© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Contents

Preface xvii

1 Introduction 1

1.1 What are strings? 1
1.2 Why study string algorithms? 1
1.3 What are our goals? 2
1.4 A roadmap 3
1.5 A Primer 3
1.5.1 String notations and definitions 3
1.5.2 Graph theory 4
1.5.3 Running times 5
1.5.4 Sets and combinatorics 6
1.5.5 Numbers 6
1.5.6 Data structures 6
1.5.7 Probability 7

I EXACT MATCHING 9

2 The Z Algorithm 11

2.1 The exact matching problem 11
2.2 Simple (slow) solution 11
2.3 The Z algorithm 12
2.4 Computing the Z values 13
2.5 Summary and notes 15
2.6 Exercises 16

3 Boyer-Moore 17

3.1 High-level description 17
3.1.1 First rule: Next Matching Character 17

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

3.1.2 Second rule: Good Shift Rule 18
3.1.3 Complete algorithm 19
3.2 Computing the R_i values for the next matching character rule 19
3.3 Formalizing the Good Shift Rule 20
3.4 Implementing the Good Shift Rule 21
3.4.1 Computing the $L(i)$ values 22
3.4.2 Computing the $\ell(i)$ values 23
3.5 Summary and notes 24
3.6 Exercises 24

4 Knuth-Morris-Pratt 25

4.1 KMP via deterministic finite automata 25
4.1.1 The KMP DFA 26
4.1.2 Using the *memo* array for string search 27
4.1.3 Correctness and running time 28
4.1.4 Computing *memo* 29
4.2 KMP via the Z-values 30
4.2.1 Running time of the spm_i version of KMP 31
4.2.2 Computing the spm_i array 31
4.3 Summary and notes 33
4.4 Exercises 33

5 Seminumerical String Matching 35

5.1 Rabin-Karp fingerprinting 35
5.1.1 Computing p 35
5.1.2 Computing t_s for every position s 36
5.1.3 Time for addition, product, and comparison 37
5.1.4 Quantifying and reducing false positives 37
5.1.5 Reducing the error probability 39
5.2 The Shift-And algorithm 39
5.2.1 Space and runtime 41
5.2.2 Extension to approximate matching 41
5.3 Summary and notes 42
5.4 Exercises 42

6 Searching for Multiple Patterns 45

6.1 Aho-Corasick—a prefix-based approach 46
6.1.1 Search 48
6.1.2 Handling patterns contained in other patterns 48

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

6.1.3 Running time 50
6.1.4 Computing *f* 50
6.2 Wu-Manber—a suffix-based approach 52
6.3 Summary and notes 53
6.4 Exercises 53

II EDIT DISTANCE 55

7 Edit Distance for Inexact Matching 57

7.1 Edit distance and alignments 58
7.2 The string alignment problem 59
7.2.1 Algorithm for minimum cost alignments 60
7.2.2 Implementing the recursive algorithm 61
7.3 Dynamic programming 63
7.4 Finding the actual alignment: traceback 63
7.5 Local and semi-global alignment 65
7.5.1 Local alignment 65
7.5.2 Semi-global alignment 67
7.6 Summary and notes 68
7.7 Exercises 69

8 Edit Distance in Linear Space 73

8.1 Using linear space to compute edit distance values 73
8.2 Finding the actual alignment in linear space 76
8.2.1 Hirschberg's algorithm 77
8.2.2 Proof of running time 78
8.3 Summary and notes 80
8.4 Exercises 80

9 Faster Edit Distance via the "Four Russians" Trick 81

9.1 Dynamic programming blocks 81
9.2 Precomputing *f* 82
9.2.1 Offset encoding 83
9.2.2 Number of possible inputs to *f* 85
9.2.3 Storing *f* for quick access 85
9.3 Total running time 86
9.4 Finding the alignment 87
9.5 Summary and notes 87
9.6 Exercises 88

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

10 General and Affine Gap Penalties 89

10.1 General gap penalties 89
10.2 Affine gap penalties 91
10.3 Why do we “need” 3 matrices? 92
10.4 Summary and notes 93
10.5 Exercises 94

11 Output-Sensitive Algorithms for Edit Distance 95

11.1 Banded alignment 95
11.2 Myers’ Minimum Edit Script algorithm 96
11.3 Landau-Vishkin* 98
11.3.1 The *k* differences problem 98
11.3.2 An alternative dynamic program 99
11.3.3 The Landau-Vishkin algorithm 102
11.4 Summary and notes 103
11.5 Exercises 103

12 Aligning Multiple Strings 105

12.1 Multiple sequence alignments 105
12.2 Progressive alignment heuristic 106
12.3 Approximation algorithm for one MSA formulation 107
12.4 Summary and notes 111
12.5 Exercises 111

