DISCRETE MATHEMATICS FOR DATA SCIENCE

Jack Pope

A **Chapman & Hall** Book

CRC Press
Taylor & Francis Group

Discrete Mathematics for Data Science

Discrete Mathematics for Data Science provides an early course in both Data Science and Discrete Mathematics, focusing on how a deeper understanding of the former can unlock a more effective implementation of the latter. Students of Data Science come from a variety of disciplines, with Business, Statistics, Computer Science, Economics, and Psychology among the departments offering courses on the subject. Therefore, for many students, Data Science is considered a means of insight into a particular field of interest, with the study of its underlying discrete mathematics not a primary objective.

This book covers the topics of Discrete Mathematical Structures relevant to students of Data Science, offering a relevant and gentle introduction to both the theoretical and practical elements required to be a successful data scientist. The relaxed, accessible style makes it a perfect textbook for undergraduates.

Features

- Numerous exercises and examples.
- Ideal as a textbook for a Discrete Mathematics course for data science and computer science students.
- Source code and solutions provided as a supplementary resource.

Jack Pope has wrangled financial data since *Big Data* meant a big pile of floppy disks. He works at Investment Economics (aka, System Goats) providing system configuration, guidance, and training for organizations interested in data science infrastructure. He is also department coordinator for Computer Science and Data Science at North Hennepin Community College and chairman of the Twin Cities IEEE Computer Society.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Discrete Mathematics for Data Science

Jack Pope

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Designed cover image: vectorstock.com

First edition published 2026
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton, FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2026 Jack Pope

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged, please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC, please contact mpkbookspermissions@tandf.co.uk

For Product Safety Concerns and Information please contact our EU representative GPSR@taylorandfrancis.com. Taylor & Francis Verlag GmbH, Kaufingerstraße 24, 80331 München, Germany.

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

ISBN: 978-1-032-68722-3 (hbk)
ISBN: 978-1-032-68773-5 (pbk)
ISBN: 978-1-032-68775-9 (ebk)

DOI: 10.1201/9781032687759

Typeset in Latin Modern font
by KnowledgeWorks Global Ltd.

Publisher's note: This book has been prepared from camera-ready copy provided by the authors.

For Lisa

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

List of Figures xvii

List of Tables xxi

Preface xxiii

0.1 THE PURPOSE OF THIS BOOK xxiii

0.1.1 Context xxiv

0.1.2 History & Human Nature xxvi

0.2 PREREQUISITE KNOWLEDGE xxvii

0.3 UNIX xxvii

0.3.1 Coding Conventions xxviii

0.4 AUTHOR BIOGRAPHY xxviii

Acknowledgments xxix

SECTION I Problem Solving

CHAPTER 1 ▪ Your Mind: A Programming Environment 3

1.1 PSEUDO CODE 3

1.2 SYSTEMATIZING PROBLEMS VIA CODE 7

1.3 DATA CONFUSION 9

1.4 ERRONEOUS PROBLEMS & EXPENSIVE SOLUTIONS 12

1.5 PROBLEM-SOLVING AMID COMPLEXITY 13

1.6 CHALLENGES WITH PROBLEM SOLVING 14

1.7 MENTAL CONDITIONING FOR PROBLEM SOLVING 15

1.8 BOOKS FOR PROBLEM SOLVING 16

SECTION II Elements

CHAPTER 2 ■ Atoms & Abstractions 19

2.1 THE MEANING OF DATA 19
2.1.1 Atomic Data 19
2.1.2 Abstract Data 22
2.2 ABSTRACT FUNCTIONS 22

CHAPTER 3 ■ Numbers 26

3.1 NUMBER CLASSIFICATIONS 26
3.2 COMPUTER NUMBERS 28
3.3 FILE EXTENSIONS 29
3.4 COMPUTER NUMBER TERMINOLOGY 30
3.5 NUMBER BASE NOTATION 33
3.6 COMPUTER TERMINOLOGY VS MATH TERMINOLOGY 34

CHAPTER 4 ■ Number Conversion 36

4.1 BASE CONVERSIONS 36
4.1.1 The Power Method 39
4.1.2 The Remainder Method 41
4.2 BINARY CODED DECIMAL (BCD) 42
4.3 EXAMPLE PROBLEMS 43
4.4 MACHINE REPRESENTATION OF NEGATIVE NUMBERS 44
4.5 THE IEEE 754 FLOATING POINT STANDARD 46
4.6 SIMPLISTIC CONVERSIONS 48
4.7 CHECKING YOUR NUMBER CONVERSIONS 49

CHAPTER 5 ■ Digital Arithmetic & Logic 51

5.1 BINARY ADDITION 51
5.2 BINARY SUBTRACTION 53
5.3 BINARY MULTIPLICATION 54
5.4 BINARY DIVISION 55
5.5 BOOLEAN LOGIC 55

5.6 MACHINE APPROACHES TO MULTIPLICATION & DIVISION 58
5.7 BINARY SUBTRACTION USING ONE'S COMPLEMENT 60
5.8 BINARY SUBTRACTION USING TWO'S COMPLEMENT 61
5.9 SHIFTERS FOR MULTIPLICATION AND DIVISION 62
5.10 MULTIPLICATION BY LEFT-SHIFT & ADDITION 63
5.11 DIVISION BY RIGHT-SHIFT & SUBTRACT 64
5.12 PRECISION LOSS WITH RIGHT SHIFT 65
5.13 ARITHMETIC ON IEEE 754 FRACTIONAL VALUES 66
5.14 DIVISION WITH IEEE 754 FLOATING POINT NUMBERS 67

