation, our example permutation has the form $\pi = (+0, +2, -2, +1, -1, -3, +3, +5, -5, +6, -6, -4, +4, -7)$

Clearly, an oriented element $\overrightarrow{k} = -k, +k$ (or $\overleftarrow{k} = +k, -k$, respectively) cannot be split by a reversal.

Definition 9.3.1 The *reality-desire diagram* of a permutation π is an undirected graph with nodes $+0, +1, \dots, +n, -1, -2, \dots, -n, -(n+1)$ and two kinds of edges:

- for each pair π_i, π_{i+1} of consecutive oriented elements in π there is a *reality edge* that connects π_i with π_{i+1} . More precisely, it connects the right component of π_i with the left component of π_{i+1} . For example, if $\pi_i, \pi_{i+1} = \overleftarrow{1}, \overrightarrow{3} = (+1, -1), (-3, +3)$ then there is a reality edge $(-1, -3)$ in the reality-desire diagram. Reality edges reflect the actual neighbor relationships.
- For each k with $0 \leq k \leq n$, there is a *desire edge* $(+k, -(k+1))$. Desire edges reflect the desired neighbor relationships in the identity permutation $id = (\overrightarrow{0}, \overrightarrow{1}, \dots, \overrightarrow{n}, \overrightarrow{n+1})$.

Figures 9.11 and 9.12 show a linear and a circular representation of the reality-desire diagram of our example permutation.

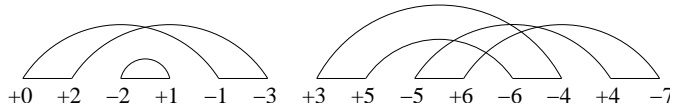


Figure 9.11: The linear reality-desire diagram of our permutation.

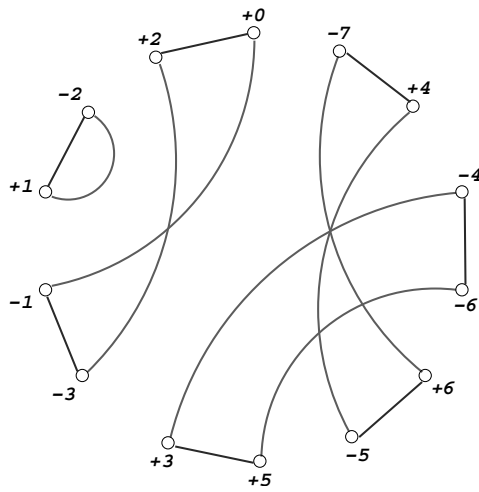


Figure 9.12: The circular reality-desire diagram of our permutation.

Definition 9.3.1 immediately implies:

- Every node in a reality-desire diagram is incident to exactly one reality edge and exactly one desire edge.
- Every connected component in the reality-desire diagram is a cycle consisting of alternating reality and desire edges.
- For an adjacency $\overrightarrow{k}, \overleftarrow{k+1}$, the reality edge $(+k, -(k+1))$ coincides with the desire edge $(+k, -(k+1))$ (the case $\overleftarrow{k+1}, \overrightarrow{k}$ is analogous). A cycle corresponds to an adjacency if and only if it consists of exactly two edges (one reality edge and one desire edge).

Lemma 9.3.2 *A permutation π is the identity permutation if and only if its reality-desire diagram has $n+1$ cycles.*

Proof If π is the identity permutation, then $a_\pi = n+1$. Hence its reality-desire diagram has $n+1$ cycles. If π is not the identity permutation, then it has at least one breakpoint. The cycle containing the reality edge induced by the breakpoint has more than two edges. Because the reality-desire diagram has a total of $2(n+1)$ edges, this implies that the overall number of cycles must be less than $n+1$. \square

In view of the preceding lemma, the process of transforming a permutation π into the identity permutation by a minimum number of reversals can be viewed as a process of transforming the reality-desire diagram of π into a reality-desire diagram with as many cycles as possible. Thus, it is helpful to characterize the effect of a reversal on a reality-desire diagram (especially on the number of cycles).

Obviously, any reversal acts on exactly two reality edges. For example, the reversal $\rho(4, 5)$ applied to our example permutation

$$\pi = (\overrightarrow{0}, \overleftarrow{2}, \overleftarrow{1}, \overrightarrow{3}, \overleftarrow{5}, \overleftarrow{6}, \overrightarrow{4}, \overrightarrow{7})$$

acts on the reality edges $(+3, +5)$ and $(-6, -4)$; see Figure 9.11. To be precise, it removes the edges $(+3, +5)$ and $(-6, -4)$, inverts the segment $\overleftarrow{5}, \overleftarrow{6}$, and adds the new reality edges $(+3, -6)$ and $(+5, -4)$; see Figure 9.13. In the following, it is convenient to specify the interval that is inverted by ρ instead of specifying the segment. In our example, the reversal $\rho(4, 5)$ of the segment $\overleftarrow{5}, \overleftarrow{6}$ is denoted by $\rho([+5, \dots, -6])$, and we say ρ inverts (or reverses) the interval $[+5, -5, +6, -6]$.

With the aid of the following definition, we will be able to characterize the effect of a reversal on the number of cycles.

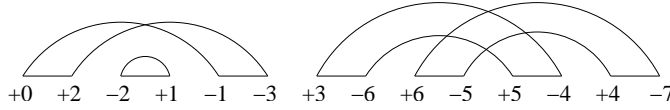


Figure 9.13: The reversal $\rho([+5, \dots, -6])$ applied to the permutation from Figure 9.11 acts on the convergent reality edges $(+3, +5)$ and $(-6, -4)$. As one can see in the figure, the number of cycles remains unchanged.

Definition 9.3.3 Two distinct reality edges belonging to the same cycle are called *convergent* if they are traversed in the same direction in a traversal of the cycle. In the linear reality-desire diagram, this means that both edges are traversed from left to right or both are traversed from right to left. In the circular reality-desire diagram, this means that both edges are traversed clockwise or both are traversed counter-clockwise. Otherwise, they are *divergent*.

Note that the phrase “two reality edges are convergent” necessarily implies that the two edges belong to the same cycle; the same is true for two divergent reality edges.

The reality edges $(+3, +5)$ and $(-6, -4)$ in Figures 9.11 and 9.12 are convergent, whereas e.g. the reality edges $(+0, +2)$ and $(-1, -3)$ are divergent.

Lemma 9.3.4 A reversal ρ applied to a permutation π changes the number of cycles as follows:

- If ρ acts on two convergent reality edges, then the number of cycles remains unchanged; see Figure 9.13.
- If ρ acts on two divergent reality edges, then the number of cycles increases by one; see Figure 9.14.
- If ρ acts on two reality edges from different cycles, then the number of cycles decreases by one; see Figure 9.15.

Proof Suppose that the reversal acts on the two reality edges (s, t) and (u, v) , and that s, t, u, v occur in that order in π . Clearly, the reversal can be written as $\rho([t, \dots, u])$ and it has the following effects on the reality-desire diagram: it removes the edges (s, t) and (u, v) , inverts the interval $[t, \dots, u]$, and adds the new reality edges (s, u) and (t, v) . Since desire edges always remain unchanged, the reversal solely effects the cycle(s) containing (s, t) and (u, v) . Now, let us consider the different cases.

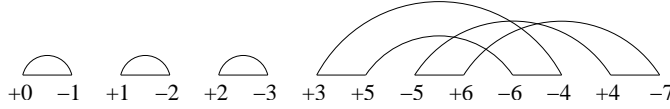


Figure 9.14: The reversal $\rho([+2, \dots, -1])$ applied to the permutation from Figure 9.11 acts on the divergent reality edges $(+0, +2)$ and $(-1, -3)$. The number of cycles increases by one.

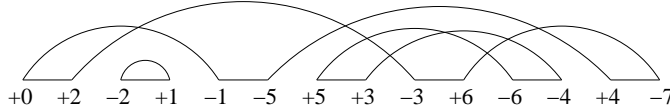


Figure 9.15: The reversal $\rho([-3, \dots, -5])$ applied to the permutation from Figure 9.11 acts on the reality edges $(-1, -3)$ and $(-5, +6)$, which belong to different cycles. The number of cycles decreases by one.

- Since the two reality edges (s, t) and (u, v) are convergent, they belong to the same cycle. A traversal of this cycle that starts with the edge (s, t) , continues from t to u , moves along the edge (u, v) , and then goes back from v to s . In other words, the path has the form $(s, t, \dots, u, v, \dots, s)$. After the reversal, there is the path $(s, u, \dots, t, v, \dots, s)$. Therefore, the new reality edges (s, u) and (t, v) belong to the same cycle. Consequently, the number of cycles remains the same.
- If the two reality edges (s, t) and (u, v) are divergent, a traversal of their cycle that starts with the edge (s, t) has the form $(s, t, \dots, v, u, \dots, s)$. The reversal replaces the edges (s, t) and (u, v) with the edges (s, u) and (t, v) . So after the reversal, there are the paths (s, u, \dots, s) and (t, \dots, v, t) . In summary, the big cycle $(s, t, \dots, u, v, \dots, s)$ is split into two smaller cycles (s, u, \dots, s) and (t, \dots, v, t) . That is, the reversal increases the number of cycles by one.
- If (s, t) and (u, v) belong to different cycles (s, t, \dots, s) and (u, v, \dots, u) , then the reversal merges them into the big cycle $(s, u, \dots, v, t, \dots, s)$. Therefore, the number of cycles decreases by one.

□

Corollary 9.3.5 *There is the following lower bound for the reversal distance d_π of a permutation π :*

$$d_\pi \geq n + 1 - c_\pi$$

where c_π denotes the number of cycles in the reality-desire diagram of π .

Proof The desired identity permutation has $n + 1$ cycles. Thus, in the transformation from the reality-desire diagram of π to that of the identity permutation, $n + 1 - c_\pi$ cycles must be created. According to Lemma 9.3.4, a reversal can increase the number of cycles by at most one. So any sequence of reversals that sorts π has length at least $n + 1 - c_\pi$. \square

Exercise 9.3.6 Give an example of a permutation that can not be sorted with $n + 1 - c_\pi$ reversals.

9.4 Components

In this section, we define the components of a reality-desire diagram and provide a linear-time algorithm to find them all.

9.4.1 Elementary intervals

Definition 9.4.1 For every $k \in \{0, 1, \dots, n\}$, the *elementary interval* I_k of the permutation π is defined as

$$I_k = \begin{cases} [+k, \dots, -(k+1)] & \text{if } +k \text{ occurs in } \pi \text{ before } -(k+1) \\ [- (k+1), \dots, +k] & \text{if } +k \text{ occurs in } \pi \text{ after } -(k+1) \end{cases}$$

An elementary interval I_k is called *good* if the oriented elements k and $k+1$ have different orientations in π ; it is *bad* if the oriented elements k and $k+1$ have the same orientation but they are not adjacent in π .⁴

The oriented elements k and $k+1$ are adjacent in π (i.e., either $\overrightarrow{k}, \overrightarrow{k+1}$ or $\overleftarrow{k+1}, \overleftarrow{k}$ occur consecutively in π) if and only if the elementary interval I_k has the form $[+k, -(k+1)]$ or $[-(k+1), +k]$. Such an interval is neither good nor bad.

The notation $\rho(I)$ means that the reversal ρ inverts the interval I . If $-p$ (or $+q$) is a boundary element of I , then $+p$ (or $-q$) must also be contained in I because a reversal cannot split an oriented element. In this chapter,

⁴As first done by Setubal and Meidanis [289], we use the attributes *good* and *bad* instead of the originally suggested attributes *oriented* and *unoriented* because a sentence like “An elementary interval I_k is called *oriented* if the oriented elements k and $k+1$ have different orientations in π ” may easily confuse the reader.

we implicitly assume that this is indeed the case. But there is one exception: elementary intervals may violate this assumption. Therefore, we define the meaning of a reversal applied to an elementary interval I_k as follows: $\rho(I_k)$ inverts all oriented elements that are completely contained in I_k . That is, if $-k$ ($+(k+1)$, respectively) is not in I_k , then the oriented element k ($k+1$, respectively) is not affected by $\rho(I_k)$. For example, the reversal of the elementary interval $I_2 = [+2, \dots, -3]$ in the permutation

$$\pi = (+0, +2, -2, +1, -1, -3, +3, +5, -5, +6, -6, -4, +4, -7)$$

does not move $\overrightarrow{3}$ because it inverts the interval $[+2, -2, +1, -1]$.

An elementary interval I_k directly corresponds to the desire edge from node $+k$ to node $-(k+1)$ in the reality-desire diagram of π . The neighboring reality edges of this desire edge are divergent if I_k is good and convergent if I_k is bad. Note that the reversal $\rho(I_k)$ acts on these neighboring reality edges. By Lemma 9.3.4, the reversal of a good elementary interval increases the number of cycles by one, whereas the reversal of a bad elementary interval has no influence on the number of cycles. This explains the attributes good and bad. According to Definition 9.4.1, the elementary intervals of adjacencies are neither good nor bad. However, the reader should not think of them as being neutral. They are excellent because they are already part of the desired result.

The status (good or bad) of an elementary interval I_k can also be inferred from the linear representation of the reality-desire diagram. There, the desire edge corresponding to I_k is drawn as an arc. If both neighboring reality edges lie inside (outside, respectively) the arc, then I_k is bad; otherwise it is good. Let us take a look at Figure 9.11 (page 441). The elementary interval $I_5 = [+5, -5, +6, -6]$ is bad because both reality edges $(+3, +5)$ and $(-6, -4)$ lie outside the arc $(+5, -6)$. By contrast, the elementary interval $I_4 = [-5, +6, -6, -4, +4]$ is good because $(-5, +6)$ is inside the arc $(-5, +4)$ but $(+4, -7)$ is outside.

Exercise 9.4.2 Show that the reversal of a good elementary interval I_k creates the adjacency $\overrightarrow{k}, \overrightarrow{k+1}$ or $\overleftarrow{k+1}, \overleftarrow{k}$.

Definition 9.4.3 For every i with $1 \leq i \leq 2(n+1)$ let p_i denote the signed element at position i in $\pi = (+0, \dots, -(n+1))$.

Two intervals $[p_i, \dots, p_j]$ and $[p_k, \dots, p_l]$ *overlap* if either $i < k \leq j < l$ or $k < i \leq l < j$ holds true. To put it differently, two intervals overlap if their intersection is non-empty but neither is a subinterval of the other.

Lemma 9.4.4 An elementary interval I_k that does not overlap with any other elementary interval has the form $[+k, -(k+1)]$ or $[-(k+1), +k]$ (in other words, $\overrightarrow{k}, \overrightarrow{k+1}$ or $\overleftarrow{k+1}, \overleftarrow{k}$ is an adjacency in π).

Proof Let p (q , respectively) be the maximum (minimum, respectively) absolute value of all elements in I_k . If $+p$ were an element of I_k , then $-(p+1)$ would not be an element of I_k and thus I_k would overlap with the elementary interval I_p . Hence $-p$ is an element of I_k , but $+p$ is not. Similarly, one can show that $+q$ is an element of I_k , but $-q$ is not. It follows as a consequence that $+q$ and $-p$ coincide with the boundary elements $+k$ and $-(k+1)$ of I_k . Because there is no natural number i satisfying $k < i < k+1$, I_k is either the interval $[+k, -(k+1)]$ or the interval $[-(k+1), +k]$. \square

For our purposes, the converse statement of Lemma 9.4.4 is important: if the oriented elements k and $k+1$ are not adjacent in π , then the elementary interval I_k must overlap with another elementary interval.

In what follows, we will identify the elementary interval I_k with the desire edge $(+k, -(k+1))$. For example, the desire edge $(+k, -(k+1))$ is said to be *good* if the elementary interval I_k is good. Furthermore, we say that two desire edges $(+j, -(j+1))$ and $(+k, -(k+1))$ *overlap* if the elementary intervals I_j and I_k overlap.

Definition 9.4.5 Two different cycles c and d in the reality-desire diagram of π *overlap* if there is a desire edge in c that overlaps with a desire edge from d .

In the reality-desire diagram of Figure 9.11, the cycles $(+3, +5, -6, -4)$ and $(-5, +6, -7, +4)$ overlap.

Definition 9.4.6 The *overlap graph* G of a permutation π is an undirected graph defined as follows:

- The cycles in the reality-desire diagram of π are the nodes of G .
- There is an edge (c, d) if and only if the cycles c and d overlap.

The connected components of G are simply referred to as *components*.

The *overlap graph* of the permutation π from Figure 9.11 has four nodes: the cycles $c_1 = (+0, +2, -3, -1)$, $c_2 = (-2, +1)$, $c_3 = (+3, +5, -6, -4)$, and $c_4 = (-5, +6, -7, +4)$. Because it has just one edge (c_3, c_4) , there are three components.

In the following definition, we exclude cycles and components that correspond to an adjacency (these are neither good nor bad).

Definition 9.4.7 A cycle c in the reality-desire diagram of permutation π is called *good* if it contains at least one good desire edge; otherwise we speak of a *bad* cycle. A component is *good* if it contains at least one good cycle; otherwise it is *bad*.

9.4.2 Finding cycles and components

Finding cycles in the reality-desire diagram of a permutation π is rather easy. Suppose that the $2n + 2$ nodes in the reality-desire diagram are stored left-to-right in an array v . Initially, each node is untagged. A node will be tagged when it is visited for the first time. The algorithm scans the nodes from left to right and searches for the first untagged node. If the first untagged node appears at position i_1 , the algorithm starts at node $v[i_1]$, follows the incident reality edge to node $v[i_2]$, tags $v[i_2]$, follows the incident desire edge to node $v[i_3]$, tags $v[i_3]$, follows the incident reality edge etc. until $v[i_1]$ is reached again. Clearly, $v[i_1], v[i_2], v[i_3], \dots, v[i_1]$ is a cycle. Then it resumes the left-to-right scan (and identifies further cycles) until it reaches the last position $2n + 2$. In this way, all cycles can be found in $O(n)$ time (there are $2n + 2$ nodes and $2n$ edges in the reality-desire diagram of π). It is quite obvious how to modify this algorithm so that it also delivers the status of each cycle. This easy exercise is left to the reader.

By contrast, it is much harder to find all components in linear time. This is because the overlap graph can be of quadratic size. Bader et al. [20] solved the problem by constructing an *overlap forest* so that two cycles of the reality-desire diagram of π belong to the same tree in the forest if and only if they belong to the same component in the overlap graph of π . An overlap forest has exactly one tree per component and thus is of linear size.

Definition 9.4.8 The *extent* of a cycle c in the reality-desire diagram of a permutation π is the interval $[c.b..c.e]$, where $c.b = \min\{i \mid v[i] \text{ belongs to } c\}$ and $c.e = \max\{i \mid v[i] \text{ belongs to } c\}$. The *extent* of a set of cycles $\{c_1, \dots, c_k\}$ is the interval $[b..e]$, where $b = \min\{c_i.b \mid 1 \leq i \leq k\}$ and $e = \max\{c_i.e \mid 1 \leq i \leq k\}$.

Suppose all cycles and their extents have been computed by the algorithm described above, and that the cycles are numbered consecutively from left to right. In the following, the number of a cycle is used as an identifier of the cycle. In the initial forest F_0 , every cycle in the reality-desire diagram of π is the root of a single-node tree. Proceeding inductively, let F_{i-1} be the forest obtained by processing the first $i - 1$ nodes of the reality-desire diagram of π . The forest F_i is constructed from F_{i-1} as follows: Let the i -th node belong to the cycle c . If c starts at position i , i.e., position i is the leftmost node that belongs to the cycle c (so $i = c.b$), then no information about overlaps of c with other cycles is yet available. Thus, $F_i = F_{i-1}$. Otherwise, we infer that every cycle c' starting at a position in between the positions $c.b$ and i and ending after position i must overlap with c . Consequently, c and c' belong to the same component C in the overlap graph and c becomes the parent node of c' in the overlap forest.

Furthermore, the combined extent of c' and the tree rooted at c is computed (this corresponds to the extent of the component C up to position i) and stored in $c.b$ and $c.e$. We say that a tree rooted at c is *active* at position i if i lies properly within the extent of c , and store the extent of active trees on a stack. Pseudo-code of the algorithm can be found in Algorithm 9.1. Each tree T in the overlap forest can be represented by the root r of the tree. We use the identifier $r.id$ of the cycle r at the root of T as a unique identifier of T (and thus of the component corresponding to T). In the last for-loop, the identifier $c.id$ of a cycle c is changed to the identifier of the root of the tree to which c belongs. This can be done by a left-to-right scan of the array *parent* because the parent of a cycle c has an identifier that is strictly smaller than $j = c.id$ (i.e., $parent[j] < j$). Afterwards, the component to which node $v[i]$ belongs can be determined by the component identifier $C[ptr[i]].id$.

Theorem 9.4.9 *Algorithm 9.1 constructs a forest F so that the trees in the forest correspond exactly to the components of the overlap graph.*

Proof We show by induction that Algorithm 9.1 maintains the following invariant for each i with $1 \leq i \leq 2n + 2$: after the $(i - 1)$ -th iteration of the second for-loop, the trees in the forest F_{i-1} correspond exactly to the components of the overlap graph determined by the overlaps detected up to position $i - 1$. More precisely, the set of nodes (cycles) of a tree containing a cycle c coincides with the set of nodes (cycles) of the component containing c .

The base case $i = 1$ (i.e., $i - 1 = 0$) is trivial because the overlap graph has no edges yet (no overlap has been detected so far). By the inductive hypothesis we may assume that the invariant holds for $i - 1$. In the inductive step, let the i -th node in the reality-desire diagram of π belong to cycle c . We prove that after the i -th iteration of the second for-loop, the tree to which c belongs contains another cycle c' *if and only if* the component to which c belongs contains c' . Because the other trees and components are unaffected, this shows that the invariant also holds after the i -th iteration of the second for-loop. Observe that if $i = c.b$ (i.e., i is the leftmost node of cycle c), then neither the overlap forest nor the overlap graph changes. Therefore, the invariant is preserved in this case. Now we consider a position i at which the overlap forest or the overlap graph (or both) change.

“if”: If $c.b < i$ and $c.b < top.b$, then Algorithm 9.1 sets $parent[top.id] \leftarrow c.id$. In other words, the tree rooted at top becomes a subtree of the tree rooted at c . By the inductive hypothesis, the nodes contained in the tree rooted at c are contained in a component C and the nodes contained in the tree rooted at top are contained in a different component C' in the overlap graph determined by the overlaps detected up to position $i - 1$. We must show that after the i -th iteration, every cycle c' contained in the tree rooted

Algorithm 9.1 Constructing an overlap forest.

```

initialize an array  $ptr[1, \dots, 2n + 2]$ 
 $k \leftarrow 0$       /*  $k$  counts the number of cycles */
in a left-to-right scan of the reality-desire diagram of  $\pi$  do
  whenever a new cycle  $c = v[i_1], v[i_2], \dots, v[i_m], v[i_1]$  is found do
     $k \leftarrow k + 1$ 
     $c.b \leftarrow \min\{i_j \mid 1 \leq j \leq m\}$ 
     $c.e \leftarrow \max\{i_j \mid 1 \leq j \leq m\}$       /*  $[b..e]$  is the extent of cycle  $c$  */
    for  $j \leftarrow 1$  to  $m$  do
       $ptr[i_j] \leftarrow k$       /* node  $v[i_j]$  belongs to the  $k$ -th cycle */
       $c.id \leftarrow k$       /* identify a cycle by its number */
       $C[k] \leftarrow c$       /*  $C[k]$  is the cycle  $c$  with number  $c.id = k$  */
for  $j \leftarrow 1$  to  $k$  do      /* initialize the overlap forest */
   $parent[j] \leftarrow \perp$       /*  $parent[j]$  is undefined */
for  $i \leftarrow 1$  to  $2n + 2$  do
   $c \leftarrow C[ptr[i]]$       /* node  $v[i]$  belongs to cycle  $c$  */
  if  $i = c.b$  then
     $push(c)$ 
     $extent \leftarrow c$ 
    while ( $c.b < top.b$ ) do
       $extent.b \leftarrow \min\{extent.b, top.b\}$ 
       $extent.e \leftarrow \min\{extent.e, top.e\}$ 
       $parent[top.id] \leftarrow c.id$ 
       $pop()$ 
       $top.b \leftarrow \min\{extent.b, top.b\}$ 
       $top.e \leftarrow \max\{extent.e, top.e\}$ 
    if  $i = top.e$  then
       $pop()$ 
for  $j \leftarrow 1$  to  $k$  do
  if  $parent[j] \neq \perp$  then
     $C[j].id \leftarrow C[parent[j]].id$ 
/* now node  $v[i]$  belongs to the component with identifier  $C[ptr[i]].id$  */

```

at top is contained in the component containing c . Because top was on the stack before the i -th iteration, it follows that $c.b < top.b < i < top.e$. This implies that there must be a desire edge belonging to cycle c that overlaps with a desire edge belonging to top . In other words, the components C and C' are connected by a new edge in the overlap graph. Consequently, after the i -th iteration cycle c' belongs to the component containing c .

“only-if”: The new edges added to the overlap graph are those overlaps that are detected at position i , i.e., they have not been detected up to position $i - 1$. Each of these overlaps is an overlap between cycle c and another cycle c' . If c and c' were already in the same component after the $(i - 1)$ -th iteration, then it follows from the inductive hypothesis that they are in the same tree. So suppose that they were not in the same component, say c belonged to component C and c' belonged to component C' . According to the inductive hypothesis, the components C and C' correspond to trees T and T' . After the i -th iteration, there is an edge between c and c' in the overlap graph. Because the overlap of c and c' is detected at position i , there are desire edges $(v[j], v[i])$ and $(v[k], v[l])$ belonging to c and c' , respectively, so that $j < k < i < l$. In this case, however, Algorithm 9.1 sets $parent[c'.id] \leftarrow c.id$, i.e., the tree rooted at c' becomes a subtree of the tree rooted at c . Thus, every cycle of the former component C' (or equivalently, of the former tree T') now belongs to the tree rooted at c . \square

9.5 Sorting a permutation without bad components

In the overall sorting-by-reversals algorithm we are going to develop, all bad components are eliminated first and then the resulting permutation (without bad components) is sorted. We defer the elimination of bad components to Section 9.6. In this section, we develop an algorithm that is able to sort a permutation without bad components in $O(n^3)$ time. Later, we shall see that the problem can be solved in $O(n^2)$ time.

Lemma 9.5.1 *A reversal $\rho(I)$ changes the status of an elementary interval I_k (good into bad or vice versa) if and only if I contains one of the oriented elements k and $k + 1$ but not the other.*

Proof If I contains one of the oriented elements k and $k + 1$, but not the other, then it changes the orientation of exactly one of them. So if they have the same orientation (different orientation, respectively) before the reversal $\rho(I)$, then they have different orientation (the same orientation, respectively) after the reversal. That is, the status of I_k changes. If I contains both k and $k + 1$, then $\rho(I)$ changes the orientation of both k and $k + 1$. In this case, the status of I_k remains unchanged. Obviously, the same is true if I neither contains k nor $k + 1$. \square

Lemma 9.5.2 *A reversal $\rho(I_j)$ of an elementary interval I_j changes the status of another elementary interval I_k if and only if I_j and I_k overlap.*

Proof Clearly, if I_j and I_k overlap, then their boundary elements $+j$, $-(j+1)$, $+k$, and $-(k+1)$ occur in alternating order in the permutation π . We consider the orders (a) $+j$, $+k$, $-(j+1)$, $-(k+1)$ and (b) $-(j+1)$, $+k$, $+j$, $-(k+1)$; the other possible orders can be treated similarly. Note that $j \neq k$. In case (a), $-k$ must lie within the interval I_k ; hence $+j$, $+k$, $-k$, $-(j+1)$, $-(k+1)$ occur in that order in π , where possibly $-k = -(j+1)$. In case (b), the elements occur in the order $-(j+1)$, $+k$, $-k$, $+j$, $-(k+1)$ or in the order $-(j+1)$, $-k$, $+k$, $+j$, $-(k+1)$. In both cases (a) and (b), $+k$ and $-k$ are contained in I_j but $-(k+1)$ is not. Thus, Lemma 9.5.1 implies that the reversal $\rho(I_j)$ changes the status of I_k .

If I_j and I_k do not overlap, then their boundary elements $+j$, $-(j+1)$, $+k$, and $-(k+1)$ do not occur in alternating order. If both $+j$ and $-(j+1)$ precede (or succeed) $+k$ and $-(k+1)$, then I_j contains none of the oriented elements k and $k+1$. The same is true if $+j$ and $-(j+1)$ occur in between $+k$ and $-(k+1)$. Finally, if $+k$ and $-(k+1)$ occur in between $+j$ and $-(j+1)$, then I_j contains both oriented elements k and $k+1$. In all these cases $\rho(I_j)$ does not change the status of I_k by Lemma 9.5.1. \square

What happens when the inverted interval in the preceding lemma is a non-elementary interval I ? On the one hand, the if-part is still true: if I and I_k overlap, then $\rho(I)$ changes the status of I_k . To see this, suppose the overlap of I and I_k contains $+k$. Then, however, $-k$ must also be contained in I by the definition of a non-elementary interval. Therefore, I contains the oriented element k but not $k+1$. Hence $\rho(I)$ changes the status of I_k by Lemma 9.5.1. On the other hand, the only-if-part of the preceding lemma crucially depends on the fact that the inverted interval is an elementary interval. If a non-elementary interval I is contained in I_k , then $\rho(I)$ may change the status of I_k . For example, in the permutation

$$\pi = (+0, +2, -2, +1, -1, -3, +3, +5, -5, +6, -6, -4, +4, -7)$$

the elementary interval $I_2 = [+2, \dots, -3]$ is good. The interval $I = [+2, -2]$ is contained in I_2 , and in the permutation

$$\pi' = \pi\rho(I) = (+0, -2, +2, +1, -1, -3, +3, +5, -5, +6, -6, -4, +4, -7)$$

the elementary interval $I_2' = [+2, \dots, -3]$ is bad.

In what follows, the *overlap status* between two elementary intervals I_j and I_k is *true* if I_j and I_k overlap; otherwise it is *false*.

Lemma 9.5.3 *A reversal $\rho(I)$ changes the overlap status between two elementary intervals I_j and I_k if and only if I overlaps with both I_j and I_k .*

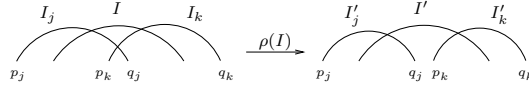


Figure 9.16: Before the reversal, I overlaps with both I_j and I_k , and the overlap status between I_j and I_k is true.

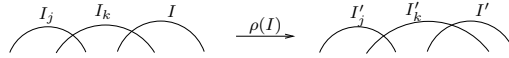


Figure 9.17: Before the reversal, I overlaps with I_k but not with I_j .

Proof Let $I_j = [p_j, \dots, q_j]$ and $I_k = [p_k, \dots, q_k]$. That is, p_j, q_j and p_k, q_k are the boundary elements of the intervals I_j and I_k , respectively. Note that $\{p_j, q_j\} = \{+j, -(j+1)\}$, $\{p_k, q_k\} = \{+k, -(k+1)\}$, and $\{p_j, q_j\} \cap \{p_k, q_k\} = \emptyset$. The overlap status between I_j and I_k is true if and only if their boundary elements occur in alternating order in π , i.e., their order is either p_j, p_k, q_j, q_k or p_k, p_j, q_k, q_j . If I overlaps with both I_j and I_k , then $\rho(I)$ inverts the order of exactly one boundary element of I_j and one boundary element of I_k . In other words, if the boundary elements occur in alternating order before the reversal $\rho(I)$, then this is not true afterwards (see Figure 9.16) and vice versa. Thus, if I overlaps with both I_j and I_k , then $\rho(I)$ changes the overlap status between I_j and I_k .

If I overlaps with only one of the intervals or with none of them, then the order of the boundary elements of I_j and I_k remains unchanged; see Figure 9.17. Therefore, the overlap status remains the same. \square

Definition 9.5.4 Let I_k be a *good* elementary interval of the permutation π . The *score* $\text{score}(I_k)$ of I_k is the number of good elementary intervals in the permutation $\pi\rho(I_k)$.

So the score of a good elementary interval I_k can be computed by inverting I_k and counting the number of good elementary intervals in the resulting permutation. The following theorem is due to Bergeron [37].

Theorem 9.5.5 *The reversal of a good elementary interval with maximum score does not create new bad components.*

Proof Let t be the total number of good elementary intervals in the permutation π , and for each good elementary interval I_i let $g(I_i)$ ($b(I_i)$, respectively) be the number of good (bad, respectively) elementary intervals that overlap with I_i .

Consider a good elementary interval I_k with maximum score. After the reversal $\rho(I_k)$, the oriented elements k and $k+1$ are adjacent; see Exercise 9.4.2. Furthermore, by Lemma 9.5.2, the reversal $\rho(I_k)$ changes the status of another elementary interval if and only if it overlaps with that interval. Putting it all together, we infer that $\text{score}(I_k) = t + b(I_k) - g(I_k) - 1$. For a proof by contradiction, suppose that $\rho(I_k)$ creates a new bad component C . So all elementary intervals (desire edges) in C are bad, but at least one of them, say I'_j , was good before the reversal. According to Lemma 9.5.2, I_j and I_k must overlap. Note that $\text{score}(I_j) = t + b(I_j) - g(I_j) - 1$.

We claim that every bad elementary interval I_i that overlaps with I_k must also overlap with I_j . Note that this directly implies $b(I_j) \geq b(I_k)$. We prove the claim by contradiction. Suppose that there is a bad elementary interval I_i that overlaps with I_k but not with I_j . Because I_i and I_k as well as I_j and I_k overlap, it follows from Lemma 9.5.3 that the elementary intervals I'_i and I'_j overlap in the permutation $\pi' = \pi\rho(I_k)$. Moreover, by Lemma 9.5.2 I'_i is a good elementary interval in π' . In conclusion, the component C contains the good elementary interval (desire edge) I'_i . Hence C is a good component. This contradiction proves the claim.

Analogously, one can show that each good elementary interval I_i that overlaps with I_j must also overlap with I_k . (If there were a good elementary interval I_i that overlaps with I_j but not with I_k , then I'_i and I'_j would still overlap in the permutation $\pi' = \pi\rho(I_k)$, and C would be a good component.) Consequently, $g(I_j) \leq g(I_k)$. Now we distinguish the following cases:

- If $b(I_j) = b(I_k)$ and $g(I_j) = g(I_k)$, then I_j and I_k overlap with the same elementary intervals. According to Lemma 9.5.3, this means that I'_j does not overlap with any other elementary interval in π' . By Lemma 9.4.4, it must have the form $[+j, -(j+1)]$ or $[-(j+1), +j]$. Thus, it is neither good nor bad. This contradicts our assumption that I'_j is a bad elementary interval that belongs to the bad component C .
- If $b(I_j) > b(I_k)$ or $g(I_j) < g(I_k)$, then

$$\text{score}(I_j) = t + b(I_j) - g(I_j) - 1 > t + b(I_k) - g(I_k) - 1 = \text{score}(I_k)$$

contradicts the fact that the score of I_k is maximum.

To sum up, $\rho(I_k)$ does not create a new bad component. □

Corollary 9.5.6 *If the permutation π has no bad component, then the reversal distance d_π satisfies:*

$$d_\pi = n + 1 - c_\pi$$

where c_π is the number of cycles in the reality-desire diagram of π .

Algorithm 9.2 Sorting a permutation π without bad components.

```

while  $\pi \neq id$  do
   $max \leftarrow 0$ 
  for  $k \leftarrow 0$  to  $n$ 
    compute  $score(I_k)$ 
    if  $score(I_{max}) < score(I_k)$  then
       $max \leftarrow k$ 
  output  $\rho(I_{max})$ 
   $\pi \leftarrow \pi \rho(I_{max})$ 

```

Proof Corollary 9.3.5 states that $n + 1 - c_\pi$ is a lower bound for d_π , and Algorithm 9.2 achieves this lower bound: the reversal of a good elementary interval with maximum score increases the number of cycles by one because it creates a new adjacency and it does not create new bad components by Theorem 9.5.5. \square

On the one hand, it is not difficult to see that $O(n^3)$ is an upper bound for the run time of Algorithm 9.2. On the other hand, Ozery-Flato and Shamir [256] showed that $\Omega(n^3)$ is a lower bound. Hence the worst-case time complexity of Algorithm 9.2 is $\Theta(n^3)$.

9.6 Dealing with bad components

According to Lemma 9.3.4, if a reversal ρ acts on two reality edges from the same bad component or from two different components, then this does not increase the number of cycles. Therefore, we now investigate how bad components can be dealt with. There are two alternative ways to turn bad components into good components. The first one is described in Lemma 9.6.1.

Lemma 9.6.1 *Let C be a bad component and let I_k be an elementary interval (a desire edge) in C . The reversal $\rho(I_k)$ turns C into a good component and the number of cycles remains unchanged.*

Proof Because C is a bad component, it consists solely of bad cycles. By the converse statement of Lemma 9.4.4, the bad elementary interval I_k must overlap with another elementary interval, say I_j . Clearly, I_j also belongs to C ; hence it is also bad. According to Lemma 9.5.2, the elementary interval I'_j after the reversal $\rho(I_k)$ is good and so is the component to which it belongs.

Furthermore, $\rho(I_k)$ acts on two convergent reality edges, so the number of cycles remains unchanged by Lemma 9.3.4. \square

Henceforth, a reversal of an elementary interval that turns a bad component into a good component is called a *flipping reversal*. The next lemma says that such a flipping reversal has no influence on the other components.

Lemma 9.6.2 *Let C_1, \dots, C_m be the components of a permutation π . A reversal $\rho(I_k)$ of an elementary interval I_k that belongs to C_1 does not affect the remaining components C_2, \dots, C_m , i.e., they are also components of the permutation $\pi' = \pi\rho(I_k)$ and their status remains unchanged.*

Proof Fix a component $C_i \neq C_1$. Let I_j be an elementary interval (a desire edge) that belongs to C_i . I_j does not overlap with I_k because C_1 and C_i are different components. According to Lemmata 9.5.2 and 9.5.3, the reversal $\rho(I_k)$ does neither change the status of I_j nor does it change the overlaps of I_j with other elementary intervals. Therefore, the component C_i is still a component after the reversal, and it has the same status as before. \square

Let bc_π be the number of bad components of the permutation π . As we have seen, a flipping reversal decreases the measure $n+1-c_\pi+bc_\pi$ by one. If all bad components are eliminated by flipping reversals (Lemma 9.6.1) and the resulting permutation is sorted by Algorithm 9.2, then this requires exactly $n+1-c_\pi+bc_\pi$ reversals. However, this approach is not optimal. (To be precise, $n+1-c_\pi+bc_\pi$ is an upper bound for d_π but not a lower bound.) To see this, consider the permutation $\pi = (\vec{0}, \vec{8}, \vec{1}, \vec{6}, \vec{2}, \vec{4}, \vec{3}, \vec{5}, \vec{7}, \vec{9})$. It has three bad components, each of which is a cycle; see Figure 9.18. Thus, the approach described above needs $n+1-c_\pi+bc_\pi = 9$ reversals to sort π . However, if one first applies the reversal $\rho([-1, \dots, +2])$ to π , then $\pi' = \pi\rho([-1, \dots, +2])$ has just one good component, which consists of two cycles; see Figure 9.19. Since π' can be sorted with $n+1-c_{\pi'}+bc_{\pi'} = 7$ reversals, we infer that π can be sorted with 8 reversals.

The reversal $\rho([-1, \dots, +2])$ applied to π is called a *merging reversal* because it merges the two bad components containing the reality edges that it acts on. A merging reversal is always as good as a flipping reversal because it decreases the measure $n+1-c_\pi+bc_\pi$ by at least one (it eliminates at least two bad components but also one cycle). If it eliminates more than two bad components (as in our example above), then it is better than a flipping reversal because it decreases the measure $n+1-c_\pi+bc_\pi$ by more than one.

In the sequel, we say that a component C *overlaps* with an interval I if C contains an elementary interval I_k that overlaps with I .

Lemma 9.6.3 *If a reversal $\rho(I)$ acts on reality edges of two different components C_ℓ and C_r (where C_ℓ occurs before C_r in the permutation π), then C_ℓ and C_r as well as all components that overlap with I are merged into one big component C' in the permutation $\pi' = \pi\rho(I)$, but the other components*

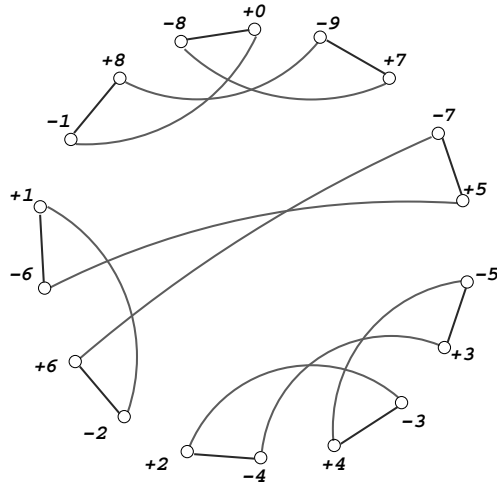


Figure 9.18: The permutation π has three bad components (cycles).

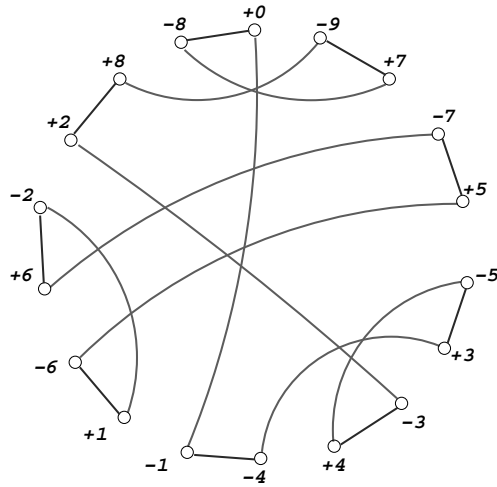


Figure 9.19: The permutation $\pi' = \pi\rho([-1, \dots, +2])$ has one good component consisting of two cycles.

are not affected. Furthermore, if one of the components C_ℓ and C_r is bad, then C' is good.

Proof Let e be the left reality edge on which $\rho(I)$ acts. Since e belongs to a cycle of the left component C_ℓ and one endpoint of e is in I and the other endpoint is outside I , it follows that there must be a desire edge in the cycle with one endpoint inside I and one endpoint outside I . The elementary interval corresponding to this desire edge overlaps with the interval I . Let I_j be this elementary interval that belongs to C_ℓ and overlaps with I . Analogously, there is an elementary interval I_k that belongs to C_r and overlaps with I . Because I_j and I_k do not overlap before the reversal, we infer from Lemma 9.5.3 that they overlap after the reversal. Therefore, $\rho(I)$ merges C_ℓ and C_r into one big component C' . Similar reasoning shows that any other component that overlaps with I must be part of C' (after the reversal). Moreover, because I overlaps with I_j and I_k , the reversal $\rho(I)$ changes their status. So if at least one of the components C_ℓ and C_r is bad, then C' is a good component because it contains a good elementary interval.

Now consider a component C that does not overlap with I . We claim that $\rho(I)$ does not change the status of any elementary interval I_i of C . For a proof by contradiction, suppose that $\rho(I)$ changes the status of I_i . By Lemma 9.5.1, I must contain one of the oriented elements i or $i+1$ but not the other. Without loss of generality, suppose that $+i$ occurs before $-(i+1)$ in π , i.e., $I_i = [+i, \dots, -(i+1)]$. If I contains the oriented element i but not $i+1$, then $I = [+i, -i, \dots]$ must be a proper subinterval of $I_i = [+i, -i, \dots, -(i+1)]$ (recall that I does not overlap with I_i). Because I_j overlaps with I , it follows that I_j must also overlap with I_i . However, this contradicts the fact that C_ℓ and C are different components. Hence $\rho(I)$ does not change the status of any elementary interval I_i of C . Furthermore, according to Lemma 9.5.3 the reversal $\rho(I)$ does not change the overlap status between I_i and any other elementary interval. Summing up, the reversal $\rho(I)$ does not affect components that do not overlap with I . \square

9.6.1 Hurdles

Using the linear-time Algorithm 9.1 (page 450), we can tag every node $v[i]$ at position i in the reality-desire diagram with its status $v[i].status$ and an identifier $v[i].id$ that indicates to which component the node belongs. By enumerating all component identifiers, we obtain a circular sequence of component identifiers. Then, this sequence is compacted:

- Every identifier belonging to a good component is discarded.
- The following process is repeated until all consecutive component identifiers are different: If two consecutive identifiers belong to the same (bad) component, then one of them is discarded.

Algorithm 9.3 Constructing a compacted circular sequence of component identifiers.

```

for  $i \leftarrow 1$  to  $2n + 1$  do
  if  $v[i].status = bad$  and  $v[i].id \neq v[i + 1].id$  then
    output  $v[i].id$ 
  if  $v[2n + 2].status = bad$  and  $v[2n + 2].id \neq v[1].id$  then
    output  $v[2n + 2].id$ 

```

Of course, the compacted circular sequence of component identifiers can be obtained more directly, as Algorithm 9.3 shows.

The next definition stems from [172].

Definition 9.6.4 A bad component C is called a *hurdle* if its component identifier occurs exactly once in the compacted circular sequence of component identifiers. Otherwise, C is called a *non-hurdle*. The number of hurdles of π is denoted by h_π .

As an example, let us reconsider the permutation π from Figure 9.18. The circular sequence of (bad) component identifiers is

$$1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 2, 2, 1, 1$$

and the compacted circular sequence is 1, 2, 3, 2. Consequently, C_1 and C_3 are hurdles, whereas C_2 is a non-hurdle.

As shown above, non-hurdles can be removed *en passant* by merging hurdles. By contrast, we shall next see that it needs an extra reversal to eliminate a hurdle.

Lemma 9.6.5 *A reversal can remove at most two hurdles. If it removes two hurdles, it decreases the number of cycles by one.*

Proof Lemma 9.6.2 states that if a reversal $\rho(I)$ acts on two reality edges of the same component, say C , then the remaining components are not affected. In this case, $\rho(I)$ can remove at most one hurdle (namely C).