III DATA STRUCTURES 113

13 Suffix Trees 115

13.1 The fundamental string data structure 115
13.2 Tries 115
13.3 Defining suffix trees 116
13.4 Applications of suffix trees 118
13.4.1 Simple queries 118
13.4.2 Longest common substring of two strings 119
13.4.3 Generalized suffix trees 120
13.4.4 All pairs suffix-prefix matches 121
13.5 Summary and notes 122
13.6 Exercises 122

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

14 More Applications of Suffix Trees 125

14.1 Lempel-Ziv compression 125
14.2 Longest common extension 127
14.3 Finding palindromes 128
14.4 Finding all k-mismatch occurrences of a pattern 129
14.5 Longest common substring with online queries 130
14.6 Summary and notes 133
14.7 Exercises 133

15 Suffix Tree Construction 135

15.1 Constructing suffix tries 135
15.2 Ukkonen's algorithm 138
15.2.1 Updating the tree 139
15.2.2 Starting each phase 141
15.2.3 Example run of Ukkonen's algorithm 142
15.2.4 Running time of Ukkonen's algorithm 144
15.3 Summary and notes 145
15.4 Exercises 146

16 Suffix Arrays 147

16.1 Suffix tree operations in less space 147
16.2 Suffix array ↔ suffix tree 148
16.3 Searching for substrings using the suffix array 149
16.4 Faster search 149
16.4.1 Obtaining the $lcp(i, j)$ values 152
16.5 Summary and notes 154
16.6 Exercises 154

17 Suffix Array Construction 157

17.1 An $O(|T| \log |T|)$ algorithm for suffix array construction 157
17.1.1 Radix sort 158
17.2 A linear-time construction algorithm 159
17.2.1 Running time 161
17.3 Summary and notes 162
17.4 Exercises 162

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

18 Burrows-Wheeler Transform 165

18.1 Definition of the BWT 165
18.2 Why is this useful for compression? 166
18.3 Recovering the original string from its BWT 166
18.3.1 The LF mapping 167
18.3.2 Faster inversion algorithm 168
18.4 Faster computation of the BWT 169
18.5 Applications of the BWT 170
18.5.1 Search 170
18.5.2 Compression 171
18.6 Summary and notes 173
18.7 Exercises 174

19 RRR Compressed Bit Vectors 175

19.1 Rank and select operations 175
19.2 RRR bit vector data structure 176
19.3 Computing $\text{rank}_1(S, i)$ 177
19.4 Compactly encoding the RRR information 177
19.4.1 Prefix-sum data structure 177
19.4.2 Space to store the vectors of integers 178
19.4.3 Space for the tables 179
19.4.4 Space for the p values 180
19.5 Supporting the select operation* 181
19.5.1 The select data structures 181
19.5.2 Answering select queries 183
19.6 Summary and notes 183
19.7 Exercises 184

20 Wavelet Trees 187

20.1 The wavelet tree data structure 187
20.1.1 The access $S[i]$ operation 188
20.1.2 The $\text{rank}_c(S, i)$ operation 189
20.1.3 The $\text{select}_c(S, j)$ operation 190
20.1.4 Running times 190
20.2 Applications of wavelet trees 191
20.2.1 Inverted indices 191
20.2.2 Document retrieval 191
20.2.3 Storing graphs 192
20.3 Summary and notes 193
20.4 Exercises 193

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

21 The FM Index for Compressed Searching 195

21.1 BWT and wavelet trees 195
21.2 A more compression-focused approach 196
21.2.1 Computing rank on the compressed string 196
21.2.2 Space usage for count arrays 197
21.2.3 Computing occurrence counts 198
21.2.4 Computing counts within a block 198
21.3 Summary and notes 199
21.4 Exercises 199

22 Storing Text for Editing 201

22.1 Gap buffers 202
22.2 Piece tables 202
22.3 Ropes 203
22.4 Summary and notes 207
22.5 Exercises 207

IV SKETCHING 209

23 Locality Sensitive Hashing 211

23.1 Estimating similarity between strings 211
23.2 An LSH family for the Jaccard similarity 213
23.3 An LSH family for the weighted Jaccard 216
23.4 An LSH family for a cosine-related similarity 217
23.5 A gapped LSH family for edit distance 218
23.6 Summary and notes 220
23.7 Exercises 220

24 Bloom Filters 223

24.1 How do Bloom filters work? 223
24.2 How should we set the filter parameters? 224
24.3 Notes on and extensions to Bloom filters 226
24.4 Cascading Bloom filters 227
24.5 Summary and notes 230
24.6 Exercises 230

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

25 Sketching with Minimizers 231

25.1 Accelerating read overlap computation 232
25.2 Definition of minimizers 232
25.3 Properties of minimizers 233
25.4 Expected density 234
25.5 Random minimizers 235
25.6 Selecting minimizers 237
25.7 De Bruijn sequences and de Bruijn graphs 238
25.8 Set of unavoidable words 241
25.9 Summary and notes 242
25.10 Exercises 242

V GRAPHS AND GENERATIVE MODELS 243

26 Regular Grammars and the Chomsky Hierarchy 245

26.1 Regular grammars 246
26.1.1 Finite state automata 246
26.1.2 Deterministic vs. non-deterministic FSAs 248
26.1.3 Regular grammars and regular expressions 249
26.1.4 Summary of regular grammars 252
26.2 Context-free grammars 252
26.2.1 CFG parse trees 252
26.2.2 Chomsky normal form 253
26.3 Context-sensitive and unrestricted grammars 253
26.4 Summary and notes 254
26.5 Exercises 254

27 Hidden Markov Models 257

27.1 Definition of hidden Markov models 257
27.2 HMM decoding problems 259
27.3 Viterbi algorithm 260
27.4 The Forward-Backward algorithm 261
27.4.1 Forward algorithm 261
27.4.2 Backward algorithm 262
27.5 Estimating HMM parameters 263
27.5.1 Labeled training 263
27.5.2 Viterbi training 264
27.5.3 Baum-Welch algorithm 264
27.6 Summary and notes 265
27.7 Exercises 265

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

28 Stochastic Context-Free Grammars 267

28.1 The Inside algorithm 268
28.2 Cocke-Younger-Kasami (CYK) algorithm 269
28.3 Estimating the production probabilities 269
28.3.1 The Outside algorithm 272
28.4 Summary and notes 273
28.5 Exercises 273

29 Genome Graphs 275

29.1 Some types of genome graphs 275
29.1.1 De Bruijn graphs 275
29.1.2 Colored de Bruijn graphs 277
29.1.3 Variation graphs 277
29.2 Constructing genome graphs via compression 278
29.3 Alignment of strings to genome graphs 280
29.3.1 Alignment graph 280
29.4 Comparing genome graphs 282
29.5 Summary and notes 284
29.6 Exercises 284

30 BWT on Labeled Trees 287

30.1 String trees 287
30.2 String tree BWT 288
30.2.1 The interval corresponding to the ith node 290
30.2.2 A child interval 290
30.3 Extending to substring intervals 292
30.4 Backwards search 294
30.5 Summary and notes 294
30.6 Exercises 295

31 Transformers 297

31.1 Embeddings 297
31.2 Encoder-decoder architectures 299
31.3 Feedforward neural networks 300
31.4 Attention 302
31.5 Transformers 303
31.5.1 The transformer encoder 303
31.5.2 The transformer decoder 304
31.6 Training and using transformers 305

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

31.7 Summary and notes 305
31.8 Exercises 306

VI MISCELLANEOUS 307

32 Huffman Codes 309

32.1 Prefix-free codes 309
32.1.1 UTF-8 encodings 310
32.2 Huffman coding algorithm 311
32.2.1 Proof of optimality 313
32.2.2 Linear-time code construction 315
32.3 Summary and notes 316
32.4 Exercises 316

33 Shortest Superstring 317

33.1 Relationship to the traveling salesman problem 317
33.2 The Cycle Cover algorithm 319
33.3 Some lemmas about repeated strings 322
33.4 The Cycle Cover algorithm gives a 4-approximation 324
33.5 Summary and notes 326
33.6 Exercises 326

34 Limits on What's Possible 327

34.1 A brief introduction to NP-completeness 327
34.2 Shortest superstring is NP-complete 329
34.2.1 The reduction 329
34.2.2 The proof 331
34.2.3 Additional notes 333
34.3 Shortest supersequence is NP-complete 333
34.4 MSA with SP-score is NP-complete 335
34.5 Summary and notes 338
34.6 Exercises 338

35 Conclusion 339

Bibliography 341
List of Algorithms 349
Index 351

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

CHAPTER 1

Introduction

1.1 What are strings?

In this book we focus on algorithms that operate on strings. What do we mean by strings?

Definition 1.1 (String). A *string* is a one-dimensional, ordered list of symbols drawn from some finite alphabet Σ. ■

Typically, we assume $|\Sigma|$ is "small" compared with the length of the string. Usually, in fact, we will assume $|\Sigma|$ is a constant. For example, Σ may be the ASCII character set, or the Unicode character set, or the set of stock ticker symbols. On the other hand, choosing $\Sigma = \mathbb{R}$ (the real numbers) would produce something that is *not* a string.

A string is typically encoded into the memory of a computer using a sequence of contiguous, fixed-length fields, where each field (of say 8 bits) codes for some character of Σ. In the C language, for example, `char s[10]` represents a string of length 9 where Σ equals the set of symbols represented by the type `char`. In C, and some other languages, rather than encode the length of a string, a terminal special character (`\0`) is placed at the end of the representation. In other representations, an explicit length is stored.

None of these representations are essential for our definition of "string." They are implementation details. A string stored with each character in successive nodes of a linked list is also a string. Although in that case accessing the ith character incurs a cost of $O(i)$ instead of the cost $O(1)$ in the more typical encoding. While storage of a string in linked list or other more complex data structures still encodes something we will call a string, unless otherwise noted, we will assume that we are using a representation where accessing the ith character takes $O(1)$ time.