Lemma 9.6.3 states that if a reversal $\rho(I)$ acts on reality edges of two different components, then all components that overlap with I are merged into one big component. Of these, at most two bad components can be hurdles (only one can overlap I on the left-hand side and only one can overlap I on the right-hand side). Lemma 9.6.3 also states that components that do not overlap with I are unaffected by the reversal $\rho(I)$. Hence the reversal $\rho(I)$ removes at most two hurdles. If it removes two hurdles, then it also decreases the number of cycles by one; see Lemma 9.3.4. \square

Theorem 9.6.6 *We have*

$$d_\pi \geq n + 1 - c_\pi + h_\pi$$

Proof Let $\pi' = \pi\rho$ be the permutation after the reversal ρ has been applied to the permutation π . Furthermore, let $c_{\pi'}$ and $h_{\pi'}$ be the number of cycles and hurdles in the reality-desire diagram of π' , respectively. Define

$$\begin{aligned} \Delta(\rho) &= (n + 1 - c_\pi + h_\pi) - (n + 1 - c_{\pi'} + h_{\pi'}) \\ &= (c_{\pi'} - c_\pi) + (h_\pi - h_{\pi'}) \end{aligned}$$

We show $\Delta(\rho) \leq 1$ by case analysis.

1. If the reversal ρ acts on two divergent reality-edges of the same cycle c , then $c_{\pi'} = c_\pi + 1$ by Lemma 9.3.4. Let C be the component to which the cycle c belongs. According to Lemma 9.6.2, ρ does neither remove nor change the status of the remaining components. Because C is a good component, it follows $h_\pi \leq h_{\pi'}$. Hence $\Delta(\rho) \leq 1$.
2. If the reversal ρ acts on two convergent reality-edges of the same cycle c , then $c_{\pi'} = c_\pi$ by Lemma 9.3.4. Again, we take advantage of the fact that ρ can neither remove nor change the status of a component that is different from the component C to which the cycle c belongs. So if C is not a hurdle, then $h_\pi \leq h_{\pi'}$. If C is a hurdle and ρ removes it, we obtain $h_{\pi'} \geq h_\pi - 1$. In both cases, we have $\Delta(\rho) = (c_{\pi'} - c_\pi) + (h_\pi - h_{\pi'}) = 0 + (h_\pi - h_{\pi'}) \leq 1$.
3. If the reversal ρ acts on reality-edges of different cycles c_1 and c_2 , then $c_{\pi'} = c_\pi - 1$ by Lemma 9.3.4. According to Lemma 9.6.5, the inequality $(h_\pi - h_{\pi'}) \leq 2$ holds. Consequently, $\Delta(\rho) = (c_{\pi'} - c_\pi) + (h_\pi - h_{\pi'}) = -1 + (h_\pi - h_{\pi'}) \leq 1$.

Because $\Delta(\rho) \leq 1$ implies that every reversal can decrease the quantity $n + 1 - c_\pi + h_\pi$ by at most one, we conclude that every sequence of reversals that sorts a permutation π into the identity permutation must contain at least $n + 1 - c_\pi + h_\pi$ reversals. In other words, $n + 1 - c_\pi + h_\pi$ is a lower bound on the reversal distance d_π . This proves the theorem. \square

Next we try to develop an algorithm that achieves this bound. By Lemma 9.6.5 two hurdles can be removed by a merging reversal at the cost of increasing the number of cycles by one. So merging two hurdles decreases the quantity $n + 1 - c_\pi + h_\pi$ by one *except* for the case when it creates a *new* hurdle.

Exercise 9.6.7 Give an example of a permutation in which the merging of two hurdles creates a new hurdle.

Definition 9.6.8 Two hurdles H_1 and H_2 are *non-consecutive* if in the compacted circular sequence of component identifiers there is at least—both in clockwise and counterclockwise directions—one other hurdle between them.

Lemma 9.6.9 *A reversal that merges two non-consecutive hurdles H_1 and H_2 does not create a new hurdle.*

Proof Suppose that a non-hurdle NH is turned into a hurdle H' by the merging reversal $\rho(I)$. By Definition 9.6.4, the component identifier NH_{id} of NH occurs at least twice in the compacted circular sequence of component identifiers of π , whereas the component identifier H'_{id} of H' occurs exactly once in the compacted circular sequence of component identifiers of $\pi' = \pi\rho(I)$.

According to Lemma 9.6.3, if the reversal $\rho(I)$ merges two hurdles H_1 and H_2 , then all components that overlap with I are merged into one good component, whereas the other components are not affected. So if the non-hurdle NH would overlap with I , then it would disappear. If the non-hurdle NH is contained in I , then it clearly remains a non-hurdle. Otherwise, if NH is not contained in I , then it can become a hurdle only if there is no bad component contained in I . However, since H_1 and H_2 are non-consecutive hurdles, there is at least one other hurdle in between H_1 and H_2 , both in clockwise direction and in counterclockwise direction. So there is at least one hurdle contained in I . \square

So as long as the number of hurdles satisfies $h_\pi \geq 4$, we can successively merge non-consecutive hurdles until there are only two or three left. Exercise 9.6.10 asks you to prove that the former case causes no problem.

Exercise 9.6.10 Show the following statement: If $h_\pi = 2$, then a reversal that merges the two hurdles does not create a new hurdle.

However, we may run into trouble when dealing with permutations having three hurdles because there is a particular constellation called a fortress.

9.6.2 A fortress

Before we can explain what a fortress is, we must first introduce one more preliminary notion.

Definition 9.6.11 A hurdle H is called a *super hurdle* if the removal of H would turn a non-hurdle into a hurdle; otherwise H is called a *simple hurdle*.

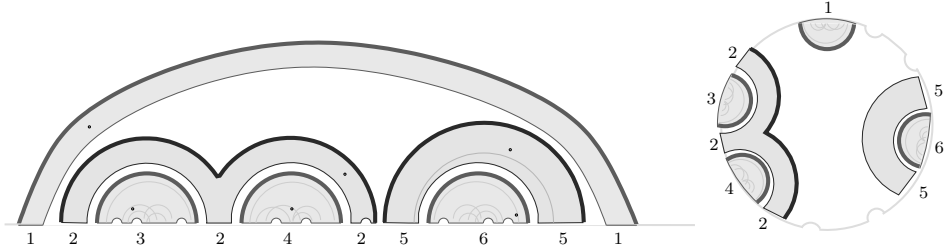


Figure 9.20: In this schematic diagram, there are six bad components. The compacted circular sequence of component identifiers is 1, 2, 3, 2, 4, 2, 5, 6, 5 (right-hand side). The components 2 and 5 are non-hurdles, whereas 1, 3, 4, 6 are hurdles. The hurdle 6 is a super hurdle, while 1, 3, 4 are simple hurdles.

Figure 9.20 explains the difference between super hurdles and simple hurdles.

Lemma 9.6.12 *Let the compacted circular sequence of component identifiers be given. A hurdle H is a super hurdle if and only if its identifier H_{id} is flanked by the same component identifier NH_{id} , and NH_{id} occurs exactly twice in the sequence.*

Proof Recall that a bad component is a hurdle H if and only if its component identifier occurs exactly once in the compacted circular sequence of component identifiers. If H_{id} is flanked by the same component identifier NH_{id} , and NH_{id} occurs exactly twice in the sequence, then after the removal of H the component identifier NH_{id} occurs exactly once in the resulting sequence (because one of the two consecutive identifiers NH_{id} is discarded). Since NH is a hurdle afterwards, H is a super hurdle. On the other hand, if H_{id} is flanked by two different component identifiers, then H is certainly not a super hurdle. Furthermore, if H_{id} is flanked by the same component identifier NH_{id} , but NH_{id} occurs more than twice in the sequence, then after the removal of H the bad component NH is still a non-hurdle because its identifier occurs more than once in the resulting sequence. \square

If a super hurdle H is flanked by the non-hurdle NH , we say that NH shields H .

Again, we consider the permutation π from Figure 9.18 (page 457) as an example. We have seen that its compacted circular sequence of component identifiers is 1, 2, 3, 2. The identifier 3 is obviously flanked by 2 on both sides, and the same is true for 1 because the sequence is circular.

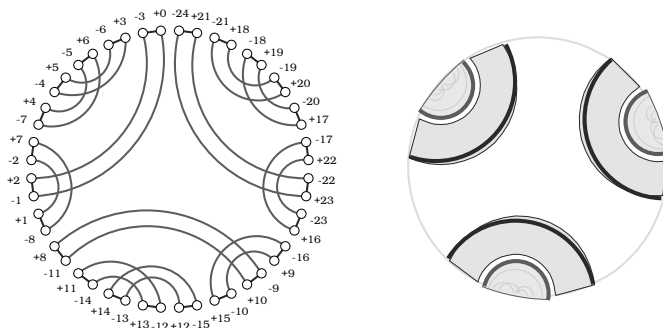


Figure 9.21: The smallest possible fortress is shown on the left-hand side, and a schematic diagram of this fortress is shown on the right. If one of the super hurdles is removed by a flipping reversal, then its shielding non-hurdle becomes a hurdle. If one merges two of the super hurdles (and their shielding non-hurdles), then the remaining non-hurdle becomes a hurdle.

Because 2 occurs exactly twice in the compacted sequence, it follows that both components C_1 and C_3 are super hurdles.

Let us resume our discussion on clearing hurdles. If there are three hurdles and at least one of them is simple, then we can remove the simple hurdle by a flipping reversal. Because this does not create a new hurdle, the quantity $n+1-c_\pi+h_\pi$ decreases by one, and the remaining two hurdles can safely be merged; cf. Exercise 9.6.10. However, if there are three hurdles and all of them are super hurdles, both possibilities (a) flipping one of them or (b) merging two of them will create a new hurdle! Such a constellation is called a fortress. The definition below is a bit more general because the sheer existence of such a constellation has the consequence that we must carefully revise our procedure of clearing hurdles.

Definition 9.6.13 A permutation π is called a *fortress* if it has an odd number of hurdles, each of which is a super hurdle.

Figure 9.21 shows the smallest possible fortress; cf. [289, Figure 7.19]. It also gives the idea why one needs an extra reversal to eliminate a fortress. Theorem 9.6.14 formalizes this fact.

Theorem 9.6.14 We have

$$d_\pi \geq n+1-c_\pi+h_\pi+f_\pi$$

where f_π is defined by

$$f_\pi = \begin{cases} 1 & \text{if } \pi \text{ is a fortress} \\ 0 & \text{otherwise} \end{cases}$$

Proof Let $\pi' = \pi\rho$ be the permutation after the reversal ρ has been applied to the permutation π , and let $c_{\pi'}$, $h_{\pi'}$, and $f_{\pi'}$ be the number of cycles, hurdles, and the fortress indicator of π' , respectively. Define

$$\begin{aligned} \Delta(\rho) &= (n+1 - c_\pi + h_\pi + f_\pi) - (n+1 - c_{\pi'} + h_{\pi'} + f_{\pi'}) \\ &= (c_{\pi'} - c_\pi) + (h_\pi - h_{\pi'}) + (f_\pi - f_{\pi'}) \end{aligned}$$

We show that $\Delta(\rho) \leq 1$ by case analysis.

If π is not a fortress or both π and π' are fortresses, we have $(f_\pi - f_{\pi'}) \leq 0$. In these cases, the theorem follows from Theorem 9.6.6. Let us consider the remaining case: π is a fortress, but π' is not. As in the proof of Theorem 9.6.6, we proceed by case distinction.

1. If the reversal ρ acts on two divergent reality-edges of the same cycle, then $c_{\pi'} = c_\pi + 1$ according to Lemma 9.3.4. By Lemma 9.6.2, all super hurdles in π are also super hurdles in π' . So ρ can destroy the fortress solely by *increasing* the number of hurdles (e.g. $h_{\pi'} = h_\pi + 1$ implies that $h_{\pi'}$ is even, so that π' is not a fortress). This yields $\Delta(\rho) = (c_{\pi'} - c_\pi) + (h_\pi - h_{\pi'}) + (f_\pi - f_{\pi'}) \leq 1 + (h_\pi - h_{\pi'}) + 1 \leq 1$ because $h_\pi - h_{\pi'} \leq -1$.
2. If the reversal ρ acts on two convergent reality-edges of the same cycle c , then $c_{\pi'} = c_\pi$ by Lemma 9.3.4. For an indirect proof, suppose that $\Delta(\rho) > 1$. This is only possible if $h_\pi - h_{\pi'} \geq 1$. Once again, we use the fact that ρ can neither remove nor change the status of a component that is different from the component C to which the cycle c belongs, and infer that C must be a hurdle and ρ removes it. However, since π is a fortress, all hurdles in the reality-desire diagram of π are super hurdles. By definition, the removal of a super hurdle creates a new hurdle. To sum up, if ρ removes the super hurdle C , it follows that $h_\pi - h_{\pi'} = 0$. This contradiction shows that our supposition $\Delta(\rho) > 1$ was wrong, i.e., $\Delta(\rho) \leq 1$ must be true.
3. If the reversal ρ acts on reality-edges of different cycles c_1 and c_2 , then $c_{\pi'} = c_\pi - 1$ according to Lemma 9.3.4. Again, suppose that $\Delta(\rho) > 1$. This is only possible if $h_\pi - h_{\pi'} = 2$ (recall that $h_\pi - h_{\pi'} \leq 2$ by Lemma 9.6.5). That is, π' has an odd number of hurdles. So $\rho(I)$ removes two super hurdles H_1 and H_2 by merging them and all other components that overlap with I into one good component; see Lemma 9.6.3. It is not difficult to verify that none of the non-hurdles shielding the remaining super hurdles can overlap with I . Because all components

Algorithm 9.4 Sort a permutation π by reversals.

```

repeat
  compute components, hurdles, simple hurdles, and super hurdles
  if there is a bad component then      /*  $h_\pi > 0$  */
    if  $h_\pi$  is even then
      if  $h_\pi \geq 4$  then
        output a reversal  $\rho$  merging two non-consecutive hurdles
      else      /*  $h_\pi = 2$  */
        output a reversal  $\rho$  merging the two hurdles
      else      /*  $h_\pi$  is odd */
        if there is a simple hurdle then
          output a reversal  $\rho$  flipping this simple hurdle
        else      /* fortress */
          if  $h_\pi \geq 5$  then
            output a reversal  $\rho$  merging two non-consecutive hurdles
          else      /*  $h_\pi = 3$  */
            output a reversal  $\rho$  merging any two hurdles
       $\pi \leftarrow \pi\rho$ 
  until there is no bad component
  apply Algorithm 9.2 to  $\pi$ 

```

that do not overlap with I are unaffected by the reversal $\rho(I)$ (Lemma 9.6.3), none of the remaining super hurdles is turned into a simple hurdle. In other words, π' is a fortress. This contradiction shows that our supposition $\Delta(\rho) > 1$ was wrong.

Because $\Delta(\rho) \leq 1$ implies that every reversal can decrease the quantity $(n + 1 - c_\pi + h_\pi + f_\pi)$ by at most one, we conclude $d_\pi \geq n + 1 - c_\pi + h_\pi + f_\pi$. \square

In order to show that the lower bound given in Theorem 9.6.6 is tight, we now give an algorithm that achieves this lower bound. Clearly, the algorithm must avoid the creation of a new fortress. Algorithm 9.4 does so by a simple case differentiation. When applied to a permutation π , it first computes components, hurdles, simple hurdles, and super hurdles. If π has no bad component, then it is sorted by Algorithm 9.2. Otherwise, there must be at least one hurdle and Algorithm 9.4 proceeds as follows:

- If the number h_π of hurdles is even, then π cannot be a fortress. If $h_\pi = 2$, the algorithm merges the two hurdles; otherwise it merges two non-consecutive hurdles. In both cases, this removes two (old) hurdles but also a cycle. Furthermore, it does not create a new hurdle by Lemma 9.6.9 and Exercise 9.6.10. Consequently, there are $h_\pi - 2$ hurdles afterwards, so the resulting permutation is not a fortress. To sum up, the measure $n + 1 - c_\pi + h_\pi + f_\pi$ decreases by 1.

- If the number h_π of hurdles is odd but there is a simple hurdle, then π cannot be a fortress. In this case, the algorithm eliminates a simple hurdle by a flipping reversal. This neither creates a new hurdle nor does it change the number of cycles. Because there are $h_\pi - 1$ hurdles afterwards and $h_\pi - 1$ is even, it follows that the resulting permutation cannot be a fortress. Again, the measure $n + 1 - c_\pi + h_\pi + f_\pi$ decreases by 1.
- If the number h_π of hurdles is odd and there is no simple hurdle, then π must be a fortress. If $h_\pi \geq 5$, the algorithm merges two non-consecutive hurdles. Although the resulting permutation is still a fortress (cf. case (3) in the proof of Theorem 9.6.14), the measure $n + 1 - c_\pi + h_\pi + f_\pi$ decreases by 1. If $h_\pi = 3$, then the algorithm merges any two hurdles. This removes two old hurdles (and a cycle) but also creates a new hurdle. So the resulting permutation has two hurdles and thus cannot be a fortress. Again, the measure $n + 1 - c_\pi + h_\pi + f_\pi$ decreases by 1.

Now we are in a position to state the main theorem, which is due to Hannenhalli and Pevzner [145].

Theorem 9.6.15 *The reversal distance d_π of a permutation π can be computed by the formula*

$$d_\pi = n + 1 - c_\pi + h_\pi + f_\pi$$

Proof Theorem 9.6.14 states that $n + 1 - c_\pi + h_\pi + f_\pi$ is a lower bound for d_π , and Algorithm 9.4 achieves this bound because each of the computed reversals decreases the measure $n + 1 - c_\pi + h_\pi + f_\pi$ by one. \square

Corollary 9.6.16 *The reversal distance d_π of a permutation π can be computed in linear time.*

Proof According to Theorem 9.6.15, $d_\pi = n + 1 - c_\pi + h_\pi + f_\pi$. We have seen in Section 9.4.2 that cycles and components can be determined in linear time. Furthermore, it is clear from Section 9.6.1 that hurdles and super hurdles can also be identified in linear time. Therefore, the values c_π , h_π , and f_π can be computed in linear time. \square

The worst-case time complexity of the repeat-loop of Algorithm 9.4 is $O(n^2)$ because in each iteration it computes components, hurdles, simple hurdles, and super hurdles. Kaplan et al. [172] observed that this is overkill: it suffices to compute them only once and store a list of hurdles in the order they occur in the compacted circular sequence of component identifiers. To cite Kaplan et al. [172]:

At the next stage this list is used to identify correct hurdles to merge.

This is because the sequence of reversals that clears all hurdles can be determined in advance:

1. If the number h_π of hurdles is even, then the algorithm merges non-consecutive hurdles until there are two hurdles left, which are subsequently merged.
2. If h_π is odd and there is a simple hurdle, then this simple hurdle is removed and the algorithm proceeds as in case (1).
3. If π is a fortress, then the algorithm merges non-consecutive hurdles until there are three hurdles left. Two of them are merged, creating a new hurdle, and the third hurdle is then merged with the new hurdle.

Consequently all bad components can be removed in linear time. Bergeron et al. [38] presented an alternative linear-time algorithm.

9.7 Sorting by reversals in quadratic time

Algorithm 9.4 runs in $\Theta(n^3)$ time because Algorithm 9.2 (page 455) does. We draw the conclusion that the whole sorting algorithm can be made to run in $O(n^2)$ time provided that a permutation without bad components can be sorted in $O(n^2)$ time. Kaplan et al. [172] showed that this is indeed possible, and we follow their presentation below.

For every permutation π of size n , we partition its set of elementary intervals $\mathcal{I} = \{I_0, \dots, I_n\}$ into the set \mathcal{I}_g of all good and the set \mathcal{I}_b of all bad elementary intervals.

Definition 9.7.1 A *happy clique* is a non-empty set $\mathcal{H} \subseteq \mathcal{I}_g$ so that:

- The elements of \mathcal{H} are pairwise overlapping, i.e., any two good elementary intervals $I_i, I_j \in \mathcal{H}$ overlap.
- For every good elementary interval $I_k \notin \mathcal{H}$ that overlaps with a good elementary interval $I_i \in \mathcal{H}$ there is another good elementary interval $I_l \notin \mathcal{H}$ that overlaps with I_k but not with I_i .

Let I_i be an elementary interval of a permutation π . As in Theorem 9.5.5, $g(I_i)$ ($b(I_i)$, respectively) denotes the number of good (bad, respectively) elementary intervals that overlap with I_i .

Theorem 9.7.2 Let $\mathcal{H} \subseteq \mathcal{I}_g$ be a happy clique and let $I_k \in \mathcal{H}$ so that $b(I_l) \leq b(I_k)$ for any $I_l \in \mathcal{H}$. Then the reversal of I_k does not generate new bad components.

Proof For a proof by contradiction, suppose that $\rho(I_k)$ generates a new bad component C' . Thus, C' has only bad elementary intervals and at least one of them, say I'_j , originates from a good elementary interval I_j . That is, I_j and I_k overlap. By the definition of a happy clique, if I_j were not an element of \mathcal{H} , then there would be another good elementary interval $I_l \notin \mathcal{H}$ that overlaps with I_j but not with I_k . Because $\rho(I_k)$ has no effect on I_l , $I'_l = I_l$ would still be good after the reversal, and according to Lemma 9.5.3 I'_l would still overlap with I'_j . In other words, the component C' would be good. These arguments show that I_j must be in \mathcal{H} . It follows exactly as in the proof of Theorem 9.5.5 that every bad elementary interval that overlaps with I_k must also overlap with I_j . Hence $b(I_j) \geq b(I_k)$. It is a precondition that conversely $b(I_j) \leq b(I_k)$ holds true. All in all, we have $b(I_j) = b(I_k)$.

Again, as in the proof of Theorem 9.5.5 one can show that every good elementary interval that overlaps with I_j must also overlap with I_k . Thus, $g(I_j) \leq g(I_k)$. A proof by contradiction will show that $g(I_j) < g(I_k)$ is impossible and hence $g(I_j) = g(I_k)$ must hold. So suppose $g(I_j) < g(I_k)$. Then, there must be a good elementary interval I_i that overlaps with I_k but not with I_j . We stress that after the reversal $\rho(I_k)$ the resulting intervals I'_i and I'_j must overlap by Lemma 9.5.3 because both I_i and I_j overlap with I_k but not each other. Moreover, the interval I_i cannot be a member of the happy clique \mathcal{H} . (If it were, then it would overlap both I_j and I_k because $I_j, I_k \in \mathcal{H}$.) According to the definition of a happy clique, there is another good elementary interval $I_l \notin \mathcal{H}$ that overlaps with I_i but not with I_k . Again, because $\rho(I_k)$ has no effect on I_l , $I'_l = I_l$ is still good after the reversal, and I'_l still overlaps with I'_i according to Lemma 9.5.3. To sum up, I'_l is good and overlaps with I'_i , while I'_i overlaps with I'_j . We derive as a consequence that I'_j belongs to a good component. This contradiction proves $g(I_j) = g(I_k)$.

The rest of the proof is verbatim the same as in Theorem 9.5.5: I_j and I_k overlap with the same elementary intervals. According to Lemma 9.5.3, this means that I'_j does not overlap with any other elementary interval in π' . By Lemma 9.4.4, it must have the form $[+j, -(j+1)]$ or $[-(j+1), +j]$. Thus, it is neither good nor bad. This contradicts our assumption that I'_j is a bad elementary interval that belongs to the bad component C' . \square

9.7.1 Finding a happy clique

In the following, for every elementary interval I_i , let $s(I_i)$ and $e(I_i)$ denote the start and end position of I_i in $\pi = (+0, \pi_1, \dots, \pi_n, -(n+1))$. That is, $s(I_i)$ and $e(I_i)$ are natural numbers between 1 and $2n+2$. Moreover, let $\mathcal{I}_g = \{\tilde{I}_1, \dots, \tilde{I}_m\}$ be the set of good elementary intervals. Without loss of generality, we may assume that $\tilde{I}_1, \dots, \tilde{I}_m$ are ordered according to their start positions, i.e., $s(\tilde{I}_1) < s(\tilde{I}_2) < \dots < s(\tilde{I}_m)$. (It is an easy exercise to show

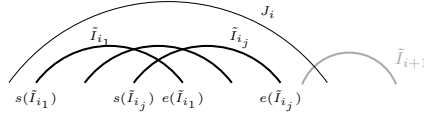


Figure 9.22: Finding a happy clique: Case (1).

that the left-to-right order of all good elementary intervals can be obtained in linear time.) Algorithm 9.5 computes a happy clique by scanning the good elementary intervals from left to right. During the scan, it maintains a happy clique \mathcal{H}_i of the set $\{\tilde{I}_1, \dots, \tilde{I}_i\}$: if $|\mathcal{H}_i| = j$ and $\mathcal{H}_i = \{\tilde{I}_{i_1}, \dots, \tilde{I}_{i_j}\}$, where $1 \leq i_1 < \dots < i_2 < \dots < i_j \leq i$, then \mathcal{H}_i is represented by a linked list containing the intervals $\tilde{I}_{i_1}, \dots, \tilde{I}_{i_j}$ in increasing order of their start positions. Furthermore, the algorithm maintains a good interval J_i that contains all the intervals in \mathcal{H}_i provided that such an interval exists. If it does not exist, then J_i is undefined, denoted by $J_i = \perp$.

When the algorithm scans the next interval \tilde{I}_{i+1} , it proceeds according to the following case differentiation (the cases are illustrated in Figures 9.22 and 9.23):

- (1) If $e(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1})$, then \tilde{I}_{i+1} and all remaining good elementary intervals do not overlap with an interval from \mathcal{H}_i . In this case, the algorithm stops and returns \mathcal{H}_i as a happy clique.
- (2) If \tilde{I}_{i+1} overlaps with \tilde{I}_{i_j} but also with J_i , then \mathcal{H}_i is also a happy clique of the set $\{\tilde{I}_1, \dots, \tilde{I}_i, \tilde{I}_{i+1}\}$ and the algorithm scans the next interval \tilde{I}_{i+2} . Observe that in this case, the interval J_i acts as a “shield” for the current happy clique: as long as \tilde{I}_{i+1} overlaps with J_i , no harm is done to the happy clique \mathcal{H}_i .
- (3) Otherwise, \tilde{I}_{i+1} overlaps with \tilde{I}_{i_j} but not with J_i (Cases 3a and 3b) or \tilde{I}_{i+1} is contained in \tilde{I}_{i_j} (Case 3c).
 - (3a) If \tilde{I}_{i+1} overlaps with all intervals in \mathcal{H}_i , then \mathcal{H}_i can be expanded with \tilde{I}_{i+1} , i.e., the new happy clique is $\mathcal{H}_i \cup \{\tilde{I}_{i+1}\}$.
 - (3b) If \tilde{I}_{i+1} does not overlap with all intervals in \mathcal{H}_i , then $\{\tilde{I}_{i+1}\}$ is a happy clique of the set $\{\tilde{I}_1, \dots, \tilde{I}_i, \tilde{I}_{i+1}\}$.
 - (3c) If \tilde{I}_{i+1} is contained in \tilde{I}_{i_j} , then $\{\tilde{I}_{i+1}\}$ is the new happy clique, protected by the new shield $J_{i+1} = \tilde{I}_{i_j}$.

Algorithm 9.5 contains a description of this procedure in pseudo-code.

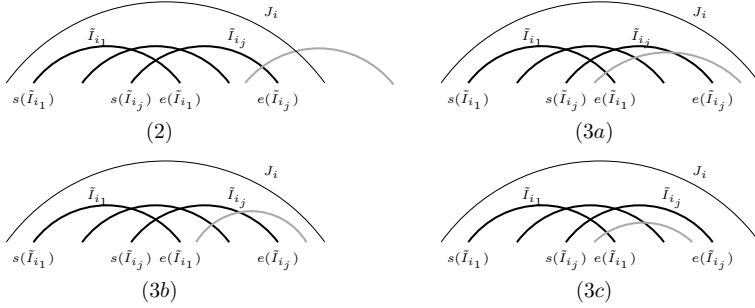


Figure 9.23: Finding a happy clique: Cases (2) and (3).

Algorithm 9.5 Finding a happy clique

```

 $\mathcal{H}_1 \leftarrow \{\tilde{I}_1\}$ 
 $J_1 \leftarrow \perp$ 
for  $i \leftarrow 1$  to  $m - 1$ 
  if  $e(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1})$  then return  $\mathcal{H}_i$           /* Case (1) */
  else          /*  $s(\tilde{I}_{i+1}) < e(\tilde{I}_{i_j})$  */
    if  $J_i \neq \perp$  and  $e(J_i) < e(\tilde{I}_{i+1})$  then      /* Case (2) */
       $\mathcal{H}_{i+1} \leftarrow \mathcal{H}_i$ 
       $J_{i+1} \leftarrow J_i$ 
    else          /*  $J_i = \perp$  or  $e(\tilde{I}_{i+1}) < e(J_i)$  */
      if  $e(\tilde{I}_{i_j}) < e(\tilde{I}_{i+1})$  then
        if  $s(\tilde{I}_{i+1}) < e(\tilde{I}_{i_1})$  then          /* Case (3a) */
           $\mathcal{H}_{i+1} \leftarrow \mathcal{H}_i \cup \{\tilde{I}_{i+1}\}$ 
           $J_{i+1} \leftarrow J_i$ 
        else          /*  $s(\tilde{I}_{i+1}) > e(\tilde{I}_{i_1})$ , Case (3b) */
           $\mathcal{H}_{i+1} \leftarrow \{\tilde{I}_{i+1}\}$ 
           $J_{i+1} \leftarrow J_i$ 
        else          /*  $e(\tilde{I}_{i+1}) < e(\tilde{I}_{i_j})$ , Case (3c) */
           $\mathcal{H}_{i+1} \leftarrow \{\tilde{I}_{i+1}\}$ 
           $J_{i+1} \leftarrow \tilde{I}_{i_j}$ 
  return  $\mathcal{H}_m$ 

```

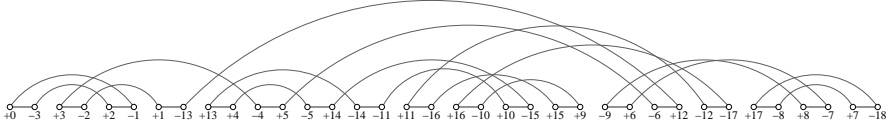


Figure 9.24: The reality-desire-diagram for Exercise 9.7.3.

Exercise 9.7.3 Apply Algorithm 9.5 to the example in Figure 9.24.

For a formal correctness proof, we need the following lemma.

Lemma 9.7.4 *Algorithm 9.5 maintains the following invariant for every i with $1 \leq i \leq m$:*

- (a) \mathcal{H}_i contains pairwise overlapping good elementary intervals. Moreover, if J_i is defined, then it is a good elementary interval that contains all intervals of \mathcal{H}_i .
- (b) If $J_i \neq \perp$, then every good elementary interval $\tilde{I}_k \notin \mathcal{H}_i$ with $k \leq i$ and $s(\tilde{I}_{i_j}) < s(\tilde{I}_k)$ overlaps with J_i . If $J_i = \perp$, then there is no such \tilde{I}_k .
- (c) Every good elementary interval $\tilde{I}_k \notin \mathcal{H}_i$ with $s(\tilde{I}_k) < s(\tilde{I}_{i_j})$ that overlaps with an interval from \mathcal{H}_i either overlaps with a good elementary interval \tilde{I}_l so that $e(\tilde{I}_l) < s(\tilde{I}_{i_1})$ or, provided that $J_i \neq \perp$, it overlaps with J_i .

Proof We prove the lemma by induction on i . The base case $i = 1$ is clear. In the inductive step, we show by case analysis that the lemma holds for $i + 1$, under the assumption that it holds for i .

- (1) $e(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1})$: All remaining good elementary intervals do not overlap with an interval from \mathcal{H}_i . Clearly, the invariant holds in this case. Note that in the remaining cases we have $s(\tilde{I}_{i+1}) < e(\tilde{I}_{i_j})$.
- (2) $J_i \neq \perp$ and $e(J_i) < e(\tilde{I}_{i+1})$: In this case, we have $\mathcal{H}_{i+1} = \mathcal{H}_i$ and $J_{i+1} = J_i$. By the inductive hypothesis and $\tilde{I}_{i+1} \notin \mathcal{H}_{i+1}$, it must be shown in the second part of the invariant: $s(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1})$ implies that \tilde{I}_{i+1} overlaps with J_{i+1} . It is not difficult to see that this is indeed the case, because
 - (i) $s(J_{i+1}) < s(\tilde{I}_{i_j})$ (as $J_{i+1} = J_i$ contains all intervals from $\mathcal{H}_{i+1} = \mathcal{H}_i$) in combination with $s(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1})$ has $s(J_{i+1}) < s(\tilde{I}_{i+1})$ as a consequence and
 - (ii) we have $e(J_i) < e(\tilde{I}_{i+1})$. The third part of the invariant holds by the inductive hypothesis in conjunction with $s(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1})$.
- (3) $J_i = \perp$ or $e(\tilde{I}_{i+1}) < e(J_i)$: This case is split into three subcases.

- (3a) $e(\tilde{I}_{i_j}) < e(\tilde{I}_{i+1})$ and $s(\tilde{I}_{i+1}) < e(\tilde{I}_{i_1})$: In this case, $\mathcal{H}_{i+1} = \mathcal{H}_i \cup \{\tilde{I}_{i+1}\}$ and $J_{i+1} = J_i$. For the first part of the invariant, we must show that \tilde{I}_{i+1} overlaps with each interval from \mathcal{H}_i . This is readily proven:

$$s(\tilde{I}_{i_1}) < \cdots < s(\tilde{I}_{i_j}) < s(\tilde{I}_{i+1}) < e(\tilde{I}_{i_1}) < \cdots < e(\tilde{I}_{i_j}) < e(\tilde{I}_{i+1})$$

Moreover, $J_{i+1} = J_i$ contains \tilde{I}_{i+1} as $s(J_i) < s(\tilde{I}_{i_1})$ and $e(\tilde{I}_{i+1}) < e(J_i)$. The second part of the invariant holds vacuously. In the third part, the inductive hypothesis implies that every good elementary interval $\tilde{I}_k \notin \mathcal{H}_{i+1}$ with $s(\tilde{I}_k) < s(\tilde{I}_{i_j})$ that overlaps with an interval from \mathcal{H}_{i+1} either overlaps with a good elementary interval \tilde{I}_l so that $e(\tilde{I}_l) < s(\tilde{I}_{i_1})$ or, provided that $J_{i+1} \neq \perp$, it overlaps with $J_{i+1} = J_i$. It remains to be shown that the same is true for every $\tilde{I}_k \notin \mathcal{H}_{i+1}$ with $s(\tilde{I}_{i_j}) < s(\tilde{I}_k) < s(\tilde{I}_{i+1})$. We observe that $e(\tilde{I}_{i_j}) < e(\tilde{I}_k) < e(\tilde{I}_{i+1})$ would mean that $\tilde{I}_k \in \mathcal{H}_{i+1}$. Consequently, $e(\tilde{I}_k) < e(\tilde{I}_{i_j})$ must hold and hence \tilde{I}_k is contained in \tilde{I}_{i_j} . In this case, however, case (3c) of the algorithm would have been applicable to \tilde{I}_k and \tilde{I}_{i_j} (i.e., $\{\tilde{I}_k\}$ is a happy clique and \tilde{I}_{i_j} is the interval containing this happy clique). Because case (3c) of the algorithm was not applied to \tilde{I}_k and \tilde{I}_{i_j} , also the third part of the invariant is satisfied.

- (3b) $e(\tilde{I}_{i_j}) < e(\tilde{I}_{i+1})$ and $s(\tilde{I}_{i+1}) > e(\tilde{I}_{i_1})$: In this case, $\mathcal{H}_{i+1} = \{\tilde{I}_{i+1}\}$ and $J_{i+1} = J_i$. The first part of the invariant is satisfied because J_i (provided it exists) contains \tilde{I}_{i+1} . Again, the second part of the invariant holds vacuously. For the third part of the invariant, note that the preconditions imply that \tilde{I}_{i+1} overlaps with \tilde{I}_{i_j} but not with \tilde{I}_{i_1} , and let \tilde{I}_k be a good elementary interval that overlaps with \tilde{I}_{i+1} . We show this third part by another case analysis. If $s(\tilde{I}_k) < s(\tilde{I}_{i_1})$, then \tilde{I}_k contains \tilde{I}_{i_1} and case (3c) of the algorithm would have been applicable to \tilde{I}_{i_j} and \tilde{I}_k . Hence $s(\tilde{I}_{i_1}) < s(\tilde{I}_k)$. Furthermore, if $e(\tilde{I}_k) < e(\tilde{I}_{i_1})$, then \tilde{I}_{i_1} contains \tilde{I}_k and case (3c) of the algorithm would have been applicable to \tilde{I}_k and \tilde{I}_{i_j} . Therefore, $s(\tilde{I}_{i_1}) < s(\tilde{I}_k)$ and $e(\tilde{I}_{i_1}) < e(\tilde{I}_k)$. This means that \tilde{I}_k overlaps with \tilde{I}_{i_1} . Since \tilde{I}_{i+1} does not overlap with \tilde{I}_{i_1} , the interval \tilde{I}_{i_1} does the job.

- (3c) $e(\tilde{I}_{i+1}) < e(\tilde{I}_{i_j})$: In this case, $\mathcal{H}_{i+1} = \{\tilde{I}_{i+1}\}$ and $J_{i+1} = \tilde{I}_{i_j}$. Clearly, the first part of the invariant holds true. Again, the second part of the invariant holds vacuously. For the third part of the invariant, let \tilde{I}_k be a good elementary interval that overlaps with \tilde{I}_{i+1} . If \tilde{I}_k does not overlap with \tilde{I}_{i_j} , then it must be contained in \tilde{I}_{i_j} and hence case (3c) of the algorithm would have been applicable to \tilde{I}_k and \tilde{I}_{i_j} . Thus, \tilde{I}_k overlaps with $J_{i+1} = \tilde{I}_{i_j}$ and we are done.

□

Theorem 9.7.5 *The output \mathcal{H}_i of Algorithm 9.5 is a happy clique in \mathcal{I}_g .*

Proof Because Algorithm 9.5 maintains the invariant of Lemma 9.7.4, every \mathcal{H}_i , $1 \leq i \leq m$, is a happy clique of the set $\{\tilde{I}_1, \dots, \tilde{I}_i\}$. If the algorithm stops in case (1) for some $1 \leq i < m$, then the remaining good elementary intervals do not overlap with an interval from \mathcal{H}_i . Thus, the output \mathcal{H}_i is a happy clique of the set \mathcal{I}_g . Otherwise, the algorithm returns \mathcal{H}_m , which is a happy clique of the set $\mathcal{I}_g = \{\tilde{I}_1, \dots, \tilde{I}_m\}$. \square

9.7.2 Searching the happy clique

Once we have identified a happy clique \mathcal{H} , we must search it for an elementary interval with maximum number of overlaps with bad elementary intervals, i.e., for an interval $I_k \in \mathcal{H}$ so that $b(I_l) \leq b(I_k)$ for any $I_l \in \mathcal{H}$. Let $\hat{I}_1, \dots, \hat{I}_p$ be the intervals in \mathcal{H} ordered in increasing order of their start positions. That is,

$$s(\hat{I}_1) < \dots < s(\hat{I}_p) < e(\hat{I}_1) < \dots < e(\hat{I}_p)$$

The start and end positions of the intervals from \mathcal{H} partition the straight line from 1 to $2n + 2$ into $2p + 1$ disjoint intervals

$$\begin{aligned} K_0 &= [1..s(\hat{I}_1)] \\ K_j &= [s(\hat{I}_j)..s(\hat{I}_{j+1})] \text{ for } 1 \leq j < p \\ K_p &= [s(\hat{I}_p)..e(\hat{I}_1)] \\ K_j &= [e(\hat{I}_{j-p})..e(\hat{I}_{j-p+1})] \text{ for } p < j < 2p \\ K_{2p} &= [e(\hat{I}_p)..2n + 2] \end{aligned}$$

The algorithm maintains an array $\text{cnt}[1..p]$ of p counters. For each k with $1 \leq k \leq p$ the prefix sum $\sum_{j=1}^k \text{cnt}[j]$ equals the number of bad elementary intervals seen so far that overlap with \hat{I}_k . Thus, at the beginning each counter $\text{cnt}[i]$ is initialized to 0. The algorithm scans the straight line from 1 to $2n + 2$ twice in left-to-right direction. In a first scan, it computes for each bad elementary interval $I \in \mathcal{I}_b$ the intervals K_l and K_r that contain $s(I)$ and $e(I)$, respectively. In a second scan, it updates at most three entries in the array of counters. This is done by case analysis on the values l and r . This analysis is explained in Theorem 9.7.6.

Theorem 9.7.6 *Algorithm 9.6 returns an interval from \mathcal{H} with maximum number of overlaps with bad elementary intervals.*

Proof We claim that after each iteration of the third for-loop of Algorithm 9.6 the prefix sum $\sum_{j=1}^k \text{cnt}[j]$, $1 \leq k \leq p$, equals the number of bad elementary intervals seen so far that overlap with \hat{I}_k . This is certainly true

Algorithm 9.6 Searching a happy clique

```

for  $i \leftarrow 1$  to  $p$ 
     $\text{cnt}[i] \leftarrow 0$ 
for each bad elementary interval  $I \in \mathcal{I}_b$  compute the intervals  $K_l$  and
     $K_r$  so that  $K_l$  contains  $s(I)$  and  $K_r$  contains  $e(I)$ , and store the
    values  $l$  and  $r$ 
for each  $I \in \mathcal{I}_b = \{\hat{I}_1, \dots, \hat{I}_p\}$  in increasing order of the start positions
    if  $l < r$  then
        if  $r \leq p$  then
             $\text{cnt}[l+1] \leftarrow \text{cnt}[l+1] + 1$ 
            if  $r < p$  then
                 $\text{cnt}[r+1] \leftarrow \text{cnt}[r+1] - 1$ 
            else if  $l \geq p$  then
                 $\text{cnt}[l-p+1] \leftarrow \text{cnt}[l-p+1] + 1$ 
                if  $r < 2p$  then
                     $\text{cnt}[r-p+1] \leftarrow \text{cnt}[r-p+1] - 1$ 
            else /*  $l < p$  and  $p < r$  */
                 $m \leftarrow \min\{l, r-p\}$ 
                 $\text{cnt}[1] \leftarrow \text{cnt}[1] + 1$ 
                 $\text{cnt}[m+1] \leftarrow \text{cnt}[m+1] - 1$ 
                 $M \leftarrow \max\{l, r-p\}$ 
                if  $M < p$  then
                     $\text{cnt}[M+1] \leftarrow \text{cnt}[M+1] + 1$ 
 $q \leftarrow \arg \max_{1 \leq k \leq p} \sum_{j=1}^k \text{cnt}[j]$ 
return  $\hat{I}_q$ 

```

at the very beginning because $\text{cnt}[i]$ is initialized to 0 for all $1 \leq i \leq p$. So, suppose the claim holds before an execution of the for-loop. To show that it is also satisfied after the execution of the for-loop, we proceed by case differentiation. Let $I \in \mathcal{I}_b$ be the bad elementary interval under consideration, and let l and r be the indices so that K_l contains $s(I)$ and K_r contains $e(I)$. If $l = r$, then I does not overlap any interval from \mathcal{H} . Thus Algorithm 9.6 only proceeds when $l > r$.

Case (1): If $r \leq p$, then I overlaps with the intervals $\hat{I}_{l+1}, \dots, \hat{I}_r$. Thus, we increment $\text{cnt}[l+1]$ and decrement $\text{cnt}[r+1]$ (provided that $r < p$).

Case (2): If $p \leq l$, then I overlaps with the intervals $\hat{I}_{l-p+1}, \dots, \hat{I}_{r-p}$. Thus, we increment $\text{cnt}[l-p+1]$ and decrement $\text{cnt}[r-p+1]$ (provided that $r < 2p$).

Case (3): If $l < p$ and $p < r$, then I overlaps with all intervals that (i) started before $s(I)$ and ended before $e(I)$ and all intervals that (ii) started after $s(I)$ and ended after $e(I)$. These are all intervals \hat{I}_k for which (i) $k \leq l$ and $k+p \leq r$ (hence $k \leq \min\{l, r-p\}$) or (ii) $k > l$ and $k+p > r$ (hence $k > \max\{l, r-p\}$) holds. Therefore, (i) we increment $\text{cnt}[1]$ and decrement

		I_3	I_2	I_4	I_1	I_{13}	I_{10}	I_8	I_9	I_7	I_6
1	0	0	0	0	0	1	2	2	2	2	2
2	0	0	1	2	2	2	2	2	3	4	4
3	0	0	-1	-2	-1	-1	-1	-1	-1	-1	0
4	0	0	0	0	0	-1	-2	-2	-2	-2	-2

Figure 9.25: The values of the array $cnt[1..4]$ whenever the beginning of a new bad elementary interval was scanned.

$cnt[m+1]$, where $m = \min\{l, r-p\}$, and (ii) if $M < p$ we increment $cnt[M+1]$, where $M = \max\{l, r-p\}$.

It is readily verified that after the execution of the for-loop the prefix sum $\sum_{j=1}^k cnt[j]$, $1 \leq k \leq p$, equals the number of bad elementary intervals seen so far that overlap with \hat{I}_k . \square

As an example, we apply Algorithm 9.6 to the reality-desire-diagram in Figure 9.26. There, the desire-edges are drawn as a thick (thin) line if they are good (bad) elementary intervals. All four good elementary intervals $\hat{I}_1 = I_0$, $\hat{I}_2 = I_5$, $\hat{I}_3 = I_{12}$, and $\hat{I}_4 = I_{11}$ (ordered in increasing order of their start positions) belong to the happy clique \mathcal{H} . Thus, the algorithm maintains an array of four counters. The values of the entries in this array $cnt[1..4]$ are depicted in Figure 9.25. For instance, immediately after the second bad interval I_2 was scanned, we have $cnt[3] = -1$. At the end of the scan, the prefix sums are $\sum_{j=1}^1 cnt[j] = 2$, $\sum_{j=1}^2 cnt[j] = 6$, $\sum_{j=1}^3 cnt[j] = 6$, and $\sum_{j=1}^4 cnt[j] = 4$. Therefore, the first good elementary interval \hat{I}_1 overlaps with 2 bad elementary intervals, the second and the third overlap with 6 bad elementary intervals, while \hat{I}_4 overlaps with 4 bad elementary intervals. Consequently, both \hat{I}_2 and \hat{I}_3 attain the maximum number of overlaps with bad elementary intervals.