To encode a symbol from Σ using a simple encoding requires

$$\lceil \log_2 |\Sigma| \rceil \tag{1.1}$$

bits. The notation $\lceil \ldots \rceil$ indicates rounding up to the next integer; $\lfloor \ldots \rfloor$ indicates rounding down.

1.2 Why study string algorithms?

The ability to process, store, search, and manipulate strings with computational efficiency has changed the world in many ways. Web search engines regularly process terabytes of

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

strings, internet shopping sites deal with many product descriptions, and social networks handle large collections of comments and posts. Version control systems and compilers must process text that forms the source code of programs. Expert systems (like IBM Watson [Ferrucci et al., 2010] or NELL [Mitchell et al., 2015; Carlson et al., 2010]) must process strings to form databases of knowledge. Large Language Models (LLMs) [Radford et al., 2018; Devlin et al., 2018] train on massive amounts of text to generate human-like answers to queries.

The field of genomics has been a fruitful and motivating area for string algorithms, where, to a good first approximation, the genome of an organism can be encoded as a long string of letters representing the nucleotides that make up the DNA (or RNA) molecules. These genome strings can be huge—megabytes for a bacterium, gigabytes for an organism like human. Storing, searching, comparing, and analyzing these strings requires efficient algorithms. Reconstructing genomes or parts of them from fragmented measurements requires reconstructing strings from shorter strings. Collections of protein sequences are also well represented by strings. Genome graphs in Chapter 29 are motivated by applications in pan-genomics, and minimizers in Chapter 25 have their primary current application in genomics, though they were originally proposed in other contexts.

String algorithms are also fundamental from a computer science perspective. One can view the sequence of bits in a computer's RAM as a long string of 1s and 0s. Questions about what problems on strings can be efficiently solved have spurred innovation in computational techniques that can be widely applied and adapted to other problems that do not at first appear to be string-related. Solving a one-dimensional version of a problem is often a good first step toward understanding pattern matching problems in higher dimensions. Much of the theory of computational complexity is based around sets of strings. Finally, string algorithms are a fun and interesting type of algorithm that have given rise to interesting algorithmic techniques and neat data structures that are elegant and useful.

1.3 What are our goals?

This book aims to describe algorithmic solutions to string problems. We will generally take a "theoretical" viewpoint, meaning that we will be concerned with proving correctness and runtime for our various approaches rather than empirical evaluation, though we will discuss some practical considerations. While the emphasis is on provable properties, we aim to not be unnecessarily mathematical, but instead aim to draw out the key ideas that may be more generally useful. We are not trying to treat you as a "mathematics compiler" nor to provide the definitive proof of every result. Instead, we present the material in such a way that we hope you will understand why the algorithms work the way they do. We strive to cover everything you need to believe that the presented algorithms have the properties we claim they have, though, in the interest of focusing on the key ideas, some details are omitted from proofs and other material.

If you master the topics in this book, you will be well positioned to use and adapt string algorithms to solve new problems. You will also have the necessary background to begin making contributions to the design of new string algorithms and applications.

The field of string algorithms is vast, and we cannot hope to cover everything. There are many more advanced or esoteric topics that are of great interest—both theoretically and for

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

particular applications—that we must omit. An entire second volume could be written covering advanced topics that are only hinted at here. We also don't aim to cover the complete history of the presented algorithms. Once you grasp what is included here, you will have the tools to learn about these other topics on your own.

1.4 A roadmap

We start (Part I) with the classical problem of finding occurrences of a short string P exactly within a longer text T. This is, of course, the string problem that is most used and most useful, on its own or as a building block for bigger string processing systems.

In Part II, we move to the edit distance problem that supports finding matches with some deviation from the pattern (extra characters, mismatching characters, etc.); this allows us to introduce the classical dynamic programming approach for this problem and also some tricks that enable faster or more space-efficient realizations of the underlying dynamic programming technique. This collection of problems is particularly important in genomic analyses, where sequences have mutations.

Part III introduces data structures that have been developed to store strings and solve a variety of problems associated with them. These include the classical (and very important) suffix trees and suffix arrays, as well as newer (though still decades old) data structures such as wavelet trees and RRR compressed bit vectors. The building blocks introduced in this part are foundational to a number of string processing algorithms, and often useful outside of string problems.

Part IV expands on the techniques for creating small, lossy representations of strings or sets of strings allowing for some loss of accuracy but with great gains in space or time efficiency. Classical approaches such as locality sensitive hashing and Bloom filters are introduced, as are techniques that are more modern such as sketching for edit distances and the minimizers technique. These sketching techniques are frequently used to deal with extremely large strings or collections of strings.

Part V turns to looking at, broadly, generative models for strings. By encoding a family (perhaps of infinite size) of strings in a compact way, we can answer questions about that family of strings and relate other strings to the family. Techniques introduced in this part include the classical hidden Markov models, but also newer approaches such as genome graphs.

Finally, Part VI collects miscellaneous topics that do not fit in other parts.

1.5 A Primer

Here, we give a primer on computational and discrete mathematical concepts that will be useful in the text.

1.5.1 String notations and definitions

$|S|$ is the length of string S. ($|A|$ is also the size of set A.) Σ as a variable will always be an alphabet over which strings are formed. Typically, we start strings at index 1, which is

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

convenient for explaining the algorithms, but not the norm for actually programming the algorithm, where indexing starting at 0 is more common. $T[i]$ is the character of string T at position i.

A *substring* of a string S is a contiguous sequence of characters from S. $T[i \dots j]$ denotes a substring of string T starting at index i and continuing to index j, inclusive. We will sometimes use other notation for substrings that will be introduced when used.

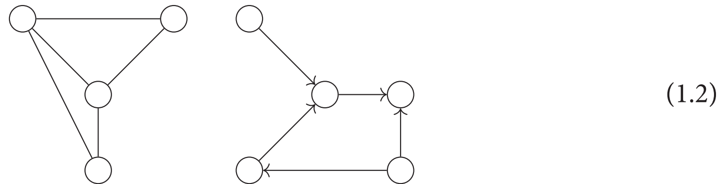
A *prefix* is a substring of characters starting from the start of the string to some higher index in the string. A *suffix* is a substring of characters starting from some position in the string and continuing until the end of the string. A *proper* prefix (suffix) is a prefix (suffix) that is neither empty nor the entire string. The *empty string*, denoted ϵ, consists of 0 characters. It is a (non-proper) suffix and prefix of any string.

Concatenation of two strings α and β is denoted $\alpha\beta$ or $\alpha \circ \beta$ when we want to emphasize the operation.

1.5.2 Graph theory

An *undirected graph* is a pair of sets (V, E), where each element of V is a *vertex* (also called a *node*), and E is a subset of $\{\{u, v\} : u \in V, v \in V\}$, each element of which is an *edge* that connects two vertices. If E is instead a subset of $\{(u, v) : u, v \in V\}$, the resulting pair (V, E) is a *directed graph*.

Examples of undirected (left) and directed (right) graphs are:



(1.2)

A *path* through a graph is a sequence of vertices $v_1, \dots, v_k$ such that each successive pair in the sequence is connected by an edge (if the graph is directed, the edge must be directed from a vertex to the successive one, i.e., $(v_i, v_{i+1}) \in E$ for edge set E). A *cycle* is a path where $v_1 = v_k$. A *simple* path or cycle is one in which no two vertices (except v_1, v_k in the case of a cycle) are equal. An undirected graph is *connected* if there is a path between every pair of vertices.

A graph is *acyclic* if it does not contain any cycles. If the graph is directed and acyclic, it is called a *DAG* (for directed acyclic graph). If an undirected graph is connected and contains no cycles it is a *tree*. If it is not connected and contains no cycles, it is a *forest*.

A *rooted tree* is a directed graph for which the corresponding undirected graph is a tree and for which there is some node r (the *root*) such that every edge goes from a vertex closer to the root to a vertex that is farther from the root. For an edge (u, v) in a rooted tree, u is called the *parent* of v and v is a *child* of u. In a rooted tree, there is a unique path from the root to any node. If v is contained on that path to a node u then v is an *ancestor* of u and u is a *descendent* of v. If a node has no descendants, it is a *leaf*. An example of a rooted tree:

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

(1.3)

A *traversal* of a tree is an algorithm to visit each vertex of a (rooted) tree. A *breadth-first search* (BFS) of a tree visits each node in increasing distance from a starting node r; that is a vertex at distance i from the root is visited before any vertices at distance $i+1$. A *depth-first search* of a tree starting from node r recursively visits children until a leaf is reached, then returns to the most recently visited node with unvisited children, and continues by visiting one of its unvisited children.

1.5.3 Running times

Big-O notation $O(f(n))$, where f is a non-negative function of an integer n, represents an upper bound on the asymptotic behavior of a resource usage as the problem instance size, n, grows. For example, with $f(n)=n$, $O(f(n))=O(n)$ indicates linear growth; $O(n^2)$ indicates quadratic growth, and $O(1)$ indicates growth that is a constant independent of the size of the problem.

More formally, let $f(n)$ be a function that gives the worst-case runtime (in terms of number of fundamental computational steps) of an algorithm on an instance of size n. We say an algorithm with running time $f(n)$ is in $O(g(n))$ if there exists constants c, n_0 such that for all $n \geq n_0, f(n) \leq cg(n)$. A consequence of this asymptotic definition is that only the "leading" term in $f(n)$ matters: $f(n)=10n^3+5n$ is in $O(n^3)$ and $f(n)=\log n+2\log\log n$ is in $O(\log n)$. An important special case is when $f(n) \leq k$ for some constant (meaning not depending on n) k. Then $f(n)$ is in $O(1)$ since with $g(n)=1, f(n) \leq kg(n)$. The $O(\cdot)$ notation can be used for resources besides runtime (like memory usage).