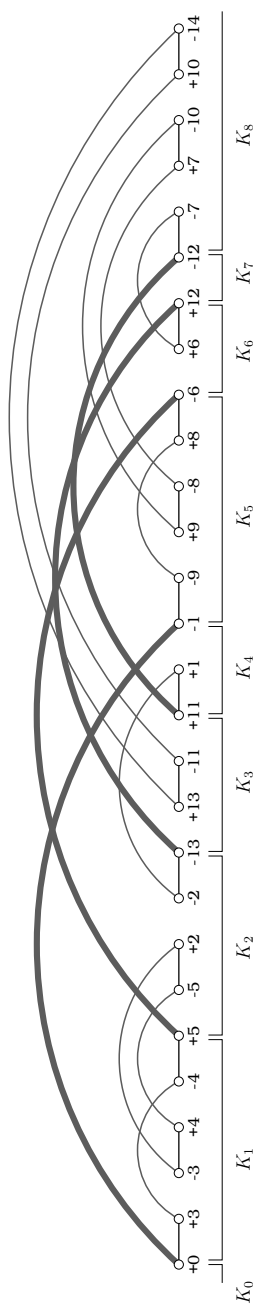


Figure 9.26: Apply Algorithm 9.6 to this reality-desire-diagram.

Phylogenetic Reconstruction

10.1 Introduction

There is strong evidence that all life on earth is descended from a single common ancestor. Over a period of at least 3.8 billion years that life form has split repeatedly into new and independent lineages. The evolutionary relationships among these species is referred to as their *phylogeny* and phylogenetic reconstruction is concerned with inferring the phylogeny of groups of organisms. These groups are called *taxa* (singular: *taxon*). Figure 10.1 shows a phylogenetic tree of the great apes. This tree has been constructed based on genetic and fossil evidence but the actual phylogeny is unknown because ancestral species have become extinct.

The splitting of lineages is called *speciation*. The most common reason for a speciation event is that one population becomes split into two sub-populations that can eventually no longer interbreed with each other. There are several ways this can happen, but the easiest one to visualize is geographical isolation. For example, the formation of the Congo River separated two chimpanzee populations because chimpanzees are not proficient swimmers, and the sea separates Sumatran from Bornean orang-utans; cf. Figure 10.1. Once that happens, each of the two populations evolve independently of each other. They undergo different random mutations and are exposed to different selection pressures. After many generations, the changes accumulate to the extent that the two populations evolve into separate species.

The tree of life is a phylogeny of species, but methods of phylogenetic reconstruction can also be applied in other biological contexts, such as inferring the phylogeny of different populations within a species. These methods can even be applied in non-biological settings: e.g. Barbrook et al. [27] described the phylogeny of different fifteenth-century manuscripts of “The Wife of Bath’s Prologue” from The Canterbury Tales.

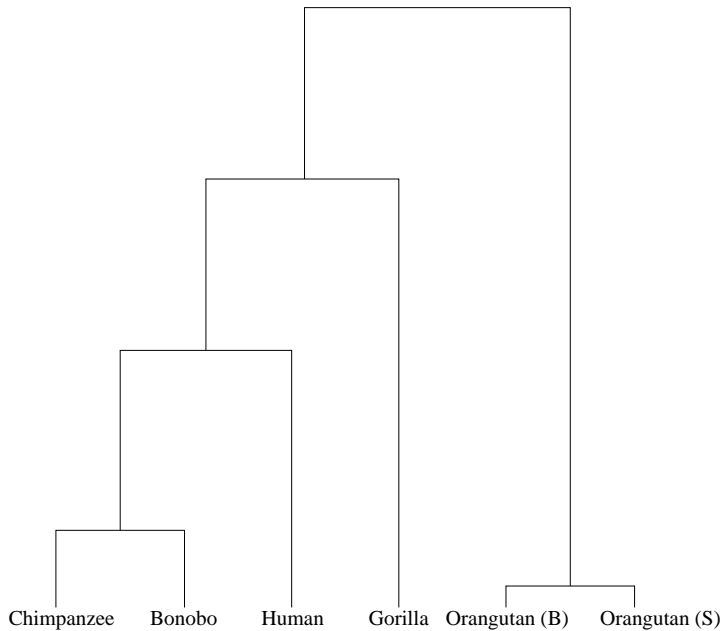


Figure 10.1: Phylogeny of the great apes (Hominidae), including six extant species: chimpanzees (*Pan troglodytes*), bonobos (*Pan paniscus*), humans (*Homo sapiens*), gorillas (*Gorilla gorilla*), Bornean orangutans (*Pongo pygmaeus*), and Sumatran orangutans (*Pongo abelii*). The length of a vertical line (branch length) represents the amount of time that lies between two speciation events. Chimpanzee and bonobo split with the formation of the Congo River, around 2 million years (myr) ago [261]. Scally et al. [282] placed the human-chimpanzee and human-chimpanzee-gorilla speciation events at approximately 6 and 10 myr ago. However, the estimates differ from study to study: according to Ingman et al. [160], the last common ancestor (LCA) of chimpanzees and human lived about 5 myr ago, while [57] estimate that it lived 5-6 myr ago. According to [207], the orangutans diverged from the Hominidae family about 12-16 myr ago. They estimate that the two orangutan species diverged about 400,000 years ago [207].

In a rooted phylogenetic tree T , the root node r corresponds to the last common ancestor (LCA) of all species in T . In this case, a path from the root to a leaf is called an *evolutionary path*. If an equal amount of changes occurs on every evolutionary path, i.e., evolutionary changes occur in a more-or-less clocklike fashion, then this tree satisfies the *molecular clock hypothesis*. In this case, one can assign a time $t(v)$ to every internal node v in the tree and a length of $t(v) - t(w)$ to an edge (v, w) in the tree. Every extant species corresponds to time 0 and a speciation event corresponding to an internal node v in the tree occurred the amount of $t(v)$ time ago. Consequently, the length of an edge represents the amount of time that lies between two speciation events. Furthermore, the last common ancestor of all species in T lived $t(r)$ time ago and all evolutionary paths have the same length $t(r)$ (where the length of a path is the sum of the lengths of all edges along the path). We shall see in Section 10.3 that these trees correspond to so-called ultrametric trees. There was a time in which the molecular clock hypothesis was widely accepted among biologists and thus the reconstruction of ultrametric trees was quite popular. Nowadays, however, one knows that DNA sequences rarely evolve at a constant rate across different lineages. From this perspective, edge lengths do not represent time, but instead represent the expected amounts of evolution (usually expressed as the expected numbers of nucleotide substitutions per site). This leads to so-called additive phylogenetic trees, which are studied in Section 10.4. Methods that reconstruct additive phylogenetic trees usually deliver an unrooted tree. Rooting such a tree—finding a unique root node corresponding to the (usually imputed) last common ancestor of all the taxa at the leaves of the tree—is itself a challenging problem. The most common method for rooting trees is the use of an *outgroup*. This is a taxon that is more distantly related to all other taxa than any other of the taxa. In practice, finding an uncontroversial outgroup can be difficult: it must be close enough to the other taxa to perform meaningful comparisons but far enough from these to be a clear outgroup.

10.1.1 Methods of phylogenetic inference

Essentially, there are three types of methods for phylogenetic inference:

- distance methods,
- maximum parsimony (MP) methods, and
- maximum likelihood (ML) methods.

In this book, we provide an in-depth study of distance methods from a computer science point of view, but we neither discuss MP methods nor ML methods.

Distance methods construct a phylogenetic tree from a distance matrix that contains the evolutionary distances between all pairs of taxa. Distances that are employed are e.g. the number of nucleotide differences per site (Section 10.1.2), the alignment-free distance measure of Section 5.6.6, or the number of genome rearrangements (Chapter 9). As already mentioned, we will study ultrametric and additive phylogenetic trees in Sections 10.3 and 10.4. The length (or weight) of an edge (or branch) in a tree represents the expected amount of evolution. In an ultrametric tree, where we assume a constant rate of molecular evolution, the length of an edge represents the elapsed time. If one ignores edge weights, one speaks of the *topology* (shape) of a tree. Distance methods that have proved to be useful for actual data analysis, such as UPGMA or neighbor-joining, *always* produce a phylogenetic tree. So the question is, under what conditions/assumptions can we trust the tree? We shall show in Section 10.3 that UPGMA *provably* constructs the correct tree provided that the input distance matrix is ultrametric. Analogously, we shall see in Section 10.5 that several neighbor-joining algorithms (among them *the* neighbor-joining algorithm developed by Saitou and Nei [276]) *provably* construct the correct tree provided that the distance matrix is additive. However, in most applications, the observed pairwise distances between taxa are only estimates of the real distances. In other words, the observed data deviate from the (unknown) real additive data. In this case, one can still use neighbor-joining algorithms to reconstruct the *topology* of the tree. Under certain criteria, this tree topology is *provably* trustworthy; see Section 10.6. If only the topology of the tree is known, then one must assign weights to the edges of the tree that best fit the data. This can be done with standard least squares methods; see Section 10.6.2.

Maximum parsimony methods were originally developed for morphological characters. Nei and Kumar [242] describe MP methods that are useful for analyzing molecular data as follows:

In these MP methods, four or more aligned nucleotide (or amino acid) sequences ($m \geq 4$) are considered, and the nucleotides (amino acids) of ancestral taxa are inferred separately at each site for a given topology under the assumption that mutational changes occur in all directions among the four nucleotides (or 20 amino acids). The smallest number of nucleotide (or amino acid) substitutions that explain the entire evolutionary process for the topology is then computed. This computation is done for all potentially correct topologies, and the topology that requires the smallest number of substitutions is chosen to be the best tree. The theoretical basis of this method is William of Ockham's philosophical idea that the best hypothesis to explain a process is the one that requires the smallest number of assumptions.

The main idea of maximum likelihood methods can be summarized as follows [242]:

In ML methods, the likelihood of observing a given set of sequence data for a specific substitution model is maximized for each topology, and the topology that gives the highest maximum likelihood is chosen as the final tree. The parameters to be considered are not the topologies but the branch lengths for each topology, and the likelihood is maximized to estimate branch lengths.

As said before, we neither discuss MP nor ML methods here. The interested reader is e.g. referred to [97, 242].

10.1.2 Molecular anthropology

Molecular anthropology is the science that uses the methods of modern molecular genetics to investigate questions that anthropologists are interested in concerning human evolution. It has been extremely useful in establishing the evolutionary tree of humans and other primates, but it also includes such areas of research as genetically reconstructing man's ancient migrations. Where did we come from? As already observed by Charles Darwin in his 1871 book "The Descent of Man" the family tree in Figure 10.1 suggests that Africa was the cradle of humans because our two closest living relatives—chimpanzees and gorillas—live there. However, it was unclear where anatomically modern humans (that is, humans with skeletons similar to those of present-day humans) evolved. Ingman et al. [160] describe the debate over recent human origins as follows:

The two main hypotheses for the evolution of modern humans agree that *Homo erectus* spread from Africa around 2 myr ago. The 'recent African origin' hypothesis states that anatomically modern humans originated in Africa 100,000–200,000 years ago and subsequently spread to the rest of the world, replacing archaic human forms with little or no genetic mixing. The alternative, 'multi-regional' hypothesis proposes that the transformation to anatomically modern humans occurred in different parts of the world, and supports this with fossil evidence of cultural and morphological continuity between archaic and modern humans outside Africa.

Starting with the landmark study of Cann et al. [49], this question has been approached by analyzing DNA from individuals belonging to different human populations. Mitochondrial DNA (mtDNA) is well suited for this purpose because of high substitution rate and the lack of recombination, but nuclear chromosomes (particularly the Y chromosome) can

also be employed. While mtDNA is inherited from the mother (maternally inherited), the Y chromosome is inherited from the father (paternally inherited). This enables researchers to trace back both the maternal and paternal lineages far back in time.

In a landmark study, Ingman et al. [160] sequenced the complete mtDNA of 53 humans of diverse origin. Apparently, they built a multiple alignment of the mtDNA sequences (which varied in length from 16,558 to 16,576 bp), excluded gaps (yielding a gapless alignment of length 16,553), and obtained pairwise distances in this way (the distance between two taxa is the number of nucleotide differences at the 16,553 sites).¹ The phylogenetic tree constructed from the pairwise distances is shown in Figure 10.2. The mtDNA sequences (excluding the D-loop) have evolved at roughly constant rates, i.e., the molecular clock hypothesis is satisfied. Assuming that human-chimpanzee speciation happened around 5 myr ago, the mutation rate (excluding the D-loop) is estimated to be $1.7 \cdot 10^{-8}$ substitutions per site per year. Ingman et al. [160] conclude:

The age of the most recent common ancestor (MRCA) for mtDNA, on the basis of the maximum distance between two humans ($5.82 \cdot 10^{-3}$ substitutions per site between the Africans Mkamba and San), is estimated to be $171,500 \pm 50,000$ yr BP. We can also estimate the age of the MRCA for the youngest clade that contains both African and non-African sequences (Fig. 2, asterisk) from the mean distance of all members of that clade to their common node ($8.85 \cdot 10^{-4}$ substitutions per site) as $52,000 \pm 27,500$ yr BP. Because genetic divergence is expected to precede the divergence of populations, this date can be considered as the lower bound for an exodus from Africa. Notably, a group of six African sequences (Fig. 4a, sequences 33-38) are genetically distant to those of other Africans, but share a common ancestor with non-Africans. These lineages represent descendants of a population that evidently gave rise to all the non-African lineages. Whether the ancestors of these six extant lineages originally came from a specific geographic region is not possible to determine, but we note that these sequences are from five populations that are now geographically unrelated.

¹In mathematical terms, the distance between two taxa i and j is the Hamming distance between the DNA sequences in rows i and j in the gapless alignment. When there is no molecular clock, one should use sophisticated models of DNA evolution, proposed e.g. by Jukes and Cantor [170] or Kimura [181]; see also [97, 242].

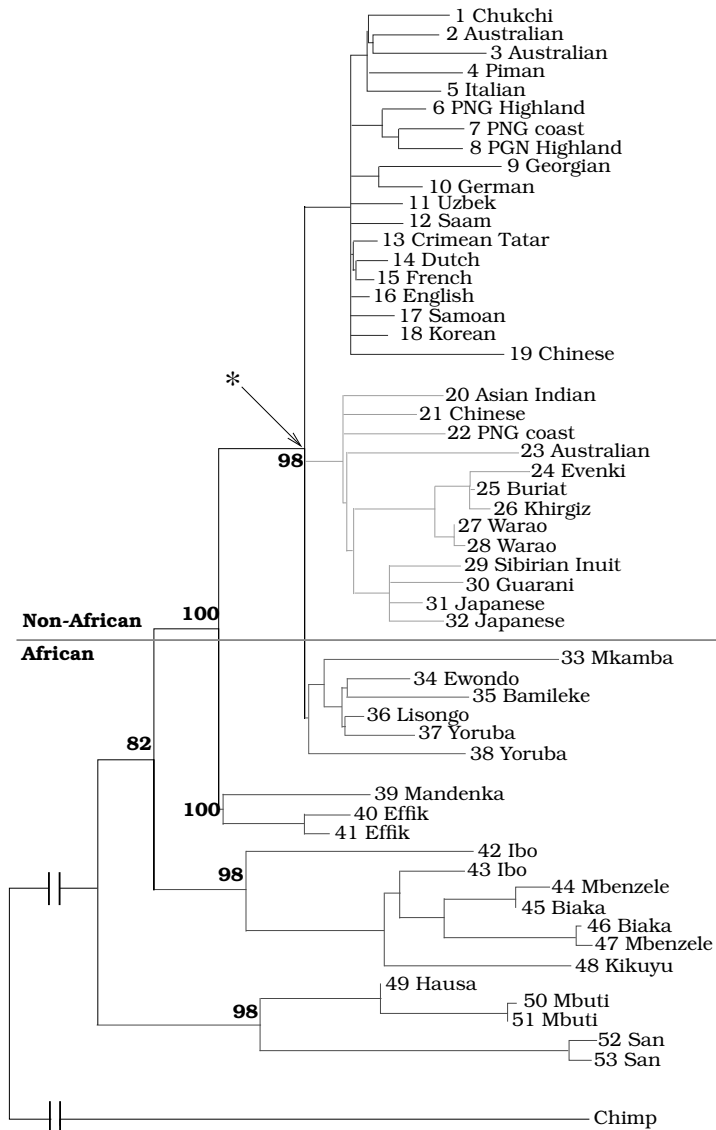


Figure 10.2: Phylogenetic tree of 53 humans of diverse origin and the chimpanzee as outgroup, constructed with the neighbor-joining algorithm described in Section 10.5.2. Edge lengths represent the number of nucleotide differences per site. Bootstrap values, shown at 6 nodes, are explained in Section 10.6.3. Adapted with permission from Ingman et al. [160].

In other words, the study provides compelling evidence of an “Out of Africa” migration. Because the MRCA² is a woman, she is called “Mitochondrial Eve” or “African Eve.”

So Figure 10.1 provides a picture of human-ape common ancestry and the human-chimpanzee divergence about 6 myr ago, while Figure 10.2 provides evidence that anatomically modern humans arose in Africa about 200,000 years ago. But what happened in between? It could be that the fossil skeleton of a female nicknamed “Lucy” found in 1974 in the Afar Depression (Ethiopia) is one of the missing links. Lucy belongs to the species *Australopithecus afarensis*, which lived between 3.9 and 3 million years ago. *Australopithecus*, the name of the genus, comes from the Latin word *australis* meaning southern and the Greek word *pithekos* meaning ape. The name “southern ape” was chosen because the first fossil, the fossilized skull of a child, was discovered in 1924 in Taung, South Africa. The genus *Homo* (*homo* is a Latin word meaning human or man) is estimated to be about 2.3 million years old, possibly having evolved from *A. afarensis* ancestors. The genus is characterized by an upright posture, large brains, high intelligence, and hairlessness. Apart from *Homo sapiens* (*sapiens* is a Latin word meaning wise, intelligent), some of the most famous members of the genus *Homo* are (dates are taken from [166]):

- *Homo habilis* (lived 2.3 to 1.6 myr ago),
- *Homo ergaster* (1.8 to 1.5 myr ago),
- *Homo erectus* (1.7 million to 40,000 years ago),
- *Homo heidelbergensis* (700,000 to 200,000 years ago),
- *Homo neanderthalis* (200,000 to 30,000 years ago), and
- *Homo floresiensis* (74,000 to 12,000 years ago).

H. habilis (“handy man”) was the earliest prehuman/human species to show a significant increase in brain size and also the first to be found associated with stone tools (hence the name “handy man”). There is a controversy over whether this species should be placed in the genus *Homo* because new fossil discoveries show that this species shares some important physical similarities with members of the genus *Australopithecus*. It is debated whether *H. habilis* is a direct human ancestor.

H. erectus (“upright man”) fossils have been found in Africa (e.g. Lake Turkana), Georgia (Dmanisi, located in the Caucasus, is the earliest known hominid site outside of Africa), and in Asia, mostly in Indonesia

²The term MRCA is usually used to describe a common ancestor of individuals within a species. We use the term last common ancestor (LCA) when discussing ancestry between species.

(“Java Man”) and China (“Peking Man”). A growing number of scientists have redefined the species *H. erectus* so that it now contains only Asian fossils. Many of the older African fossils formerly known as *H. erectus* have now been placed into a separate species called *H. ergaster* (the name “workman” was chosen because large stone tools were found near some of its fossils), and this species is considered to be ancestral to *H. erectus*. The redefined *H. erectus* is now generally believed to be a side branch on our family tree, whereas *H. ergaster* is now viewed as one of our direct ancestors. Some researchers believe that the hominin skulls discovered in Ngandong (Indonesia) represent a small colony of *H. erectus* that survived long after the rest had died off. Recent dates seem to suggest that the skulls are actually between 27,000 and 53,000 years old. However, these dates were taken on faunal remains that other researchers believe are not of the same age as the hominins.

H. heidelbergensis (“Heidelberg man;” in 1907 the first fossil was discovered near the city of Heidelberg, Germany) evolved in Africa but by 500,000 years ago some populations were in Europe. *H. heidelbergensis* began to develop regional differences that eventually gave rise to two species of humans. European populations of *H. heidelbergensis* evolved into *H. neanderthalensis*, while a separate population in Africa evolved into *H. sapiens*. *H. heidelbergensis* was probably a descendant of *H. ergaster*. Fossils from Atapuerca in Spain date to 800,000 years old, and may be *H. heidelbergensis* or a different species, *H. antecessor*.

H. neanderthalensis (“Man from the Neander Valley;” in 1856 the first major specimen was found in the Neanderthal,³ Germany) was an advanced humans species, capable of intelligent thought processes and able to adapt to and survive in some of the harshest environments known to humans. Thousands of fossils representing the remains of many hundreds of Neanderthal individuals have been recovered from sites across Europe and southwestern to central Asia. Neanderthals and modern humans (“Cro-Magnons”) coexisted in Europe for several thousand years, but the duration of this period is uncertain. Modern humans may have first migrated to Europe 40,000–43,000 years ago and the Neanderthal became extinct about 30,000 years ago. There is some evidence that members of the two species may have met much earlier in the Middle East. In 1982, the most complete Neanderthal skeleton found to date was discovered in the Kebara Cave (Israel); it is estimated to be about 60,000 years old. In 2005, a set of 7 teeth from Tabun Cave in Israel were studied and found to most likely belong to a Neanderthal that may have lived around 90,000 years ago [60], and another Neanderthal from Tabun was estimated to be ca. 122,000 years old. So Neanderthals lived in the Middle East for a long period of time. The earliest anatomically modern

³The ‘th’ is pronounced as ‘t’.

specimens, found in Skhul and Qafzeh in Israel, are probably more than 90,000 years old [301]. Although it is possible that the early presence of modern humans in the area was episodic, Neanderthals and early modern humans might have made contact in the region more than 90,000 years ago. So an interesting question arises: Did Neanderthals interbreed with modern humans? Comparing DNA sequences from both species could help to answer this question, but the recovery of DNA from ancient bones and teeth is difficult because DNA degrades over time (the speed of the degradation process depends on a number of factors, such as temperature and soil acidity). So exceptional circumstances are required for DNA to survive over long time periods. Moreover, contamination by modern DNA is a particularly difficult problem when the ancient DNA comes from close relatives like *H. neanderthalensis*. Despite these difficulties, in 1997 Neanderthal mtDNA was successfully extracted from bones. Studies of human and Neanderthal mtDNA found no evidence of interbreeding; see e.g. [189]. In 2010, Svante Pääbo's team from the Max Planck Institute for Evolutionary Anthropology in Germany announced a draft sequence of the Neanderthal genome [131]. Their comparison of the genomes of five modern humans with the Neanderthal genome produced evidence consistent with inter-species mating, known as hybridization, between *H. neanderthalensis* and *H. sapiens* [131]. According to the study, Neanderthals have contributed approximately 1-4% to the genomes of non-African modern humans. This suggests that modern humans bred with Neanderthals in the Middle East, after they left Africa, but before they spread to Asia and Europe. However, Eriksson and Manica [92] provided an alternative explanation: they claim that common ancestry, without any hybridization, explains the genetic similarities between Neanderthals and modern humans. Green et al. [131] point out themselves that alternative scenarios cannot conclusively be ruled out:

Although gene flow from Neandertals into modern humans when they first left sub-Saharan Africa seems to be the most parsimonious model compatible with the current data, other scenarios are also possible. For example, we cannot currently rule out a scenario in which the ancestral population of present-day non-Africans was more closely related to Neandertals than the ancestral population of present-day Africans due to ancient sub-structure within Africa ...

Denisovans are the most recently discovered hominins. In 2008, a finger bone of a juvenile hominin was excavated in Denisova Cave in the Altai Mountains in southern Siberia. The bone was found in a layer dated to 50,000–30,000 years ago. Krause et al. [187] were able to extract mtDNA and their comparison with present-day human mtDNA yielded a big surprise:

Whereas Neanderthals differ from modern humans at an average of 202 nucleotide positions, the Denisova individual differs at an average of 385 positions (...), and the chimpanzee at 1,462 positions (...). The Denisova hominin mtDNA thus carries almost twice as many differences to the mtDNA of present-day humans as do Neanderthal mtDNAs.

That is, Denisovans are a genetically distinct group of humans, distantly related to Neanderthals and even more distantly related to modern humans. For the first time in history, the discovery of an unknown hominin was based on molecular sequence data and not on morphological data! Because the finger bone was in excellent condition, it was possible to sequence the whole genome of the Denisova hominin. Reich et al. [266] compared the Denisovan DNA sequences with those of Neanderthals and present-day humans and concluded:

Assuming 6.5 million years for human-chimpanzee divergence, this implies that DNA sequences of Neanderthals and the Denisova individual diverged on average 640,000 years ago, and from present-day Africans 804,000 years ago.

Their study also revealed that the Denisovans possibly interbred with the ancestors of modern Melanesians when the latter migrated to Melanesia:

..., we estimate that $2.5 \pm 0.6\%$ of the genomes of non-African populations derive from Neanderthals, in agreement with our previous estimate of 1-4%. In addition, we estimate that $4.8 \pm 0.5\%$ of the genomes of Melanesians derive from Denisovans. Altogether, as much as $7.4 \pm 0.8\%$ of the genomes of Melanesians may thus derive from recent admixture with archaic hominins.

According to Reich et al. [266], the most plausible model of population history is the following:

After the divergence of the Denisovans from Neanderthals, there was gene flow from Neanderthals into the ancestors of all present-day non-Africans. Later there was admixture between the Denisovans and the ancestors of Melanesians that did not affect other non-African populations.

This model is illustrated in Figure 10.3.

Homo floresiensis ("Man from Flores;" in 2003 remains were discovered in the cave of Liang Bua on the Island of Flores, Indonesia) is a hominin with unusual features. It is assumed that the remains of a largely complete skeleton with skull belong to a female aged about 30 years old, who stood about 1 meter tall (hence the nickname "Hobbit") and had a brain

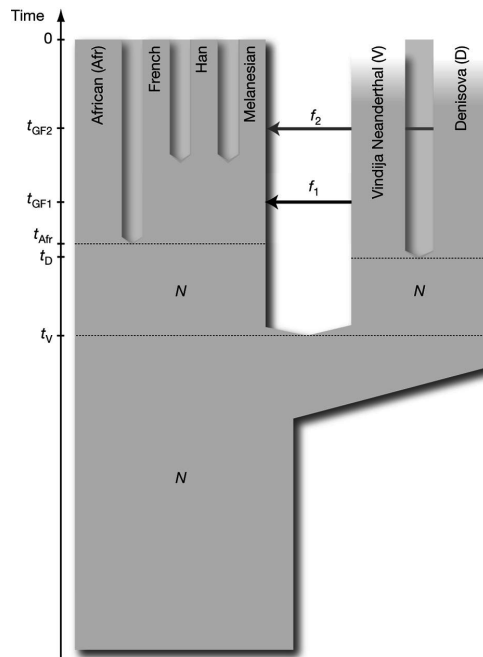


Figure 10.3: A model of the population history of modern humans, Neanderthals, and Denisovans. N denotes the effective populations size, t denotes the time of population separation, f denotes the amount of gene flow, and t_{GF} denotes the time of the gene flow. Reprinted with permission from Reich et al. [266].

volume of about 380-420 cm³ (by contrast, the brains of human pygmies are almost as large as those of normal-sized humans). Despite of their small brain size, *H. floresiensis* made and used stone tools and hunted pygmy elephants (mostly juvenile ones). The remains of at least twelve *H. floresiensis* individuals date from about 38,000 to 18,000 years old, but archaeological evidence suggests *H. floresiensis* lived at Liang Bua for a much longer period of time. Modern humans arrived in Indonesia between 55,000 and 35,000 years ago, and may have interacted with *H. floresiensis*, but there is no evidence of this at Liang Bua. Scientists are trying to figure out how *H. floresiensis* is related to other human species. Stone tools dating to 840,000 years ago were discovered on Flores, indicating that a hominin species was living on the island at that time, but there is a controversy over whether these stone tools have been made by *H. erectus* or hominids of another smaller species (such as those found at Dmanisi). It is possible that *H. floresiensis* shared a common ancestor with *H. erectus* but was not descended from it. Whatever the origins of the ancestral population, it is accepted that the population underwent long term isolation on the small island, with limited food resources and a lack of predators. This results in island dwarfism, a common phenomenon seen in other mammals in similar environments. For example, the pygmy elephants on Flores showed the same adaptation: they are one of the smallest species of *Stegodon* ever found. Apart from morphological criteria, DNA sequences extracted from hominin remains can help in understanding hominin evolution. Given the relatively recent age of the unfossilized remains of *H. floresiensis*, there is hope that it might be possible to retrieve DNA from the bones. However, DNA degrades fast in the tropical climate of Indonesia and initial efforts were unsuccessful (Neanderthal and Denisovan bones from which DNA has been extracted all came from much colder climates).

So there is evidence that at least three other hominin species were still in existence between 50,000 and 30,000 years ago, when modern humans were migrating out of Africa into Asia and Europe: Neanderthals, Denisovans, and *H. floresiensis*.

10.2 Basic definitions

Phylogenies are usually represented as binary trees because speciation events are generally bifurcating, that is, speciation occurs when an ancestral lineage splits into two new independent lineages. This is not entirely correct. Although hybrid speciation and horizontal gene transfer are rare, they do occur. Thus, the evolutionary history of all life on earth would more appropriately be modeled by a phylogenetic network. For the sake

of simplicity, we will only deal with phylogenetic trees, but we do not insist that the trees must be binary.

Definition 10.2.1 Let $S = \{s_1, \dots, s_n\}$ be a set of taxa. A *phylogenetic tree* on S is a triple $T = (V, E, \lambda)$, where

- (V, E) is an acyclic connected graph (V is the set of nodes and E is the set of undirected edges) in which there is either a distinguished root node of degree ≥ 2 and all other internal nodes have a degree ≥ 3 or there is no such distinguished root node and all internal nodes have a degree ≥ 3 . In the former case we speak of a *rooted tree*, and in the latter case of an *unrooted tree*. Henceforth, we denote the set of leaves (nodes of degree 1) by V_L , and the set of internal nodes by V_I .
- λ is a bijection $\lambda : S \rightarrow V_L$ between the set of taxa and the set of leaves.

An edge $(v, w) \in E$ is an *external edge* if either v or w is a leaf, otherwise it is an *internal edge*. A phylogenetic tree is *binary* if every internal node has degree 3, except for the root node in a rooted tree.

This definition of a phylogenetic tree does not include edge weights. These will be implicitly introduced in ultrametric trees (Definition 10.3.1) and explicitly in additive trees (Definition 10.4.1).

In a phylogenetic tree T on S , leaves are usually labeled with extant species. In this chapter we often identify a leaf with its label. For example, when we speak of leaf s_i , where $s_i \in S$, then we mean the leaf $\lambda(s_i)$. Internal nodes correspond to ancestral species, which in most cases are extinct. Thus, the topology of the tree describes the putative order of speciation events that gave rise to the extant taxa. The extreme case in which a phylogenetic tree has just one internal node results in a topology that resembles a star. Hence it is called a *star phylogeny*.

The phylogenetic reconstruction algorithms presented in this chapter are based on distances between taxa, and we formally define the relevant concepts below. In the following, let S be the set of taxa.

Definition 10.2.2 A *semimetric* on S is a function $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ that satisfies for all $x, y \in S$:

- $d(x, y) = 0 \Leftrightarrow x = y$ (identity of indiscernibles)
- $d(x, y) = d(y, x)$ (symmetry)

A *metric* or *distance function* on S is a semimetric $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the *triangle inequality*, i.e. for all $x, y, z \in S$:

- $d(x, y) \leq d(x, z) + d(z, y)$

A metric $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is called *additive* if it satisfies the *additive inequality*, i.e. for all $w, x, y, z \in S$:

$$\bullet \quad d(w, x) + d(y, z) \leq \max\{d(x, y) + d(w, z), d(x, z) + d(w, y)\}$$

An *ultrametric* on S is an additive metric $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the *ultrametric inequality*, i.e. for all $x, y, z \in S$:

$$\bullet \quad d(x, y) \leq \max\{d(x, z), d(y, z)\}$$

Lemma 10.2.3 *A semimetric $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the additive inequality is in fact a metric.*

Proof Suppose that the additive inequality

$$d(w, x) + d(y, z) \leq \max\{d(x, y) + d(w, z), d(x, z) + d(w, y)\}$$

holds for all $w, x, y, z \in S$. Choosing $w = x$, we obtain

$$d(x, x) + d(y, z) \leq \max\{d(x, y) + d(x, z), d(x, z) + d(x, y)\} = d(y, x) + d(x, z)$$

for all $x, y, z \in S$. In other words, the triangle inequality is satisfied. \square

Lemma 10.2.4 *A semimetric $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ that satisfies the ultrametric inequality is in fact an additive metric.*

Proof We have to show that the additive inequality $d(w, x) + d(y, z) \leq \max\{d(x, y) + d(w, z), d(x, z) + d(w, y)\}$ holds for $w, x, y, z \in S$. According to the ultrametric inequality we have:

$$\begin{aligned} d(w, x) &\leq \max\{d(w, z), d(x, z)\} \\ d(w, x) &\leq \max\{d(w, y), d(x, y)\} \\ d(y, z) &\leq \max\{d(y, x), d(z, x)\} \\ d(y, z) &\leq \max\{d(y, w), d(z, w)\} \end{aligned}$$

Without loss of generality, assume that $d(x, z)$ is the largest element of $\{d(w, z), d(x, z), d(w, y), d(x, y)\}$. In particular, this implies $d(w, x) \leq d(x, z)$ and $d(y, z) \leq d(x, z)$. Now, if $d(w, x) \leq d(w, y)$ or $d(y, z) \leq d(w, y)$, then the additive inequality is a consequence of $d(w, x) + d(y, z) \leq d(x, z) + d(w, y) \leq \max\{d(x, y) + d(w, z), d(x, z) + d(w, y)\}$. Otherwise, if $d(w, x) > d(w, y)$ and $d(y, z) > d(w, y)$, then the inequalities $d(w, x) \leq d(x, y)$ and $d(y, z) \leq d(w, z)$ follow. In this case, the additive inequality is also true because $d(w, x) + d(y, z) \leq d(x, y) + d(w, z) \leq \max\{d(x, y) + d(w, z), d(x, z) + d(w, y)\}$. \square

Thus, in order to prove that a function $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is (i) a metric, (ii) an additive metric, or (iii) an ultrametric, it suffices to verify *three* properties: identity of indiscernibles, symmetry, and the respective inequality.

For ease of presentation, we number the n taxa consecutively and identify a taxon with its number. More precisely, we assume that the set S of taxa is the set $\{1, \dots, n\}$.

Definition 10.2.5 A symmetric $n \times n$ matrix $D = (d_{ij})$ satisfying $d_{ii} = 0$ and $d_{ij} > 0$ for all $i \neq j$ with $i, j \in \{1, \dots, n\}$ is called *dissimilarity matrix*.

Throughout this chapter, we assume that the input to a phylogenetic reconstruction algorithm is a dissimilarity matrix. (Note that one can test in $O(n^2)$ time whether or not an $n \times n$ matrix is a dissimilarity matrix.) Since $S = \{1, \dots, n\}$, an $n \times n$ dissimilarity matrix $D = (d_{ij})$ induces a function $d : S \times S \rightarrow \mathbb{R}_{\geq 0}$ defined by $d(x, y) = d_{xy}$. Clearly, the definition of a dissimilarity matrix implies that d is a semimetric.

Definition 10.2.6 An $n \times n$ dissimilarity matrix $D = (d_{ij})$ is called *distance matrix* if the induced function d is a *distance function* (i.e., it satisfies the metric axioms). Furthermore, we say that the matrix D is *additive* (ultrametric, respectively) if the induced function d is additive (ultrametric, respectively).

One can check in $O(n^3)$ time whether or not a matrix D is a distance matrix (an ultrametric matrix, respectively). Take all 3-element subsets of the set $S = \{1, \dots, n\}$ and test the triangle inequality (the ultrametric inequality, respectively). Analogously, one can check in $O(n^4)$ time whether or not a matrix D is an additive distance matrix.

Exercise 10.2.7 Give an example of a non-additive distance matrix.

Exercise 10.2.8 Provide an additive matrix that is not ultrametric.

10.3 Ultrametric distance matrices and trees

Definition 10.3.1 Let $T = (V, E, \lambda)$ be a rooted phylogenetic tree on a set S of taxa. Then T together with a marking $\mu : V_I \rightarrow \mathbb{R}_{>0}$ (of the internal nodes with positive numbers) is an *ultrametric tree* provided that for each path v_1, v_2, \dots, v_m, k from the root $r = v_1$ to a leaf k the sequence of marks $\mu(v_1), \mu(v_2), \dots, \mu(v_m)$ is strictly decreasing.

Given a rooted phylogenetic tree T on a set of n taxa together with a marking μ of the internal nodes, one can test by a depth-first traversal of T whether it is ultrametric or not. This takes only $O(n)$ time because T has n leaves and at most $n - 1$ internal nodes (every internal node has at least two children).

Before defining consistency of an ultrametric tree with a dissimilarity matrix D , we recall the definition of the lowest common ancestor. Given two nodes v and w in a rooted tree, their *lowest common ancestor* $\text{LCA}(v, w)$ is the node farthest from the root that is an ancestor of v and w (or, to put it differently, it is the node u so that u is ancestor of v and w and there is no proper descendant of u that is also an ancestor of v and w).

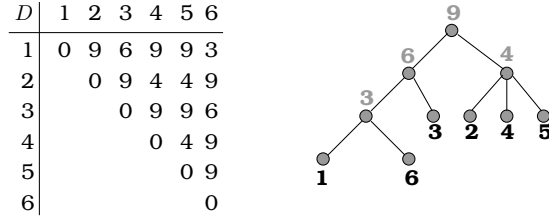


Figure 10.4: The dissimilarity matrix D and the ultrametric tree T are consistent.

Definition 10.3.2 Let $S = \{1, \dots, n\}$ be a set of taxa, let D be an $n \times n$ dissimilarity matrix, and let $T = (V, E, \lambda, \mu)$ be an ultrametric tree on S . We say that D and T are *consistent* if $\mu(\text{LCA}(i, j)) = d_{ij}$ holds true for any two leaves (taxa) i and j of T .

We will also say that D is consistent with T , and vice versa. Figure 10.4 shows a dissimilarity matrix D on the left that is consistent with the ultrametric tree T on the right.

We remark that it is possible to check in $O(n^2)$ time whether an $n \times n$ dissimilarity matrix D and an ultrametric tree T on a set of n taxa are consistent: Preprocess T in $O(n)$ time so that lowest common ancestor queries can be answered in constant time (see Chapter 3), and then verify $\mu(\text{LCA}(i, j)) = d_{ij}$ for each of the $O(n^2)$ pairs of taxa.

Ultrametric trees are those phylogenetic trees that satisfy the molecular clock hypothesis. We will come back to this topic in Section 10.4.1. Moreover, we would like to stress that ultrametric trees can alternatively be defined as additive trees in which the lengths of all evolutionary paths (all paths from the root to a leaf) are equal. We will show the equivalence of both definitions in Section 10.4.1.

By definition, a rooted phylogenetic tree T is an unordered tree in the sense that the order of the children of a node in T is not specified. However, every drawing of such a tree is, in fact, an ordered tree. Because there is a bijection between the set $S = \{1, \dots, n\}$ of taxa and the leaves of T , it is natural to use the *canonical form* of T in drawings. This canonical form is the ordered tree that is obtained by ordering the children of nodes in T as follows: Let v be an internal node in T , let $\{v_1, \dots, v_m\}$ be the set of its children, and for each child node v_i let s_i be the minimum of all taxa in the subtree rooted at v_i . Then, order the children so that node v_j appears before node v_k if and only if $s_j < s_k$. Figure 10.5 illustrates this.

Sections 10.3.1–10.3.2 are based on Gusfield's book [139, 17.1] and the work of Heun [150, 151].

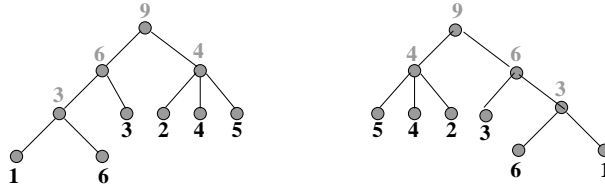


Figure 10.5: Two ordered representations of the same ultrametric tree. The canonical form is shown on the left-hand side.

10.3.1 Characterization of ultrametric matrices

In the following, we are going to characterize ultrametric dissimilarity matrices: they satisfy the so-called 3-point condition and are consistent with an ultrametric tree.

Definition 10.3.3 An $n \times n$ dissimilarity matrix D satisfies the *3-point condition* if for all $i, j, k \in \{1, \dots, n\}$ the two largest values out of d_{ij}, d_{ik}, d_{jk} are equal.

The next theorem not only shows that an ultrametric matrix is consistent with an ultrametric tree, but it is constructive in the sense that it gives an algorithm for constructing the ultrametric tree.

Theorem 10.3.4 For any dissimilarity matrix D , the following statements are equivalent:

- (1) D is ultrametric.
- (2) D satisfies the 3-point condition.
- (3) D is consistent with an ultrametric tree. Moreover, this tree is unique.

Proof We prove (1) \Rightarrow (2), (2) \Rightarrow (3), and (3) \Rightarrow (1).

(1) \Rightarrow (2): It must be shown that D satisfies the 3-point condition for all $i, j, k \in \{1, \dots, n\}$. Fix three points (taxa) i, j, k . Since D is ultrametric, the ultrametric inequality $d_{ij} \leq \max\{d_{ik}, d_{jk}\}$ holds true. If $d_{ik} = d_{jk}$, then the 3-point condition is obviously satisfied. Now suppose that $d_{ik} \neq d_{jk}$. Without loss of generality, assume that $d_{ik} < d_{jk}$. In conjunction with $d_{ij} \leq \max\{d_{ik}, d_{jk}\}$, this implies $d_{ij} \leq d_{jk}$. Because D is ultrametric, we also have $d_{jk} \leq \max\{d_{ij}, d_{ik}\} = d_{ij}$ (the last equality follows from $d_{ik} < d_{jk}$). In summary, we have shown $d_{ij} \leq d_{jk} \leq d_{ij}$. Hence $d_{ik} < d_{jk} = d_{ij}$.

(2) \Rightarrow (3): The proof is by induction on the number n of taxa. If $n = 1$,

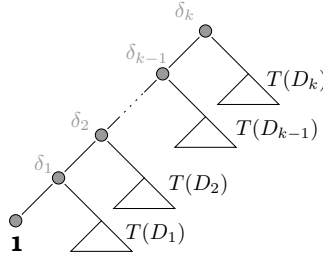


Figure 10.6: Construction of an ultrametric tree.

then the tree consists of only one leaf (taxon). The base case $n = 2$ is also clear: the root of the tree is marked with the distance d_{12} between the two taxa 1 and 2, and the leaves 1 and 2 are its children. So let $n \geq 3$. For the induction step, suppose that the claim holds for every natural number n_i with $n_i < n$. We arbitrarily choose some taxon out of the n taxa, say taxon 1, and consider its distance to all other taxa. Let $\{d_{12}, d_{13}, \dots, d_{1n}\} = \{\delta_1, \dots, \delta_k\}$ and suppose that the δ -values are numbered in increasing order, i.e., $\delta_1 < \delta_2 < \dots < \delta_k$. The δ -values partition the set $\{2, \dots, n\}$ into disjoint subsets $S_i = \{j \in \{2, \dots, n\} \mid d_{1j} = \delta_i\}$. For each S_i let D_i be the dissimilarity matrix obtained from D by deleting all rows and columns of D that do not correspond to a taxon from S_i . Clearly, D_i is ultrametric. Let $n_i = |S_i|$ be the number of taxa in S_i . Since $n_i < n$, the inductive hypothesis implies that D_i is consistent with an ultrametric tree $T(D_i)$. Now we construct a path $1, v_1, \dots, v_k$ from leaf 1 to the root $r = v_k$ consisting of new nodes v_1, \dots, v_k . Each node v_i is defined to be the parent node of the tree $T(D_i)$, and it is marked with δ_i (so $\mu(v_i) = \delta_i$). This approach is illustrated in Figure 10.6. The tree constructed in this way may not yet be ultrametric, so it may be necessary to alter it, but we shall see that after small modifications the resulting tree is indeed ultrametric. By construction, the sequence of marks on the path from the root to leaf 1 is strictly decreasing. Furthermore, by the inductive hypothesis the sequence of marks on the path from the root of a subtree $T(D_i)$ to a leaf is strictly decreasing. We next show that the mark of the root of a subtree $T(D_i)$ is less than or equal to the mark $\mu(v_i) = \delta_i$ of its parent node v_i . Because the mark of the root of a subtree $T(D_i)$ is equal to d_{xy} , where x and y are two leaves in $T(D_i)$ whose lowest common ancestor is the root of $T(D_i)$, we can rephrase our claim as $d_{xy} \leq \delta_i$; see Figure 10.7. By assumption, D satisfies the 3-point condition. Therefore, the two largest values out of d_{xy}, d_{1x}, d_{1y} are equal. This, in combination with $d_{1x} = d_{1y} = \delta_i$, proves our claim $d_{xy} \leq \delta_i$. In case $d_{xy} < \delta_i$, the sequence of marks on any path from the root to a leaf in the subtree $T(D_i)$ is strictly decreasing and

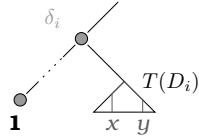


Figure 10.7: If x and y are two leaves in $T(D_i)$ whose lowest common ancestor is the root of $T(D_i)$, then $d_{xy} \leq \delta_i$.

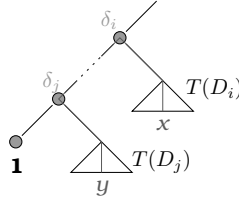


Figure 10.8: The two leaves x and y stem from different subtrees.

we are done. If $d_{xy} = \delta_i$, then we remove the edge from node v_i to the root of $T(D_i)$ (in other words, we identify node v_i with the root of $T(D_i)$). After this small modification, the sequence of marks on any path from the root to a leaf in the subtree $T(D_i)$ is strictly decreasing. It is clear that the overall result of these modifications, the tree $T(D)$, is ultrametric.