If a computational problem has an algorithm that runs in time $O(n^k)$ for some fixed k, we say that the problem is solvable in *polynomial time*. The set of problems that are solvable in polynomial time is denoted P. While it is an imperfect definition, P is viewed as the set of problems with *efficient* algorithms. See Chapter 34 for more discussion of the set P and related sets of problems. If an algorithm is in $O(n)$ for problem size n, we say the algorithm runs in *linear time*, which is usually the ideal time (not always achievable) for string algorithms.

A relative of Big-O is Ω notation, where $\Omega(f(n))$ indicates a *lower bound* on the growth of f for increasing n. An algorithm said to take $\Omega(n^2)$ time will take at least quadratic time asymptotically. If $f(n)$ is both $O(f(n))$ and $\Omega(f(n))$ then we say it is $\Theta(f(n))$.

Sometimes it is convenient to express the running time of an algorithm on instances of size n as a *recurrence relation*, where the time $T(n)$ is expressed as a function of the time for a smaller problem:

$$T(n) \leq f(n, T(n')), \tag{1.4}$$

where $n' < n$ and f is some function of the instance size and the running time of smaller instances. "Solving" such a recurrence means finding a non-recursive expression that is an

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

upper bound on $T(n)$. This is especially relevant when the algorithm itself is recursive or can be analyzed inductively.

1.5.4 Sets and combinatorics

A *set* is a collection of unique items. If A and B are sets, $A \cup B$ is the set of items that are in one or both of B, called the *union*. $A \cap B$ is the set of items that are in both A and B, called the *intersection*. A *multiset* is a set that allows duplicated items.

Given a set A of size n, the number of ways to choose k distinct items from that set is denoted $\binom{n}{k}$, pronounced *n choose k*. We have $\binom{n}{k} = \frac{n!}{(n-k)!k!}$, where $x!$ is the *factorial* function $x \cdot x-1 \cdot x-2 \cdot \ldots \cdot 1$.

An important special case of the *binomial theorem* is:

$$2^n = \sum_{k=0}^{n} \binom{n}{k}. \tag{1.5}$$

1.5.5 Numbers

A base-2 (aka *binary*) representation of a number is a sequence of n bits $b_{n-1}, b_{n-2}, \ldots b_0$, where each $b_i \in \{0, 1\}$ and that represents the number $\sum_{i=0}^{n-1} 2^i b_i$. Adding an additional bit doubles the range of numbers that can be represented. Hence, representing a number M requires $\lceil \log_2 M \rceil$ bits.

Base-16 is *hexadecimal*, where a number is represented with digits $\{0, \ldots, 15\}$, denoted $\{0, \ldots, 9, A, B, C, D, E, F\}$. In hexadecimal, a number $d_{n-1}d_{n-2} \ldots d_0$ represents $x = \sum_{i=0}^{n-1} d_i 16^i$. To distinguish hexadecimal from other bases, the number is typically preceded by the symbol `0x`.

The bitwise "and" function is defined as $x\&y = 1$ if and only if bits $x = 1$ and $y = 1$ (otherwise 0). The bitwise "or" function is defined as $x|y = 1$ if and only if $x = 1$ or $y = 1$ (otherwise 0). When x and y are equal-length bit vectors, the & and | operations are applied bitwise to each bit.

If x and y are positive integers, $x \pmod y$ is the remainder after x is divided by y.

1.5.6 Data structures

An array A is a contiguous sequence of values, indexed by an integer i. $A[i]$ is the value stored at position i. Arrays, like strings, may be indexed starting at 0 or 1 as needed.

A *hash table* is a data structure that stores key-value pairs (k, v). There are many implementations of this data structure. We assume that the average time to access the value v associated with key k is $O(1)$. This is a non-trivial assumption, but there are many data structures where this is true. A *hash function* is a function $h(k)$ that maps a key to a location in a hash table. Both hash functions and hash tables have a rich theory surrounding specific implementations, which we assume the reader can find if they are interested.

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

1.5.7 Probability

Let U be a universe of things that could happen. An *event* A is a subset of U meaning that one of the things in A happened. $\Pr[A]$ is a function that assigns a *probability* to one of the things in A happening. This probability is a number between 0 and 1. The notation $\Pr_X[A]$ means that probability that A happens over random variable X.

The conditional probability $\Pr[A \mid X]$ is the probability that A happens given that X happens. $\Pr[A \mid X] = \frac{\Pr[A \cap X]}{\Pr[X]}$.