To show that the tree $T(D)$ is consistent with D , one must prove that $d_{xy} = \mu(\text{LCA}(x, y))$ is true for any two leaves x and y . By construction, it suffices to verify this equality in the cases in which x is a leaf in $T(D_i)$ and y is a leaf in $T(D_j)$, where $i \neq j$. The situation is illustrated by Figure 10.8. Without loss of generality, we may assume $\delta_i > \delta_j$. Obviously, the lowest common ancestor of x and y is the node v_i with $\mu(v_i) = \delta_i$, and we must prove that $d_{xy} = \delta_i$. By assumption, D satisfies the 3-point condition. Therefore, the two largest values out of d_{xy}, d_{1x}, d_{1y} are equal. Since $d_{1y} = \delta_j < \delta_i = d_{1x}$, it follows that $d_{xy} = d_{1x} = \delta_i$.

It still must be shown that $T(D)$ is unique. It is not difficult to see that any ultrametric tree, that is consistent with D must contain a path $1, u_1, \dots, u_k$ from leaf 1 to the root $r = u_k$ so that $\mu(u_i) = \delta_i$. Moreover, the subtree at node u_i must contain all taxa that have distance δ_i to taxon 1. In other words, it must contain all taxa from the set S_i . By the inductive hypothesis, $T(D_i)$ is the unique ultrametric tree that is consistent with D_i . Consequently, $T(D)$ is the unique ultrametric tree that is consistent with D .

(3) \Rightarrow (1): By the definition of a dissimilarity matrix, it is sufficient to

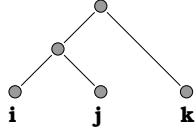
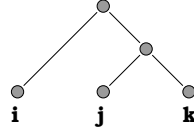
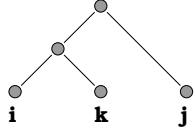
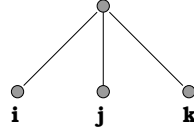
Case 1: $\text{LCA}(i, j) \neq \text{LCA}(i, k) = \text{LCA}(k, j)$ **Case 2:** $\text{LCA}(i, j) = \text{LCA}(i, k) \neq \text{LCA}(k, j)$ **Case 3:** $\text{LCA}(i, j) = \text{LCA}(j, k) \neq \text{LCA}(i, k)$ **Case 4:** $\text{LCA}(i, j) = \text{LCA}(i, k) = \text{LCA}(k, j)$ 

Figure 10.9: The four possible cases.

show that D satisfies the ultrametric inequality. That is, we must show that for $i, j, k \in \{1, \dots, n\}$ the inequality $d_{ij} \leq \max\{d_{ik}, d_{jk}\}$ holds true. Let $T(D)$ be the unique ultrametric tree that is consistent with D . We use a case-by-case analysis, and the four possible cases are depicted in Figure 10.9. In case 1, we have $\text{LCA}(i, j) \neq \text{LCA}(i, k) = \text{LCA}(k, j)$. Because $\mu(\text{LCA}(x, y)) = d_{xy}$ for any two leaves x and y in $T(D)$ and every sequence of marks on a path from the root to a leaf is strictly decreasing, it follows that $d_{ij} < d_{ik} = d_{jk}$. Similarly, we derive $d_{jk} < d_{ij} = d_{ik}$ in case 2, $d_{ik} < d_{ij} = d_{jk}$ in case 3, and $d_{ij} = d_{ik} = d_{jk}$ in case 4. In case 1, $d_{ij} < \max\{d_{ik}, d_{jk}\}$ and in the remaining cases we have $d_{ij} = \max\{d_{ik}, d_{jk}\}$. That is, the ultrametric inequality is satisfied in all four cases. \square

10.3.2 Construction algorithm

According to Theorem 10.3.4, every ultrametric matrix is consistent with an ultrametric tree, and the proof of that theorem already comprises a (quadratic time) algorithm to construct this tree. By Theorem 10.3.4, this construction algorithm will only work correctly if the input matrix D is indeed an ultrametric matrix. As mentioned earlier, we can check this by testing the ultrametric inequality (or, equivalently, the 3-point condition) for all possible 3-element subsets of $S = \{1, \dots, n\}$. However, this takes $O(n^3)$ time, whereas the construction algorithm takes only $O(n^2)$ time (as we shall see). There are two ways out of this trap: (a) let the algorithm construct a tree and test afterwards in $O(n^2)$ time if this tree is ultrametric and consistent with D , or (b) integrate the test into the construction algorithm itself. Approach (b) is taken in Algorithm 10.1.

Algorithm 10.1 Construction of an ultrametric tree (if it exists).

Input: $n \times n$ dissimilarity matrix D .

1. Choose a taxon s out of the set S , say $s = 1$.
2. Determine $k = |\{d_{1j} \mid 2 \leq j \leq n\}|$ as well as the sequence $\delta_1, \dots, \delta_k$ so that $\{\delta_1, \dots, \delta_k\} = \{d_{12}, \dots, d_{1n}\}$ and $\delta_1 < \dots < \delta_k$. Simultaneously partition $\{2, \dots, n\}$ into disjoint sets S_1, \dots, S_k :

$$S_i = \{j \mid 2 \leq j \leq n \text{ and } d_{1j} = \delta_i\}$$

3. Recursively compute the subtrees $T(D_i)$. Let r_i be the root of $T(D_i)$ and—if r_i is not a leaf—let m_i be its mark. During the computation, maintain a pair of taxa (x_i, y_i) with $d_{x_i y_i} = m_i$.
4. Create a path $1, v_1, \dots, v_k$ from leaf 1 to the root $r = v_k$ consisting of nodes v_1, \dots, v_k with marks $\delta_1, \dots, \delta_k$. If r_i is a leaf, then connect it to node v_i by adding the edge (v_i, r_i) . With the remaining subtrees, proceed as follows:
 - If $m_i < \delta_i$, then connect $T(D_i)$ to node v_i by adding the edge (v_i, r_i) .
 - If $m_i = \delta_i$, then connect $T(D_i)$ to the tree by identifying v_i with r_i .
 - If $m_i > \delta_i$, then output the triple $(x_i, y_i, 1)$ as a witness that D is not ultrametric and terminate. (Note that $(x_i, y_i, 1)$ does not satisfy the 3-point condition because $d_{x_i y_i} = m_i > \delta_i = d_{1x_i} = d_{1y_i}$.)
5. For each pair $T(D_i)$ and $T(D_j)$ of subtrees with $\delta_i > \delta_j$, test for each leaf x in $T(D_i)$ and for each leaf y in $T(D_j)$ whether or not $d_{xy} = \delta_i$. If $d_{xy} \neq \delta_i$, then output the triple $(x, y, 1)$ as a witness that D is not ultrametric and terminate. (If D were ultrametric, then $d_{1x} = \delta_i > \delta_j = d_{1y}$ would imply $d_{xy} = \delta_i$ by the 3-point condition.)

Output: The tree $T(D)$ and a pair of taxa $(1, j)$, where $j \in S_k$.

It is not hard to show that Algorithm 10.1 is correct. If it outputs a triple of taxa, then these three taxa show that D does not satisfy the 3-point condition. Hence D cannot be consistent with an ultrametric tree by Theorem 10.3.4. Otherwise, Algorithm 10.1 outputs a tree $T(D)$ and a pair $(1, j)$ of taxa with $j \in S_k$. By step 4 of the algorithm, $T(D)$ is an ultrametric tree. Moreover, step 5 ensures that $T(D)$ is consistent with D . Finally, note that the pair $(1, j)$ of taxa satisfies $d_{1j} = \delta_k$, and δ_k is the mark of the root of $T(D)$.

Our next goal is to show that the worst-case time complexity of Algorithm 10.1 is $O(n^2)$. Step 2 can be implemented in $O(kn)$ time using a linked list⁴ of “buckets” (sets that can also be implemented as linked lists). The linked list maintains the δ -values in increasing order and each value δ_i is associated with a bucket, called the δ_i -bucket. We scan the values d_{12}, \dots, d_{1n} and for each value d_{1j} we linearly search through the list until either (a) an element δ_i with $\delta_i = d_{1j}$ is found or (b) the right place to insert the new d_{1j} -value is found. In case (a) we put j into the already existing δ_i -bucket, and in case (b) we make up a new d_{1j} -bucket containing j . Clearly, each of the $n - 1$ scans through the list takes $O(k)$ time, so the overall time complexity is $O(kn)$. After all values d_{12}, \dots, d_{1n} have been processed, the list has the form $[\delta_1, \dots, \delta_k]$, where $\{\delta_1, \dots, \delta_k\} = \{d_{12}, \dots, d_{1n}\}$ and $\delta_1 < \dots < \delta_k$. Furthermore, each δ_i -bucket contains the set S_i .

It is readily verified that step 4 of Algorithm 10.1 takes $O(k)$ time.

In the analysis of step 5, it is important to note that $\sum_{i=1}^k n_i = n - 1$, where $n_i = |S_i|$. Consequently, there are

$$\frac{1}{2} \left(\sum_{i=1}^k n_i(n - 1 - n_i) \right) = \frac{1}{2} \left((n - 1) \sum_{i=1}^k n_i - \sum_{i=1}^k n_i^2 \right) = \frac{1}{2} \left((n - 1)^2 - \sum_{i=1}^k n_i^2 \right)$$

many pairs (x, y) of taxa to which the (constant time) test $d_{xy} = \delta_i$ must be applied. Thus, step 5 takes $O(n^2 - \sum_{i=1}^k n_i^2)$ time.

To sum up, the worst case running time $T(n)$ of Algorithm 10.1 can be described by the recurrence

$$T(n) = O(kn) + \sum_{i=1}^k T(n_i) + O(n^2 - \sum_{i=1}^k n_i^2) \quad \text{where} \quad \sum_{i=1}^k n_i = n - 1$$

We claim that the solution is $T(n) = O(n^2)$. To prove that $T(n) \leq c \cdot n^2$ for an appropriate choice of the constant $c > 0$, the following lemma is useful.

⁴Using balanced search trees, step 2 can be implemented in $O(n \log k)$ time, but we are content with the $O(kn)$ time complexity.

Lemma 10.3.5 *The following inequality holds:*

$$\sum_{i=1}^k n_i^2 \leq k - 1 + (n - k)^2$$

Proof Without loss of generality, let $1 \leq n_1 \leq n_2 \leq \dots \leq n_k$. For $2 \leq n_\ell \leq n_k$, we have

$$\begin{aligned} (n_\ell - 1)^2 + (n_k + 1)^2 &= n_\ell^2 - 2n_\ell + 1 + n_k^2 + 2n_k + 1 \\ &= n_\ell^2 + 2(n_k - n_\ell) + n_k^2 + 2 \\ &> n_\ell^2 + n_k^2 \end{aligned}$$

That is, $\sum_{i=1}^k n_i^2$ attains its maximum at $n_1 = n_2 = \dots = n_{k-1} = 1$ and $n_k = n - k$. \square

Theorem 10.3.6 $T(n) \leq c \cdot n^2$ for an appropriate choice of the constant $c > 0$ and all $n \in \mathbb{N}$.

Proof For an appropriate choice of the constants $\bar{c} > 0$ and $\tilde{c} > 0$ the recurrence can be written as $T(n) \leq \bar{c} \cdot k \cdot n + \sum_{i=1}^k T(n_i) + \tilde{c} \cdot (n^2 - \sum_{i=1}^k n_i^2)$. We choose $c = \bar{c} + \tilde{c}$ and prove the theorem by induction on n . For the induction step, suppose that the claim holds for every natural number n_i with $n_i < n$. Then,

$$\begin{aligned} T(n) &\leq \bar{c} \cdot k \cdot n + \sum_{i=1}^k T(n_i) + \tilde{c} \cdot (n^2 - \sum_{i=1}^k n_i^2) \\ &\stackrel{I.H.}{\leq} \bar{c} \cdot k \cdot n + \sum_{i=1}^k c \cdot n_i^2 + \tilde{c} \cdot (n^2 - \sum_{i=1}^k n_i^2) \\ &= \bar{c} \cdot k \cdot n + (\bar{c} + \tilde{c}) \cdot \sum_{i=1}^k n_i^2 + \tilde{c} \cdot n^2 - \tilde{c} \cdot \sum_{i=1}^k n_i^2 \\ &= \bar{c} \cdot (k \cdot n + \sum_{i=1}^k n_i^2) + \tilde{c} \cdot n^2 \\ &\leq (\bar{c} + \tilde{c}) \cdot n^2 \end{aligned}$$

provided that $kn + \sum_{i=1}^k n_i^2 \leq n^2$. The latter inequality is implied by the stronger inequality $kn + k - 1 + (n - k)^2 \leq n^2$ because $\sum_{i=1}^k n_i^2 \leq k - 1 + (n - k)^2$ according to Lemma 10.3.5. Now we have

$$\begin{aligned} &kn + k - 1 + (n - k)^2 \leq n^2 \\ \Leftrightarrow &kn + k - 1 + n^2 - 2kn + k^2 \leq n^2 \\ \Leftrightarrow &0 \leq kn - k^2 - k + 1 \\ \Leftrightarrow &0 \leq k \cdot (n - k - 1) + 1 \end{aligned}$$

The last inequality is true because $1 \leq k \leq n - 1$ implies $n - k - 1 \geq 0$. \square

It should be pointed out that the $O(n^2)$ worst-case time complexity of Algorithm 10.1 is optimal because the input matrix D has the size $\Theta(n^2)$.

10.3.3 The UPGMA-algorithm

UPGMA is an agglomerative clustering method used in bioinformatics for the creation of phylogenetic trees. Sokal and Sneath [298] attribute it to Rohlf [268] and Sneath [295]. The acronym UPGMA stands for *un-weighted pair group method using arithmetic averages*. Let us start with some general notes on clustering.

Clustering (or cluster analysis) is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. In our context, the set of observations is the set $S = \{1, \dots, n\}$ of taxa and the dissimilarity matrix D specifies how (dis)similar two clusters are.

Hierarchical clustering creates a hierarchy of clusters represented by a rooted tree (often also called a dendrogram). The root of the tree corresponds to a single cluster containing all taxa (observations), and each leaf corresponds to a single taxon (an individual observation). Algorithms for hierarchical clustering generally fall into two categories:

- **Agglomerative:** Each taxon (observation) starts in its own cluster, and at each step of the algorithm, the pair of clusters with the shortest distance (highest similarity) are combined into a single cluster. The algorithm stops when all taxa (observations) are combined into a single cluster. So this is a kind of bottom-up approach.
- **Divisive:** All taxa (observations) start in one cluster, and at each step of the algorithm, clusters are partitioned into a pair of daughter clusters, selected to maximize the distance (minimize the similarity) between each daughter. So this is a kind of top-down approach.

Since we number the n taxa consecutively and identify a taxon with its number, it is natural to do the same with clusters. To be precise, in the agglomerative method we start with the clusters $C_1 = \{1\}, C_2 = \{2\}, \dots, C_n = \{n\}$; the first new cluster that is obtained by merging two clusters is the cluster C_{n+1} , the second one is the cluster C_{n+2} , and so on. Furthermore, we use the phrases “cluster i ” and “cluster C_i ” interchangeably.

The agglomerative method can further be divided into several types according to how the distance between clusters is defined. Usually the distance $d(i, j)$ between two clusters C_i and C_j of sizes $n_i = |C_i|$ and $n_j = |C_j|$ is one of the following:

- The maximum distance between elements of each cluster (also called complete linkage clustering):

$$d(i, j) = \max\{d_{xy} : x \in C_i, y \in C_j\}$$

- The minimum distance between elements of each cluster (also called single-linkage clustering):

$$d(i, j) = \min\{d_{xy} : x \in C_i, y \in C_j\}$$

- The mean distance between elements of each cluster (also called average linkage clustering, used e.g. in UPGMA):

$$d(i, j) = \frac{1}{n_i n_j} \sum_{x \in C_i, y \in C_j} d_{xy}$$

In this section, we focus on the UPGMA-algorithm, which is shown in Algorithm 10.2. The general procedure of the agglomerative method has already been explained above, so Algorithm 10.2 should be easy to understand (an example is illustrated below). However, there is a discrepancy between Algorithm 10.2 and what was explained above: Algorithm 10.2 uses the equation

$$d(k, \ell) = \frac{n_i \cdot d(i, \ell) + n_j \cdot d(j, \ell)}{n_i + n_j}$$

to calculate the distance of the new cluster $C_k = C_i \cup C_j$ to the “old” cluster C_ℓ , whereas—according to the above definition—it should be calculated as

$$d(k, \ell) = \frac{1}{n_k n_\ell} \sum_{x \in C_k, y \in C_\ell} d_{xy}$$

The discrepancy is settled by the next lemma.

Lemma 10.3.7 *Let $C_k = C_i \cup C_j$. For any cluster C_ℓ with $i \neq \ell \neq j$ we have*

$$d(k, \ell) = \frac{n_i \cdot d(i, \ell) + n_j \cdot d(j, \ell)}{n_i + n_j}$$

Algorithm 10.2 UPGMA algorithm.

Input: $n \times n$ dissimilarity matrix D .

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon $i \in S$ is a leaf in the tree T .
3. Each taxon $i \in S$ defines a cluster $C_i = \{i\}$ of size $n_i = 1$.
4. For all $i, j \in S$, define $d(i, j) = d_{ij}$.

while $|S| \geq 2$ **do**

1. Determine $i, j \in S$ with $i \neq j$ so that $d(i, j)$ is minimal.
2. Let $C_k = C_i \cup C_j$ be a new cluster of size $n_k = n_i + n_j$, and define

$$d(k, \ell) = \frac{n_i \cdot d(i, \ell) + n_j \cdot d(j, \ell)}{n_i + n_j}$$

for each $\ell \in S \setminus \{i, j\}$.

3. Set $S = (S \setminus \{i, j\}) \cup \{k\}$.
4. Add a new node k to T with mark $d(i, j)$.
5. Add edges from k to i and from k to j to the tree T .

Output: The tree T .

D	1	2	3	4	5
1	0	6	9	9	9
2		0	9	9	9
3			0	8	8
4				0	7
5					0

	3	4	5	6
3	0	8	8	9
4		0	7	9
5			0	9
6				0

Figure 10.10: The original dissimilarity matrix D is shown on the left, while the right-hand side shows the dissimilarity matrix after clusters C_1 and C_2 have been merged into the new cluster C_6 .

Proof

$$\begin{aligned}
 d(k, \ell) &= \frac{1}{n_k n_\ell} \sum_{x \in C_k, y \in C_\ell} d_{xy} \\
 &= \frac{1}{(n_i + n_j) n_\ell} \sum_{x \in C_i \cup C_j, y \in C_\ell} d_{xy} \\
 &= \frac{1}{(n_i + n_j) n_\ell} \left(\sum_{x \in C_i, y \in C_\ell} d_{xy} + \sum_{x \in C_j, y \in C_\ell} d_{xy} \right) \\
 &= \frac{1}{(n_i + n_j) n_\ell} (n_i n_\ell d(i, \ell) + n_j n_\ell d(j, \ell)) \\
 &= \frac{n_i d(i, \ell) + n_j d(j, \ell)}{n_i + n_j}
 \end{aligned}$$

□

Exercise 10.3.8 Analyze the running time of Algorithm 10.2. What is the crucial factor—the most time consuming operation—in that algorithm? You should be able to argue that $O(n^3)$ is an upper bound. Can you give an implementation of Algorithm 10.2 with a better worst-case time complexity? In Section 10.3.4, we will show that an $O(n^2)$ time implementation of Algorithm 10.2 is possible, using a data structure called a quadtree.

We now illustrate Algorithm 10.2 by applying it to the dissimilarity matrix D of Figure 10.10. Initially, each taxon i starts in its own cluster C_i , and the distance $d(i, j)$ between two clusters C_i and C_j is d_{ij} . The minimum in D is $d_{12} = 6$. Thus, clusters C_1 and C_2 are merged into a new cluster C_6 . The distances of the new cluster C_6 to the remaining cluster

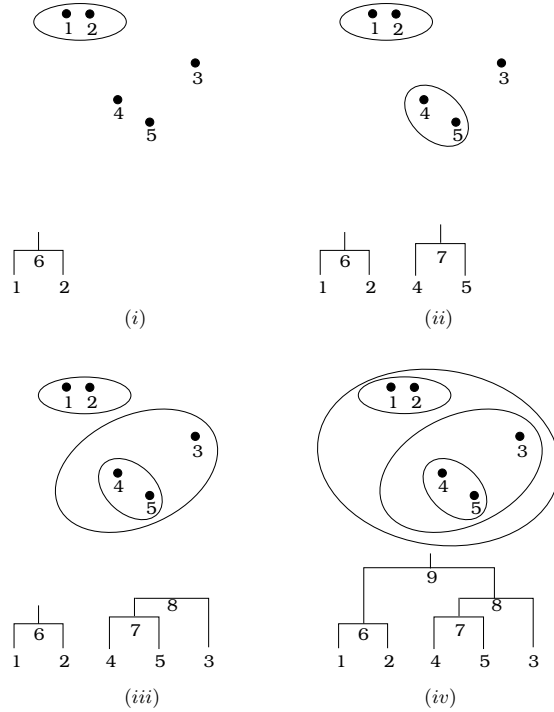


Figure 10.11: The UPGMA-algorithm applied to the dissimilarity matrix D of Figure 10.10 yields the phylogenetic tree shown in (iv). For clarity, the marks at internal nodes are omitted. Intermediate trees are shown in (i)–(iii), and clusters are illustrated above the trees.

C_3 , C_4 , and C_5 are

$$\begin{aligned}
 d(3, 6) &= \frac{1 \cdot d(1, 3) + 1 \cdot d(2, 3)}{1 + 1} = 9 \\
 d(4, 6) &= \frac{1 \cdot d(1, 4) + 1 \cdot d(2, 4)}{1 + 1} = 9 \\
 d(5, 6) &= \frac{1 \cdot d(1, 5) + 1 \cdot d(2, 5)}{1 + 1} = 9
 \end{aligned}$$

The set S is updated to $\{3, 4, 5, 6\}$, the updated dissimilarity matrix is shown on the right-hand side of Figure 10.10, and Figure 10.11 (i) depicts the updated tree.

	3	4	5	6
3	0	8	8	9
4		0	7	9
5			0	9
6				0

	3	6	7
3	0	9	8
6		0	9
7			0

Figure 10.12: Left-hand side: The dissimilarity matrix of clusters 3, 4, 5, 6. Right-hand side: The matrix after clusters 4 and 5 have been combined into the new cluster 7.

	3	6	7
3	0	9	8
6		0	9
7			0

	6	8
6	0	9
8		0

Figure 10.13: Another update of the dissimilarity matrix.

Now $d(4, 5)$ is the shortest distance between two clusters. Therefore, clusters 4 and 5 are merged into a new cluster 6. Figure 10.12 shows the corresponding update of the dissimilarity matrix and Figure 10.11 (ii) depicts the corresponding tree. Then, clusters 3 and 7 are combined into a new cluster 8; see Figures 10.13 and 10.11 (iii). In the final step, the remaining clusters 6 and 8 are merged.

Lemma 10.3.9 *The sequence of marks generated in the $n - 1$ iterations of the UPGMA-algorithm is increasing.*

Proof We prove the lemma by induction on the number q of iterations of the while-loop of the UPGMA-algorithm. The lemma is vacuously true for $q = 0$. Let $\mu_1, \mu_2, \dots, \mu_q$ be the sequence of marks generated in the first q iterations of the while-loop. According to the inductive hypothesis, we have $\mu_1 \leq \mu_2 \leq \dots \leq \mu_q$. Suppose that in the q -th iteration the clusters C_i and C_j are merged into a new cluster because $d(i, j)$ is the shortest distance between two clusters. Note that $\mu_q = d(i, j)$. In the $(q + 1)$ -th iteration, let $d(k, \ell)$ be the shortest distance between two clusters, so that C_k and C_ℓ are merged into a new cluster. That is, the new node gets the mark $\mu_{q+1} = d(k, \ell)$. In order to show $\mu_q = d(i, j) \leq d(k, \ell) = \mu_{q+1}$, we have to consider three possibilities. In the q -th iteration either

- clusters C_i and C_j are merged into the cluster C_k , or
- clusters C_i and C_j are merged into the cluster C_ℓ , or
- clusters C_i , C_j , C_k , and C_ℓ are pairwise disjoint (i.e., none of the preceding two cases applies).

We show that $d(i, j) \leq d(k, \ell)$ for the first case (the second case is symmetric, and the third case is easy). The fact that $d(i, j)$ is the shortest distance in the q -th iteration has $d(i, j) \leq d(i, \ell)$ and $d(i, j) \leq d(j, \ell)$ as a consequence. Thus,

$$\begin{aligned} d(k, \ell) &= \frac{n_i d(i, \ell) + n_j d(j, \ell)}{n_i + n_j} \\ &\geq \frac{n_i d(i, j) + n_j d(i, j)}{n_i + n_j} \\ &= d(i, j) \end{aligned}$$

To sum up, $\mu_q = d(i, j) \leq d(k, \ell) = \mu_{q+1}$. \square

The next goal is to show that UPGMA produces the correct tree provided that the input is an ultrametric dissimilarity matrix D . There is one obstacle though: UPGMA always constructs a binary tree and there may not be an ultrametric binary tree that is consistent with D . As an example, consider three taxa with a pairwise dissimilarity of 1. The star phylogeny of these three taxa, where the internal node r is the root of the tree and $\mu(r) = 1$, is consistent with this dissimilarity matrix, but UPGMA cannot construct it. Instead, it constructs a binary tree in which both internal nodes have mark 1. The problem of having edges (v, w) with $\mu(v) = \mu(w)$ is avoided in Algorithm 10.1 (page 498) by identifying such nodes v and w . Of course, we could modify UPGMA accordingly, but refrain from doing so (to avoid confusion with the usual definition of UPGMA given in the literature). Instead, we will prove that UPGMA produces the correct tree provided that the dissimilarity matrix D is consistent with an ultrametric binary tree.

Lemma 10.3.10 *Let D be an ultrametric dissimilarity matrix, and let $T(D)$ be the unique ultrametric tree that is consistent with D . Furthermore, let T be the binary tree returned by UPGMA when applied to D . If we delete all edges (v, w) from T for which $\mu(v) = \mu(w)$ holds and identify the nodes v and w , then we obtain $T(D)$.*

Proof According to Lemma 10.3.9, the sequence of marks generated in the $n - 1$ iterations of the UPGMA-algorithm is increasing. This implies that for each path v_1, v_2, \dots, v_m, k in T from the root $r = v_1$ to a leaf k , the sequence of marks $\mu(v_1), \mu(v_2), \dots, \mu(v_m)$ is decreasing. Thus, if we delete all edges (v, w) from T for which $\mu(v) = \mu(w)$ holds and identify the nodes v and w , then we obtain an ultrametric tree. It remains to be shown that this tree is consistent with D , i.e., for any two taxa $x, y \in S$ the equality $\mu(\text{LCA}(x, y)) = d_{xy}$ must hold. This, in turn, is a consequence of the following statement.

If UPGMA is applied to an ultrametric matrix D , then after each iteration of its while-loop the following holds:

- For all clusters C_ℓ and C_m with $\ell, m \in S$ and $\ell \neq m$ and all taxa $x \in C_\ell$ and $y \in C_m$, we have $d_{xy} = d(\ell, m)$.

The proof is by induction on the number q of iterations of the while-loop. The lemma is certainly true for $q = 0$ (before the while-loop is entered for the first time). After the q -th iteration, for all clusters C_ℓ and C_m with $\ell, m \in S$ and $\ell \neq m$ and all taxa $x \in C_\ell$ and $y \in C_m$, we have $d_{xy} = d(\ell, m)$ by the inductive hypothesis. Suppose that in the $(q + 1)$ -th iteration the clusters C_i and C_j are merged into a new cluster C_k because $d(i, j)$ is the shortest distance between two clusters. It must be shown that $d_{xz} = d(k, \ell)$ for all taxa $x \in C_k$ and $z \in C_\ell$, where $k \neq \ell$. Let $x \in C_i$, $y \in C_j$, and $z \in C_\ell$ be arbitrary but fixed. The ultrametric matrix D satisfies the 3-point condition. Thus, the two largest values out of $d_{xy} = d(i, j)$, $d_{xz} = d(i, \ell)$, $d_{yz} = d(j, \ell)$ are equal. Since $d(i, j)$ is the shortest distance between two clusters, it follows that $d_{xz} = d_{yz}$. Consequently,

$$\begin{aligned}
 d(k, \ell) &= \frac{n_i \cdot d(i, \ell) + n_j \cdot d(j, \ell)}{n_i + n_j} \\
 &= \frac{n_i \cdot d_{xz} + n_j \cdot d_{yz}}{n_i + n_j} \\
 &= \frac{(n_i + n_j) \cdot d_{xz}}{n_i + n_j} \\
 &= d_{xz}
 \end{aligned}$$

This concludes the proof. \square

Corollary 10.3.11 *If the dissimilarity matrix D is consistent with an ultrametric binary tree, then UPGMA applied to D constructs this tree.*

Proof According to Theorem 10.3.4, D is consistent with an ultrametric tree if and only if D is ultrametric. Moreover, the ultrametric tree $T(D)$ for D is unique. According to the preceding lemma, $T(D)$ can be obtained from the binary tree T constructed by UPGMA by deleting all edges (v, w) from T for which $\mu(v) = \mu(w)$ holds and identifying the nodes v and w . However, if there were such an edge in T , then the resulting tree $T(D)$ would not be binary. Hence $T = T(D)$. \square

Exercise 10.3.12 The WPGMA-algorithm [296] (WPGMA is an acronym for *weighted pair group method using arithmetic averages*) is almost identical to the UPGMA-algorithm. The sole difference between these algorithms is the way in which the distance of an “old” C_ℓ cluster to the new cluster C_k is defined. The WPGMA-algorithm uses the formula $d(k, \ell) = \frac{d(i, \ell) + d(j, \ell)}{2}$. Show that if the input is an ultrametric matrix D , then the outputs of the UPGMA-algorithm and the WPGMA-algorithm coincide.

Note that the terms weighted and unweighted refer to the labels created by the algorithms, not the mathematics by which it is achieved. Thus the simple averaging in WPGMA produces a weighted result, and the proportional averaging in UPGMA produces an unweighted result.

10.3.4 Fast UPGMA implementation based on quadrees

In this section, we present an implementation of the UPGMA-algorithm that is based on a quadtree and runs in quadratic time; see [89]. (It has been known since at least 1984 that UPGMA can be implemented in $O(n^2)$ time [133, 232].) A *quadtree* is a tree data structure in which each internal node has four children. Its name was coined by its inventors Finkel and Bentley [103]. Quadrees are most often used to partition a two dimensional space (in our context, the dissimilarity matrix D) by recursively subdividing it into four quadrants (or regions).

Let $D = (d_{ij})_{1 \leq i, j \leq n}$ be an $n \times n$ matrix. For ease of presentation, we will henceforth assume that n is a power of 2. The matrix is subdivided into four submatrices:

- The northwest quadrant I: this is the matrix (d_{ij}) where $1 \leq i, j \leq n/2$.
- The northeast quadrant II: this is the matrix (d_{ij}) where $1 \leq i \leq n/2$ and $n/2 + 1 \leq j \leq n$,
- The southwest quadrant III: the matrix (d_{ij}) where $n/2 + 1 \leq i \leq n$ and $1 \leq j \leq n/2$.
- The southeast quadrant IV: the matrix (d_{ij}) where $n/2 + 1 \leq i, j \leq n$.

Figure 10.14 illustrates this decomposition.

Definition 10.3.13 Let D be an $n \times n$ matrix of elements from a totally ordered set. A *quadtree* of D is a labeled quaternary tree defined as follows:

- The root of the tree is labeled with the minimum element of D . If $n > 1$, then the tree has four children:
 - The first child of the root is the quadtree of the quadrant I.
 - The second child of the root is the quadtree of the quadrant II.
 - The third child of the root is the quadtree of the quadrant III.
 - The fourth child of the root is the quadtree of the quadrant IV.

Figure 10.14 illustrates this recursive definition for $n = 4$.

We construct the quadtree of an $n \times n$ matrix D , where $n = 2^k$, in a bottom-up fashion. For ease of presentation, we simultaneously define a (merely conceptual) matrix A_l for each level l with $0 \leq l \leq k$:

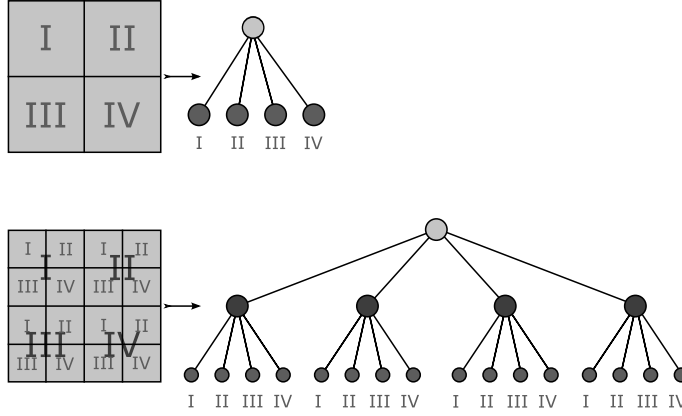


Figure 10.14: Quadtree

- $l = k$:
 - $A_k = D$
 - The k -th level of the quadtree consists of $n^2 = (2^k)^2 = 4^k$ leaves, labeled with the values d_{ij} , where $1 \leq i, j \leq n$.
- $l < k$:
 - The matrix $A_l = (a_{ij}^l)$, $1 \leq i, j \leq 2^l$, is defined by

$$a_{ij}^l = \min\{a_{2i-1, 2j-1}^{l+1}, a_{2i-1, 2j}^{l+1}, a_{2i, 2j-1}^{l+1}, a_{2i, 2j}^{l+1}\}$$
 - Given the 4^{l+1} nodes with labels a_{ij}^{l+1} ($1 \leq i, j \leq 2^{l+1}$) at level $l+1$ ($0 \leq l < k$), the l -th level of the quadtree consists of 4^l nodes with labels a_{pq}^l ($1 \leq p, q \leq 2^l$) and the node a_{pq}^l is the parent node of the four nodes labeled with $a_{2p-1, 2q-1}^{l+1}$ (first child), $a_{2p-1, 2q}^{l+1}$ (second child), $a_{2p, 2q-1}^{l+1}$ (third child), and $a_{2p, 2q}^{l+1}$ (fourth child).

The quadtree of an $n \times n$ matrix D , where $n = 2^k$, can be constructed in $O(n^2)$ time and space. This is because the k -th level of the tree has n^2 nodes, the $(k-1)$ -th level has $\frac{n^2}{4}$ nodes, the $(k-2)$ -th level has $\frac{n^2}{4^2}$ nodes, etc. So the overall number of nodes in the quadtree is

$$n^2 \sum_{i=0}^k \left(\frac{1}{4}\right)^i = n^2 \frac{1 - \left(\frac{1}{4}\right)^{k+1}}{1 - \frac{1}{4}} < n^2 \frac{1}{1 - \frac{1}{4}} = \frac{4}{3} n^2$$

Lemma 10.3.14 Given the quadtree of an $n \times n$ matrix D , the minimum element of D can be identified in constant time. A position (i, j) at which the minimum occurs in D can be determined in $O(\log n)$ time.

Proof By construction, the label of the root of the quadtree is the minimum element of D . A position (i, j) at which this minimum occurs in D can be determined by following the path from the root to a leaf v_l with the same label as the root. To be more precise, at each node v the path passes through a child node having the same label as v . Because every internal node has four children, following this path takes time proportional to the depth of the quadtree, which is $\log_4 n^2 = \log_2 n$. It is not difficult to see that the position (i, j) corresponding to leaf v_l can be computed by means of the path from the root to v_l : each time the path passes from a parent node to one of its four children, the choice of the child node determines the next smaller quadrant in which the position (i, j) can be found. \square

It is clear that the time to determine a position (i, j) of the minimum element of D can be improved to $O(1)$ time if one uses elements *and* their position in the matrix D as labels of the nodes of the quadtree.

Lemma 10.3.15 *If the matrix D' is obtained from an $n \times n$ matrix D by updating one row (or one column), then the quadtree of D' can be obtained in $O(n)$ time by updating the quadtree of D .*

Proof At most n elements change in the update of matrix $D = A_k$, where $k = \log n$. Because only two (non-diagonal) quadrants are affected by the update, at most $\frac{n}{2}$ elements change in matrix A_{k-1} . Analogously, at most $\frac{n}{2^2}$ elements change in matrix A_{k-2} , and so on. So the overall number of nodes in the quadtree whose labels may possibly be updated is

$$n \sum_{i=0}^k \left(\frac{1}{2}\right)^i = n \frac{1 - \left(\frac{1}{2}\right)^{k+1}}{1 - \frac{1}{2}} < n \frac{1}{1 - \frac{1}{2}} = 2n$$

This proves that at most $2n - 1$ nodes in the quadtree of D are affected by the update. Since n of them are leaves, at most $n - 1$ labels at internal nodes must be updated. Because the new label of an internal node is the minimum of its current label and the new labels of the affected children, the update takes constant time provided that the labels of its children have already been updated. Thus, if we update the labels in a bottom-up fashion, the update of the quadtree takes $O(n)$ time. \square

Obviously, the preceding lemma remains true if we update one row *and* one column (or even two rows and two columns). An illustration of the necessary changes can be found in Figure 10.15.

It is almost clear now that the quadtree of a dissimilarity matrix D enables a quadratic time implementation of the UPGMA-algorithm, but there are still some subtleties. Because we wish to maintain a closest pair of clusters, the 0-entries on the main diagonal of matrix D must be set to infinity i.e., $d_{ii} = \infty$ for all i with $1 \leq i \leq n$. Moreover, either the entries

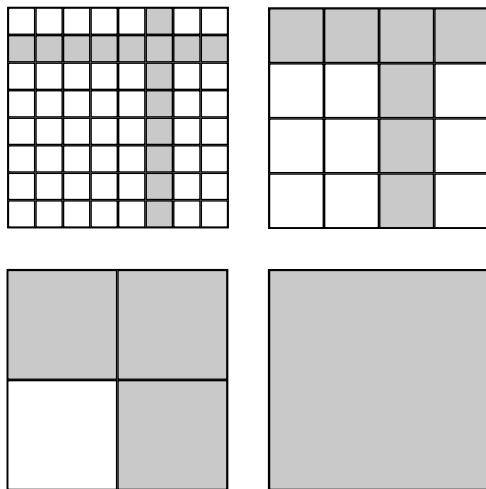


Figure 10.15: Changes in the quadtree (in the matrices A_3, A_2, A_1, A_0) of an 8×8 matrix D , when row $i = 2$ and column $j = 6$ are updated.

D	1	2	3	4
1	0	9	9	9
2		0	7	7
3			0	4
4				0

D	1	2	3	4
1	∞	9	9	9
2	∞	∞	7	7
3	∞	∞	∞	4
4	∞	∞	∞	∞

Figure 10.16: The entries in the lower triangle and on the main diagonal of the dissimilarity matrix D are set to infinity.

below the main diagonal (the lower triangle) or the entries above the main diagonal (the upper triangle) are redundant because D is symmetric. For this reason, the entries in the lower triangle are set to infinity as well.

Figure 10.16 shows a dissimilarity matrix D and its modified form. The quadtree of the modified matrix is shown in Figure 10.17.

In each of the $n - 1$ iterations, UPGMA merges the two closest clusters C_i and C_j into a new cluster C_k . According to Lemma 10.3.14, it takes $O(\log n)$ time to determine the two closest clusters. Then, the new distance matrix is obtained by deleting the rows and columns corresponding to the clusters C_i and C_j , and adding a new row and column for cluster C_k . We reuse the row and column corresponding to cluster C_i , that is, we store the new distances of cluster C_k to the remaining clusters there. The deletion of the row and column corresponding to cluster C_j is implemented by setting

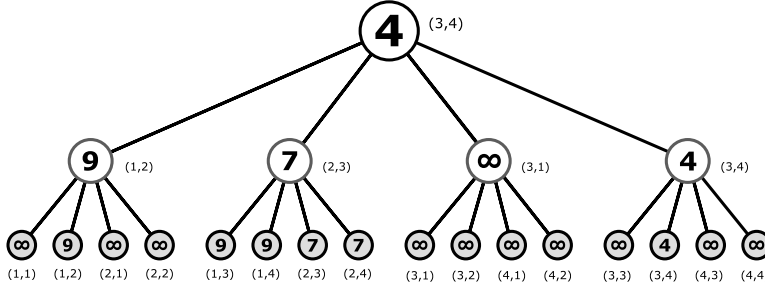


Figure 10.17: The quadtree of the modified dissimilarity matrix D of Figure 10.16. For ease of exposition, the nodes are labeled with the minima and their positions in the matrix D .

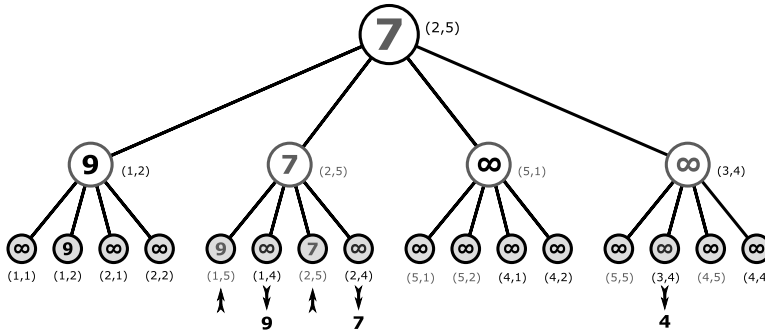


Figure 10.18: Quadtree after the first iteration of UPGMA.

their entries to infinity. The quadtree of the new distance matrix can be obtained in $O(n)$ time according to Lemma 10.3.15. Thus, the quadtree implementation of UPGMA has an overall worst-case time complexity of $O(n^2)$.

In the example of Figures 10.16 and 10.17, UPGMA merges the two closest clusters (taxa) 3 and 4 into a new cluster 5. The distances of this new cluster to the remaining clusters (taxa) are $d(5, 1) = 9$ and $d(5, 2) = 7$. Figure 10.18 depicts the resulting update of the quadtree.

Exercise 10.3.16 To continue the example above, infer from the label of the root of the quadtree of Figure 10.18 that now clusters 2 and 5 have minimum distance, namely $d(2, 5) = 7$. Consequently, in the next iteration UPGMA merges these two clusters into a new cluster 6. The distance of this new cluster to the only remaining cluster 1 is $d(6, 1) = 9$. Update the quadtree of Figure 10.18 accordingly.

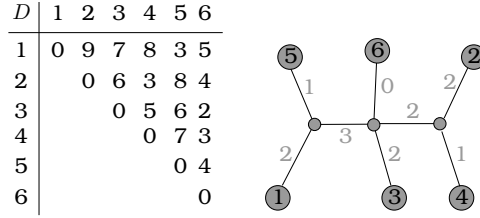


Figure 10.19: The dissimilarity matrix D and the additive tree T are consistent.

10.4 Additive distance matrices and trees

In an influential paper, Waterman et al. [328] gave an $O(n^2)$ time algorithm to construct additive trees; see [28] for a historical sketch of work on additive trees. In Sections 10.4.2–10.4.4, we will follow Gusfield’s approach [139, 17.4] that reduces the additive tree problem (devise an efficient algorithm to construct an additive tree) to the ultrametric tree problem; see also [151].

Definition 10.4.1 An *additive tree* is a phylogenetic tree $T = (V, E, \lambda)$ on a set of taxa S together with a weight function $\gamma : E \rightarrow \mathbb{R}_{\geq 0}$ so that $\gamma(v, w) > 0$ for each internal edge (v, w) . The distance between two nodes v and w in T is defined by $d_T(v, w) = \sum_{k=1}^{m-1} \gamma(v_k, v_{k+1})$, where v_1, v_2, \dots, v_m is the (unique) path between $v = v_1$ and $w = v_m$ in T .

In an additive tree, external edges may have weight 0. This is because we want to avoid taxa at internal nodes.

Definition 10.4.2 Let $S = \{1, \dots, n\}$ be a set of taxa, let D be an $n \times n$ dissimilarity matrix, and let $T = (V, E, \lambda, \gamma)$ be an additive tree on S . D and T are *consistent* if for any two leaves (taxa) i and j in T the equality $d_T(i, j) = d_{ij}$ holds.

If an additive tree T is consistent with a dissimilarity matrix D , then there cannot be two leaves $i \neq j$ adjacent to the same internal node v so that $\gamma(v, i) = 0 = \gamma(v, j)$ because $d_T(i, j) = d_{ij} > 0$.

Figure 10.19 shows an additive tree T that is consistent with the given dissimilarity matrix D .

Note that it is possible to test in $O(n^2)$ time whether an $n \times n$ dissimilarity matrix D and an additive tree T on a set of n taxa are consistent. Because each internal node has a degree ≥ 3 and there n leaves, it follows that T has at most $n - 1$ internal nodes. So, for each leaf i ($1 \leq i \leq n$) in T , one can calculate the distance of i to any other node in $O(n)$ time.

10.4.1 Ultrametric trees revisited

This section shows that ultrametric trees correspond to additive trees that satisfy the molecular clock hypothesis. In an ultrametric tree $T' = (V, E, \lambda, \mu)$ that is consistent with a dissimilarity matrix D , the value $\mu(v)$ is the distance d_{ij} between any two taxa i and j with last common ancestor v . If an additive tree T is consistent with the same dissimilarity matrix D , then the distance d_{ij} between i and j is the sum of the edge weights on the path from i to j , where edge weights represent expected amounts of evolution. In particular, $d_{ij} = d_T(i, v) + d_T(v, j)$. If evolutionary changes occur at a constant rate (i.e., the molecular clock hypothesis holds true), then the species (taxa) i and j diverged $t(v)$ time ago, and the distance $d_{ij} = \mu(v)$ between i and j should be $d_T(i, v) + d_T(v, j) = t(v) + t(v)$ because both lineages evolved in the same amount $t(v)$ of time. In summary, if we set $t(v) = \frac{\mu(v)}{2}$ for every internal node $v \in V$ in an ultrametric tree $T' = (V, E, \lambda, \mu)$, then we obtain an additive tree in which the lengths of all evolutionary paths are equal. This is formally proven in the next lemma.

Lemma 10.4.3 *Let $T' = (V, E, \lambda, \mu)$ be an ultrametric tree that is consistent with a dissimilarity matrix D . For every edge $(v, w) \in E$, define*

$$\gamma(v, w) = \begin{cases} \left| \frac{\mu(v) - \mu(w)}{2} \right| & \text{if both } v \text{ and } w \text{ are internal nodes} \\ \frac{\mu(v)}{2} & \text{if } w \text{ is a leaf} \end{cases}$$

Then the rooted additive tree $T = (V, E, \lambda, \gamma)$ is consistent with D and the lengths of all evolutionary paths are equal.

Proof First, we observe that if v_1 is an internal node in T that lies on a path from the root r to leaf k , then the distance between v_1 and k in T can be computed as follows. Let v_1, v_2, \dots, v_t, k be the path from v_1 to k . As illustrated in Figure 10.20, we have

$$\begin{aligned} d_T(v_1, k) &= \sum_{i=1}^{t-1} \gamma(v_i, v_{i+1}) + \gamma(v_t, k) \\ &= \sum_{i=1}^{t-1} \frac{1}{2}(\mu(v_i) - \mu(v_{i+1})) + \frac{\mu(v_t)}{2} \\ &= \frac{\mu(v_1)}{2} \end{aligned}$$

It follows as a direct consequence that T is consistent with D , because for two leaves i and j in T with $\text{LCA}(i, j) = v$ we have

$$d_T(i, j) = d_T(i, v) + d_T(v, j) = \frac{\mu(v)}{2} + \frac{\mu(v)}{2} = \mu(v) = d_{ij}$$

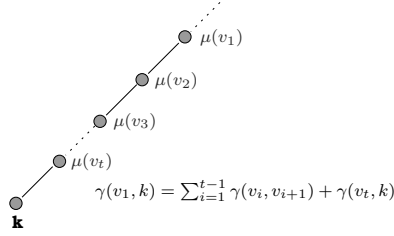


Figure 10.20: The distance $d_T(v_1, k)$ from the internal node v_1 to a leaf k on a path from the root r to k is $\frac{\mu(v_1)}{2}$ because $\sum_{i=1}^{t-1} \gamma(v_i, v_{i+1}) + \gamma(v_t, k) = \sum_{i=1}^{t-1} \frac{1}{2}(\mu(v_i) - \mu(v_{i+1})) + \frac{\mu(v_t)}{2} = \frac{\mu(v_1)}{2}$.

Moreover, for every evolutionary path from the root r to a leaf k , the equality $d_T(r, k) = \frac{\mu(r)}{2}$ holds true. This proves the lemma. \square

Conversely, if a rooted additive tree $T = (V, E, \lambda, \gamma)$ is consistent with a dissimilarity matrix D and the lengths of all evolutionary paths in T are equal, then we obtain an ultrametric tree $T' = (V, E, \lambda, \mu)$ that is consistent with D : For every internal node $v \in V$ define $\mu(v) = 2 \cdot d_T(v, k)$, where k is any leaf in the subtree of T rooted at v . The proof is left as an exercise to the reader.

These facts explain the usual definition of an ultrametric tree: An ultrametric tree is a rooted additive tree in which all evolutionary paths have equal lengths.

It should be pointed out that in its most common formulation, the UPGMA-algorithm constructs a rooted additive tree. This formulation can be obtained by the following modification of Algorithm 10.2 (page 503). When UPGMA merges two clusters C_i and C_j into a new cluster C_k , it places a new node k at height $h(k) = \frac{d_{ij}}{2}$ (instead of marking node k with d_{ij}) and adds an edge from k to i with weight $h(k) - h(i)$ and an edge from k to j with weight $h(k) - h(j)$ to the tree.

10.4.2 Reduction of the additive tree problem

In this section, we sketch a method that constructs an additive tree T that is consistent with an additive (input) matrix D . The method consists of three phases:

1. The matrix D is transformed into an ultrametric matrix D' .
2. The unique ultrametric tree T' that is consistent with D' is constructed by Algorithm 10.1 (page 498).

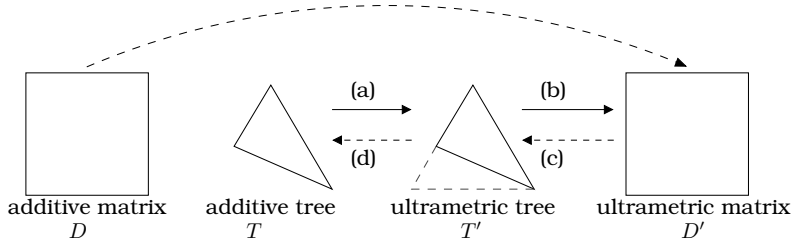


Figure 10.21: Transformations (a) and (b) can be inverted by (c) and (d).

3. The tree T' is transformed into the tree T .

These three phases correspond to the dotted arrows in Figure 10.21. For didactic reasons, we shall assume for a moment that we already know an (unrooted) additive tree T that is consistent with D . It will be shown how (a) T can be transformed into an ultrametric tree T' , from which (b) the ultrametric matrix D' can easily be inferred; see Figure 10.21. Armed with this knowledge, it will become easier to understand phases (1) and (3) of the above-mentioned method.

Let us elaborate on the transformations (a) – (d) of Figure 10.21. We will illustrate each transformation by means of the additive tree depicted in Figure 10.19 (page 514).

(a) Transform the additive tree T into an ultrametric tree T' .

- i. Let $d_T(i, j) = d_{ij}$ be the longest distance between two leaves in T .
- ii. In essence, root the tree T at leaf i . Strictly speaking, we must proceed a little differently because the resulting tree is not a phylogenetic tree. If i is adjacent to the node p , we remove the edge (p, i) and root the tree at a new node r by adding the edges (r, p) with weight $\gamma(p, i)$ and (r, i) with weight 0.

For two leaves (taxa) k and l in the rooted tree with $v = \text{LCA}(k, l)$, we have $2d_T(v, k) = d_{ik} + d_{kl} - d_{il}$ by additivity; see Figure 10.22.

- iii. The adapted tree T' is obtained from the rooted tree by elongating external edges so that each evolutionary path has length d_{ij} . To be precise, the weight d of an external edge (v, k) in the rooted tree is changed to $d' = d + d_{ij} - d_{ik}$ in the adapted tree T' ; see Figure 10.23. It then follows $d_{T'}(r, k) = d_{T'}(r, v) + d + d_{ij} - d_{ik} = d_T(i, v) + d + d_{ij} - d_{ik} = d_T(i, k) + d_{ij} - d_{ik} = d_{ik} + d_{ij} - d_{ik} = d_{ij}$.

For every external edge (v, k) , there must be a leaf $l \neq k$ in the subtree rooted at v ; see Figure 10.24. Since $v = \text{LCA}(k, l)$ it fol-

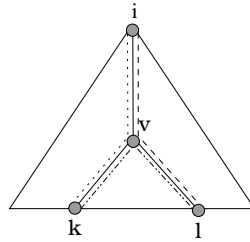


Figure 10.22: $2 d_T(v, k) = d_{ik} + d_{kl} - d_{il}$ and $2 d_T(v, l) = d_{il} + d_{kl} - d_{ik}$.

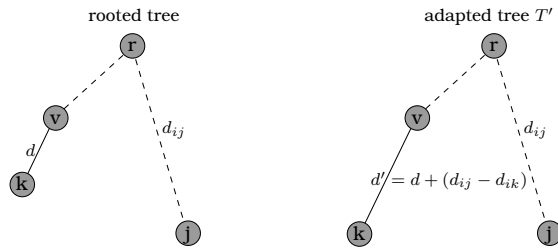


Figure 10.23: Elongation of external edges.

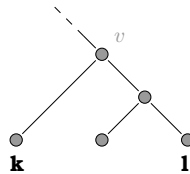


Figure 10.24: In the subtree of T' with root v , there is a leaf $l \neq k$.

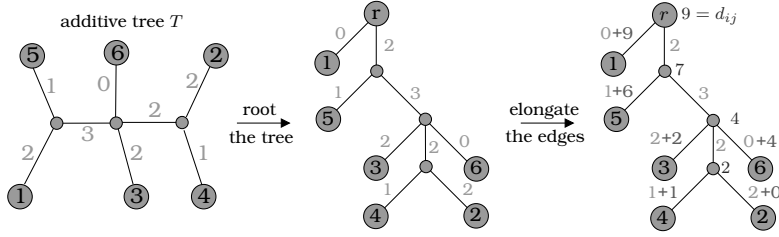


Figure 10.25: The additive tree T from Figure 10.19 is transformed into the ultrametric tree T' .

lows from additivity (Figure 10.22) that

$$\begin{aligned}
 d_{T'}(v, k) &= d_T(v, k) + d_{ij} - d_{ik} \\
 &= \frac{1}{2}(d_{ik} - d_{il} + d_{kl}) + d_{ij} - d_{ik} \\
 &= d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})
 \end{aligned}$$

and analogously

$$\begin{aligned}
 d_{T'}(v, l) &= d_T(v, l) + d_{ij} - d_{il} \\
 &= \frac{1}{2}(d_{il} - d_{ik} + d_{kl}) + d_{ij} - d_{il} \\
 &= d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})
 \end{aligned}$$

- iv. Let V'_I be the set of internal nodes of T' . Define $\mu : V'_I \rightarrow \mathbb{R}_{>0}$ by $\mu(v) = d_{T'}(v, k)$, where k is a leaf in the subtree rooted at v (note that $d_{T'}(v, k) = d_{T'}(v, l)$ for any other leaf $l \neq k$ in that subtree). Mark each internal node v with $\mu(v)$.

Figure 10.25 illustrates transformation (a). In the additive tree T , the longest distance between two leaves (taxa) is $d_T(1, 2) = 9$. Therefore, we root T at node 1 as explained above. Then, we elongate external edges so that each evolutionary path $d_{T'}(r, k)$ has length $d_T(i, j) = d_{ij}$. Finally, we mark each internal node v with $\mu(v)$.

- (b) The ultrametric matrix D' that is consistent with the ultrametric tree T' can be directly read off the tree: d'_{kl} is defined by $d'_{kl} = \mu(\text{LCA}(k, l))$ for each pair of taxa k and l . In our example, we obtain the matrix D' of Figure 10.26.

D'	1	2	3	4	5	6
1	0	9	9	9	9	9
2		0	4	2	7	4
3			0	4	7	4
4				0	7	4
5					0	7
6						0

Figure 10.26: The ultrametric matrix D' is consistent with the ultrametric tree T' from Figure 10.25.

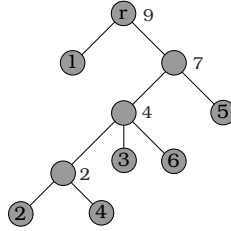


Figure 10.27: Algorithm 10.1 applied to D' yields the ultrametric tree T' .

We stress that the matrix D' can be directly inferred from the matrix D because for each pair of taxa k and l with $v = \text{LCA}(k, l)$, we have

$$d'_{kl} = \mu(v) = d_{T'}(v, k) = d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})$$

as explained above. This transition from the matrix D to the matrix D' is called a *Farris transform* in the literature [95, 96].

- (c) Algorithm 10.1 applied to the ultrametric matrix D' yields the ultrametric tree T' shown in Figure 10.27.
- (d) The additive tree T emerges from the ultrametric tree T' by reversing the transformation (a). Marks at internal nodes are removed and edge weights are introduced, where external edges are shortened appropriately:

$$\gamma(v, k) = \begin{cases} |\mu(v) - \mu(k)| & \text{if neither } v \text{ nor } k \text{ is a leaf} \\ \mu(v) - (d_{ij} - d_{ik}) & \text{if } k \text{ is a leaf} \end{cases}$$

Moreover, the root of T' is replaced with the distinguished taxon i . This back transformation is depicted in Figure 10.28. In the next section, it will be proven that the additive tree T is indeed consistent with D .

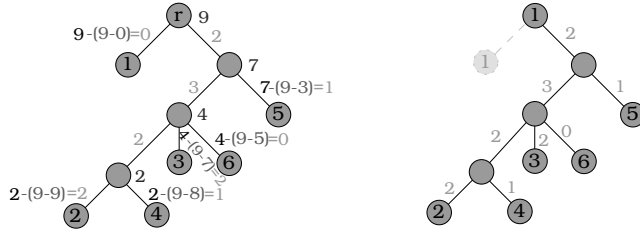


Figure 10.28: The tree T' (upper left) is transformed in the additive tree T (upper right).

10.4.3 Characterization of additive matrices

In this section, we show that additive matrices can be characterized as follows: they satisfy the so-called 4-point condition [47, 333] and they are consistent with an additive tree.

Definition 10.4.4 An $n \times n$ dissimilarity matrix D satisfies the *4-point condition* if for all $i, j, k, l \in \{1, \dots, n\}$ the two largest values out of $d_{ij} + d_{kl}$, $d_{ik} + d_{jl}$, and $d_{il} + d_{jk}$ are equal.

As in the characterization of ultrametric matrices, the proof of the following theorem is constructive in the sense that it not only shows that an additive matrix is consistent with an additive tree, but it also gives an algorithm for constructing the additive tree. This algorithm was instigated in the previous section.

Theorem 10.4.5 Let d_{ij} be the maximum entry in the $n \times n$ dissimilarity matrix D . Let the matrix D' be defined by $d'_{kk} = 0$ and $d'_{kl} = d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})$ for $k \neq l$. Then the following statements are equivalent.

1. D satisfies the 4-point condition.
2. D is additive.
3. D' is ultrametric and D satisfies the triangle inequality.
4. D is consistent with an additive tree.

Proof We prove (1) \Rightarrow (2), (2) \Rightarrow (3), (3) \Rightarrow (4), and (4) \Rightarrow (1).

(1) \Rightarrow (2): It must be shown that the additive inequality

$$d_{pq} + d_{rs} \leq \max\{d_{pr} + d_{qs}, d_{ps} + d_{qr}\}$$

holds for all $p, q, r, s \in \{1, \dots, n\}$. Clearly, this is an immediate consequence of the fact that the two largest values out of $d_{pq} + d_{rs}$, $d_{pr} + d_{qs}$, and $d_{ps} + d_{qr}$ are equal because D satisfies the 4-point condition.

(2) \Rightarrow (3): According to Lemma 10.2.3, the additive matrix D satisfies the triangle inequality. Furthermore, it is readily verified that D' is a dissimilarity matrix. So all we have to prove is that the ultrametric inequality $d'_{pq} \leq \max\{d'_{pr}, d'_{qr}\}$ holds for all $p, q, r \in \{1, \dots, n\}$. For an indirect proof, suppose that $d'_{pq} > \max\{d'_{pr}, d'_{qr}\}$, i.e., (i) $d'_{pq} > d'_{pr}$ and (ii) $d'_{pq} > d'_{qr}$. By the definition of D' , (i) implies that

$$d'_{pq} = d_{ij} - \frac{1}{2}(d_{ip} + d_{iq} - d_{pq}) > d'_{pr} = d_{ij} - \frac{1}{2}(d_{ip} + d_{ir} - d_{pr})$$

It is readily verified that this is equivalent to $d_{pq} + d_{ir} > d_{pr} + d_{iq}$. Similarly, we obtain $d_{pq} + d_{ir} > d_{qr} + d_{ip}$ from (ii). Putting the two inequalities together into a compound inequality, we get $d_{pq} + d_{ir} > \max\{d_{pr} + d_{iq}, d_{qr} + d_{ip}\}$. This, however, contradicts the fact that D is additive. Hence D' is ultrametric.

(3) \Rightarrow (4): Let T' be the unique ultrametric tree that is consistent with D' . Since D' is the Farris transform of D , we can transform T' into a tree T by transformation (d) from Section 10.4.2: marks at the internal nodes of T' are removed and edge weights are introduced as follows:

$$\gamma(v, k) = \begin{cases} |\mu(v) - \mu(k)| & \text{if } (v, k) \text{ is an internal edge} \\ \mu(v) - (d_{ij} - d_{ik}) & \text{if } (v, k) \text{ is an external edge} \end{cases}$$

However, the resulting (rooted) tree T is additive only if $\mu(v) - (d_{ij} - d_{ik}) \geq 0$ for each external edge (v, k) . To show that this is indeed the case, let $l \neq k$ be a leaf in the subtree rooted at v ; see Figure 10.24 (page 518). Because D' is ultrametric and $v = \text{LCA}(k, l)$, we derive

$$\mu(v) = d'_{kl} = d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})$$

Thus, $\mu(v) \geq d_{ij} - d_{ik}$ is equivalent to

$$\begin{aligned} d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl}) &\geq d_{ij} - d_{ik} \\ \Leftrightarrow \frac{1}{2}(d_{ik} + d_{kl}) &\geq \frac{1}{2}d_{il} \\ \Leftrightarrow d_{ik} + d_{kl} &\geq d_{il} \end{aligned}$$

Clearly, the last inequality holds true because D satisfies the triangle inequality. Hence $\gamma(v, k) = \mu(v) - (d_{ij} - d_{ik}) \geq 0$.

We still have to prove that the (rooted) additive tree T is consistent with D . To this end, let k and l be two arbitrary leaves in T and let $v = \text{LCA}(k, l)$. According to Figure 10.29, the distance between v and k in T is $d_T(v, k) =$

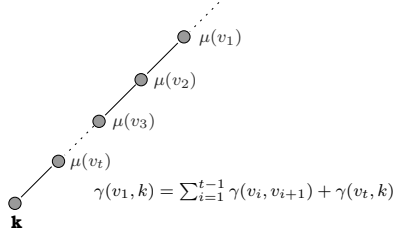


Figure 10.29: The distance $d_T(v_1, k)$ from a leaf k to a node v_1 lying on the path from k to the root of T can be computed by $\sum_{i=1}^{t-1} \gamma(v_i, v_{i+1}) + \gamma(v_t, k) = \sum_{i=1}^{t-1} (\mu(v_i) - \mu(v_{i+1})) + \mu(v_t) - (d_{ij} - d_{ik})$. Thus, $d_T(v_1, k) = \mu(v_1) - (d_{ij} - d_{ik})$.

$\mu(v) - (d_{ij} - d_{ik})$. Clearly, $d_T(v, l)$ can be computed analogously. So we obtain

$$\begin{aligned}
 d_T(k, l) &= d_T(v, k) + d_T(v, l) \\
 &= \mu(v) - (d_{ij} - d_{ik}) + \mu(v) - (d_{ij} - d_{il}) \\
 &= 2\mu(v) - 2d_{ij} + d_{ik} + d_{il} \\
 &\stackrel{\mu(v)=d'_{kl}}{=} 2(d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})) - 2d_{ij} + d_{ik} + d_{il} \\
 &= 2d_{ij} - d_{ik} - d_{il} + d_{kl} - 2d_{ij} + d_{ik} + d_{il} \\
 &= d_{kl}
 \end{aligned}$$

Hence T is consistent with D .

Finally, we show that there is an external edge (r, i) in T with $\gamma(r, i) = 0$, where r is the root of T . In other words, T can further be transformed into an unrooted tree by replacing the root r with the leaf i . Note that for all taxa $k \neq l$ we have $d'_{kl} = d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl}) \leq d_{ij}$ because $d_{ki} + d_{il} \geq d_{kl}$ by the triangle inequality. Therefore, $d'_{ij} = d_{ij}$ is the maximum entry in D' . Consequently, the mark of the root r of the ultrametric tree T' satisfies $\mu(r) = d'_{ij} = d_{ij}$. By the definition of ultrametric trees, the marks of all other internal nodes in T' are strictly smaller than $\mu(r) = d_{ij}$. Now consider the edge (v, i) in T' that leads to the leaf i . Since (v, i) is an external edge, it follows from the discussion above that $\mu(v) \geq d_{ij} - d_{ii} = d_{ij}$. This is possible only if $v = r$. So (r, i) is an external edge in T' and in T . Moreover, it follows from the construction of T that $\gamma(r, i) = \mu(r) - (d_{ij} - d_{ii}) = d_{ij} - d_{ij} = 0$.

(4) \Rightarrow (1): Let the matrix D be consistent with the additive tree T . We must show that D satisfies the 4-point condition, i.e., for all $p, q, r, s \in \{1, \dots, n\}$, the two largest values out of $d_{pq} + d_{rs}$, $d_{pr} + d_{qs}$, and $d_{ps} + d_{qr}$ must be equal. This is done by a case analysis on the location of the leaves (taxa) p, q, r, s in the tree T . Figure 10.30 shows one possible constellation. In that

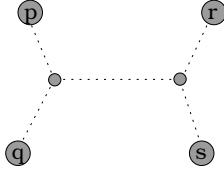


Figure 10.30: In this additive tree T , we have $d_T(p, q) + d_T(r, s) < d_T(p, r) + d_T(q, s) = d_T(p, s) + d_T(q, r)$.

constellation, we have $d_T(p, q) + d_T(r, s) < d_T(p, r) + d_T(q, s) = d_T(p, s) + d_T(q, r)$. Since T and D are consistent, we infer $d_{pq} + d_{rs} < d_{pr} + d_{qs} = d_{ps} + d_{qr}$. The other cases follow similarly. \square

10.4.4 Construction algorithm

According to Theorem 10.4.5, a dissimilarity matrix D is consistent with an additive tree if and only if D is additive, and the proof of that theorem contains an algorithm to construct such a tree. Instead of testing whether D is additive (prior to the construction), Algorithm 10.3 contains this test as a built-in.

If Algorithm 10.3 returns a tree T , then the proof of Theorem 10.4.5 implies that T is additive and consistent with D (hence D is additive). Otherwise, if Algorithm 10.3 terminates in step 2 with failure because Algorithm 10.1 returns a triple (x, y, z) , then we have $d'_{xy} > d'_{xz} = d'_{yz}$. As shown in the proof of the implication (2) \Rightarrow (3) in Theorem 10.4.5, this has

$$d_{iz} + d_{xy} > \max\{d_{iy} + d_{xz}, d_{ix} + d_{yz}\}$$

as a consequence. That is, the additive inequality does not hold for the taxa i, x, y, z . By Theorem 10.4.5, no additive tree can be consistent with the non-additive matrix D . In other words, Algorithm 10.3 correctly terminates with failure. Similarly, if Algorithm 10.3 terminates in step 3 with failure (see Figure 10.31 for an example), then this is because there is an external edge (v, k) so that $\mu(v) < d_{ij} - d_{ik}$. As shown in the proof of the implication (3) \Rightarrow (4) in Theorem 10.4.5, this means that there must be a leaf l in the subtree rooted at v so that $d_{ik} + d_{kl} < d_{il}$. So D does not satisfy the triangle inequality. According to Lemma 10.2.3, D is not additive. Again, Algorithm 10.3 correctly terminates with failure.

It is not difficult to verify that the Algorithm 10.3 runs in $\Theta(n^2)$ time (the formal proof is left to the reader).

Algorithm 10.3 Construction of an additive tree (if it exists).

Input: $n \times n$ dissimilarity matrix D .

1. Compute the matrix D' by $d'_{kk} = 0$ and $d'_{kl} = d_{ij} - \frac{1}{2}(d_{ik} + d_{il} - d_{kl})$ for $k \neq l$, where d_{ij} is a maximum entry in D .
2. Apply Algorithm 10.1 (page 498) to D' . If D' is consistent with an ultrametric tree T' , then Algorithm 10.1 returns T' . In this case, proceed with step 3. Otherwise, if Algorithm 10.1 returns a triple (x, y, z) , then output the quadruple (i, x, y, z) as a witness that D does not satisfy the additive inequality and terminate with failure.
3. For each external edge (v, k) in the ultrametric tree T' , test whether $\mu(v) \geq d_{ij} - d_{ik}$. If so, proceed with step 4. Otherwise, there is an external edge (v, k) so that $\mu(v) < d_{ij} - d_{ik}$. Then there must be a leaf l in the subtree rooted at v so that $d_{ik} + d_{kl} < d_{il}$. Find such a taxon l , output the triple (i, k, l) as a witness that D does not satisfy the triangle inequality, and terminate with failure.
4. Transform T' into the tree T by deleting the marks at internal nodes, introducing the edge weights

$$\gamma(v, k) = \begin{cases} |\mu(v) - \mu(k)| & \text{if } (v, k) \text{ is an internal edge} \\ \mu(v) - (d_{ij} - d_{ik}) & \text{if } (v, k) \text{ is an external edge} \end{cases}$$

and replacing the root node with the leaf i .

Output: The tree T .

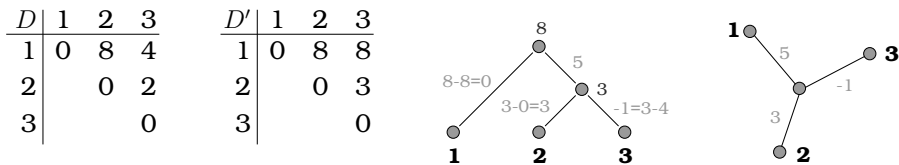


Figure 10.31: Algorithm 10.3 applied to the matrix D outputs the triple $(1, 3, 2)$ and terminates with failure in step 3. The fact that $d_{13} + d_{32} < d_{12}$ shows that D does not satisfy the triangle inequality for the taxa 1, 3, and 2.

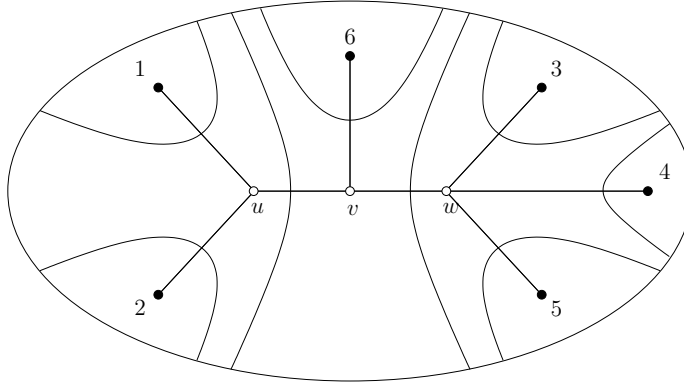


Figure 10.32: A phylogenetic tree and its splits.

10.4.5 Splits and quartets

In the previous section, we have seen that an additive matrix D is consistent with an unrooted additive tree T . Our next goal is to show that this tree is unique. It should be stressed that this is not true for rooted additive trees. So in the rest of this chapter, the phrase “ T is a tree” always means that T is an *unrooted* phylogenetic tree on the set $S = \{1, \dots, n\}$ of the taxa, unless stated otherwise. The reader should keep this in mind.

Definition 10.4.6 Let $e = (u, v)$ be an edge in the tree T . The removal of e partitions T into two connected components. Moreover, it partitions S into the set of taxa X_e that appear as leaves in the connected component of $T \setminus \{e\}$ containing u and the set of taxa $\bar{X}_e = S \setminus X_e$ that appear as leaves in the connected component of $T \setminus \{e\}$ containing v . Such a partition is called a *split* at edge e .

Figure 10.32 shows a phylogenetic tree on six taxa and its splits.

Definition 10.4.7 For any (not necessarily distinct) $i, j, k, l \in S$, the tree T induces the *quartet* $(ij : kl)$ if there is a split at an edge e so that $i, j \in X_e$ and $k, l \in \bar{X}_e$. We will say that i and j are *separated* from k and l by the edge e . The set of all quartets induced by T is denoted by $Q(T)$.

To put it differently, $(ij : kl)$ is a quartet induced by T if the path connecting i and j has no node in common with the path connecting k and l . By the above definition, $(ii : jk)$ is a quartet for any $i, j, k \in S$ provided that $j \neq i \neq k$ (but j may be equal to k).

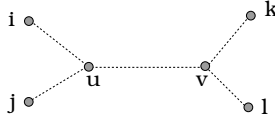


Figure 10.33: Let the internal edge e on the path from u to v separate i and j from k and l . If the additive tree T is consistent with D , then we have $d_{ik} + d_{jl} - (d_{ij} + d_{kl}) = 2d_T(u, v) \geq 2\gamma(e) > 0$.

Lemma 10.4.8 *In an additive tree T , let the leaves i and j be separated from the leaves k and l by an internal edge e . If T is consistent with the dissimilarity matrix D , then $d_{ij} + d_{kl} < d_{ik} + d_{jl} = d_{il} + d_{jk}$.*

Proof As explained in Figure 10.33, we derive $d_{ik} + d_{jl} - (d_{ij} + d_{kl}) > 0$. Hence $d_{ij} + d_{kl} < d_{ik} + d_{jl}$. The equality $d_{ik} + d_{jl} = d_{il} + d_{jk}$ is a direct consequence of the fact that the additive tree T is consistent with D . \square

Note that Lemma 10.4.8 is true even if $i = j$ or $k = l$.

Corollary 10.4.9 *Let $(ij : kl)$ be a quartet in an additive tree T . If T is consistent with the dissimilarity matrix D and $i, j, k, l \in S$ are pairwise distinct, then $d_{ij} + d_{kl} < d_{ik} + d_{jl} = d_{il} + d_{jk}$.*

Proof The fact that $i, j, k, l \in S$ are pairwise distinct in combination with $(ij : kl) \in Q(T)$ implies that i and j must be separated from k and l by an internal edge e . Thus, the corollary follows from Lemma 10.4.8. \square

Exercise 10.4.10 Show that Corollary 10.4.9 no longer holds if we drop the condition that $i, j, k, l \in S$ must be pairwise distinct.

The following notion will be useful in subsequent sections.

Definition 10.4.11 Let T be a phylogenetic tree. Two distinct leaves i and j that are adjacent to the same internal node in T are called *leaf neighbors*.

Lemma 10.4.12 *Suppose that the additive tree T is consistent with the dissimilarity matrix D . Let i and j be leaf neighbors in T and let $k, l \in S \setminus \{i, j\}$. Then, $d_{ij} + d_{kl} \leq d_{ik} + d_{jl} = d_{il} + d_{jk}$. Moreover, if T is binary and $|S| \geq 4$, then $d_{ij} + d_{kl} < d_{ik} + d_{jl} = d_{il} + d_{jk}$.*

Proof Let i and j be adjacent to the internal node v . Because the additive tree T is consistent with D , it follows that $d_{ik} + d_{jl} = d_T(i, k) + d_T(j, l) = d_T(i, v) + d_T(v, k) + d_T(j, v) + d_T(v, l) = d_T(i, v) + d_T(v, l) + d_T(j, v) + d_T(v, k) = d_T(i, l) + d_T(j, k) = d_{il} + d_{jk}$. Since D is additive according to Theorem 10.4.5,

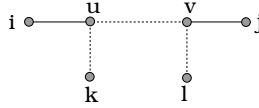


Figure 10.34: Taxa i and j are not leaf neighbors in the tree T .

the 4-point condition implies that the two largest values out of $d_{ij} + d_{kl}$, $d_{ik} + d_{jl}$, and $d_{il} + d_{jk}$ are equal. Hence $d_{ij} + d_{kl} \leq d_{ik} + d_{jl} = d_{il} + d_{jk}$.

If T is binary and $|S| \geq 4$, then i and j cannot have another leaf neighbor. Thus, i and j must be separated from k and l by an internal edge e . So the second statement is an immediate consequence of Lemma 10.4.8. \square

Lemma 10.4.13 *Let T be a phylogenetic tree on the set S of taxa, where $|S| \geq 4$. If the two taxa i and j are not leaf neighbors in T , then there are two leaves k and l so that $i, j, k, l \in S$ are pairwise distinct and $(ik : jl) \in Q(T)$. Furthermore, $d_{ik} + d_{jl} < d_{ij} + d_{kl} = d_{il} + d_{jk}$.*

Proof Let i and j be adjacent to the nodes u and v in T , where $u \neq v$. Let e be an edge on the path from u to v and consider the split at edge e . We have $i \in X_e$ and $j \in \overline{X}_e = S \setminus X_e$. Because every internal node has a degree ≥ 3 , there must be taxa $k \in X_e$ with $i \neq k$ and $l \in \overline{X}_e$ with $j \neq l$; see Figure 10.34. Clearly, $(ik : jl)$ is a quartet induced by T . The last statement is a direct consequence of Corollary 10.4.9. \square

10.4.6 Uniqueness of the additive tree

Lemma 10.4.14 *There is exactly one unrooted phylogenetic tree for two or three taxa. This tree is binary (i.e., every internal node has degree 3).*

Proof If there are just two taxa, then their phylogenetic tree consists of two leaves and an edge between them. If there are three taxa, then the topology of the tree must be a star. To be precise, all three leaves (taxa) are connected by an edge to an internal node v (the center of the star). \square

Lemma 10.4.15 *A distance matrix D for two or three taxa is consistent with exactly one unrooted additive tree. This tree is binary.*

Proof If $S = \{1, 2\}$, then the phylogenetic tree consists of two leaves 1 and 2 and an edge between them with weight d_{12} . If $S = \{1, 2, 3\}$, then the topology of the tree must be a star with center v . By additivity, the weights

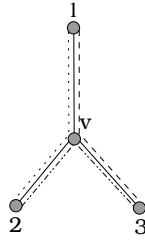


Figure 10.35: In this additive tree, edge weights are unique.

of the edges are uniquely determined (see Figure 10.35) by

$$\begin{aligned}\gamma(1, v) &= \frac{d_{12} + d_{13} - d_{23}}{2} \\ \gamma(2, v) &= \frac{d_{12} + d_{23} - d_{13}}{2} \\ \gamma(3, v) &= \frac{d_{13} + d_{23} - d_{12}}{2}\end{aligned}$$

These weights are greater than or equal to 0 because D satisfies the triangle inequality. \square

Lemma 10.4.16 Suppose that $|S| \geq 4$ and that the additive tree T is consistent with a dissimilarity matrix D on $S = \{1, \dots, n\}$. Let the taxa i and j be adjacent to the same node v in T (i.e., i and j are leaf neighbors), and let \bar{T} be the tree obtained from T as follows:

- Delete the leaves i and j and the edges (v, i) and (v, j) .
- If v is a leaf in the resulting tree, then label v with a new (artificial) taxon k . In this case, v must be adjacent to an internal node w because $|S| \geq 4$, and $\gamma(v, w) > 0$ because (v, w) is an internal edge in T . Otherwise, if v is not a leaf in the resulting tree,⁵ then add a new leaf labeled with k and a new edge (v, k) with weight $\gamma(v, k) = 0$.

The tree \bar{T} is consistent with the $(n - 1) \times (n - 1)$ dissimilarity matrix \bar{D} obtained from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with

$$\bar{d}_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$$

for every $m \in S \setminus \{i, j, k\} = S \setminus \{i, j\}$.

⁵This case can occur only if the degree of node v is strictly greater than 3. So this case cannot occur if T is a binary tree.

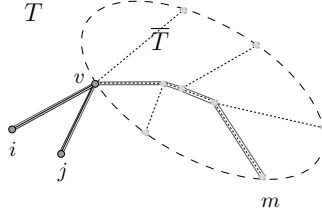


Figure 10.36: Let the additive tree T be consistent with the dissimilarity matrix D . Suppose that taxa i and j are adjacent to the same node v in T . By additivity, for any other taxon m , the equalities $d_{im} = d_T(i, v) + d_T(v, m)$, $d_{jm} = d_T(j, v) + d_T(v, m)$, and $d_{ij} = d_T(i, v) + d_T(v, j)$ hold. Thus, $2 \cdot d_T(v, m) = d_{im} - d_T(i, v) + d_{jm} - d_T(j, v) = d_{im} + d_{jm} - d_{ij}$.

Proof Consider the new taxon k and another leaf $m \neq k$ in \bar{T} . With the derivation in Figure 10.36, it follows $d_{\bar{T}}(k, m) = d_{\bar{T}}(v, m) = d_T(v, m) = \frac{1}{2}(d_{im} + d_{jm} - d_{ij}) = \bar{d}_{km}$. Now consider two leaves l and m in \bar{T} with $l \neq k \neq m$. Because these are also leaves in T and T is consistent with D , we infer $d_{\bar{T}}(l, m) = d_T(l, m) = d_{lm} = \bar{d}_{lm}$. Thus, \bar{T} is consistent with \bar{D} . \square

Lemma 10.4.17 *Let the additive tree T be consistent with a dissimilarity matrix D , and let the taxa i and j be adjacent to the same node v in T . Then $\gamma(v, i) = \frac{1}{2}(d_{im} + d_{ij} - d_{jm})$ and $\gamma(v, j) = d_{ij} - \gamma(v, i) = \frac{1}{2}(d_{jm} + d_{ij} - d_{im})$.*

Proof According to the derivation in Figure 10.36, we have

$$\gamma(v, i) = d_T(v, i) = d_{im} - d_T(v, m) = d_{im} - \frac{1}{2}(d_{im} - d_{ij} + d_{jm}) = \frac{1}{2}(d_{im} + d_{ij} - d_{jm})$$

Clearly, $\gamma(v, j) = d_{ij} - \gamma(v, i)$ because T is additive. Therefore,

$$\gamma(v, j) = d_{ij} - \gamma(v, i) = d_{ij} - \frac{1}{2}(d_{im} + d_{ij} - d_{jm}) = \frac{1}{2}(d_{jm} + d_{ij} - d_{im})$$

\square

Theorem 10.4.18 *If an additive tree T is consistent with a dissimilarity matrix D , then there is no other additive tree T' (different from T) that is consistent with D .*

Proof Let T and T' be additive trees that are consistent with D . We show by induction on the number n of taxa that T and T' must coincide. Lemma 10.4.15 covers the cases $n = 2$ and $n = 3$. Suppose $n \geq 4$ and let the taxa

i and j be leaf neighbors in T . We claim that i and j must also be leaf neighbors in T' . For an indirect proof of this claim, suppose that i and j are not leaf neighbors in T' . By Lemma 10.4.13, there are two other taxa p and q so that i, j, p, q are pairwise distinct, $(ip : jq)$ is a quartet induced by T' , and $d_{ip} + d_{jq} < d_{ij} + d_{pq} = d_{iq} + d_{jp}$. On the other hand, we have $d_{ij} + d_{pq} \leq d_{ip} + d_{jq} = d_{iq} + d_{jp}$ by Lemma 10.4.12 because i and j are leaf neighbors in T . The contradiction $d_{ij} + d_{pq} \leq d_{ip} + d_{jq} < d_{ij} + d_{pq}$ shows that i and j must be leaf neighbors in T' , too. Let \bar{T} and \bar{T}' be the trees obtained from T and T' , respectively, by deleting i and j and adding the new taxon k (as specified in Lemma 10.4.16). By Lemma 10.4.16, both \bar{T} and \bar{T}' are consistent with the $(n-1) \times (n-1)$ dissimilarity matrix \bar{D} obtained from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k (as specified in Lemma 10.4.16). According to the inductive hypothesis, \bar{T} and \bar{T}' coincide. In $\bar{T} = \bar{T}'$, k is a leaf that is adjacent to an internal node, say w . If the edge (w, k) has weight $\gamma(w, k) > 0$, then both T and T' are obtained from $\bar{T} = \bar{T}'$ by adding the edges (k, i) and (k, j) . Because the edge weights of (v, i) and (v, j) are uniquely determined by Lemma 10.4.17, it follows that T and T' coincide. The case $\gamma(w, k) = 0$ similarly follows. \square

Once again, we stress that we are dealing with *unrooted* trees. We know from Theorem 10.4.5 that every additive matrix is consistent with an additive tree and Theorem 10.4.18 implies that this tree is unique. In other words, every additive matrix is consistent with exactly one additive tree. By contrast, an additive matrix is not necessarily consistent with an additive *binary* tree. Consider for example the additive matrix

D	1	2	3	4
1	0	2	2	2
2		0	2	2
3			0	2
4				0

It is consistent with the (additive) star phylogeny of the four taxa in which every edge has weight 1, but it is not consistent with an additive binary tree. Theorem 10.4.18 states that if there *exists* an additive binary tree T that is consistent with D , then T is unique.

10.5 Neighbor-joining algorithms

The discussion of the preceding section directly leads to the so-called neighbor-joining algorithms, algorithms that reconstruct an unrooted *binary* phylogenetic tree by successively joining leaf neighbors. A generic neighbor-joining algorithm is formulated in Algorithm 10.4. We will get

to know several concrete neighbor-joining algorithms, which differ by the specific neighbor selection criterion they use.

In essence, the correctness of the generic neighbor-joining algorithm is an immediate consequence of the discussion of the preceding section. For the convenience of the reader, however, we work out the details of the correctness proof.

Theorem 10.5.1 *If a dissimilarity matrix D is consistent with an additive binary tree T and the neighbor selection criterion truly identifies leaf neighbors, then the generic neighbor-joining algorithm (Algorithm 10.4) applied to D constructs T .*

Proof We will prove the theorem by induction on the number n of the taxa. If $n = 3$, then the generic neighbor-joining algorithm returns the correct tree; see Lemma 10.4.15. In the inductive step, we show the theorem for $n > 3$, under the hypothesis that it holds for $n - 1$. Let T be an additive binary tree that is consistent with D . Suppose that the neighbor selection criterion selects the taxa i and j . By assumption, i and j are leaf neighbors in T , i.e., they are adjacent to the same node k in T . Let w be the third node adjacent to k (T is a binary tree, so k has degree 3). Because $n > 3$, w must be an internal node of T . Moreover, since T is additive, the edge (k, w) has a strictly positive weight. Let \bar{T} be the additive binary tree obtained from T by deleting the edges from k to i and from k to j . According to Lemma 10.4.16, \bar{T} is consistent with the $(n - 1) \times (n - 1)$ dissimilarity matrix \bar{D} obtained from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new (artificial) taxon k with $\bar{d}_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in S \setminus \{i, j, k\}$. By the inductive hypothesis, the generic neighbor-joining algorithm applied to \bar{D} constructs \bar{T} . It returns the tree obtained from \bar{T} by adding node k and edges (k, i) and (k, j) with weights $\gamma(k, i) = \frac{1}{2}(d_{im} + d_{ij} - d_{jm})$ and $\gamma(k, j) = \frac{1}{2}(d_{jm} + d_{ij} - d_{im})$, where $m \in S \setminus \{i, j, k\}$. It is readily verified that the returned tree is consistent with D (this is because \bar{T} is consistent with \bar{D}). By Theorem 10.4.18, the returned tree coincides with T . \square

The crux of developing a concrete neighbor-joining algorithm consists of finding a criterion that provably selects *leaf neighbors* in the (yet unknown) tree T for D . One might be tempted to select taxa i and j so that d_{ij} is a minimum entry in the dissimilarity matrix D . This naive approach, however, fails. To see this, consider the additive binary tree T and the matrix D from Figure 10.37. Note that D and T are consistent. The minimum entry in D is d_{12} , but the taxa 1 and 2 are not leaf neighbors.

In the following, we will develop several neighbor selection criteria.

Algorithm 10.4 Generic neighbor-joining algorithm.

Input: $n \times n$ dissimilarity matrix D , where $n \geq 3$.

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon i is a leaf in the tree T .

while $|S| > 3$ **do**

1. Using a specific neighbor selection criterion, select two taxa i and j that are leaf neighbors in the (yet unknown) tree T .
2. Add a new node k to the tree T .
3. Choose an $m \in S \setminus \{i, j\}$ and add edges (k, i) and (k, j) with weights $\gamma(k, i) = \frac{1}{2}(d_{im} - d_{jm} + d_{ij})$ and $\gamma(k, j) = d_{ij} - \gamma(k, i) = \frac{1}{2}(d_{jm} - d_{im} + d_{ij})$ to the tree T .
4. Update the dissimilarity matrix by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in S \setminus \{i, j, k\}$.
5. Delete i and j from S and add the new (artificial) taxon k to S .

Termination:

Let i, j, m be the remaining three taxa. Add a new internal node v to the tree T , and add edges (v, i) , (v, j) , and (v, m) to the tree T with weights

$$\begin{aligned}\gamma(v, i) &= \frac{d_{ij} + d_{im} - d_{jm}}{2} \\ \gamma(v, j) &= \frac{d_{ij} + d_{jm} - d_{im}}{2} \\ \gamma(v, m) &= \frac{d_{im} + d_{jm} - d_{ij}}{2}\end{aligned}$$

Output: The tree T .

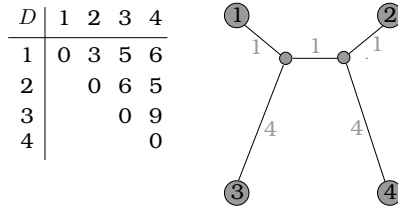


Figure 10.37: The naive approach to select leaf neighbors fails: d_{12} is a minimum entry in D , but the leaves 1 and 2 are not neighbors.

10.5.1 Farris' neighbor-joining algorithm

Our first concrete neighbor-joining algorithm is based on a generalization of the Farris transform introduced in Section 10.4.2. Farris [95, 96] applied his transformation to the original dissimilarity matrix D and then used UPGMA on the transformed matrix D' ; see also [7, 59, 241].

Definition 10.5.2 Farris' neighbor selection criterion: Given the dissimilarity matrix D for the set $S = \{1, \dots, n\}$ of taxa, let c be a constant that is greater than or equal to the maximum entry in D and let $r \in S$ be some distinguished taxon. Define the matrix D' by $d'_{ij} = c - \frac{1}{2}(d_{ri} + d_{rj} - d_{ij})$ for all $i, j \in S$. Select taxa i and j so that d'_{ij} is a minimum entry in D' .

The constant c serves the purpose to turn D' into a dissimilarity matrix.⁶ Note that minimizing $c - \frac{1}{2}(d_{ri} + d_{rj} - d_{ij})$ for all $i, j \in S$ is equivalent to maximizing $\frac{1}{2}(d_{ri} + d_{rj} - d_{ij})$ for all $i, j \in S$.

In Section 10.4.2, we have already become acquainted with a special form of the Farris transform, in which c is the maximum entry in D and r is a taxon having the maximum distance c to another taxon.

Theorem 10.5.3 If a dissimilarity matrix D is consistent with an additive binary tree T , then the Farris neighbor-joining algorithm constructs T .

Proof Without loss of generality, $|S| = n > 3$. According to Theorem 10.5.1, it is sufficient to show that i and j are leaf neighbors in T whenever d'_{ij} is a minimum entry in the D' matrix corresponding to D .

We first claim that Farris' neighbor selection criterion will not select the distinguished taxon r . For an indirect proof of the claim, suppose that

⁶In fact, any constant that is greater than or equal to $\max\{d_{ri} \mid 1 \leq i \leq n\}$ does the job, but the above definition of c is independent of the choice of r .

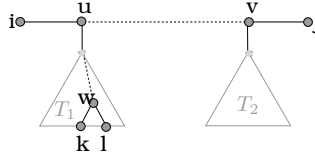


Figure 10.38: If neither i nor j has a leaf neighbor $\neq r$, then one of the subtrees T_1 or T_2 does not contain r but rather the leaf neighbors k and l .