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Index

AngleSim, 217, 218
D-path, 97
$K(P)$, 46
$L(i)$, 20, 22
$L(v)$, 46
$N_j(P)$, 21
R_i, 17
Z_i, 21
Child, 291, 292
Ω notation, 5
Σ, 1, 3
Θ notation, 5
cIndex, 291
$\ell(i)$, 20, 23
ϵ, 4
extendLeft, 293
$\mathcal{I}_{\text{node}}$, 289
$\mathcal{I}_{\text{str}}$, 289
d_M, 107
$f(v)$, 46
k differences, 95, 98
k-mismatch, 129
$lp(v)$, 46
memo, 47, 48
spm_i, 30, 47
t_s, 36
ZMatch, 15
NP, 328
NP-complete, 328
P, 327

accept, 246
activation function, 301
acyclic, 4, 277
affine gap, 91
affine gap penalties, 91
Aho, 46
Aho-Corasick, 45, 46
alignment, 57, 58, 280, 335
alignments, 232
all pairs suffix-prefix, 121
AllYPrefixCosts, 77
AllYSuffixCosts, 77
alphabet, 1
alphabet independent, 15
amortized analysis, 340
ancestor, 4
approximate matching, 41
approximate membership queries, 230
approximation algorithm, 107, 317, 324
ASCII, 1
attention, 302
automata, 339

backward algorithm, 261
banded alignment, 95
Baum-Welch, 264, 265
BFS, 50
big-O, 5
binary, 6
binomial theorem, 6
bit vectors, 175
bitwise and, 6
bitwise or, 6
block, 197
Bloom filter, 3, 223
Boyer-Moore, 17, 45, 52, 339
breadth-first search, 5, 50
Burrows-Wheeler transform, 165
BWT, 165, 195, 288, 291
BWT inversion, 168

cascading Bloom filters, 227
ceiling, 1
certificate, 328
CFG, 267
charged context, 235
Chernoff bounds, 215
child, 4
Chomsky, 246
Chomsky normal form, 253
choose, 6
chunking, 340
cities, 317
CNF, 253
Cocke-Younger-Kasami, 269
code points, 310
color, 277
color class, 277
colored de Bruijn graph, 277
commute, 322
compatible order, 241
compression, 171, 309, 317
computational complexity, 327
concatenation, 4
conjecture, 326
connected, 4
context, 235
context-free grammar, 252
context-sensitive grammar, 253
continuation byte, 311
Corsick, 46
cosine similarity, 217, 218
counting Bloom filters, 226
CountRange, 194
cross attention, 305
cycle, 4
cycle cover, 319, 320, 324
CYK, 269

DAG, 4
de Bruijn graph, 239, 275
decision problem, 327
decoder, 299
deletion, 58
density, 234
descendent, 4
deterministic finite automaton, 25, 246
DFA, 25
diagonal, 95, 99
directed graph, 4
dissimilarity, 217
divide-and-conquer, 73, 76
document retrieval, 191
DP matrix, 95
dynamic programming, 3, 63, 73, 81, 260, 261, 339

edit distance, 58, 81, 231, 280, 282, 339
embedding, 297
embedding function, 297
empty string, 4
encoder, 299
encoder-decoder, 299
EPM, 278
error probability, 39
Eulerian, 276

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

Eulerian graph, 239
Eulerian tour, 282
event, 7
exact matching, 11
expected density, 234
expert systems, 2
external pointer macro scheme, 278

factorial, 6
failure function, 47
feedforward, 300
FFN, 300
fingerprinting, 35
finite state automata, 246
finite state automaton, 257
finite state machine, 92
floor, 1
FM-index, 195, 196, 231, 309
forest, 4, 311
forward algorithm, 261
forward-backward algorithm, 261
Four Russians, 81, 95, 198
FSA, 257

gap, 57
gap buffers, 202
gap character, 59
gapped LSH, 218
general FSA, 251
general gap penalties, 89
generalized suffix trees, 120
generative models, 3
genome graph, 3, 275
genomics, 2
global alignment, 65
good shift rule, 18, 20
GoodShift, 53
grammar, 245, 267
graph, 4, 192, 275
graph alignment graph, 283
graph traversal edit distance, 282
greedy algorithm, 326
GTED, 282

Hamiltonian graph, 239
Hamiltonian path, 319, 329
Hamiltonian tour, 239
hash collision, 212
hash function, 6
hash table, 6
hashing, 35, 212
hexadecimal, 6
hidden Markov model, 3, 257
hidden neuron, 301
Hirschberg's algorithm, 73
HMM, 3, 257, 339
HMM decoding, 260
HMM parameters, 263
Horner's rule, 35

Huffman coding, 196, 309, 311
hyperplane, 217

identity, 233
indels, 96
induction, 79
induction hypothesis, 162
insertion, 58
Inside algorithm, 268, 269
instance, 327
integer linear program, 284
intersection, 6
inverted index, 191
IthInRange, 194

Jaccard, 213, 216

Karp reduction, 337
key frame, 177
keyword tree, 46, 86, 115
Kleene star, 249
KMP, 25, 45, 48
Knuth-Morris-Pratt, 25, 339