Farris' neighbor selection criterion selects r , i.e., d'_{rm} is a minimum entry in D' for some $m \in S$. We have

$$d'_{rm} = c - \frac{1}{2}(d_{rr} + d_{rm} - d_{rm}) = c$$

Because T is binary and $|S| \geq 4$, there are leaf neighbors k and l in T with $k \neq r \neq l$. It follows from Lemma 10.4.12 that $d_{kl} = d_{kl} + d_{rr} < d_{rk} + d_{rl}$. Hence

$$d'_{kl} = c - \frac{1}{2}(d_{rk} + d_{rl} - d_{kl}) < c$$

This contradiction proves the claim.

For an indirect proof of the theorem, suppose that d'_{ij} is a minimum entry in the D' matrix but i and j are not leaf neighbors in T . According to the discussion above, neither i nor j coincides with r . We proceed by case analysis.

Case 1: i or j has a leaf neighbor $k \neq r$. Without loss of generality, let i be this taxon. On the one hand, $d'_{ik} - d'_{ij} \geq 0$ because d'_{ij} is minimal. On the other hand, this contradicts

$$\begin{aligned} d'_{ik} - d'_{ij} &= c - \frac{1}{2}(d_{ri} + d_{rk} - d_{ik}) - c + \frac{1}{2}(d_{ri} + d_{rj} - d_{ij}) \\ &= \frac{1}{2}(d_{ik} + d_{rj} - (d_{rk} + d_{ij})) \\ &< 0 \end{aligned}$$

where the last inequality follows from Lemma 10.4.12: $d_{ik} + d_{rj} < d_{rk} + d_{ij}$ because i and k are leaf neighbors in T .

Case 2: Neither i nor j has a leaf neighbor $\neq r$. Let i be adjacent to node u and j be adjacent to node v in T , where $u \neq v$. Let T_1 and T_2 be the subtrees of T as depicted in Figure 10.38. Obviously, either T_1 or T_2 (or both) do not contain the distinguished taxon r . Without loss of generality, let T_1 be this subtree. T_1 must contain leaf neighbors k

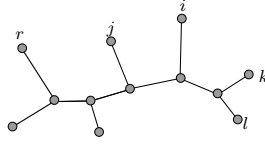


Figure 10.39: The situation in case 2 of the proof of Theorem 10.5.3.

and l ; let these leaves be adjacent to the node w as depicted in Figure 10.38. Note that $u \neq w$. Furthermore, observe that the taxa r, i, j, k, l are pairwise distinct. The situation is illustrated in Figure 10.39. Again, the inequality $d'_{kl} - d'_{ij} \geq 0$ holds true because d'_{ij} is a minimum entry in the D' matrix. Because $(ri : kl)$ and $(rj : il)$ are quartets induced by T , and D is consistent with the additive binary tree T , we conclude with Corollary 10.4.9 that (a) $d_{ri} + d_{kl} < d_{rk} + d_{il}$ and (b) $d_{rj} + d_{il} < d_{rl} + d_{ij}$. Using these inequalities, we derive a contradiction as follows

$$\begin{aligned}
 d'_{kl} - d'_{ij} &= c - \frac{1}{2}(d_{rk} + d_{rl} - d_{kl}) - c + \frac{1}{2}(d_{ri} + d_{rj} - d_{ij}) \\
 &= \frac{1}{2}(d_{ri} + d_{kl} + d_{rj} - d_{rk} - d_{rl} - d_{ij}) \\
 &\stackrel{(a)}{<} \frac{1}{2}(d_{rk} + d_{il} + d_{rj} - d_{rk} - d_{rl} - d_{ij}) \\
 &= \frac{1}{2}(d_{rj} + d_{il} - (d_{rl} + d_{ij})) \\
 &\stackrel{(b)}{<} 0
 \end{aligned}$$

□

It is readily verified that the maximum entry in the current D matrix is greater than or equal to the maximum entry in the D matrix of the next iteration. Therefore, one can choose the same constant c in each iteration of the Farris neighbor-joining algorithm (it is also possible to get rid of the constant by maximizing $\frac{1}{2}(d_{ri} + d_{rj} - d_{ij})$ for all $i, j \in S$).

The choice of the distinguished taxon r is arbitrary. If one chooses different distinguished taxa in each iteration, then the algorithm has a worst-case time complexity of $O(n^3)$ because the matrix D' must be recomputed in each iteration. Because Farris' neighbor selection criterion will never select the distinguished taxon r , it is also possible to choose the same taxon r in every iteration. Pseudo-code for this variant of Farris' neighbor-joining algorithm is given in Algorithm 10.5.

Let us analyze the worst-case time complexity of Algorithm 10.5. In each of the $(n-3)$ iterations of the while-loop, the update of the matrices D and D' takes $O(n)$ time. As in the fast implementation of the UPGMA-algorithm

Algorithm 10.5 Farris' neighbor-joining algorithm.

Input: $n \times n$ dissimilarity matrix D , where $n \geq 3$.

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon i is a leaf in the tree T .
3. Let c be the maximum entry in D .
4. Select a distinguished taxon $r \in S$.
5. Compute the matrix D' by $d'_{ij} = c - \frac{1}{2}(d_{ri} + d_{rj} - d_{ij})$ for all $i, j \in S$.

while $|S| > 3$ **do**

1. Select $i, j \in S$ so that d'_{ij} is a minimum entry in D' .
2. Add a new node k to the tree T .
3. Add edges (k, i) and (k, j) with weights $\gamma(k, i) = \frac{1}{2}(d_{ri} + d_{ij} - d_{rj})$ and $\gamma(k, j) = d_{ij} - \gamma(k, i) = \frac{1}{2}(d_{rj} + d_{ij} - d_{ri})$ to the tree T .
4. a) Update the matrix D by deleting the rows and columns corresponding to i and j and adding new rows and columns for the new taxon k with $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in S \setminus \{i, j, k\}$.
 b) Update the matrix D' by deleting the rows and columns corresponding to i and j and adding new rows and columns for the new taxon k with $d'_{km} = c - \frac{1}{2}(d_{rk} + d_{rm} - d_{km})$ for all $m \in S \setminus \{i, j, k\}$.
5. Delete i and j from S and add the new (artificial) taxon k to S .

Termination:

Let i, j, r be the remaining three taxa. Add a new internal node v to the tree T , and add edges (v, i) , (v, j) , and (v, r) to the tree T with weights

$$\begin{aligned}\gamma(v, r) &= \frac{d_{ir} + d_{jr} - d_{ij}}{2} \\ \gamma(v, i) &= \frac{d_{ij} + d_{ir} - d_{jr}}{2} \\ \gamma(v, j) &= \frac{d_{ij} + d_{jr} - d_{ir}}{2}\end{aligned}$$

Output: The tree T .

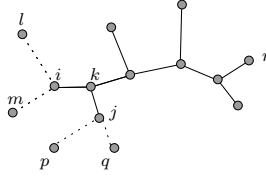


Figure 10.40: Nodes i and j are connected to the new node k . If $t(i) = \{l, m\}$ and $t(j) = \{p, q\}$, where $l, m, p, q \in \{1, \dots, n\}$, then $t(k)$ is obtained by choosing one element from $t(i)$ and one element from $t(j)$, say $t(k) = \{l, p\}$. Moreover, the equalities $d_T(k, r) = \frac{d_{rl} + d_{rp} - d_{lp}}{2}$, $d_T(k, i) = \frac{d_{ri} + d_{rm} - d_{im}}{2} - d_T(k, r)$, and $d_T(k, j) = \frac{d_{rp} + d_{rq} - d_{pq}}{2} - d_T(k, r)$ hold.

(Section 10.3.4), we can use a quadtree to determine a minimum entry in D' in $O(\log n)$ (Lemma 10.3.14) and the update of the quadtree takes $O(n)$ time in each iteration (Lemma 10.3.15). Thus, the Farris neighbor-joining algorithm can be implemented so that its worst-case time complexity is $O(n^2)$. Note that this is optimal because the size of the input matrix D is proportional to n^2 .

Taking this approach one step further [116], we can even refrain from updating the input matrix D . Then, however, a difficulty arises from the fact that the edge weights in Algorithm 10.5 are defined by means of the current D matrix. So we must be able to assign the correct edge weights based on the original input matrix. This is possible if we store an original taxon (an element of $\{1, \dots, n\}$) from each of the two subtrees at a newly created internal node. Pseudo-code for the resulting algorithm can be found in Algorithm 10.6. To distinguish this variant of the Farris neighbor-joining algorithm from Algorithm 10.5, we call it a “second version of the Farris neighbor-joining algorithm.”

The differences between Algorithms 10.5 and 10.6 lie in steps 2-4 of the while-loop. When a new node k is added to the tree in step 2, Algorithm 10.6 stores two original taxa in the set $t(k)$: one from the subtree containing i and one from the subtree containing j ; see Figure 10.40 for an illustration of the situation. Using Figure 10.40, it is a simple exercise to prove that the edge weights assigned in step 3 of the while-loop of Algorithm 10.6 are correct. It remains to be shown that in each iteration of the while-loop Algorithms 10.5 and 10.6 compute the same matrix D' . The correctness of Algorithm 10.6 is then a consequence of Theorem 10.5.3.

Theorem 10.5.4 *If the input matrix D_{in} is consistent with an additive binary tree T , then Algorithms 10.5 and 10.6 compute the same dissimilarity matrix D' in each iteration of the while-loop.*

Algorithm 10.6 Second version of Farris' neighbor-joining algorithm.

Input: $n \times n$ dissimilarity matrix D , where $n \geq 3$.

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon i is a leaf in the tree T .
3. Let c be the maximum entry in D .
4. Select a distinguished taxon $r \in S$.
5. Compute the matrix D' by $d'_{ij} = c - \frac{1}{2}(d_{ri} + d_{rj} - d_{ij})$ for all $i, j \in S$.

while $|S| > 3$ **do**

1. Select $i, j \in S$ so that d'_{ij} is a minimum entry in D' .
2. Add a new node k to the tree T and store $t(k) = \{l, p\}$, where $l \in t(i)$ if $i \notin \{1, \dots, n\}$ and $l = i$ otherwise, and $p \in t(j)$ if $j \notin \{1, \dots, n\}$ and $p = j$ otherwise.
3. Compute $d_T(k, r) = \frac{d_{ri} + d_{rp} - d_{ip}}{2}$, where $t(k) = \{l, p\}$.
Add an edge from k to i with weight

$$\gamma(k, i) = \begin{cases} d_{ri} - d_T(k, r) & \text{if } i \in \{1, \dots, n\} \\ \frac{d_{ri} + d_{rm} - d_{im}}{2} - d_T(k, r) & \text{otherwise, where } t(i) = \{l, m\} \end{cases}$$

and an edge from k to j with weight

$$\gamma(k, j) = \begin{cases} d_{rj} - d_T(k, r) & \text{if } j \in \{1, \dots, n\} \\ \frac{d_{rp} + d_{rq} - d_{pq}}{2} - d_T(k, r) & \text{otherwise, where } t(j) = \{p, q\} \end{cases}$$

4. Update the matrix D' by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $d'_{km} = \frac{1}{2}(d'_{im} + d'_{jm})$ for all $m \in S \setminus \{i, j, k\}$.
5. Delete i and j from S and add the new (artificial) taxon k to S .

Termination:

Let i, j, r be the remaining three taxa. Add a new internal node k to the tree T , and add edges (k, i) , (k, j) , and (k, r) to the tree T with weights $\gamma(k, r) = d_T(k, r)$, $\gamma(k, i)$, and $\gamma(k, j)$ as defined in step (3) of the while-loop.

Output: The tree T .

Proof We show the theorem by induction on the number q of iterations of the while-loop. The base case $q = 0$ is obvious. Let D be the dissimilarity matrix after the q -th iteration of the while-loop in Algorithm 10.5 (so for $q > 1$, D is *not* the original input matrix D_{in}), and let D' be its Farris transformed matrix. By the inductive hypothesis, Algorithm 10.6 computes the same matrix D' . Suppose that in the $(q + 1)$ -th iteration, taxa i and j are selected. Algorithm 10.5 obtains the new dissimilarity matrix \bar{D} from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $\bar{d}_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all m with $i \neq m \neq j$. Furthermore, it obtains the new Farris transformed matrix \bar{D}' from D' by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $\bar{d}'_{km} = c - \frac{1}{2}(\bar{d}_{rk} + \bar{d}_{rm} - \bar{d}_{km})$. Thus, for all m with $i \neq m \neq j$, we have

$$\begin{aligned}
 \bar{d}'_{km} &= c - \frac{1}{2}(\bar{d}_{rk} + \bar{d}_{rm} - \bar{d}_{km}) \\
 &= c - \frac{1}{2} \left(\frac{1}{2}(d_{ri} + d_{rj} - d_{ij}) + d_{rm} - \frac{1}{2}(d_{im} + d_{jm} - d_{ij}) \right) \\
 &= \frac{1}{2} \left(2c - \frac{1}{2}(d_{ri} + d_{rj} + 2d_{rm} - d_{im} - d_{jm}) \right) \\
 &= \frac{1}{2} \left(c - \frac{1}{2}(d_{ri} + d_{rm} - d_{im}) + c - \frac{1}{2}(d_{rj} + d_{rm} - d_{jm}) \right) \\
 &= \frac{1}{2}(d'_{im} + d'_{jm})
 \end{aligned}$$

This shows the theorem. □

10.5.2 Saitou and Nei's neighbor-joining algorithm

The most popular neighbor-joining algorithm is due to Saitou and Nei [120, 276, 303]. In the literature, it is most often referred to as “the neighbor-joining algorithm.” Its neighbor selection criterion uses the matrix $N = (n_{ij})_{i,j \in S}$ defined by

$$n_{ij} = d_{ij} - (r_i + r_j),$$

where

$$r_i = \frac{1}{|S| - 2} \sum_{m \in S} d_{im}$$

and selects $i, j \in S$ so that n_{ij} is a minimum entry in N . Saitou and Nei's neighbor-joining algorithm is presented in Algorithm 10.7.

According to Theorem 10.5.1, if a dissimilarity matrix D is consistent with an additive binary tree T , then Algorithm 10.7 applied to D constructs T provided that (a) its neighbor selection criterion truly identifies

Algorithm 10.7 Saitou and Nei's neighbor-joining algorithm.

Input: $n \times n$ dissimilarity matrix D , where $n \geq 3$.

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon i is a leaf in the tree T .

while $|S| > 3$ **do**

1. a) Compute the matrix $N = (n_{ij})_{i,j \in S}$, where $n_{ij} = d_{ij} - (r_i + r_j)$ and $r_i = \frac{1}{|S|-2} \sum_{m \in S} d_{im}$.
 b) Select $i, j \in S$ so that n_{ij} is a minimum entry in N .
2. Add a new node k to the tree T .
3. Add edges (k, i) and (k, j) with weights $\gamma(k, i) = \frac{1}{2}(d_{ij} + r_i - r_j)$ and $\gamma(k, j) = d_{ij} - \gamma(k, i) = \frac{1}{2}(d_{ij} + r_j - r_i)$ to the tree T .
4. Update the dissimilarity matrix by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in S \setminus \{i, j, k\}$.
5. Delete i and j from S and add the new (artificial) taxon k to S .

Termination:

Let i, j, m be the remaining three taxa. Add a new internal node v to the tree T , and add edges (v, i) , (v, j) , and (v, m) to the tree T with weights

$$\begin{aligned}\gamma(v, i) &= \frac{d_{ij} + d_{im} - d_{jm}}{2} \\ \gamma(v, j) &= \frac{d_{ij} + d_{jm} - d_{im}}{2} \\ \gamma(v, m) &= \frac{d_{im} + d_{jm} - d_{ij}}{2}\end{aligned}$$

Output: The tree T .

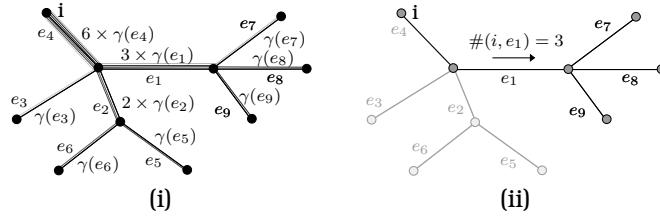


Figure 10.41: r_i is computed by summing up the distances from leaf i to all other leaves (and then dividing by $n - 2$). Thereby, the contribution of each edge e is $\gamma(e) \#(i, e)$.

leaf neighbors (step 1 of the while-loop), and (b) the edge weights of the newly created edges are correct (step 3 of the while-loop). In order to enhance readability, we prove these statements only for the case $|S| = n$ (in general, n must be replaced with $|S|$ in the formulae below). Statement (a) will be proven in Theorem 10.5.6. Statement (b) can easily be verified (cf. Figure 10.36 on page 530):

$$\begin{aligned}
 \gamma(k, i) &= \frac{1}{2}(d_{ij} + r_i - r_j) \\
 &= \frac{1}{2}d_{ij} + \frac{1}{2(n-2)} \left(\sum_{m \neq i} d_{im} - \sum_{m \neq j} d_{jm} \right) \\
 &= \frac{1}{2(n-2)} \sum_{m \notin \{i, j\}} d_{ij} + \frac{1}{2(n-2)} \sum_{m \notin \{i, j\}} (d_{im} - d_{jm}) \\
 &= \frac{1}{2(n-2)} \sum_{m \notin \{i, j\}} (d_{ij} + d_{im} - d_{jm}) \\
 &= \frac{1}{(n-2)} \sum_{m \notin \{i, j\}} d_T(i, k) \\
 &= d_T(i, k)
 \end{aligned}$$

We need a few prerequisites to be able to prove Theorem 10.5.6. Our exposition follows [280].

For a leaf i and an edge e in the tree T , let $\#(i, e)$ denote the number of leaves in T that are reachable from i by a path that contains e . An r_i -value can be written with this new notation as:

$$r_i = \frac{1}{n-2} \sum_{m \neq i} d_{im} = \frac{1}{n-2} \sum_{e \in E} \gamma(e) \#(i, e)$$

The formula is illustrated in Figure 10.41; a formal proof is left to the reader. Yet another notation will prove useful: For two nodes u and v in

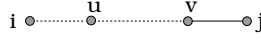


Figure 10.42: The situation of Lemma 10.5.5.

the tree T , let $E(u, v)$ denote the set of all edges on the (unique) path from u to v . Now we are in a position to prove the following lemma.

Lemma 10.5.5 *Let the additive binary tree T be consistent with the dissimilarity matrix D . If there is a path from leaf i via the internal node u to the internal node $v \neq u$ and an external edge (v, j) as depicted in Figure 10.42, then we have*

$$r_i - r_j \leq d_T(i, u) - d_T(v, j) + \frac{1}{n-2} \sum_{e \in E(u, v)} \gamma(e) (\#(i, e) - \#(j, e))$$

Proof We use the above formula and repeatedly apply the definition of $\#(i, e)$:

$$\begin{aligned} & r_i - r_j \\ &= \frac{1}{n-2} \sum_{e \in E} \gamma(e) \underbrace{(\#(i, e) - \#(j, e))}_{=0 \text{ if } e \notin E(i, j)} \\ &= \frac{1}{n-2} \sum_{e \in E(i, j)} \gamma(e) (\#(i, e) - \#(j, e)) \\ &= \frac{1}{n-2} \left(\sum_{e \in E(i, u)} \gamma(e) \underbrace{(\#(i, e) - \#(j, e))}_{\leq n-1} \right) + \frac{1}{n-2} \left(\sum_{e \in E(u, v)} \gamma(e) (\#(i, e) - \#(j, e)) \right) \\ &\quad + \frac{1}{n-2} \left(\sum_{e=(v, j)} \gamma(e) \underbrace{(\#(i, e) - \#(j, e))}_{=1} \right) \\ &\leq \frac{1}{n-2} \left(\sum_{e \in E(i, u)} \gamma(e) (n-2) \right) + \frac{1}{n-2} \left(\sum_{e \in E(u, v)} \gamma(e) (\#(i, e) - \#(j, e)) \right) - \gamma(v, j) \\ &= d_T(i, u) + \frac{1}{n-2} \left(\sum_{e \in E(u, v)} \gamma(e) (\#(i, e) - \#(j, e)) \right) - d_T(v, j) \end{aligned}$$

□

Theorem 10.5.6 *If a dissimilarity matrix D is consistent with an additive binary tree T and n_{ij} is a minimum entry in the corresponding N matrix, then i and j are leaf neighbors in T .*

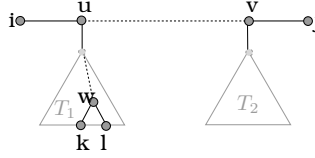


Figure 10.43: If neither i nor j has a leaf neighbor, then both subtrees T_1 and T_2 contain leaf neighbors.

Proof For an indirect proof, suppose that n_{ij} is a minimum entry in the matrix N , but i and j are not leaf neighbors in T . We will derive a contradiction by the following case distinction.

Case 1: i or j (or both) have a leaf neighbor. Without loss of generality, assume that i has a leaf neighbor, say k . Because n_{ij} is minimal, $n_{ik} - n_{ij} \geq 0$ should be true. Let us try to verify this:

$$\begin{aligned}
 & n_{ik} - n_{ij} \\
 = & d_{ik} - (r_i + r_k) - (d_{ij} - (r_i + r_j)) \\
 = & d_{ik} - d_{ij} - r_k + r_j \\
 = & d_{ik} - d_{ij} - \frac{1}{n-2} \sum_{m \neq k} d_{km} + \frac{1}{n-2} \sum_{m \neq j} d_{jm} \\
 = & d_{ik} - d_{ij} + \frac{1}{n-2} \left(\sum_{m \notin \{i,j,k\}} (d_{jm} - d_{km}) \right) + \frac{1}{n-2} (-d_{ki} - d_{kj} + d_{ji} + d_{jk}) \\
 = & \frac{1}{n-2} \left(\sum_{m \notin \{i,j,k\}} (d_{jm} - d_{km}) \right) + \frac{n-3}{n-2} d_{ik} - \frac{n-3}{n-2} d_{ij} + \frac{1}{n-2} \underbrace{(-d_{kj} + d_{jk})}_{=0} \\
 = & \frac{1}{n-2} \sum_{m \notin \{i,j,k\}} (d_{ik} + d_{jm} - (d_{ij} + d_{km}))
 \end{aligned}$$

Because i and k are leaf neighbors in the binary tree T , it follows from Lemma 10.4.12 that $d_{ik} + d_{jm} < d_{ij} + d_{km}$ for any other taxon $m \notin \{i, j, k\}$. Consequently, $n_{ik} - n_{ij} < 0$. This, however, contradicts the assumption that n_{ij} is minimal.

Case 2: Neither i nor j has a leaf neighbor. Let i be adjacent to node u and j be adjacent to node v in T , where $u \neq v$. Let T_1 and T_2 be the subtrees of T as depicted in Figure 10.43. Without loss of generality, we may assume that $|T_1| \leq |T_2|$, where $|T_1|$ and $|T_2|$ denote the number of leaves in T_1 and T_2 , respectively. T_1 must contain leaf neighbors k and l ; let these leaves be adjacent to the node $w \neq u$; see Figure 10.43. Next, we

will show $n_{kl} < n_{ij}$. By the definition of the N matrix, we have

$$\begin{aligned}
 & n_{kl} - n_{ij} \\
 &= (d_{kl} - (r_k + r_l)) - (d_{ij} - (r_i + r_j)) \\
 &= d_{kl} - d_{ij} - r_k - r_l + r_i + r_j \\
 &= d_{kl} - d_{ij} + (r_i - r_k) + (r_j - r_l)
 \end{aligned}$$

Since there is a path from leaf i via the internal node u to the internal node $w \neq u$ and an external edge (w, k) , an application of Lemma 10.5.5 yields

$$r_i - r_k \leq d_T(i, u) - d_T(w, k) + \frac{1}{n-2} \sum_{e \in E(u, w)} \gamma(e) (\#(i, e) - \#(k, e))$$

Any path that starts from leaf i and uses an edge $e \in E(u, w)$ cannot reach a leaf outside T_1 . Hence $\#(i, e) \leq |T_1|$ for all $e \in E(u, w)$. Furthermore, i , j , and all leaves in T_2 are reachable from k by a path that uses every edge e in $E(u, w)$. Thus, $\#(k, e) \geq |T_2| + 2$ for all $e \in E(u, w)$. By these two facts in conjunction with $|T_1| \leq |T_2|$, we obtain

$$\begin{aligned}
 r_i - r_k &\leq d_T(i, u) - d_T(w, k) + \frac{1}{n-2} d_T(u, w) (|T_1| - |T_2| - 2) \\
 &\leq d_T(i, u) - d_T(w, k) - \frac{2}{n-2} d_T(u, w)
 \end{aligned}$$

There is also a path from leaf j via the internal node u to the internal node $w \neq u$ and an external edge (w, l) . Therefore, we can similarly derive an upper bound for $r_j - r_l$:

$$r_j - r_l \leq d_T(j, u) - d_T(w, l) - \frac{2}{n-2} d_T(u, w)$$

Putting all pieces together, we obtain

$$\begin{aligned}
 & n_{kl} - n_{ij} \\
 &= d_{kl} - d_{ij} + (r_i - r_k) + (r_j - r_l) \\
 &\leq d_{kl} - d_{ij} + \underbrace{(d_T(i, u) + d_T(u, j))}_{=d_{ij}} - \underbrace{(d_T(k, w) + d_T(w, l))}_{=d_{kl}} - \frac{4}{n-2} d_T(u, w) \\
 &= -\frac{4}{n-2} d_T(u, w)
 \end{aligned}$$

Because T is additive, the path from u to w has a strictly positive weight, i.e., $d_T(u, w) > 0$. Hence $n_{kl} - n_{ij} < 0$. This, however, contradicts the assumption that n_{ij} is minimal. \square

It is not difficult to see that Algorithm 10.7 has a worst-case time complexity of $O(n^3)$.

10.5.3 Fast neighbor-joining

The fast neighbor-joining algorithm devised by Elias and Lagergren [87] has an $O(n^2)$ running time. It improves upon Saitou and Nei's neighbor-joining algorithm by using the following two ideas: First, the minimum is taken over the so-called visible set, which has cardinality $O(n)$. Second, the update of the n_{ij} values can be performed in constant time because the row sums $R_i = \sum_{m \in S} d_{im}$ can be updated in constant time.

Definition 10.5.7 Given a dissimilarity matrix D , let the matrix N be defined as in Saitou and Nei's neighbor-joining algorithm. For a fixed but arbitrary $i \in S$, a pair of taxa (i, l) is called *visible* from i if

$$n_{il} = \min\{n_{ij} \mid j \in S, j \neq i\}$$

In other words, (i, l) is visible from i if n_{il} is a minimum entry in the i -th row of the matrix N . The *visible set* \mathcal{V} contains for each taxon i one pair (i, l) that is visible from i .

The next lemma is called the “visibility lemma.”

Lemma 10.5.8 *Let the dissimilarity matrix D be consistent with an additive binary tree T . If i and l are neighbors in T , then the pair (i, l) is visible from i . Moreover, (i, l) is the sole pair that is visible from i .*

Proof It must be shown that $n_{il} = \min\{n_{ij} \mid j \in S, j \neq i\}$. We know from Case (1) in Theorem 10.5.6 that $n_{il} < n_{ij}$ for all j with $i \neq j \neq l$ because i and l are leaf neighbors in T . This proves the lemma. \square

Pseudo-code for the fast neighbor-joining algorithm can be found in Algorithm 10.8.

Theorem 10.5.9 *If the dissimilarity matrix D is consistent with an additive binary tree T , then the fast neighbor-joining algorithm constructs T .*

Proof We will show by induction on the number t of iterations of the while-loop that in each iteration the current visible set \mathcal{V} contains all pairs of neighbors in the additive binary tree yet to be constructed, and that the n_{pq} -values for all $(p, q) \in \mathcal{V}$ coincide with those computed by Saitou and Nei's neighbor-joining algorithm. It then follows that the fast neighbor-joining algorithm constructs T because Saitou and Nei's neighbor-joining algorithm does.

The base case $t = 1$ is a consequence of the visibility lemma 10.5.8. Let S be the set of the taxa, D be the dissimilarity matrix, \mathcal{V} be the visible set, and R_m (for any $m \in S$) be the row sum of m in the t -th iteration. According to the inductive hypothesis, D is consistent with an additive binary tree

Algorithm 10.8 Fast neighbor-joining algorithm.

Input: $n \times n$ dissimilarity matrix $D = (d_{ij})$.

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon i is a leaf in the tree T .
3. For each taxon i , compute $R_i = \sum_{m \in S} d_{im}$
4. Compute the visible set \mathcal{V} .

while $|S| > 3$ **do**

1. a) For each $(p, q) \in \mathcal{V}$ compute $n_{pq} = d_{pq} - (r_p + r_q)$,
where $r_p = \frac{R_p}{|S|-2}$ and $r_q = \frac{R_q}{|S|-2}$.
b) Select $(i, j) \in \mathcal{V}$ so that $n_{ij} = \min\{n_{pq} \mid (p, q) \in \mathcal{V}\}$.
2. Add a new node k to the tree T .
3. Add edges (k, i) and (k, j) with weights $\gamma(k, i) = \frac{1}{2}(d_{ij} + r_i - r_j)$ and $\gamma(k, j) = d_{ij} - \gamma(k, i)$ to the tree T .
4. Update the dissimilarity matrix by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in S \setminus \{i, j, k\}$.
5. Delete i and j from S and add k to S .
6. Compute $R_k = \sum_{m \in S} d_{km}$.
7. For all $m \in S$ with $m \neq k$ update R_m by $R_m \leftarrow R_m - \frac{1}{2}(d_{im} + d_{jm} + d_{ij})$.
8. Delete all pairs (p, q) from \mathcal{V} for which p or q is an element of $\{i, j\}$.
9. Determine a pair (k, l) that is visible from k and add it to \mathcal{V} .

Termination:

Let i, j, m be the remaining three taxa. Add a new internal node v to the tree T , and add edges (v, i) , (v, j) , and (v, m) to the tree T with weights as in Algorithm 10.4 (page 533).

Output: The tree T

T , $R_m = \sum_{l \in S} d_{lm}$ (for any $m \in S$), and \mathcal{V} contains all pairs of neighbors in T . More precisely, if p and q are neighbors in T , then $(p, q) \in \mathcal{V}$ or $(q, p) \in \mathcal{V}$ (or both).

By the correctness of Saitou and Nei's neighbor selection criterion, if n_{ij} is a minimum entry in the matrix N corresponding to D , then i and j are leaf neighbors in T . Hence $(i, j) \in \mathcal{V}$ or $(j, i) \in \mathcal{V}$. Thus, if Saitou and Nei's neighbor-joining algorithm selects taxa i and j in the $(t+1)$ -th iteration, then so does the fast neighbor-joining algorithm. Let \bar{S} be the set of the taxa in the $(t+1)$ -th iteration, i.e., $\bar{S} = (S \setminus \{i, j\}) \cup \{k\}$. Furthermore, let \bar{D} be the dissimilarity matrix obtained in the $(t+1)$ -th iteration from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $\bar{d}_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in \bar{S} \setminus \{k\}$. By Lemma 10.4.16, \bar{D} is consistent with the additive binary tree \bar{T} obtained from T by deleting the leaves i and j and their edges.

Moreover, \bar{R}_m , the row sum of m in the $(t+1)$ -th iteration, satisfies

$$\begin{aligned} \bar{R}_m &= R_m - \frac{1}{2}(d_{im} + d_{jm} + d_{ij}) \\ &= R_m - d_{im} - d_{jm} + \frac{1}{2}(d_{im} + d_{jm} - d_{ij}) \\ &= R_m - d_{im} - d_{jm} + \bar{d}_{km} \\ &= \sum_{l \in \bar{S}} \bar{d}_{lm} \end{aligned}$$

Consequently, the values of \bar{n}_{pq} computed in the $(t+1)$ -th iteration of the fast neighbor-joining algorithm coincide with those computed in the $(t+1)$ -th iteration of Saitou and Nei's neighbor-joining algorithm. The visible set $\bar{\mathcal{V}}$ in the $(t+1)$ -th iteration is obtained from the visible set \mathcal{V} by deleting all pairs (p, q) from \mathcal{V} for which p or q is an element of $\{i, j\}$ and adding a pair that is visible from the new taxon k . It remains to be shown that $\bar{\mathcal{V}}$ contains all pairs of neighbors in the additive binary tree \bar{T} . According to the inductive hypothesis, this is true for all pairs of neighbors p and q with $p, q \in S \setminus \{i, j\}$ because p and q were already neighbors in T . If the node k does not have a leaf neighbor in \bar{T} , then we are done. Otherwise, let l be the neighbor of k in \bar{T} . By the visibility lemma 10.5.8, the pair (k, l) is the sole pair that is visible from k (w.r.t \bar{D}). Thus, (k, l) is the pair that was added to $\bar{\mathcal{V}}$. This proves the theorem. \square

Exercise 10.5.10 Prove that Algorithm 10.8 has a time complexity of $O(n^2)$.

10.6 Non-additive dissimilarity matrices

In the preceding sections, we have seen that there are several $O(n^2)$ time algorithms that reconstruct the correct phylogenetic tree provided that

the dissimilarity matrix is additive. However, in most applications, the observed pairwise dissimilarities between taxa are only estimates of the real distances. In other words, the observed data deviate from the (unknown) real additive data.

To cope with a non-additive dissimilarity matrix $D = (d_{ij})$, one can search for a phylogenetic tree T that best fits the data, i.e., an additive tree T that minimizes the weighted residual sum of squares

$$R = \sum_{i=1}^n \sum_{j=i+1}^n w_{ij} (d_{ij} - d_T(i, j))^2$$

Cavalli-Sforza and Edwards [52] used $w_{ij} = 1$ whereas Fitch and Margoliash [112] used $w_{ij} = 1/d_{ij}^2$. This task, however, is NP-hard as shown by Day [74].⁷

On the other hand, given a tree topology, it is possible to solve for the edge weights (branch lengths) that minimize R by standard least squares methods; see Section 10.6.2. Again, we will use neighbor-joining algorithms to reconstruct the tree topology and derive criteria under which the tree is provably trustworthy. It should be stressed that neighbor-joining can in many cases be successfully applied, even if none of these criteria is met. In fact, various empirical studies have shown that Saitou and Nei's neighbor-joining algorithm performs very well in practice [167, 192, 304]. Although the Farris neighbor-joining algorithm has better theoretical properties than Saitou and Nei's neighbor-joining algorithm (see Section 10.6.1), it yields on average less accurate reconstructions [132].

10.6.1 Nearly additive matrices and quartet-consistency

In this section, it will be shown that all neighbor-joining algorithms considered so far will construct the correct phylogenetic tree (i.e., the correct tree *topology*) provided that the observed dissimilarities do not deviate too much in the L_∞ metric from the real additive distances.

This is made precise in the next definition.

Definition 10.6.1 Let T be an additive tree and let D^τ be the induced additive dissimilarity matrix. A dissimilarity matrix D is said to be *nearly additive w.r.t. T* if

$$d_\infty(D, D^\tau) = \max_{i,j \in S} |d_{ij} - d_{ij}^\tau| < \frac{1}{2} \cdot \min_{e \in T} \{\gamma(e)\}$$

A dissimilarity matrix D is said to be *nearly additive* if there is an additive tree T so that D is nearly additive w.r.t. T .

⁷Agarwala et al. [7] showed NP-hardness for the L_∞ norm. They also gave an $O(n^2)$ time approximation algorithm for the L_∞ norm.

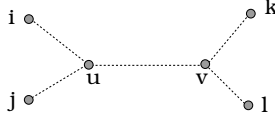


Figure 10.44: $d_{ik}^r + d_{jl}^r - d_{ij}^r - d_{kl}^r = 2d_T(u, v) \geq 2\varepsilon$.

Exercise 10.6.2 Give an example of a nearly additive dissimilarity matrix that is not additive.

Definition 10.6.3 A dissimilarity matrix D is said to be *consistent with a quartet* $(ij : kl)$ if

$$d_{ij} + d_{kl} < \min\{d_{ik} + d_{jl}, d_{il} + d_{jk}\}$$

D is said to be *quartet-consistent* with a tree T if it is consistent with all quartets induced by T .

Theorem 10.6.4 The following statements hold:

1. If a dissimilarity matrix D is consistent with an additive tree T and all edges in T have a strictly positive weight, then D is nearly additive w.r.t. T .
2. If a dissimilarity matrix D is nearly additive w.r.t. an additive tree T , then it is quartet-consistent with T .
3. If a dissimilarity matrix D is quartet-consistent with a tree T , then it is a distance matrix.

Proof (1) Obvious.

(2) Let $\varepsilon = \min_{e \in T} \{\gamma(e)\}$ and note that $\varepsilon > 0$. Let $(ij : kl)$ be a quartet induced by T . According to the assumption, we have $d_{ij} + d_{kl} < d_{ij}^r + d_{kl}^r + \varepsilon$ as well as $d_{ik}^r + d_{jl}^r - \varepsilon < d_{ik} + d_{jl}$. Because $(ij : kl)$ is a quartet induced by T , there is an edge e so that i and j are separated from k and l by e . It follows from $\varepsilon \leq \gamma(e)$ that $d_{ij}^r + d_{kl}^r + 2\varepsilon \leq d_{ik}^r + d_{jl}^r$; see Figure 10.44. Putting all pieces together, we get

$$d_{ij} + d_{kl} < d_{ij}^r + d_{kl}^r + \varepsilon \leq d_{ik}^r + d_{jl}^r - \varepsilon < d_{ik} + d_{jl}$$

The proof of $d_{ij} + d_{kl} < d_{il} + d_{jk}$ is verbatim the same. Therefore, we have $d_{ij} + d_{kl} < \min\{d_{ik} + d_{jl}, d_{il} + d_{jk}\}$, i.e., D is consistent with the quartet $(ij : kl)$. Because the quartet was chosen arbitrarily, it follows that D is quartet-consistent with T .

(3) Let D be quartet-consistent with T , and let $i, j, k \in S$ be pairwise distinct

taxa (leaves in T). Clearly, $(ij : kk)$ is a quartet in T . Since D is consistent with this quartet it follows

$$d_{ij} = d_{ij} + d_{kk} < \min\{d_{ik} + d_{jk}, d_{ik} + d_{jk}\} = d_{ik} + d_{kj}$$

That is, the triangle inequality holds. Hence D is a distance matrix. \square

The condition that all edges in T must have a strictly positive weight cannot be dropped from the first statement of Theorem 10.6.4. To put it differently, if a dissimilarity matrix D is consistent with an additive tree T and an external edge in T has weight 0, then D cannot be nearly additive w.r.t. T because $\min_{e \in T}\{\gamma(e)\} = 0$. The next lemma shows that in this case D is not even quartet-consistent with T .

Lemma 10.6.5 *If a dissimilarity matrix D is consistent with an additive tree T and an external edge in T has weight 0, then D is not quartet-consistent with T .*

Proof Let the leaf i be adjacent to node u in T and $\gamma(u, i) = 0$. Since the degree of u is at least 3, there are two different nodes v and w so that (u, v) and (u, w) are edges in T . The removal of these edges splits T into three connected components. Let j be a leaf in the connected component containing v and let k be a leaf in the connected component containing w . Then, $(ii : jk)$ is a quartet in T . If D were quartet-consistent with T , then the inequality

$$d_{jk} = d_{ii} + d_{jk} < \min\{d_{ij} + d_{ik}, d_{ik} + d_{ij}\} = d_{ij} + d_{ik}$$

would hold true. However, we have $d_{jk} = d_T(j, u) + d_T(u, k) = d_T(j, i) + d_T(i, k) = d_{ij} + d_{ik}$. \square

The following example illustrates Lemma 10.6.5. The dissimilarity matrix

D	1	2	3	4
1	0	1	2	2
2		0	3	3
3			0	2
4				0

is consistent with the additive tree T in Figure 10.45. Clearly, $(ii : jk)$, where $i = 1$, $j = 2$, and $k = 3$ is a quartet in T . If D were quartet-consistent with T , then the inequality

$$d_{23} = d_{11} + d_{23} < \min\{d_{12} + d_{13}, d_{13} + d_{12}\} = d_{12} + d_{13}$$

would hold true. However, we have $d_{23} = 3 = 1 + 2 = d_{12} + d_{13}$.

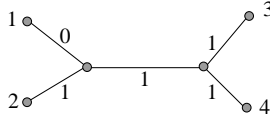


Figure 10.45: An additive tree with an external edge having weight 0.

Lemma 10.6.6 Let the dissimilarity matrix D on $S = \{1, \dots, n\}$ be quartet-consistent with a binary tree T . Suppose $|S| \geq 4$ and that the taxa i and j are adjacent to the same node k in T (i.e., i and j are leaf neighbors). Let \bar{D} be the $(n-1) \times (n-1)$ dissimilarity matrix obtained from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with

$$\bar{d}_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$$

for every $m \in S \setminus \{i, j, k\}$. Then \bar{D} is quartet-consistent with the binary tree \bar{T} obtained from T by deleting the leaves i and j as well as the edges (k, i) and (k, j) .

Proof If $n-1 = 3$, then \bar{T} is a star. Suppose that k , p , and q are the remaining three taxa. We show that \bar{D} is quartet-consistent with \bar{T} by considering the following quartets (a) $(kk : pq)$, (b) $(kp : qq)$, and (c) $(kk : qq)$ in \bar{T} . The quartets $(kq : pp)$ and $(kk : pp)$ need not be taken into account because these cases are symmetric to (b) and (c), respectively.

(a) Because $(kk : pq) \in Q(\bar{T})$, we have to show that

$$\begin{aligned} \bar{d}_{kk} + \bar{d}_{pq} &< \min\{\bar{d}_{kp} + \bar{d}_{kq}, \bar{d}_{kq} + \bar{d}_{kp}\} \\ \Leftrightarrow \bar{d}_{pq} &< \bar{d}_{kp} + \bar{d}_{kq} \\ \Leftrightarrow d_{pq} &< \frac{1}{2}(d_{ip} + d_{jp} - d_{ij}) + \frac{1}{2}(d_{iq} + d_{jq} - d_{ij}) \\ \Leftrightarrow d_{ij} + d_{pq} &< \frac{1}{2}(d_{ip} + d_{jp} + d_{iq} + d_{jq}) \end{aligned}$$

Since T is binary, $(ij : pq)$ is a quartet in T . By assumption this implies

$$d_{ij} + d_{pq} < \min\{d_{ip} + d_{jq}, d_{iq} + d_{jp}\}$$

It is easy to see that the claim follows.

(b) For $(kp : qq) \in Q(\bar{T})$, we have to show that

$$\begin{aligned} \bar{d}_{kp} + \bar{d}_{qq} &< \min\{\bar{d}_{kq} + \bar{d}_{pq}, \bar{d}_{kq} + \bar{d}_{pq}\} \\ \Leftrightarrow \bar{d}_{kp} &< \bar{d}_{kq} + \bar{d}_{pq} \\ \Leftrightarrow \frac{1}{2}(d_{ip} + d_{jp} - d_{ij}) &< \frac{1}{2}(d_{iq} + d_{jq} - d_{ij}) + d_{pq} \\ \Leftrightarrow d_{ip} + d_{jp} &< d_{iq} + d_{jq} + 2d_{pq} \end{aligned}$$

The last inequality holds true because $(ip : qq)$ and $(jp : qq)$ are quartets in T and thus $d_{ip} < d_{iq} + d_{pq}$ and $d_{jp} < d_{jq} + d_{pq}$ by the assumption that D is quartet-consistent with T .

(c) For $(kk : qq) \in Q(\bar{T})$, one must show that

$$\begin{aligned} \bar{d}_{kk} + \bar{d}_{qq} &< \min\{\bar{d}_{kq} + \bar{d}_{kq}, \bar{d}_{kq} + \bar{d}_{kq}\} \\ \Leftrightarrow 0 &< 2\bar{d}_{kq} \\ \Leftrightarrow 0 &< d_{iq} + d_{jq} - d_{ij} \end{aligned}$$

The last inequality follows as in the proof of statement (3) in Theorem 10.6.4: $(ij : qq) \in Q(t)$ implies $d_{ij} < d_{iq} + d_{jq}$.

Now suppose that $n - 1 > 3$. Let $(kl : pq)$ be a quartet induced by \bar{T} that contains the new taxon k . Without loss of generality, we may assume that $|\{k, l, p, q\}| \geq 4$ (otherwise we can argue as in the case $n - 1 = 3$). In order to prove that \bar{D} is consistent with $(kl : pq)$, we have to show that

$$\bar{d}_{kl} + \bar{d}_{pq} < \min\{\bar{d}_{kp} + \bar{d}_{lq}, \bar{d}_{kq} + \bar{d}_{lp}\}$$

where $\bar{d}_{kl} = \frac{1}{2}(d_{il} + d_{jl} - d_{ij})$, $\bar{d}_{kp} = \frac{1}{2}(d_{ip} + d_{jp} - d_{ij})$, $\bar{d}_{kq} = \frac{1}{2}(d_{iq} + d_{jq} - d_{ij})$, $\bar{d}_{lp} = d_{lp}$, $\bar{d}_{lq} = d_{lq}$, and $\bar{d}_{pq} = d_{pq}$. Thus, we must prove that

$$\frac{1}{2}(d_{il} + d_{jl} - d_{ij}) + d_{pq} < \min\left\{\frac{1}{2}(d_{ip} + d_{jp} - d_{ij}) + d_{lq}, \frac{1}{2}(d_{iq} + d_{jq} - d_{ij}) + d_{lp}\right\}$$

or equivalently

$$\frac{1}{2}(d_{il} + d_{jl}) + d_{pq} < \min\left\{\frac{1}{2}(d_{ip} + d_{jp}) + d_{lq}, \frac{1}{2}(d_{iq} + d_{jq}) + d_{lp}\right\}$$

Because $(il : pq)$ is a quartet in T and D is quartet-consistent with T , we have (1) $d_{il} + d_{pq} < d_{ip} + d_{lq}$ and (2) $d_{il} + d_{pq} < d_{iq} + d_{lp}$. Analogously, because $(jl : pq)$ is a quartet in T , we have (3) $d_{jl} + d_{pq} < d_{jp} + d_{lq}$ and (4) $d_{jl} + d_{pq} < d_{jq} + d_{lp}$. The combination of (1) and (3) yields $d_{il} + d_{jl} + 2d_{pq} < d_{ip} + d_{jp} + 2d_{lq}$ and the combination of (2) and (4) yields $d_{il} + d_{jl} + 2d_{pq} < d_{iq} + d_{jq} + 2d_{lp}$. Thus, we derive the desired inequality

$$d_{il} + d_{jl} + 2d_{pq} < \min\{d_{ip} + d_{jp} + 2d_{lq}, d_{iq} + d_{jq} + 2d_{lp}\}$$

This proves that \bar{D} is consistent with every quartet $(kl : pq)$ induced by \bar{T} that contains the new taxon k . Now consider a quartet $(lm : pq)$ induced by \bar{T} that does not contain the new taxon k . Since $(lm : pq)$ is also a quartet in T , D is quartet-consistent with T , and D restricted to $\{l, m, p, q\}$ coincides with \bar{D} restricted to $\{l, m, p, q\}$, it follows that \bar{D} is also consistent with $(lm : pq)$. In summary, \bar{D} is consistent with every quartet induced by \bar{T} , i.e., \bar{D} is quartet-consistent with \bar{T} . \square

It should be pointed out that Lemma 10.6.6 is not valid for non-binary trees. To see this, consider the star T of four taxa in which every edge has weight 1. The induced additive matrix D is quartet-consistent with T . If we join two of the taxa into a new taxon, we get an additive star \bar{T} of three taxa in which the new edge has weight 0. By Lemma 10.6.5, the additive matrix \bar{D} is not quartet-consistent with \bar{T} .

Theorem 10.6.7 *If a dissimilarity matrix D is quartet-consistent with a binary tree T , then there is no other binary tree T' (different from T) so that D is quartet-consistent with T' .*

Proof The proof is very similar to the proof of Theorem 10.4.18. Nevertheless, for the convenience of the reader, we will elaborate upon it.

Suppose that D is quartet-consistent with two binary trees T and T' . Lemma 10.4.14 covers the cases $n = 2$ and $n = 3$. For $n > 3$, we show that two leaves i and j are neighbors in T if and only if they are neighbors in T' . For an indirect proof, suppose that i and j are leaf neighbors in T but not leaf neighbors in T' . As in the proof of Theorem 10.4.18, there are two taxa $p, q \in S$ so that $(ij : pq)$ is a quartet in $Q(T)$ and $(ip : jq)$ is a quartet in $Q(T')$. On the one hand, because D is consistent with the quartet $(ij : pq) \in Q(T)$, we have $d_{ij} + d_{pq} < \min\{d_{ip} + d_{jq}, d_{iq} + d_{jp}\}$. On the other hand, since D is consistent with the quartet $(ip : jq) \in Q(T')$, it follows that $d_{ip} + d_{jq} < \min\{d_{ij} + d_{pq}, d_{iq} + d_{jp}\}$. Consequently, $d_{ij} + d_{pq} < d_{ip} + d_{jq} < d_{ij} + d_{pq}$. This contradiction shows that i and j are also leaf neighbors in T' .

Let the leaf neighbors i and j be adjacent to node v in T and node v' in T' . Label both nodes v and v' with the new taxon k . Let \bar{T} and \bar{T}' be the trees obtained from T and T' , respectively, by deleting i and j and their edges. Observe that the nodes with label k are leaves in \bar{T} and \bar{T}' because T and T' are binary trees. Moreover, \bar{T} and \bar{T}' are binary trees. By Lemma 10.6.6, both \bar{T} and \bar{T}' are quartet-consistent with the $(n - 1) \times (n - 1)$ dissimilarity matrix \bar{D} obtained from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with

$$\bar{d}_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$$

for every $m \in \{1, \dots, n\} \setminus \{i, j, k\}$. According to the inductive hypothesis, \bar{T} and \bar{T}' coincide. Therefore, T and T' coincide as well. \square

Exercise 10.6.8 Suppose that quartet-consistency is defined as follows: A dissimilarity matrix D is consistent with a quartet $(ij : kl)$ if

$$d_{ij} + d_{kl} \leq \min\{d_{ik} + d_{jl}, d_{il} + d_{jk}\}$$

and D is quartet-consistent with a tree T if it is consistent with all quartets induced by T . Show that Theorem 10.6.7 does not hold true with this definition of quartet-consistency.

Algorithm 10.9 In contrast to Algorithm 10.4 (page 533), this generic neighbor-joining algorithm does not assign weights to edges.

Input: $n \times n$ dissimilarity matrix $D = (d_{ij})$, where $n \geq 3$.

Initialization:

1. Let $S = \{1, \dots, n\}$ be the set of taxa.
2. Each taxon i is a leaf in the tree T .

while $|S| > 3$ **do**

1. Using a specific neighbor selection criterion, select two taxa i and j that are neighbors in the (yet unknown) tree T .
2. Add a new node k to the tree T .
3. Add edges (k, i) and (k, j) to the tree T .
4. Update the dissimilarity matrix by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $d_{km} = \frac{1}{2}(d_{im} + d_{jm} - d_{ij})$ for all $m \in S \setminus \{i, j, k\}$.
5. Delete i and j from S and add the new (artificial) taxon k to S .

Termination:

Connect the remaining three taxa i, j, m in a star, i.e., add a new internal node v as well as edges (v, i) , (v, j) , and (v, m) to the tree T .

Output: The tree T .

Theorem 10.6.9 *If a dissimilarity matrix D is quartet-consistent with a binary tree T , then the generic neighbor-joining algorithm (Algorithm 10.9) applied to D constructs T .*

Proof Similar to the proof of Theorem 10.5.1. □

Theorem 10.6.10 *If a dissimilarity matrix D is quartet-consistent with a binary tree T , then the Farris neighbor-joining algorithm constructs T .*

Proof According to Theorem 10.6.9, it is sufficient to show that i and j are leaf neighbors in T whenever d'_{ij} is a minimum entry in the D' matrix corresponding to D . The proof of this fact is similar to that of Theorem 10.5.3 and we just provide the key observations.

According to the proof of Theorem 10.5.3, the Farris neighbor-joining algorithm does not select the distinguished taxon r because the inequality $d_{rk} + d_{rl} > d_{kl}$ holds for leaf neighbors $k \neq r$ and $l \neq r$. This inequality holds here as well: since $(rr : kl)$ is a quartet in T and D is consistent with this quartet, it follows

$$d_{kl} = d_{rr} + d_{kl} < \min\{d_{rk} + d_{rl}, d_{rk} + d_{rl}\} = d_{rk} + d_{rl}$$

Case (1) in the proof of Theorem 10.5.3 is true because $d_{ik} + d_{rj} < d_{rk} + d_{ij}$ holds. This inequality holds here as well: because $(ik : rj)$ is a quartet in T and D is consistent with this quartet, we have

$$d_{ik} + d_{rj} < \min\{d_{ri} + d_{jk}, d_{rk} + d_{ij}\}$$

Case (2) in the proof of Theorem 10.5.3 is true because the inequalities (a) $d_{ri} + d_{kl} < d_{rk} + d_{il}$ and (b) $d_{rj} + d_{il} < d_{rl} + d_{ij}$ hold. These inequalities also hold here because $(ri : kl)$ and $(rj : kl)$ are quartets in T . □

Corollary 10.6.11 *If a dissimilarity matrix D is quartet-consistent with a binary tree T , then the second version of the Farris neighbor-joining algorithm constructs T .*

Proof According to the proof of Theorem 10.5.4, the second version of the Farris neighbor-joining algorithm computes the same matrix D' as the Farris neighbor-joining algorithm. Hence the claim is a direct consequence of Theorem 10.6.10. □

In view of the preceding results, one would expect the following statement to be true: If a dissimilarity matrix D is quartet-consistent with a binary tree T , then Saitou and Nei's neighbor-joining algorithm constructs T . Quite surprisingly, this statement is not true (for more than seven taxa). The following counterexample is due to Mihaescu et al. [222].

D	1	2	3	4	5	6	7	8
1	0	3	2	2	2	3	3	3
2	3	0	3	3	3	2	2	2
3	2	3	0	0.1	0.4	3	3	3
4	2	3	0.1	0	0.4	3	3	3
5	2	3	0.4	0.4	0	3	3	3
6	3	2	3	3	3	0	0.1	0.4
7	3	2	3	3	3	0.1	0	0.4
8	3	2	3	3	3	0.4	0.4	0

Figure 10.46: The matrix D is additive.

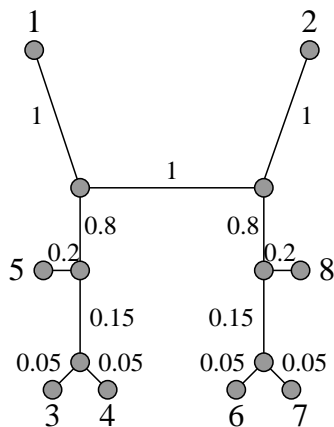


Figure 10.47: The binary additive tree T is consistent with the dissimilarity matrix D from Figure 10.46.

D^δ	1	2	3	4	5	6	7	8
1	0	2.7	2.6	2.6	2.6	4.4	4.4	4.4
2	2.7	0	4.4	4.4	4.4	2.6	2.6	2.6
3	2.6	4.4	0	0.1	0.4	2.7	2.7	2.7
4	2.6	4.4	0.1	0	0.4	2.7	2.7	2.7
5	2.6	4.4	0.4	0.4	0	2.7	2.7	2.7
6	4.4	2.6	2.7	2.7	2.7	0	0.1	0.4
7	4.4	2.6	2.7	2.7	2.7	0.1	0	0.4
8	4.4	2.6	2.7	2.7	2.7	0.4	0.4	0

Figure 10.48: The matrix D^δ is quartet-consistent with the binary additive tree T of Figure 10.47.

The dissimilarity matrix D in Figure 10.46 is additive, and the binary additive tree T of Figure 10.47 is consistent with it. Figure 10.48 shows the dissimilarity matrix D^δ , which was obtained by distorting the additive matrix D . It is readily verified that D^δ is quartet-consistent with T , and according to Theorem 10.6.7 there is no other tree with this property. However, Saitou and Nei's neighbor-joining algorithm constructs a different tree, namely the tree in Figure 10.49. In other words, it produces the wrong tree.

On the positive side, Mihaescu et al. [222] have shown that Saitou and Nei's neighbor-joining algorithm constructs the correct tree if an additional property besides quartet-consistency holds true. As a matter of fact, their result is a generalization of the following theorem.

Theorem 10.6.12 *If a dissimilarity matrix D is nearly additive w.r.t. an additive binary tree T , then Saitou and Nei's neighbor-joining algorithm applied to D constructs T .*

Proof See [19, 87]. □

It can be shown that the fast neighbor-joining algorithm also produces the correct tree if the input matrix D is nearly additive [87, 222]. In essence, this is because the visibility lemma is also valid for nearly additive dissimilarity matrices. In fact, it is even valid in the presence of quartet-consistency, as Lemma 10.6.13 shows.

Lemma 10.6.13 *Let the dissimilarity matrix D be quartet-consistent with a binary tree T . If i and k are neighbors in T , then the pair (i, k) is visible from i w.r.t. D . Moreover, (i, k) is the sole pair which is visible from i .*

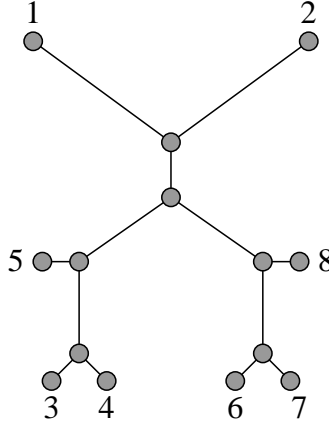


Figure 10.49: The tree constructed from D^δ by Saitou and Nei's neighbor-joining algorithm. This tree is not quartet-consistent with D^δ because $d_{12}^\delta + d_{36}^\delta = 2.7 + 2.7 > 2.6 + 2.6 = d_{13}^\delta + d_{26}^\delta$.

Proof Because i and k are neighbors in T , it follows as in Case (1) of the proof of Theorem 10.5.6 that for any j with $i \neq j \neq k$

$$n_{ik} - n_{ij} = \frac{1}{n-2} \sum_{m \notin \{i,j,k\}} (d_{ik} + d_{jm} - (d_{ij} + d_{km}))$$

For any $m \notin \{i,j,k\}$, $(ik : jm)$ is a quartet induced by T . Due to the fact that D is quartet-consistent with T , it follows $d_{ik} + d_{jm} - (d_{ij} + d_{km}) < 0$ for any $m \notin \{i,j,k\}$. Hence $n_{ik} < n_{ij}$. This proves that $n_{ik} = \min\{n_{ij} \mid j \in S, j \neq i\}$, i.e., (i, k) is visible from i , and that there is no other pair that is visible from i . \square

We conclude this section with the ADDTREE neighbor-joining algorithm, a quartet-method devised by Sattah and Tversky [279].

Definition 10.6.14 ADDTREE's neighbor selection criterion: Given a dissimilarity matrix D , for all pairs of taxa i and j , compute the number of pairs of taxa k and l (where i, j, k, l are pairwise distinct) so that D is consistent with the quartet $(ij : kl)$. In other words, compute the matrix $Q = (q_{ij})_{i,j \in S}$ with

$$q_{ij} = |\{(k, l) \in S \times S : |\{i, j, k, l\}| = 4 \text{ and } d_{ij} + d_{kl} < \min\{d_{ik} + d_{jl}, d_{il} + d_{jk}\}\}|$$

and then select two taxa that attain a maximum in Q .

It is unsurprising that the following theorem holds.

Theorem 10.6.15 *If a dissimilarity matrix D is quartet-consistent with a binary tree T , then ADDTREE constructs T .*

Proof According to Theorem 10.6.9, we must show for $n > 3$ taxa that the two taxa selected by ADDTREE's neighbor selection criterion are leaf neighbors in T . Fix a pair of leaf neighbors i and j in T . According to Lemma 10.4.12, for any other pair k and l of taxa so that i, j, k, l are pairwise distinct, we have $d_{ij} + d_{kl} < d_{ik} + d_{jl} = d_{il} + d_{jk}$. Thus,

$$q_{ij} = (n-2)(n-3)$$

attains the maximum possible value. To prove the theorem, it suffices to demonstrate that no pair k and l of non-neighbors achieves this value. If k and l are not leaf neighbors in T , then by Lemma 10.4.13 there are two leaves p and q so that $k, l, p, q \in S$ are pairwise distinct and $d_{kp} + d_{lq} < d_{kl} + d_{pq} = d_{kq} + d_{lp}$. Therefore, $q_{kl} < (n-2)(n-3)$. \square

The running time of a naive implementation of ADDTREE is $O(n^5)$: in each of the $n-3$ iterations, ADDTREE examines all quadruples, and their number is proportional to n^4 . A cleverer implementation brings this worst-case time complexity down to $O(n^4)$ [85]. In the initialization step, the matrix $Q = (q_{ij})_{1 \leq i, j \leq n}$ is computed. Since Q has $O(n^2)$ many entries and the computation of each entry takes $O(n^2)$ time, the time complexity of the initialization step is $O(n^4)$. Now consider an iteration of the while-loop and suppose that the taxa i and j are selected in Algorithm 10.9 (page 555). Let \bar{D} be the dissimilarity matrix obtained from D by deleting the rows and columns corresponding to i and j and adding a new row and column for the new taxon k with $\bar{d}_{kp} = \frac{1}{2}(d_{ip} + d_{jp} - d_{ij})$ for all p with $i \neq p \neq j$. The new matrix \bar{Q} for the new set of taxa $\bar{S} = (S \setminus \{i, j\}) \cup \{k\}$ can be obtained from Q in two phases as follows: In the first phase, for any pair $l \neq m$ of taxa from $S \setminus \{i, j\}$ we decrement the value q_{lm} by one if $d_{ij} + d_{lm} < \min\{d_{il} + d_{jm}, d_{im} + d_{jl}\}$ holds. Then, for every taxon $p \in S \setminus \{i, j, l, m\}$, we test whether $d_{ip} + d_{lm} < \min\{d_{il} + d_{pm}, d_{im} + d_{pl}\}$ or $d_{jp} + d_{lm} < \min\{d_{jl} + d_{pm}, d_{jm} + d_{pl}\}$ is true. For each positive test we decrement q_{lm} by one. This first phase takes $O(n^3)$ time and removes the contribution of the selected taxa i and j from the original q_{lm} value. The addition of the contribution of the new taxon k is done in the second phase, in which the entries \bar{q}_{kp} for the new taxon k and all p with $i \neq p \neq j$ are computed from scratch in $O(n^3)$ time. Recall that \bar{q}_{kp} is obtained by counting the number of pairs l and m so that $|\{k, p, l, m\}| = 4$ and $\bar{d}_{kp} + \bar{d}_{lm} < \min\{\bar{d}_{kl} + \bar{d}_{pm}, \bar{d}_{km} + \bar{d}_{pl}\}$. During the computation of \bar{q}_{kp} , whenever such a pair l and m is encountered, we increment the value q_{lm} by one. After these two phases, $q_{lm} = \bar{q}_{lm}$.

Needless to say that ADDTREE's overall time complexity of $O(n^4)$ is a major disadvantage compared to the other neighbor-joining algorithms.

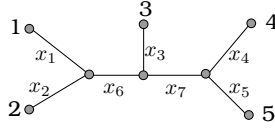


Figure 10.50: Estimating edge weights: We are searching for the weights x_1, \dots, x_7 of the edges e_1, \dots, e_7 .

10.6.2 Estimating edge weights

Once we have found a phylogenetic tree T , we must assign weights to the edges of the tree that best fit the data. As in [52], we use standard least squares methods to find the edge weights (branch lengths) that minimize the residual sum of squares

$$R = \sum_{i=1}^n \sum_{j=i+1}^n (d_{ij} - d_T(i, j))^2$$

Consider the tree in Figure 10.50. If the tree were additive, then the following system of linear equations would have a solution.

$$\begin{array}{rcl}
 x_1 + x_2 & & = d_{12} \\
 x_1 + x_3 + x_6 & & = d_{13} \\
 x_1 + x_4 + x_6 + x_7 & & = d_{14} \\
 x_1 + x_5 + x_6 + x_7 & & = d_{15} \\
 x_2 + x_3 + x_6 & & = d_{23} \\
 x_2 + x_4 + x_6 + x_7 & & = d_{24} \\
 x_2 + x_5 + x_6 + x_7 & & = d_{25} \\
 x_3 + x_4 + x_7 & & = d_{34} \\
 x_3 + x_5 + x_7 & & = d_{35} \\
 x_4 + x_5 & & = d_{45}
 \end{array}$$

This system of linear equations can also be written in matrix form as

$$\begin{pmatrix}
 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0
 \end{pmatrix}
 \begin{pmatrix}
 x_1 \\
 x_2 \\
 x_3 \\
 x_4 \\
 x_5 \\
 x_6 \\
 x_7
 \end{pmatrix}
 =
 \begin{pmatrix}
 d_{12} \\
 d_{13} \\
 d_{14} \\
 d_{15} \\
 d_{23} \\
 d_{24} \\
 d_{25} \\
 d_{34} \\
 d_{35} \\
 d_{45}
 \end{pmatrix}$$

However, here we assume that the dissimilarity matrix is non-additive and hence this equation cannot have a solution. Consequently, we search for an approximate solution of the equation

$$Ax = \mathbf{d}$$

where A is the known $\frac{n(n-1)}{2} \times 2n - 3$ coefficient matrix (a binary phylogenetic tree with n leaves has $2n - 3$ edges), $\mathbf{x} = (x_1, \dots, x_{2n-3})$ is a $2n - 3$ -dimensional parameter vector representing the unknown edge weights, and $\mathbf{d} = (d_{12}, \dots, d_{(n-1)n})$ is the known $\frac{n(n-1)}{2}$ -dimensional vector consisting of the pairwise distances between the n taxa.

We use least squares fitting to find the approximate solution. More precisely, we want to minimize the squared Euclidean norm of the residual $Ax - \mathbf{d}$, that is, the quantity

$$\|Ax - \mathbf{d}\|^2 = \sum_{i=1}^{n(n-1)/2} ([Ax]_i - \mathbf{d}_i)^2$$

where $[Ax]_i$ denotes the i -th component of the vector Ax . Note that

$$\|Ax - \mathbf{d}\|^2 = \sum_{i=1}^n \sum_{j=i+1}^n \left(\left(\sum_{e_k \in E(i,j)} x_k \right) - d_{ij} \right)^2$$

where $E(i, j)$ denotes the set of all edges on the path from leaf (taxon) i to leaf (taxon) j in the phylogenetic tree T .

Using the fact that the squared Euclidean norm of a vector \mathbf{v} is $\mathbf{v}^T \mathbf{v}$, where \mathbf{v}^T stands for the transpose of \mathbf{v} , we can rewrite the expression as

$$(Ax - \mathbf{d})^T (Ax - \mathbf{d}) = (Ax)^T (Ax) - \mathbf{d}^T Ax - (Ax)^T \mathbf{d} + \mathbf{d}^T \mathbf{d}$$

The two middle terms $\mathbf{d}^T (Ax)$ and $(Ax)^T \mathbf{d}$ are equal and the minimum is found at the zero of the derivative with respect to \mathbf{x} :

$$\frac{d}{d\mathbf{x}} [(Ax)^T (Ax) - 2(Ax)^T \mathbf{d} + \mathbf{d}^T \mathbf{d}] = 2A^T Ax - 2A^T \mathbf{d} = \mathbf{0}$$

Therefore, the minimizing vector is a solution of the equation

$$A^T Ax = A^T \mathbf{d} \tag{10.1}$$

which in fact is a system of linear equations. In our example, we have

$$A^T Ax = \begin{pmatrix} 4 & 1 & 1 & 1 & 1 & 3 & 2 \\ 1 & 4 & 1 & 1 & 1 & 3 & 2 \\ 1 & 1 & 4 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 4 & 1 & 2 & 3 \\ 1 & 1 & 1 & 1 & 4 & 2 & 3 \\ 3 & 3 & 2 & 2 & 2 & 6 & 4 \\ 2 & 2 & 2 & 3 & 3 & 4 & 6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} d_{12} + d_{13} + d_{14} + d_{15} \\ d_{12} + d_{23} + d_{24} + d_{25} \\ d_{13} + d_{23} + d_{34} + d_{35} \\ d_{14} + d_{24} + d_{34} + d_{45} \\ d_{15} + d_{25} + d_{35} + d_{45} \\ d_{13} + d_{14} + d_{15} + d_{23} + d_{24} + d_{25} \\ d_{14} + d_{15} + d_{24} + d_{25} + d_{34} + d_{35} \end{pmatrix}$$

The matrix $A^T A$ on the left-hand side is a $(2n-3) \times (2n-3)$ square matrix, which is invertible if the rank of A is $2n-3$. In our context, this is always the case; see [44]. Therefore, the solution of the system of linear equations is unique and given by

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{d}.$$

The matrix $(A^T A)^{-1} A^T$ is called the pseudoinverse of A .

In our example, the inverse of the matrix $A^T A$ has the form

$$(A^T A)^{-1} = \begin{pmatrix} \frac{5}{12} & \frac{1}{12} & 0 & 0 & 0 & -\frac{1}{4} & 0 \\ \frac{1}{12} & \frac{5}{12} & 0 & 0 & 0 & -\frac{1}{4} & 0 \\ 0 & 0 & \frac{5}{16} & 0 & 0 & -\frac{1}{16} & -\frac{1}{16} \\ 0 & 0 & 0 & \frac{5}{12} & \frac{1}{12} & 0 & -\frac{1}{4} \\ 0 & 0 & 0 & \frac{1}{12} & \frac{5}{12} & 0 & -\frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{16} & 0 & 0 & \frac{9}{16} & -\frac{3}{16} \\ 0 & 0 & -\frac{1}{16} & -\frac{1}{4} & -\frac{1}{4} & -\frac{3}{16} & \frac{9}{16} \end{pmatrix}$$

and the components of the solution vector $\mathbf{x} = (A^T A)^{-1} A^T \mathbf{d}$ are

$$\begin{aligned} x_1 &= \frac{1}{2}d_{12} + \frac{1}{6}(d_{13} + d_{14} + d_{15} - d_{23} - d_{24} - d_{25}) \\ x_2 &= \frac{1}{2}d_{12} - \frac{1}{6}(d_{13} + d_{14} + d_{15} - d_{23} - d_{24} - d_{25}) \\ x_3 &= \frac{1}{4}(d_{13} + d_{23} + d_{34} + d_{35}) - \frac{1}{8}(d_{14} + d_{15} + d_{24} + d_{25}) \\ x_4 &= \frac{1}{6}(d_{14} - d_{15} + d_{24} - d_{25} + d_{34} - d_{35}) + \frac{1}{2}d_{45} \\ x_5 &= -\frac{1}{6}(d_{14} - d_{15} + d_{24} - d_{25} + d_{34} - d_{35}) + \frac{1}{2}d_{45} \\ x_6 &= -\frac{1}{2}d_{12} + \frac{1}{4}(d_{13} + d_{23} - d_{34} - d_{35}) + \frac{1}{8}(d_{14} + d_{15} + d_{24} + d_{25}) \\ x_7 &= -\frac{1}{4}(d_{13} + d_{23} - d_{34} - d_{35}) + \frac{1}{8}(d_{14} + d_{15} + d_{24} + d_{25}) - \frac{1}{2}d_{45} \end{aligned}$$

The drawback of this standard least squares method is its time complexity. The matrix A for a binary phylogenetic tree with n leaves (taxa) is an $\frac{n(n-1)}{2} \times (2n-3)$ matrix. The multiplication of the $(2n-3) \times \frac{n(n-1)}{2}$ matrix A^T with the matrix A takes $O(n^4)$ time, using the standard implementation of matrix multiplication. The inversion of the $(2n-3) \times (2n-3)$ matrix $A^T A$ by Gaussian elimination requires $O(n^3)$ time. Both, the multiplication of

A^T with the vector \mathbf{d} and the multiplication of the $(2n-3) \times (2n-3)$ matrix $(A^T A)^{-1}$ with the vector $A^T \mathbf{d}$ take $O(n^3)$ time. Thus, the overall time complexity is $O(n^4)$.

Bryant and Waddell [44, 45] have shown that the vector

$$\mathbf{x} = (A^T A)^{-1} A^T \mathbf{d}.$$

can be computed in $O(n^2)$ time, using the following two key ideas:

- In row i of the matrix $(A^T A)^{-1}$ all entries are zero, except for those index pairs (i, j) for which e_j is an edge directly adjacent to e_i . Hence the weight x_i assigned to an edge e_i depends only on the edges directly adjacent to e_i . Because an internal edge has four directly adjacent edges (an external edge has two directly adjacent edges), it is possible to derive a system of five (three) linear equations with five (three) unknowns. We shall see that this gives closed formulae for x_i , which enable us to compute each of the $2n-3$ edge weights in constant time, provided that some information about the neighboring edges is available.
- The information required in the previous step can be directly computed on the tree T in $O(n^2)$ time.

According to Equation 10.1, we have

$$[A^T A \mathbf{x}]_i = [A^T \mathbf{d}]_i \quad (10.2)$$

for all i with $1 \leq i \leq 2n-3$. Recall that each edge e_i , $1 \leq i \leq 2n-3$, splits the set S into two clusters C_i and $\bar{C}_i = S \setminus C_i$. In the following, we make use of the equalities

$$\begin{aligned} [A^T A \mathbf{x}]_i &= \sum_{p \in C_i, q \in \bar{C}_i} \sum_{e_t \in E(p, q)} x_t \\ [A^T \mathbf{d}]_i &= \sum_{p \in C_i, q \in \bar{C}_i} d_{pq} \end{aligned}$$

If $e_i = (u, i)$ is an external edge with adjacent edges e_j and e_k as shown in Figure 10.51, then

$$[A^T A \mathbf{x}]_i = \sum_{p \in C_j \cup C_k} \sum_{e_t \in E(p, i)} x_t$$

With the definition

$$y_1 = \sum_{p \in C_j} \sum_{e_t \in E(p, u)} x_t \quad \text{and} \quad y_2 = \sum_{q \in C_k} \sum_{e_t \in E(q, u)} x_t$$

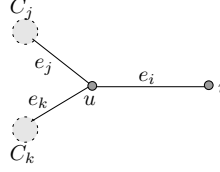


Figure 10.51: e_i is an external edge.

as well as $n_j = |C_j|$ and $n_k = |C_k|$, it follows that

$$\begin{aligned} [A^T A \mathbf{x}]_i &= \sum_{p \in C_j} \sum_{e_t \in E(p, i)} x_t + \sum_{q \in C_k} \sum_{e_t \in E(q, i)} x_t = y_1 + n_j x_i + y_2 + n_k x_i \\ [A^T A \mathbf{x}]_j &= \sum_{p \in C_j, q \in C_k} \sum_{e_t \in E(p, q)} x_t + \sum_{p \in C_j} \sum_{e_t \in E(p, i)} x_t = n_k y_1 + n_j y_2 + y_1 + n_j x_i \\ [A^T A \mathbf{x}]_k &= \sum_{q \in C_k, p \in C_j} \sum_{e_t \in E(q, p)} x_t + \sum_{q \in C_k} \sum_{e_t \in E(q, i)} x_t = n_j y_2 + n_k y_1 + y_2 + n_k x_i \end{aligned}$$

For ease of readability, we define $P_i = [A^T \mathbf{d}]_i$ for all i with $1 \leq i \leq 2n - 3$. Then, equation 10.2 yields the following system of linear equations:

$$\begin{pmatrix} 1 & 1 & n_j + n_k \\ n_k + 1 & n_j & n_j \\ n_k & n_j + 1 & n_k \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ x_i \end{pmatrix} = \begin{pmatrix} P_i \\ P_j \\ P_k \end{pmatrix}$$

The matrix is invertible, and the last row of its inverse is

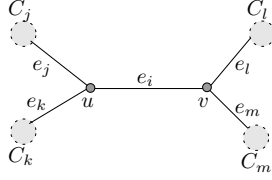
$$\frac{1}{4n_j n_k} \begin{pmatrix} (1 + n_j + n_k) & (1 + n_j - n_k) & (1 - n_j + n_k) \end{pmatrix}$$

This gives a closed formula for x_i :

$$\begin{aligned} x_i &= \frac{1}{4n_j n_k} \begin{pmatrix} (1 + n_j + n_k) & (1 + n_j - n_k) & (1 - n_j + n_k) \end{pmatrix} \begin{pmatrix} P_i \\ P_j \\ P_k \end{pmatrix} \\ &= \frac{1}{4n_j n_k} ((1 + n_j + n_k)P_i + (1 + n_j - n_k)P_j + (1 - n_j + n_k)P_k) \end{aligned}$$

In case e_i is an internal edge, we proceed in a similar fashion. In the situation of Figure 10.52, we define y_1 and y_2 as above as well as

$$y_3 = \sum_{p \in C_l} \sum_{e_t \in E(p, v)} x_t \quad \text{and} \quad y_4 = \sum_{q \in C_m} \sum_{e_t \in E(q, v)} x_t$$

Figure 10.52: e_i is an internal edge.

Let us express the respective components of $[A^T \mathbf{Ax}]$ in terms of the unknowns y_1, y_2, y_3 , and y_4 :

$$\begin{aligned}
 [A^T \mathbf{Ax}]_i &= \sum_{p \in C_j \cup C_k, q \in C_l \cup C_m} \sum_{e_t \in E(p,q)} x_t \\
 &= (n_l + n_m)y_1 + (n_l + n_m)y_2 + (n_j + n_k)y_3 + (n_j + n_k)y_4 \\
 &\quad + (n_j + n_k)(n_l + n_m)x_i \\
 [A^T \mathbf{Ax}]_j &= \sum_{p \in C_j, q \in C_k \cup C_l \cup C_m} \sum_{e_t \in E(p,q)} x_t \\
 &= (n_k + n_l + n_m)y_1 + n_j y_2 + n_j y_3 + n_j y_4 + n_j(n_l + n_m)x_i \\
 [A^T \mathbf{Ax}]_k &= \sum_{p \in C_k, q \in C_j \cup C_l \cup C_m} \sum_{e_t \in E(p,q)} x_t \\
 &= n_k y_1 + (n_j + n_l + n_m)y_2 + n_k y_3 + n_k y_4 + n_k(n_l + n_m)x_i \\
 [A^T \mathbf{Ax}]_l &= \sum_{p \in C_l, q \in C_j \cup C_k \cup C_m} \sum_{e_t \in E(p,q)} x_t \\
 &= n_l y_1 + n_l y_2 + (n_j + n_k + n_m)y_3 + n_l y_4 + n_l(n_j + n_k)x_i \\
 [A^T \mathbf{Ax}]_m &= \sum_{p \in C_m, q \in C_j \cup C_k \cup C_l} \sum_{e_t \in E(p,q)} x_t \\
 &= n_m y_1 + n_m y_2 + n_m y_3 + (n_j + n_k + n_l)y_4 + n_m(n_j + n_k)x_i
 \end{aligned}$$

Equation 10.2 yields the following system of linear equations:

$$B \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ x_i \end{pmatrix} = \begin{pmatrix} P_i \\ P_j \\ P_k \\ P_l \\ P_m \end{pmatrix}$$

where the matrix B has the shape

$$\begin{pmatrix} n_l + n_m & n_l + n_m & n_j + n_k & n_j + n_k & (n_j + n_k)(n_l + n_m) \\ n_k + n_l + n_m & n_j & n_j & n_j & n_j(n_l + n_m) \\ n_k & n_j + n_l + n_m & n_k & n_k & n_k(n_l + n_m) \\ n_l & n_l & n_j + n_k + n_m & n_l & n_l(n_j + n_k) \\ n_m & n_m & n_m & n_j + n_k + n_l & n_m(n_j + n_k) \end{pmatrix}$$

B is invertible and the last row of B^{-1} leads to the following closed formula:

$$x_i = \frac{1}{4(n_j + n_k)(n_l + n_m)} \left[\left(\frac{n}{n_j} + \frac{n}{n_k} + \frac{n}{n_l} + \frac{n}{n_m} - 4 \right) P_i \right. \\ \left. + \frac{n_j + n_k}{n_j n_k} ((2n_k - n)P_j + (2n_j - n)P_k) + \frac{n_l + n_m}{n_l n_m} ((2n_m - n)P_l + (2n_l - n)P_m) \right]$$

We claim that Algorithm 10.10 correctly computes edge weights that minimize the residual sum of squares and that it runs in $O(n^2)$ time.

Let us first prove the correctness of Algorithm 10.10. The computation of P_i for an external edge e_i in step (1) is obviously correct, and the number of leaves in the subtree below the edge e_i is apparently 1 (there is only the leaf i below e_i). When the algorithm reaches the internal edge e_i in the bottom-up traversal in step (3), the values P_j , N_j , P_k , and N_k have already been computed correctly. Lemma 10.6.16 shows that the computation of P_i is correct and the number of leaves below the edge e_i is obviously $N_i = N_j + N_k$ (the number of leaves below the edge e_j plus the number of leaves below the edge e_k). In step (4), the edge weights are computed correctly by the formulae derived above, noting that $n = 1 + |C_j| + |C_k| = 1 + |C_j| + N_k$ if e_i is an external edge and $n = |C_j| + |C_k| + |C_l| + |C_m| = |C_j| + N_k + N_l + N_m$ if e_i is an internal edge.

Lemma 10.6.16 *Let e_i be an internal edge in the binary phylogenetic tree T and let e_j and e_k be edges adjacent to the same endpoint of e_i . Let C_j , C_k , and C_i be the corresponding clusters (see Figure 10.52, noting that $C_i = C_l \cup C_m$). Then,*

$$P_i = P_j + P_k - 2 \sum_{p \in C_j, q \in C_k} d_{pq}$$

Proof This follows easily from the definition of P_i , P_j , and P_k :

$$\begin{aligned} P_i &= \sum_{p \in C_i, q \in \bar{C}_i} d_{pq} = \sum_{p \in C_i, q \in C_j \cup C_k} d_{pq} = \sum_{p \in C_i, q \in C_j} d_{pq} + \sum_{p \in C_i, q \in C_k} d_{pq} \\ P_j &= \sum_{p \in C_j, q \in \bar{C}_j} d_{pq} = \sum_{p \in C_j, q \in C_i \cup C_k} d_{pq} = \sum_{p \in C_j, q \in C_i} d_{pq} + \sum_{p \in C_j, q \in C_k} d_{pq} \\ P_k &= \sum_{p \in C_k, q \in \bar{C}_k} d_{pq} = \sum_{p \in C_k, q \in C_i \cup C_j} d_{pq} = \sum_{p \in C_k, q \in C_i} d_{pq} + \sum_{p \in C_k, q \in C_j} d_{pq} \end{aligned}$$

□

We still have to show that Algorithm 10.10 runs in $O(n^2)$ time. Step (1) takes $O(n^2)$ time because there are n external edges and for every external edge e_i the computation of P_i takes $O(n)$ time. Step (2) is negligible. At first sight, it seems that step (3) requires $O(n^3)$ time but an amortized analysis shows that this is not the case. In the computation of *all* internal

Algorithm 10.10 Bryant and Waddell's algorithm.

Input: An $n \times n$ dissimilarity matrix $D = (d_{ij})$ and a binary phylogenetic tree T for the set $S = \{1, \dots, n\}$ of taxa.

1. For each external edge e_i leading to leaf (taxon) i , compute

$$P_i = \sum_{q \in S} d_{iq}$$

and set $N_i = 1$.

2. Root the tree T at an arbitrary internal edge $e = (u, v)$, i.e., split the edge e into two edges (r, u) and (r, v) , where r is a new root node with child nodes u and v .
3. In a bottom-up traversal of the rooted tree, compute the values P_i and N_i for each internal edge e_i by

$$P_i = P_j + P_k - 2 \sum_{p \in C_j, q \in C_k} d_{pq}$$

and $N_i = N_j + N_k$, where e_j and e_k are the child edges of e_i . (For the "root"-edge $e = (u, v)$, these values can be computed either using the values of the edges directly below node u or the values of the edges directly below node v .)

4. In a second traversal of the rooted tree, compute the edge weight x_i of each edge e_i as follows:
 - If e_i is an external edge with parent edge e_j and sibling edge e_k , then set $n_j = n - (N_k + 1)$, $n_k = N_k$, and

$$x_i = \frac{1}{4n_j n_k} ((1 + n_j + n_k)P_i + (1 + n_j - n_k)P_j + (1 - n_j + n_k)P_k)$$

- If e_i is an internal edge with parent edge e_j , sibling edge e_k , and child edges e_l and e_m , then set $n_j = n - (N_k + N_l + N_m)$, $n_k = N_k$, $n_l = N_l$, $n_m = N_m$, and

$$\begin{aligned} x_i &= \frac{1}{4(n_j + n_k)(n_l + n_m)} \left[\left(\frac{n}{n_j} + \frac{n}{n_k} + \frac{n}{n_l} + \frac{n}{n_m} - 4 \right) P_i \right. \\ &\quad + \frac{n_j + n_k}{n_j n_k} ((2n_k - n)P_j + (2n_j - n)P_k) \\ &\quad \left. + \frac{n_l + n_m}{n_l n_m} ((2n_m - n)P_l + (2n_l - n)P_m) \right] \end{aligned}$$

Output: The weighted tree T .

edge weights, each distance d_{ij} , $1 \leq i < j \leq n$, is added at most *once*. Consequently, the total number of additions in step (3) is in $O(n^2)$. Clearly, step (4) requires only $O(n)$ time as there are $2n - 3$ edges and the weight of each edge can be computed in constant time with the help of the closed formulae. All in all, Algorithm 10.10 has a worst-case time complexity of $O(n^2)$.

10.6.3 Bootstrapping

Bootstrapping is a statistical technique to test the reliability or robustness of a tree T under variations of the data.

To test the reliability of a tree T produced by a certain reconstruction method from a multiple alignment A of n sequences with m columns, we draw k samples from the data and build a phylogenetic tree for each sample i with $1 \leq i \leq k$. More precisely, proceed as follows:

- Randomly draw a sample *with replacement* of size m from the columns of A ; the resulting pseudo-alignment A_i is called a *bootstrap replicate*.
- Compute the dissimilarity matrix D_i based on A_i .
- Use the same method to construct a phylogenetic tree T_i for D_i .
- Compute the set of splits induced by T_i .

For each edge e in T and each tree T_i , let $I(e, T_i) = 1$ if the split induced by e in T also occurs in the set of splits induced by T_i . Otherwise, let $I(e, T_i) = 0$. The bootstrap value of an edge e of T is defined by

$$\frac{\sum_{i=1}^k I(e, T_i)}{k}$$

Similarly, the bootstrap value of a node in T can be defined.

A large bootstrap value (95% or higher) of an edge e significantly supports the hypothesis that e is also present in the “real” (unknown) tree, while a small bootstrap value indicates that e is less reliable. See e.g. [97] for more details.

Bibliography

- [1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.
- [2] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Enhanced suffix arrays and applications. In S. Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 7. Chapman & Hall/CRC Computer and Information Science Series, 2006.
- [3] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. CoCoNUT: An efficient system for the comparison and analysis of genomes. *BMC Bioinformatics*, 9:476, 2008.
- [4] M.I. Abouelhoda and E. Ohlebusch. A local chaining algorithm and its applications in comparative genomics. In *Proc. 3rd International Workshop on Algorithms in Bioinformatics*, volume 2812 of *Lecture Notes in Bioinformatics*, pages 1–16. Springer-Verlag, 2003.
- [5] M.I. Abouelhoda and E. Ohlebusch. Multiple genome alignment: Chaining algorithms revisited. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2003.
- [6] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer-Verlag, 2008.
- [7] R. Agarwala, V. Bafna, M. Farach, M. Paterson, and M. Thorup. On the approximability of numerical taxonomy (fitting distances by tree metrics). *SIAM Journal on Computing*, 28(3):1073–1085, 1999.
- [8] A.V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

- [9] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. On finding lowest common ancestor in trees. *SIAM Journal on Computing*, 5(1):115–132, 1976.
- [10] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [11] M. Akra and L. Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10(2):195–210, 1998.
- [12] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proc. 14th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 258–264, 2002.
- [13] S. Aluru and P. Ko. Lookup tables, suffix trees and suffix arrays. In S. Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 5. Chapman & Hall/CRC Computer and Information Science Series, 2006.
- [14] S. Anderson, A.T. Bankier, B.G. Barrell, M.H. de Bruijn, A.R. Coulson, J. Drouin, I.C. Eperon, D.P. Nierlich, B.A. Roe, F. Sanger, P.H. Schreier, A.J. Smith, R. Staden, and I.G. Young. Sequence and organization of the human mitochondrial genome. *Nature*, 290(5806):457–465, 1981.
- [15] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer-Verlag, 1985.
- [16] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Doklady Akademii Nauk SSSR*, 194:487–488, 1970. In Russian.
- [17] M. Arnold. *Effiziente Algorithmen zur Suche von längsten gemeinsamen Teilstrings und Repeats*. Diploma thesis (in German), University of Ulm, Germany, 2008.
- [18] M. Arnold and E. Ohlebusch. Linear time algorithms for generalizations of the longest common substring problem. *Algorithmica*, 60(4):806–818, 2011.
- [19] K. Atteson. The performance of neighbor-joining methods of phylogenetic reconstruction. *Algorithmica*, 25(2-3):251–278, 1999.
- [20] D.A. Bader, B.M.E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with

- an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.
- [21] M. Bader. The transposition median problem is NP-complete. *Theoretical Computer Science*, 412:1099–1110, 2011.
 - [22] M. Bader and E. Ohlebusch. Sorting by weighted reversals, transpositions, and inverted transpositions. *Journal of Computational Biology*, 14:615–636, 2007.
 - [23] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate string matching. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 185–192. Springer-Verlag, 1992.
 - [24] V. Bafna and P.A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.
 - [25] V. Bafna and P.A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998.
 - [26] B.S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
 - [27] A.C. Barbrook, C.J. Howe, N. Blake, and P. Robinson. The phylogeny of The Canterbury Tales. *Nature*, 394:839, 1998.
 - [28] J.-P. Barthélemy and A. Guénoche. *Trees and proximity representations*. John Wiley and Sons Inc., New York, 1991.
 - [29] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander. Human and mouse gene structure: Comparative analysis and application to exon prediction. *Genome Research*, 10:950–958, 2000.
 - [30] M.J. Bauer, A.J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 219–231. Springer-Verlag, 2011.
 - [31] V. Becher, A. Deymonnaz, and P. Heiber. Efficient computation of all perfect repeats in genomic sequences of up to half a gigabyte, with a case study on the human genome. *Bioinformatics*, 25(14):1746–1753, 2009.
 - [32] T. Beller, K. Berger, and E. Ohlebusch. Space-efficient computation of maximal and supermaximal repeats in genome sequences. In *Proc. 19th International Symposium on String Processing and Information Retrieval*, volume 7608 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 2012.

- [33] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013.
- [34] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
- [35] M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. Latin American Theoretical INformatics*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer-Verlag, 2000.
- [36] J.L. Bentley, D.D. Sleator, R.E. Tarjan, and V.K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.
- [37] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. *Discrete Applied Mathematics*, 146(2):134–145, 2005.
- [38] A. Bergeron, J. Mixtacki, and J. Stoye. The inversion distance problem. In O. Gascuel, editor, *Mathematics of Evolution and Phylogeny*, chapter 10, pages 262–290. Oxford University Press, 2005.
- [39] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.
- [40] J.L. Boore. Animal mitochondrial genomes. *Nucleic Acids Research*, 27(8):1767–1780, 1999.
- [41] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.
- [42] N. Bray, I. Dubchak, and L. Pachter. AVID: A global alignment program. *Genome Research*, 13:97–102, 2003.
- [43] M. Brudno, C.B. Do, G.M. Cooper, M.F. Kim, E.D. Davydov, NISC Comparative Sequencing Program, E.D. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic DNA. *Genome Research*, 13(4):721–731, 2003.
- [44] D. Bryant. *Building Trees, Hunting for Trees, and Comparing Trees—Theory and Methods in Phylogenetic Analysis*. PhD thesis, University of Canterbury, New Zealand, 1997.
- [45] D. Bryant and P. Waddell. Rapid evaluation of least-squares and minimum-evolution criteria on phylogenetic trees. *Molecular Biology and Evolution*, 15(10):1346–1359, 1998.