Landau-Vishkin, 98
language, 245, 246, 249
large language models, 306
LCA, 127
LCE, 98, 102, 127
LCS, 130
leaf, 4
left shift, 36
Lempel-Ziv, 125, 278
length, 3
LF mapping, 167, 195, 291
line array, 201
line graph, 239
linear map, 298
linear time, 5
list merge, 161
local alignment, 65
locality sensitive hashing, 3, 211
longest common extension, 98, 102, 127
longest common substring, 119, 130
longest repeat, 119
lower bound, 318
lowest common ancestor, 127
LSH, 211
LSH for edit distance, 218

match, 249
match statistics, 133
maximal everywhere extension, 119
memo, 27
MergeSort, 76
minhash, 215
minimizers, 3, 231, 232
minimum cycle cover, 324
minimum edit script, 96

minwise independent, 213
move-to-front, 171, 196
MSA, 105, 335
MTF, 171
multi-graph, 276
multi-head attention, 305
multi-pattern, 45
multiple patterns, 45
multiple sequence alignment, 105, 335
multiset, 6

naïve exact matching, 11
neural networks, 300
next matching character rule, 17, 19
no-instance, 327
node, 4
node interval, 289
non-deterministic, 246, 248
non-terminal, 267
non-terminals, 245
NP-hard, 317

occurrence counts, 198
offset encoding, 83
offset vector, 84
optimization, 327
order minhash, 220
Outside algorithm, 272

pan-genomics, 275
parent, 4
parse, 247
parse tree, 252
particular density, 234
pattern, 11
perfect matching, 320
permutation, 213, 232
phase, 138
piece table, 202
pointer, 278
polynomial-time, 327
precomputation, 198, 339
predecessor, 193
prefix, 4
prefix graph, 318, 320
prefix sums, 176, 177
prefix-free, 311
prefix-free code, 309
prime number theorem, 38
probability, 7
production, 245
progressive alignment, 106
proper prefix, 4
proper suffix, 4

QuickSort, 76

Rabin-Karp, 35, 45, 52
radix sort, 158, 160, 161

For general queries, contact info@press.princeton.edu

© Copyright, Princeton University Press. No part of this book may be distributed, posted, or reproduced in any form by digital or mechanical means without prior written permission of the publisher.

RAM model, 86
random minimizer, 235
rank, 168, 171, 175, 177, 187, 196
read overlap, 232
recognize, 246
rectified linear unit, 301
recurrence, 60, 78
recurrence relation, 5
reduction, 328
regular, 239
regular expression, 249
regular grammar, 246
regular graph, 239
regular languages, 257
relative Lempel-Ziv, 278
ReLU, 301
repeated string, 322
restricted Hamiltonian path, 329
RLZ, 278
RNA folding, 267
RNA structure, 267
root, 4
rooted tree, 4
ropes, 203
RRR, 177, 198, 199
RRR data structure, 176
RRR tables, 179
RRR vector, 3
RRR vectors, 175
run-length encoding, 196

SCFG, 267
seed, 231
select, 175, 181, 187
semi-global alignment, 65, 67
seminumerical, 35
sequence, 333
sequence Bloom trees, 230
sequence-to-graph alignment, 280
set, 6
SETH, 87
Shift-And, 39, 339
shortest common supersequence, 70
shortest supersequence, 333
shortest superstring, 317, 327, 329
similarity, 211
simple, 4
simple exact match., 11
single hash filters, 230
sketch, 211
Skew algorithm, 159
solving a recurrence, 5
SP-score, 108, 335
Star algorithm, 108
states, 25, 257
stochastic context-free grammar, 267
string, 1
string alignment, 59
string interval, 289
string tree, 287
substitution, 58
substitution rule, 245
substring, 4
successor, 193
suffix, 4
suffix array, 3, 147, 148, 157, 231, 339
suffix link, 130
suffix tree, 3, 45, 102, 115–117, 125, 147, 339
suffix tries, 135
suffix-prefix, 16
super-characters, 158
superblock, 182, 197
supersequence, 333

terminals, 245
text, 11
text editor, 201
token, 297
tour, 317
traceback, 63, 70, 87
transformers, 297
traveling salesman problem, 317
traveling salesman tour, 319
traversal, 5
tree, 4
Tree BWT, 292
tree order, 291
trie, 86, 115, 135, 311
TSP, 317, 318
Turing reduction, 337

Ukkonen's algorithm, 135, 138
unavoidable words, 241
undecidable, 254
undirected graph, 4
Unicode, 1, 310
union, 6
unitigs, 276
universal hash, 230
universal hash function family, 212
universal hitting set, 241
unrestricted grammar, 254
UTF-8, 310

variation graph, 277
verifier, 328
vertex, 4
vertex cover, 333
Viterbi, 264
Viterbi algorithm, 260
Viterbi training, 264

wavelet tree, 3, 187, 195, 339
weight, 176
weighted Jaccard, 216, 217
window, 232
winnowing, 231

yes-instance, 327
YPrefix, 77
YSuffix, 77

Z algorithm, 12, 17, 20, 45
Z match, 35
Z value, 12
Z values, 31
Z-box, 13, 32
ZMatch, 22