- [46] L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *SIAM Journal on Discrete Mathematics*, 26(3):1148–1180, 2012.
- [47] P. Buneman. A note on metric properties of trees. *Journal of Combinatorial Theory*, 17(B):48–50, 1974.
- [48] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, Digital Systems Research Center, 1994.
- [49] R.L. Cann, M. Stoneking, and A.C. Wilson. Mitochondrial DNA and human evolution. *Nature*, 325:31–36, 1987.
- [50] A. Caprara. The reversal median problem. *INFORMS Journal on Computing*, 15(1):93–113, 2003.
- [51] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Mathematics*, 48(5):1073–1082, 1988.
- [52] L.L. Cavalli-Sforza and A.W.F. Edwards. Phylogenetic analysis: Models and estimation procedures. *American Journal of Human Genetics*, 19:233–257, 1967.
- [53] P. Chain, S. Kurtz, E. Ohlebusch, and T. Slezak. An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105–123, 2003.
- [54] W.I. Chang and E.L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.
- [55] K.-M. Chao and W. Miller. Linear-space algorithms that build local alignments from fragments. *Algorithmica*, 13(1-2):106–134, 1995.
- [56] G. Chen, S.J. Puglisi, and W.F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2008.
- [57] Chimpanzee Sequencing and Analysis Consortium. Initial sequence of the chimpanzee genome and comparison with the human genome. *Nature*, 437:69–87, 2005.
- [58] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
- [59] P. Clote and R. Backofen. *Computational Molecular Biology*. John Wiley and Sons Inc., New York, 2000.

- [60] A. Coppa, R. Grün, C.B. Stringer, S. Eggins, and R. Vargiu. Newly recognized Pleistocene human teeth from Tabun Cave, Israel. *Journal of Human Evolution*, 49(3):301–315, 2005.
- [61] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
- [62] F. Crick. Central dogma of molecular biology. *Nature*, 227(5258):561–563, 1970.
- [63] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, 2000.
- [64] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12:244–250, 1981.
- [65] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.
- [66] M. Crochemore, L. Ilie, C.S. Iliopoulos, M. Kubica, W. Rytter, and T. Waleń. LPF computation revisited. In *Proc. 20th International Workshop on Combinatorial Algorithms*, volume 5874 of *Lecture Notes in Computer Science*, pages 158–169. Springer-Verlag, 2009.
- [67] M. Crochemore, L. Ilie, and W.F. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *Proc. 18th Data Compression Conference*, pages 482–488. IEEE Computer Society, 2008.
- [68] M. Crochemore, L. Ilie, and L. Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011.
- [69] M. Crochemore, C.S. Iliopoulos, M. Kubica, M.S. Rahman, and T. Waleń. Improved algorithms for the range next value problem and applications. In *Proc. 25th Symposium on Theoretical Aspects of Computer Science*, pages 205–216. IBFI Schloss Dagstuhl, 2008.
- [70] M. Crochemore, C.S. Iliopoulos, and M.S. Rahman. Optimal prefix and suffix queries on texts. *Information Processing Letters*, 108(5):320–325, 2008.
- [71] M. Crochemore, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. On the maximal sum of exponents of runs in a string. *Journal of Discrete Algorithms*, 14:29–36, 2012.
- [72] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

- [73] J.S. Culpepper, G. Navarro, S.J. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *Proc. 18th Annual European Symposium on Algorithms*, volume 6347 of *Lecture Notes in Computer Science*, pages 194–205. Springer-Verlag, 2010.
- [74] W.H.E. Day. Computational complexity of inferring phylogenies from dissimilarity matrices. *Bulletin of Mathematical Biology*, 49(4):461–467, 1987.
- [75] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt. A model of evolutionary change in proteins. In M.O. Dayhoff, editor, *Atlas of Protein Sequence and Structure*, volume 5, pages 345–358. National Biomedical Research Foundation, 1978.
- [76] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, third edition, 2008.
- [77] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.
- [78] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.
- [79] J. Dhaliwal, S.J. Puglisi, and A. Turpin. Practical efficient string mining. *IEEE Transactions on Knowledge and Data Engineering*, 24(4):735–744, 2012.
- [80] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [81] T. Dobzhansky and A.H. Sturtevant. Inversions in the chromosomes of *Drosophila pseudoobscura*. *Genetics*, 23:28–64, 1938.
- [82] M. Domazet-Lošo and B. Haubold. Efficient estimation of pairwise distances between genomes. *Bioinformatics*, 25(24):3221–3227, 2009.
- [83] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.
- [84] A. Ehrenfeucht and D. Haussler. A new distance metric on strings computable in linear time. *Discrete Applied Mathematics*, 20(3):191–203, 1988.

- [85] O. Elemento and O. Gascuel. An efficient and accurate distance based algorithm to reconstruct tandem duplication trees. *Bioinformatics*, 18:S92–S99, 2002.
- [86] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):369–379, 2006.
- [87] I. Elias and J. Lagergren. Fast neighbor joining. *Theoretical Computer Science*, 410(21-23):1993–2000, 2009.
- [88] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.
- [89] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *Journal of Experimental Algorithmics*, 5:1–23, 2000.
- [90] D. Eppstein, Z. Galil, R. Giancarlo, and G.F. Italiano. Sparse dynamic programming. I: Linear cost functions; II: Convex and concave cost functions. *Journal of the ACM*, 39(3):519–567, 1992.
- [91] N. Eriksen. $(1 + \epsilon)$ -approximation of sorting by reversals and transpositions. *Theoretical Computer Science*, 289(1):517–529, 2002.
- [92] A. Eriksson and A. Manica. Effect of ancient population structure on the degree of polymorphism shared between modern human populations and ancient hominins. *Proc. National Academy of Science USA*, 109(35):13956–13960, 2012.
- [93] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.
- [94] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.
- [95] J.S. Farris. On the phenetic approach to vertebrate classification. In *Major Patterns in Vertebrate Evolution*, pages 823–850. Plenum, New York, 1977.
- [96] J.S. Farris, A.G. Kluge, and M.J. Eckardt. A numerical approach to phylogenetic systematics. *Systematic Zoology*, pages 172–189, 1970.
- [97] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004.

- [98] D. Feng and R. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360, 1987.
- [99] P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *Lecture Notes in Computer Science*, pages 697–710. Springer-Verlag, 2010.
- [100] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science*, pages 390–398, 2000.
- [101] G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of genome rearrangements*. MIT Press, 2009.
- [102] N.J. Fine and H.S. Wilf. Uniqueness theorem for periodic functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.
- [103] R.A. Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
- [104] J. Fischer. *Efficient Data Structures for String Algorithms*. PhD thesis, LMU München, Germany, 2007.
- [105] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. 9th Latin American Theoretical Informatics Symposium*, volume 6034 of *Lecture Notes in Computer Science*, pages 158–169, Berlin, 2010. Springer-Verlag.
- [106] J. Fischer. Combined data structure for previous- and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.
- [107] J. Fischer. Inducing the LCP-array. In *Proc. 12th International Symposium on Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer-Verlag, 2011.
- [108] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer-Verlag, 2006.
- [109] J. Fischer, V. Heun, and S. Kramer. Optimal string mining under frequency constraints. In *Proc. 10th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 4213 of *Lecture Notes in Computer Science*, pages 139–150. Springer-Verlag, 2006.

- [110] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.
- [111] J. Fischer, V. Mäkinen, and N. Välimäki. Space efficient string mining under frequency constraints. In *Proc. 8th IEEE International Conference on Data Mining*, pages 193–202. IEEE Computer Society, 2008.
- [112] W.M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155:279–284, 1967.
- [113] P. Flick and E. Birney. Sense from sequence reads: Methods for alignment and assembly. *Nature Methods*, 6(11 Suppl.):S6–S12, 2009.
- [114] F. Franěk, W.F. Smyth, and Y. Tang. Computing all repeats using suffix arrays. *Journal of Automata, Languages and Combinatorics*, 8(4):579–591, 2003.
- [115] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.
- [116] A. Fürstberger. *Rekonstruktion phylogenetischer Bäume*. Diploma thesis (in German), University of Ulm, Germany, 2007.
- [117] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 135–143. ACM Press, 1984.
- [118] T. Gagie, S.J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. 16th International Symposium on String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 1–6. Springer-Verlag, 2009.
- [119] R. Garesse. *Drosophila melanogaster* mitochondrial DNA: Gene organization and evolutionary considerations. *Genetics*, 118(4):649–663, 1988.
- [120] O. Gascuel. A note on Sattath and Tversky’s, Saitou and Nei’s, and Studier and Keppler’s algorithms for inferring phylogenies from evolutionary distances. *Molecular Biology and Evolution*, 11(6):961–963, 1994.
- [121] R.F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. *ACM Transactions on Algorithms*, 2(4):510–534, 2006.

- [122] R. Giegerich, M. Carsten, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.
- [123] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.
- [124] S. Gog. *Compressed Suffix Trees: Design, Construction, and Applications*. PhD thesis, University of Ulm, Germany, 2011.
- [125] S. Gog and J. Fischer. Advantages of shared data structures for sequences of balanced parentheses. In *Proc. 20th Data Compression Conference*, pages 406–415. IEEE Computer Society, 2010.
- [126] S. Gog and E. Ohlebusch. Compressed suffix trees: Efficient computation and storage of LCP-values. *Journal of Experimental Algorithmics*, 2013.
- [127] A. Golynski, J.I. Munro, and S.S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.
- [128] G. Gonnella and S. Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13:82, 2012.
- [129] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, Englewood Cliffs, NJ, 1992.
- [130] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.
- [131] R.E. Green et al. A draft sequence of the Neandertal genome. *Science*, 328:710–722, 2010.
- [132] I. Gronau and S. Moran. Neighbor joining algorithms for inferring phylogenies via LCA distances. *Journal of Computational Biology*, 14(1):1–15, 2007.
- [133] I. Gronau and S. Moran. Optimal implementations of UPGMA and other common clustering algorithms. *Information Processing Letters*, 104(6):205–210, 2007.
- [134] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

- [135] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. An earlier version of this article was presented at the *Symposium on the Theory of Computing*, 2000.
- [136] Q.-P. Gu, S. Peng, and I.H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 1999.
- [137] S.K. Gupta, J.D. Kececiloglu, and A.A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):459–472, 1995.
- [138] D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55(1):141–154, 1993.
- [139] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
- [140] D. Gusfield, G.M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992.
- [141] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004. An earlier version of this article appeared as Report CSE-98-4, University of California, Davis, 1998.
- [142] B.J. Haas and S.L. Salzberg. Finding repeats in genome sequences. In T. Lengauer, editor, *Bioinformatics — From Genomes to Therapies, Volume 1: Molecular Sequences and Structures*, chapter 7. Wiley-VCH Verlag, 2007.
- [143] Y. Han. Improving the efficiency of sorting by reversals. In *Proc. International Conference on Bioinformatics and Computational Biology*, pages 406–409. CSREA Press, 2006.
- [144] S. Hannenhalli and P.A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proc. 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 581–592, 1995.

- [145] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 48:1–27, 1999. An earlier version of this article was presented at the *Symposium on the Theory of Computing*, 1995.
- [146] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13:338–355, 1984.
- [147] T. Hartman and R. Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. *Journal of Computer and System Sciences*, 70(3):300–320, 2005.
- [148] B. Haubold, N. Pierstorff, F. Möller, and T. Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC Bioinformatics*, 6:123, 2005.
- [149] S. Henikoff and J.G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. National Academy of Science USA*, 89(22):10915–10919, 1992.
- [150] V. Heun. Analysis of a modification of Gusfield’s recursive algorithm for reconstructing ultrametric trees. *Information Processing Letters*, 108(4):222–225, 2008.
- [151] V. Heun. Skriptum zur Vorlesung Algorithmische Bioinformatik: Bäume und Graphen (in German), 2011. <http://www.bio.ifi.lmu.de/~heun/lecturenotes/>.
- [152] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
- [153] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18:S312–S320, 2002.
- [154] W.K. Hon, K. Sadakane, and W.K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 251–260, 2003.
- [155] W.K. Hon, R. Shah, S.V. Thankachan, and J.S. Vitter. String retrieval for multi-pattern queries. In *Proc. 17th International Symposium on String Processing and Information Retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 55–66. Springer-Verlag, 2010.
- [156] R.N. Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, 1980.

- [157] T.C. Hu and A.C. Tucker. Optimal computer search trees and variable-length alphabetical codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.
- [158] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.
- [159] L.C.K. Hui. Color set size problem with applications to string matching. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer-Verlag, 1992.
- [160] M. Ingman, H. Kaessmann, S. Pääbo, and U. Gyllensten. Mitochondrial genome variation and the origin of modern humans. *Nature*, 408:708–713, 2000.
- [161] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409:860–921, 2001.
- [162] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1988.
- [163] G. Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.
- [164] G. Jacobson and K.-P. Vo. Heaviest increasing/common subsequence problems. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 1992.
- [165] J.-E. Jeon, H. Park, and D.-K. Kim. Efficient construction of generalized suffix arrays by merging suffix arrays. *Journal of KISS: Computer Systems and Theory*, 32(6):268–278, 2005.
- [166] D. Johanson and B. Edgar. *From Lucy to language*. Simon & Schuster, 2006.
- [167] K.S. John, T. Warnow, B. Moret, and L. Vawter. Performance study of phylogenetic methods: (unweighted) quartet methods and neighbor joining. *Journal of Algorithms*, 48:174–193, 2003.
- [168] D.B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory*, 15:295–309, 1982.

- [169] D. Joseph, J. Meidanis, and P. Tiwari. Determining DNA sequence similarity using maximum independent set algorithms for interval graphs. In *Proc. 3rd Scandinavian Workshop on Algorithm Theory*, volume 621 of *Lecture Notes in Computer Science*, pages 326–337. Springer-Verlag, 1992.
- [170] T.H. Jukes and C.R. Cantor. Evolution of protein molecules. In *Mammalian Protein Metabolism*, page 21–132. Academic Press, New York, 1969.
- [171] W. Just. Computational complexity of multiple sequence alignment with SP-score. *Journal of Computational Biology*, 8(6):615–623, 2001.
- [172] H. Kaplan, R. Shamir, and R.E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 344–351, 1997.
- [173] J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.
- [174] J. Kärkkäinen, G. Manzini, and S.J. Puglisi. Permuted longest-common-prefix array. In *Proc. 20th Annual Symposium on Combinatorial Pattern Matching*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer-Verlag, 2009.
- [175] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer-Verlag, 2003.
- [176] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Computing and Combinatorics Conference*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer-Verlag, 1996.
- [177] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer-Verlag, 2001.
- [178] J. Kececioğlu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13:180–210, 1995.

- [179] Z. Khan, J. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence data sets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.
- [180] D.K. Kim, J.S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199. Springer-Verlag, 2003.
- [181] M. Kimura. A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *Journal of Molecular Evolution*, 16(2):111–120, 1980.
- [182] D.E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, 2011.
- [183] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
- [184] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, 2003.
- [185] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 596–604, 1999.
- [186] R. Kolpakov and G. Kucherov. On maximal repetitions in words. *Journal of Discrete Algorithms*, 1:159–186, 2000.
- [187] J. Krause, Q. Fu, J.M. Good, B. Viola, M.V. Shunkov, A.P. Derevianko, and S. Pääbo. The complete mitochondrial DNA genome of an unknown hominin from southern Siberia. *Nature*, 464:894–897, 2010.
- [188] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *Proc. 20th Data Compression Conference*, pages 239–248. IEEE Computer Society, 2010.
- [189] M. Krings, A. Stone, R.W. Schmitz, H. Krainitzki, M. Stoneking, and S. Pääbo. Neandertal DNA sequences and the origin of modern humans. *Cell*, 90(1):19–30, 1997.
- [190] A. Kügel and E. Ohlebusch. A space efficient solution to the frequent string mining problem for many databases. *Data Mining and Knowledge Discovery Journal*, 17(1):24–38, 2008.

- [191] M.O. Külekci, J.S. Vitter, and B. Xu. Efficient maximal repeat finding using the Burrows-Wheeler transform and wavelet tree. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(2):421–429, 2012.
- [192] S. Kumar and S.R. Gadagker. Efficiency of the neighbor-joining method in reconstructing evolutionary relationships in large phylogenies. *Journal of Molecular Evolution*, 51(6):544–553, 2000.
- [193] S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.
- [194] S. Kurtz. *Foundations of Sequence Analysis*. Lecture notes for a course in the summer semester 2003, Center for Bioinformatics, University of Hamburg, Germany, 2003.
- [195] S. Kurtz, J.V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.
- [196] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12), 2004.
- [197] T.-W. Lam, R. Li, A. Tam, S. Wong, E. Wu, and S.-M. Yiu. High throughput short read alignment via bi-directional BWT. In *Proc. International Conference on Bioinformatics and Biomedicine*, pages 31–36. IEEE Computer Society, 2009.
- [198] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(R25), 2009.
- [199] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.
- [200] M. Lermen and K. Reinert. The practical use of the A^* algorithm for exact multiple sequence alignment. *Journal of Computational Biology*, 7(5):655–671, 2000.
- [201] C. Leslie, E. Eskin, and W.S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Proc. 7th Pacific Symposium on Biocomputing*, pages 566–575, 2002.
- [202] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler Transform. *Bioinformatics*, 25(14):1754–1760, 2009.

- [203] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1977, 2009.
- [204] G.-H. Lin and G. Xue. Signed genome rearrangement by reversals and transpositions: Models and approximations. *Theoretical Computer Science*, 259(1-2):513–531, 2001.
- [205] D.J. Lipman, S.F. Altschul, and J.D. Kececioglu. A tool for multiple sequence alignment. *Proc. National Academy of Science USA*, 86(12):4412–4415, 1989.
- [206] R.A. Lippert, C.M. Mobarry, and B. Walenz. A space-efficient construction of the Burrows-Wheeler transforms for genomic data. *Journal of Computational Biology*, 12(7):943–951, 2005.
- [207] D.P. Locke et al. Comparative and demographic analysis of orangutan genomes. *Nature*, 469:529–533, 2011.
- [208] M.G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.
- [209] M.G. Maaß. Computing suffix links for suffix trees and arrays. *Information Processing Letters*, 101(6):250–254, 2007.
- [210] M.G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.
- [211] M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [212] V. Mäkinen and G. Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical report C-2004-20, University of Helsinki, 2004.
- [213] V. Mäkinen and G. Navarro. Position-restricted substring searching. In *Proc. Latin American Theoretical INformatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer-Verlag, 2006.
- [214] U. Manber and E.W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- [215] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

- [216] G. Manzini. Two space saving tricks for linear time LCP array computation. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer-Verlag, 2004.
- [217] G. Mauri and G. Pavesi. Pattern discovery in RNA secondary structure using affix trees. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 278–294. Springer-Verlag, 2003.
- [218] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
- [219] E.M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.
- [220] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990.
- [221] F. Meyer, S. Kurtz, R. Backofen, S. Will, and M. Beckstette. Structator: fast index-based search for RNA sequence-structure patterns. *BMC Bioinformatics*, 12:214, 2011.
- [222] R. Mihaescu, D. Levy, and L. Pachter. Why neighbor-joining works. *Algorithmica*, 54:1–24, 2009.
- [223] W. Miller, K.D. Makova, A. Nekrutenko, and R. Hardison. Comparative genomics. *Annual Review of Genomics and Human Genetics*, 5:15–56, 2004.
- [224] S.E. Mitchell, A.F. Cockburn, and J.A. Seawright. The mitochondrial genome of *Anopheles quadrimaculatus* species A: complete nucleotide sequence and gene organization. *Genome*, 36(6):1058–1073, 1993.
- [225] B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998.
- [226] D.R. Morrison. PATRICIA—Practical Algorithm to Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
- [227] D.W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, New York, 2001.

- [228] J.I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer-Verlag, 1996.
- [229] J.I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees, and planar graphs. In *Proc. 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 118–126, 1997.
- [230] J.I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
- [231] J.I. Munro, V. Raman, and S.S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
- [232] F. Murtagh. Complexities of hierarchic clustering algorithms: State of the art. *Computational Statistics Quarterly*, 1(2):101–113, 1984.
- [233] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666, 2002.
- [234] J.H. Nadeau and B.A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proc. National Academy of Science USA*, 81(3):814–818, 1984.
- [235] K. Narisawa, S. Inenaga, H. Bannai, and M. Takeda. Efficient computation of substring equivalence classes with suffix arrays. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching*, volume 4580 of *Lecture Notes in Computer Science*, pages 340–351. Springer-Verlag, 2007.
- [236] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
- [237] G. Navarro. Wavelet trees for all. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 2–26. Springer-Verlag, 2012.
- [238] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
- [239] G. Navarro and V. Mäkinen. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):Article 32, 2008.

- [240] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.
- [241] M. Nei. *Molecular Evolutionary Genetics*. Columbia University Press, New York, 1987.
- [242] M. Nei and S. Kumar. *Molecular Evolution and Phylogenetics*. Oxford University Press, New York, 2000.
- [243] G. Nong. An optimal suffix array construction algorithm. Technical report, Department of Computer Science, Sun Yat-sen University, China, 2011.
- [244] G. Nong, S. Zhang, and W.H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proc. Data Compression Conference*, pages 193–202. IEEE Computer Society, 2009.
- [245] E. Ohlebusch and M.I. Abouelhoda. Chaining algorithms and applications in comparative genomics. In S. Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 15. Chapman & Hall/CRC Computer and Information Science Series, 2006.
- [246] E. Ohlebusch, T. Beller, and M.I. Abouelhoda. Computing the Burrows-Wheeler transform of a string and its reverse. In *Proc. 23rd Annual Symposium on Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 243–256. Springer-Verlag, 2012.
- [247] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. 17th International Symposium on String Processing and Information Retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 322–333. Springer-Verlag, 2010.
- [248] E. Ohlebusch and S. Gog. A compressed enhanced suffix array supporting fast string matching. In *Proc. 16th International Symposium on String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 51–62. Springer-Verlag, 2009.
- [249] E. Ohlebusch and S. Gog. Efficient algorithms for the all pairs suffix-prefix problem and the all pairs substring-prefix problem. *Information Processing Letters*, 110(3):123–128, 2010.
- [250] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 15–26. Springer-Verlag, 2011.

- [251] E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. 17th International Symposium on String Processing and Information Retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 347–358. Springer-Verlag, 2010.
- [252] E. Ohlebusch and S. Kurtz. Space efficient computation of rare maximal exact matches between multiple sequences. *Journal of Computational Biology*, 15(4):357–377, 2008.
- [253] D. Okanohara and K. Sadakane. An online algorithm for finding the longest previous factors. In *Proc. 16th Annual European Symposium on Algorithms*, volume 5193 of *Lecture Notes in Computer Science*, pages 696–707. Springer-Verlag, 2008.
- [254] D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *Proc. 16th International Symposium on String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer-Verlag, 2009.
- [255] C. O’Keefe and E. Eichler. The pathological consequences and evolutionary implications of recent human genomic duplications. In *Comparative Genomics*, pages 29–46. Kluwer Press, 2000.
- [256] M. Ozery-Flato and R. Shamir. Two notes on genome rearrangement. *Journal of Bioinformatics and Computational Biology*, 1(1):71–94, 2003.
- [257] I. Pe’er and R. Shamir. The median problems for breakpoints are NP-complete. Technical Report TR98-071, Electronic Colloquium on Computational Complexity, 1998.
- [258] P. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.
- [259] P. Pevzner and G. Tesler. Genome rearrangements in mammalian evolution: Lessons from human and mouse genomic sequences. *Genome Research*, 13:37–45, 2003.
- [260] E. Prieur and T. Lecroq. On-line construction of compact suffix vectors and maximal repeats. *Theoretical Computer Science*, 407(1-3):290–301, 2008.
- [261] K. Prüfer et al. The bonobo genome compared with the chimpanzee and human genomes. *Nature*, 486:527–531, 2012.

- [262] S.J. Puglisi, W.F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):Article 4, 2007.
- [263] S.J. Puglisi, W.F. Smyth, and M. Yusufu. Fast, practical algorithms for computing all the repeats in a string. *Mathematics in Computer Science*, 3(4):373–389, 2010.
- [264] S.J. Puglisi and A. Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proc. 19th International Symposium on Algorithms and Computation*, volume 5369 of *Lecture Notes in Computer Science*, pages 124–135. Springer-Verlag, 2008.
- [265] M. Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165–169, 2001.
- [266] D. Reich et al. Genetic history of an archaic hominin group from Denisova Cave in Siberia. *Nature*, 468:1053–1060, 2010.
- [267] J. Rissanen and G.G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.
- [268] F.J. Rohlf. *A Numerical Taxonomic Study of the Genus Aedes (Diptera: Culicidae) with Emphasis on the Congruence of Larval and Adult Classifications*. PhD thesis, University of Kansas, 1962.
- [269] B. Ryabko. Technical correspondence on "A locally adaptive data compression scheme". *Communications of the ACM*, 30(9):792, 1987.
- [270] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th International Symposium on Algorithms and Computation*, volume 1969 of *Lecture Notes in Computer Science*, pages 410–421. Springer-Verlag, 2000.
- [271] K. Sadakane. Succinct representations of *lcp* information and improvements in the compressed suffix arrays. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 225–232, 2002.
- [272] K. Sadakane. New text indexing functionality of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
- [273] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41:589–607, 2007.

- [274] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
- [275] K. Sadakane and Navarro G. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 134–149, 2010.
- [276] N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4:406–425, 1987.
- [277] S.L. Salzberg and J.A. Yorke. Beware of mis-assembled genomes. *Bioinformatics*, 21(24):4320–4321, 2005.
- [278] D. Sankoff. Edit distance for genome comparison based on non-local operations. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching, 3rd Annual Symposium*, volume 644 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 1992.
- [279] S. Sattath and A. Tversky. Additive similarity trees. *Psychometrika*, 42(3):319–345, 1977.
- [280] M. Sauerhoff. Manuscript on neighbor-joining (in German), 2004. http://ls2-www.cs.uni-dortmund.de/~sauerhof/neighbor_joining.pdf.
- [281] G. Sauthoff, M. Möhl, S. Janssen, and R. Giegerich. Bellman’s GAP—a language and compiler for dynamic programming in sequence analysis. *Bioinformatics*, 29(5):551–560, 2013.
- [282] A. Scally et al. Insights into hominid evolution from the gorilla genome sequence. *Nature*, 483:169–175, 2012.
- [283] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
- [284] T. Schnattinger, E. Ohlebusch, and S. Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012.
- [285] B. Schölkopf and A.J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.
- [286] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001. In German.

- [287] S. Schwartz, W.J. Kent, A. Smit, Z. Zhang, R. Baertsch, R.C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Research*, 13:103–107, 2003.
- [288] S. Schwartz, Z. Zhang, K.A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. PipMaker—a web server for aligning two genomic DNA sequences. *Genome Research*, 10(4):577–586, 2000.
- [289] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing, Boston, MA, 1997.
- [290] X. Shen, X. Ma, J. Ren, and F. Zhao. A close phylogenetic relationship between Sipuncula and Annelida evidenced from the complete mitochondrial genome sequence of *Phascolosoma esculenta*. *BMC Genomics*, 10:136, 2009.
- [291] J.T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.
- [292] J.T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22:549–556, 2012.
- [293] J. Sirén. Compressed suffix arrays for massive data. In *Proc. 16th International Symposium on String Processing and Information Retrieval*, volume 5721 of *Lecture Notes in Computer Science*, pages 63–74. Springer-Verlag, 2009.
- [294] T.F. Smith, M.S. Waterman, and W.M. Fitch. Comparative biosequence metrics. *Journal of Molecular Evolution*, 18:38–46, 1981.
- [295] P.H.A. Sneath. The construction of taxonomic groups. In G.C. Ainsworth and P.H.A. Sneath, editors, *Microbial Classification*, pages 289–332. Cambridge University Press, 1962.
- [296] P.H.A. Sneath and R.R. Sokal. *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. W.H. Freeman and Company, San Francisco, 1973.
- [297] E. Sobel and H.M. Martinez. A multiple sequence alignment program. *Nucleic Acids Research*, 14(1):363–374, 1986.
- [298] R.R. Sokal and P.H.A. Sneath. *Principles of numerical taxonomy*. W.H. Freeman and Company, San Francisco, 1963.

- [299] V. Sperschneider. *Bioinformatics: Problem Solving Paradigms*. Springer-Verlag, Berlin, 2008.
- [300] J. Stoye. Affix trees. Technical report 2000-04, University of Bielefeld, Germany, 2000.
- [301] C.B. Stringer, R. Grün, H.P. Schwarcz, and P. Goldberg. ESR dates for the hominid burial site of Es Skhul in Israel. *Nature*, 338:756–758, 1989.
- [302] D. Strothmann. The affix array data structure and its applications to RNA secondary structure analysis. *Theoretical Computer Science*, 389(1-2):278–294, 2007.
- [303] J.A. Studier and K.J. Keppler. A note on the neighbor-joining algorithm of Saitou and Nei. *Molecular Biology and Evolution*, 5(6):729–731, 1988.
- [304] K. Tamura, M. Nei, and S. Kumar. Prospects for inferring very large phylogenies by using the neighbor-joining method. *Proc. National Academy of Sciences USA*, 101(30):11030–11035, 2004.
- [305] E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155:881–888, 2007.
- [306] R.E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [307] C.H. Teo and S.V.N. Vishwanathan. Fast and space efficient string kernels using suffix arrays. In *Proc. 23rd International Conference on Machine Learning*, pages 929–936. ACM Press, 2006.
- [308] G. Tesler. Efficient algorithms for multichromosomal genome rearrangements. *Journal of Computer and System Sciences*, 65(3):587–609, 2002.
- [309] J.D. Thompson, D.G. Higgins, and T.J. Gibson. CLUSTALW: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.
- [310] F. Tinti, C. Piccinetti, S. Tommasini, and M. Vallisneri. Mitochondrial DNA variation, phylogenetic relationships, and evolution of four Mediterranean genera of soles (Soleidae, Pleuronectiformes). *Marine Biotechnology*, 2(3):274–284, 2000.

- [311] G. Tischler. On wavelet tree construction. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching*, volume 6661 of *Lecture Notes in Computer Science*, pages 208–218. Springer-Verlag, 2011.
- [312] C. Trapnell and S.L. Salzberg. How to map billions of short reads onto genomes. *Nature Biotechnology*, 27(5):455–457, 2009.
- [313] T.J. Treangen and X. Messeguer. M-GCAT: interactively and efficiently constructing large-scale multiple genome comparison frameworks in closely related species. *BMC Bioinformatics*, 7:433, 2006.
- [314] E. Ukkonen. Approximate string-matching with q -grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.
- [315] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [316] N. Välimäki, S. Ladra, and V. Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Information and Computation*, 213:49–58, 2012.
- [317] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching*, volume 4580 of *Lecture Notes in Computer Science*, pages 205–215. Springer-Verlag, 2007.
- [318] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.
- [319] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
- [320] S.V.N. Vishwanathan and A.J. Smola. Fast kernels for string and tree matching. In B. Schölkopf, K. Tsuda, and J.-P. Vert, editors, *Kernel Methods in Computational Biology*, chapter 5. MIT Press, Cambridge, MA, 2004.
- [321] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.
- [322] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.
- [323] R.A. Wagner and M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

- [324] M.E.M.T. Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. In *Proc. Symposium on String Processing and Information Retrieval*, pages 96–102. IEEE Computer Society, 1998.
- [325] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.
- [326] M.S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman Hall, 1995.
- [327] M.S. Waterman, T.F. Smith, and W.A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.
- [328] M.S. Waterman, T.F. Smith, M. Singh, and W.A. Beyer. Additive evolutionary trees. *Journal of Theoretical Biology*, 64(2):199–213, 1977.
- [329] D. Weese and M.H. Schulz. Efficient string mining under constraints via the deferred frequency index. In *Proc. Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects*, volume 5077 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 2008.
- [330] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.
- [331] W.J. Wilbur and D.J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proc. National Academy of Science USA*, 80:726–730, 1983.
- [332] M. Yamamoto and K.W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, 2001.
- [333] K. Zarestkii. Reconstructing a tree from the distances between its leaves. *Uspekhi Matematicheskikh Nauk*, 20:90–92, 1965. In Russian.
- [334] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
- [335] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

Index

- ψ -function, 186, 286
- Φ -algorithm, 81, 84, 134
- $\$,$ *see* sentinel character
- $\Sigma,$ *see* alphabet
- $\Sigma^*,$ 10
- $\Sigma^+,$ 10
- $\Sigma^n,$ 10
- $\perp,$ *see* undefined value
- ω -interval, 60
- $\varepsilon,$ *see* empty string
- 3-point condition, 494
- 4-point condition, 521

- absent word, 104, 322
- additive inequality, 491
- additive metric, 491
- additive tree, 479, 514, 521, 530
 - construction algorithm, 525
- adjacency, 439, 442, 446
- Aho-Corasick algorithm, 29, 375
- alignment, 385
 - anchor-based, 417
 - cost, 388
 - graph, 391
 - length, 386, 407
 - multiple, 407
 - optimal, 388, 400
 - pairwise, 386, 394, 397, 401, 403
 - similarity score, 400
- alphabet, 9
- alphabet interval, 303
- approximate occurrence, 378
- approximate string matching, 374

- approximation algorithm, 409, 413, 434

- backward search, 299
- balanced parentheses sequence, 266
 - enhanced, 266
- ballot number, 48
- bidirectional wavelet index, 369
- binary search, 120, 262, 263, 274, 295, 300, 334
- BLOSUM matrix, 404
- bootstrapping, 483, 569
- bottom-up traversal, 94, 134, 149, 151, 156, 185, 209, 220, 229, 244, 248, 321, 568
- Boyer-Moore-Horspool algorithm, 13
- BPS, *see* balanced parentheses sequence
- $BPS_{pre},$ 259
- branch length, *see* edge length
- breadth-first traversal, 30, 99, 104
- breakpoint, 439
 - graph, 440
- Burrows and Wheeler transform, 282
- BWT, 283

- $\mathcal{C}(A),$ *see* Cartesian tree
- $\mathcal{C}^{can}(A),$ *see* canonical Cartesian tree
- $\mathcal{C}^{sup}(A),$ *see* Super-Cartesian tree
- C -array, 70, 263, 285, 299
- canonical Cartesian tree, 37
- Cartesian tree, 36
- Catalan number, 46
- cDNA, 226
- center star method, 411
- center string, 411

- chain, 419
- character, 9
- child interval, 87, 273
- child table, 107
- chromosome, 3
- CLD-table, *see* child table
- clustering, 501
- colinear, 418
- common substring, 181
- common suffix array, 173, 206
- compact trie, 31, 113
- component, 447
 - bad, 447
 - good, 447
- compressed full-text index, 281
- compressed suffix tree, 281
- convergent reality edges, 443
- correction term, 217, 249, 347
- cost
 - Levenshtein, 389
 - operation-weighted, 388
- counting query, 116, 262
- $CT(\phi)$, *see* correction term
- cycle, 442
 - bad, 447
 - good, 447
- cyclic string, 79, 141

- de Bruijn graph, 142
- de Bruijn sequence, 141, 179
- decision query, 116, 262
- depth
 - of a node in a rooted tree, 36
- depth-first traversal, 35, 99, 258
- desire edge, 440
 - bad, 447
 - good, 447
- dissimilarity matrix, 492
 - additive, 492
 - additive tree consistency, 514
 - nearly additive, 549
 - quartet-consistent with a tree, 550
 - ultrametric, 492
 - ultrametric tree consistency, 493
- $dist_\delta$, 409
- distance, 388, 490
 - edit, 377
- distance matrix, 492
- distance method, 479
- distinct elements range query, 222
- divergent reality edges, 443
- divide and conquer, 62, 395
- DNA, 1
 - double-stranded, 2
 - replication, 2
 - single-stranded, 1
- document array, 173, 206, 345
- D -array, *see* document array
- document frequency, 216, 244
- document listing problem, 221, 348
 - forbidden pattern, 350
 - multi-pattern, 349, 351
- duplication with modification, 385
- dynamic programming, 42, 390, 408, 420, 424

- edge length, 479, 483
- edist*, *see* edit distance
- edit distance, 377, 398, 411
- edit operations, 377
- elementary interval, 445
 - bad, 445
 - good, 445
 - status, 446
- embedded lcp-interval, 87
- empty string, 10
- enclosing lcp-interval, 87
- enhanced suffix array, 59, 79
 - generalized, 179, 206
- enumeration query, 116, 118, 263
- ESA, *see* enhanced suffix array
- estimation of edge weights, 561
- Euclidean algorithm, 172
- eukaryotic cell, 3
- Euler tour, 35
- Eulerian cycle, 142, 144
- evolution, 7
- evolutionary path, 479, 515
- exact match, 183
 - left maximal, 183
 - maximal, 183
 - right maximal, 183
- exponent, 170
- extent

- of a cycle, 448
 - of a set of cycles, 448
- external edge, 490, 514
- failure link, 25
- Farris transform, 520, 534
- Fibonacci string, 170
- fission, 429
- FM-index, 299
- fortress, 463
- four-point condition, 521
- Four-Russians technique, 48
- fragment, 418
 - chaining, 419
- frameshift, 8
- frequent string mining problem, 248
- fusion, 429
- gap penalty, 404
 - affine, 405
 - gap-extension, 405
 - gap-open, 405
- gene, 3
 - arrangement, 430
- gene expression, 3
- generalized suffix array, 179, 206, 345
- genetic code, 4
- genome rearrangements, 429
- GESA, *see* enhanced suffix array, generalized
- hairpin loop, 357
- Hamming distance, 375, 482
- Hamming sphere, 375
- happy clique, 467
- heaviest increasing subsequence, 424
- Hirschberg's algorithm, 397, 414
- home index, 103
- homolog, 385
- human genome, 3, 138
- hurdle, 459
 - non-consecutive hurdles, 461
 - simple, 461
 - super, 461
- hybridization, 224
- identity permutation, 432, 438
- increasing subsequence, 424
- indel, 377, 387
- induced sorting, 68
- induced sorting algorithm, 68, 292
- internal edge, 490
- inverse suffix array, 60
- inversion, 429, 438
- ISA, *see* inverse suffix array
- k -common substring problem, 208
- k -common repeated substring problem, 215
- k -mer, 183, 419
- k -differences problem, 378
- keyword tree, 24
- k -mismatch, 375
- k -mismatch problem, 375
- Knuth-Morris-Pratt algorithm, 20
- LACA, *see* LCP-array construction algorithm
- last common ancestor, 385, 479, 515
- LCA, *see* lowest common ancestor
- lcp, *see* longest common prefix
- LCP-array, 79
 - local maximum, 144
- LCP-array construction algorithm, 81, 84, 319
- lcp-index, 86
- lcp-interval, 86
- lcp-interval tree, 87
- lcs, *see* longest common suffix
- leaf neighbors, 527
- least squares method, 561
- Lempel-Ziv factorization, 125, 332
- Levenshtein costs, 389
- Levenshtein distance, 377
- lexicographic order, 60
- lexicographic product, 66
- LF -mapping, 284, 300
- ℓ -index, 86, 272
- line-sweep paradigm, 354, 420
- LMS-position, 70
- LMS-substring, 74
- LMS-suffix, 74
- longest common prefix, 79, 85, 157
- longest common substring, 181, 208
 - all-pairs, 237

- longest common suffix, 79, 85, 157
- longest increasing subsequence, 424
- longest previous substring, 126
- lowest common ancestor, 34, 92, 185, 492
- LPS-array, 126
- master theorem, 67
- matching statistics, 194, 195, 228, 241, 250, 336
 - bidirectional, 202
 - mutual, 202
- maximal exact match, 183, 340, 419
 - rare, 203
- maximal unique match, 184, 203, 419
- maximum likelihood method, 480
- maximum parsimony method, 480
- maximum similarity, 387
- MEM, *see* maximal exact match
- merging suffix arrays, 173, 203, 250
- metric, 388, 439, 490
- MFT, *see* move-to-front
- minimum distance, 387
- mitochondrial DNA, 429, 481
- molecular anthropology, 481
- molecular clock hypothesis, 479, 515
- most recent common ancestor, 482
- move-to-front, 288
- mRNA, 4, 226, 357
- mtDNA, *see* mitochondrial DNA
- multiple alignment, 407
 - problem, 408
 - progressive, 415
- MUM, *see* maximal unique match
- mutation, 7
- natural selection, 8
- nearly additive, 549
- neighbor selection criterion, 532
 - ADDTREE, 559
 - Farris, 534
 - Saitou and Nei, 540
- neighbor-joining, 416, 483
 - ADDTREE algorithm, 559
 - Farris' algorithm, 537, 539, 556
 - fast algorithm, 547, 558
 - generic algorithm, 533, 555
 - Saitou & Nei's algorithm, 541, 558
- non-hurdle, 459
 - shields super hurdle, 462
- NSV_{LCP}, 89, 93, 269
- NSV_{SA}, 128, 333
- nucleotide, 1
- oligonucleotide, 224
- oligonucleotide selection problem, 226
- on-line algorithm, 110, 374, 426
- orientation
 - negative, 432
 - positive, 432
- orthogonal range-searching, 420
- ortholog, 385
- outgroup, 479
- output function, 28
- output set, 28
- output-sensitive algorithm, 24, 143
- overlap
 - component/interval, 456
 - cycles, 447
 - desire edges, 447
 - forest, 448
 - fragments, 418
 - graph, 447
 - intervals, 61, 446
 - status, 452
- pairwise alignment, 386
- palindrome, 182
- PAM matrix, 403
- paralog, 385
- parent interval, 87, 90, 270, 337
- PATRICIA tree, 31
- pattern, 11, 24, 116, 299
- pattern matching, *see* string matching
- PCR, *see* polymerase chain reaction
- peak, 131
- peak elimination, 133, 134
- perfect binary tree, 307
- period
 - starting at a position, 159
 - to the left of a position, 159
- period-length, 157
- periodicity, 157

- left-maximal, 158
 - maximal, 158
 - right-maximal, 158
 - type 1, 163
 - type 2, 163
- permutation
 - oriented, 437
 - signed, 437
- phylogenetic tree, 415, 483, 490
 - binary, 490
 - canonical form, 493
 - rooted, 490
 - unrooted, 490
- phylogeny, 477
 - of the great apes, 478
 - star, 490
- PLCP-array, 81
- point mutation, 7, 429
- polymerase chain reaction, 224
- postorder traversal, 297
- prefix, 11
 - proper, 11
- prefix function
 - for one pattern, 16
 - for several patterns, 25
- prefix tandem repeat, 97
- preorder traversal, 35, 258
- PrevOcc-array, 130
- primer, 224
- primer selection problem, 224
- primitive string, 157
- priority queue, 423
- priority search tree, 421
- prokaryotic cell, 3
- protein, 6
- PSV_{LCP}, 89, 93, 271
- PSV_{SA}, 128, 333
- quadtrees, 509
- quartet, 526
- quartet-consistency, 550
- range maximum query, 335
 - two dimensional, 420
- range minimum query, 33, 275
- rank query, 257, 299
- rank*-array, 60
- read, 231
- reading frame, 8
- reality edge, 440
- reality-desire diagram, 440
 - circular representation, 440
 - linear representation, 440
- reciprocal translocation, 429
- relevant substring, 248
- repeat, 139
 - longest, 140, 141
 - maximal, 139, 149, 329
 - non-overlapping, 155
 - supermaximal, 139, 145, 330
 - tandem, 139
- repeated pair, 149
 - left instance, 149
 - left maximal, 150
 - maximal, 150
 - non-overlapping, 155
 - overlapping, 155
 - right instance, 149
 - right maximal, 150
- replication error, 2
- reversal, 429, 438
 - distance, 438
 - merging, 456
- reverse string, 10, 358
- rightmost path
 - in a Cartesian tree, 37
 - in a Super-Cartesian tree, 107
- RMQ, *see* range minimum query
- RNA, 3
- rRNA, 5, 357
- run, 170, 288
- SA, *see* suffix array
- SA', 206
- SACA, *see* suffix array construction
 - algorithm
- score
 - elementary interval, 453
 - similarity, 400
 - sum-of-pairs, 408
- scoring matrices, 403
- seed-and-extend paradigm, 384
- segment, 438
 - conserved, 434

- select query, 258, 299
- semimetric, 490
- sentinel character, 60
- sequence, 10
- sequence assembly, 231
- short read mapping, 374
- shotgun sequencing, 231
- similarity, 402
 - operation-weighted, 399
- singleton interval, 88
- skew algorithm, 61
- skip and count, 195
- slink, *see* suffix link table
- sorting by reversals, 432, 438
- sorting by transpositions, 434
- sparse suffix array, 263
- sparse table algorithm, 43
- speciation, 477
- split, 526
- strand
 - coding, 3
 - forward, 437
 - heavy, 430
 - lagging, 2
 - leading, 2
 - light, 430
 - reverse, 437
 - template, 3
- string, 10
 - cyclic, 79
 - empty, 10
- string matching, 11, 116
 - approximate, 374
 - safe shift, 13, 15, 16
- substring, 10
 - proper, 11
- succinct data structure, 262
- suffix, 11
 - proper, 11
- suffix array, 60
 - common, 173, 206
 - generalized, 179, 206
- suffix array construction algorithm, 61, 68
- suffix insertion algorithm, 113
- suffix link, 185
- suffix link interval, 186, 276
- suffix link table, 188
- suffix tree, 111
 - construction algorithm, 114
- suffix-prefix matching problem, 231, 352
- sum-of-pairs score, 408
- Super-Cartesian tree, 105, 266
- synteny block, 435
- tandem array, 157
 - left-maximal, 157
 - maximal, 157
 - right-maximal, 157
- tandem repeat, 157
- taxon, 477
- term frequency, 244
- text, 11
- TF-IDF score, 349
- three-point condition, 494
- top-down traversal, 98, 105, 117, 188, 192, 244, 324, 371
- transcription, 4
- translation, 4
- translocation, 429
- transposition, 429
- triangle inequality, 490
- trie, 24, 297
- tRNA, 5, 357
- ultrametric, 491
- ultrametric inequality, 491
- ultrametric tree, 479, 492, 516
 - construction algorithm, 495
- undefined value, 20
- unique substring, 104
 - shortest, 104, 225, 322, 323
- UPGMA, 501, 503
- Watson-Crick base pairs, 2
- wavelet tree, 303
 - Huffman shaped, 315
 - of the document array, 348
 - weight-balanced, 315
- whole genome alignment, 417
- word suffix array, 79
- WPGMA, 508