SECTION III Computational Logic

CHAPTER 6 ■ Propositional Logic 73

6.1 PROPOSITION 73
6.2 SYLLOGISMS 76
6.3 PROPOSITIONAL CONDITIONS 76
6.4 COMPOUND PROPOSITIONS 80
6.5 EQUATION MINIMIZATION 82
6.6 TIPS FOR EQUATION MINIMIZATION 83
6.7 SELECTED PATTERNS OF INFERENCE FOR EQUATION MINIMIZATION 84

CHAPTER 7 ■ Set Quantification 86

7.1 SETS 86
7.2 SET TYPES 88
7.3 QUANTIFICATION 90
7.4 SET RELATIONS 91
7.4.1 Domain and Range of Set Relations 94
7.5 TYPES OF SET RELATIONS 94
7.5.1 Inverse Relations 95
7.5.2 Identity Relations 95

7.5.3 Reflexive Relations 95
7.5.4 Symmetric Relations 96
7.5.5 Transitive Relations 96
7.5.6 Equality & Equivalence in Set Relations 97
7.6 VISUALIZING SET RELATIONS 98
7.7 VENN DIAGRAMS OF SETS 101
7.8 PROGRAMMATIC TREATMENT OF SETS AS LISTS 101

CHAPTER 8 ■ Proof 106

8.1 PROOFS FOR SETS 106
8.2 DEFINITIONS 107
8.3 SUBSET PROOFS 107
8.4 SET EQUALITY 109
8.5 SET EMPTINESS 109
8.6 CONDITIONAL PROOFS 110
8.7 PROOF TECHNIQUES 110
8.8 QUANTIFICATION PROOFS 111
8.9 RECURSION & PROOF BY INDUCTION 112
8.10 INDUCTION EXAMPLES 115

CHAPTER 9 ■ Computability 118

9.1 NP-COMPLETENESS 119
9.2 PROGRAMMING LANGUAGE ELEMENTS 122
9.3 LANGUAGE SYMBOLS 126
9.4 A MACHINE 129
9.5 PUSH DOWN AUTOMATON 135

SECTION IV Functions

CHAPTER 10 ■ Functions & Abstractions 141

10.1 AXIOMATIC FOUNDATIONS 141
10.2 PEANO'S AXIOMS 142
10.3 FUNCTIONS 145
10.4 RANGE, IMAGE, & CODOMAIN 145

10.5 TYPES OF FUNCTIONS 146
10.6 INVERSE FUNCTIONS 147
10.7 FUNCTION COMPOSITION 149
10.8 FUNCTION ABSTRACTION 150
10.9 CURRYING 152

CHAPTER 11 ■ Repetition & Recursion 155

11.1 RECURSION 155
11.2 INFINITE CYCLES 158
11.3 PEANO'S RECURSIVE LOGIC 159
11.4 RECURRENCE RELATION 161
11.5 TRACKING RECURSION USING A STACK 162
11.6 WHEN TO USE RECURSION 165
11.7 ADDITIONAL EXAMPLES OF RECURSIVE ALGORITHMS 166

CHAPTER 12 ■ Lambda Calculus 169

12.1 LAMBDA λ CALCULUS 169
12.2 FUNCTION APPLICATION 172
12.3 APPLICABLE ORDER 175
12.4 REDUCTION & CONVERSION 176
12.4.1 Other Conversions 179
12.5 COMPUTING WITH LAMBDA CALCULUS 179
12.5.1 Boolean Functions 181
12.6 COMBINATORS 182

CHAPTER 13 ■ Algorithm Complexity 185

13.1 ALGORITHM ANALYSIS 185
13.2 BIG-O NOTATION 187
13.2.1 *Big-O*: The Worst Case Scenario 190
13.2.2 Worst vs Average vs Best Case Scenarios 191
13.2.3 Algorithm Efficiency vs Elegance 192
13.2.4 Big-O Cases 193
13.3 TIMING COMPUTATION 197

SECTION V Data Organization

CHAPTER 14 ▪ Data Organization 203

14.1 DATA ORGANIZATION 203
14.2 ORDERING 204
14.2.1 Element Access 204
14.3 COLLECTION OPERATIONS 204
14.4 RELAXED PROPERTIES 205
14.5 UNCONNECTED DATA COLLECTIONS 206
14.6 LINEAR STRUCTURES 206
14.7 BRANCHED STRUCTURES 207
14.8 DATA COLLECTION SUMMARY TABLE 208

CHAPTER 15 ▪ Unconnected Data 210

15.1 BAGS VERSUS SETS 210
15.2 RANDOM SELECTIONS OF DATA 212
15.3 UNIFORM RANDOM DISTRIBUTION 213
15.4 PSEUDO-RANDOM NUMBER FUNCTIONS 213
15.5 EVALUATING PSEUDO RANDOM NUMBERS 215

CHAPTER 16 ▪ Linear Structures 218

16.1 PROPERTIES OF LISTS 218
16.2 STANDARD LIST OPERATIONS 220
16.3 RANDOM ACCESS & SORTED LIST 222
16.4 LISTS OF LISTS 224
16.5 LISTS OF ABSTRACT DATA TYPES 225
16.6 LIST IMPLEMENTATION OF A STACK 226
16.7 POSTFIX EXPRESSIONS 229
16.8 LIST IMPLEMENTATION OF A QUEUE 230
16.9 SIMPLE SEARCHING & SORTING ALGORITHMS 232

CHAPTER 17 ▪ Branched Structures 236

17.1 REVIEW OF LIST OBJECTS 236
17.2 TREES 239

17.2.1 Computations for Trees 240
17.2.2 Binary Tree Traversal Patterns 241
17.2.3 Parse Tree Evaluation 242
17.2.4 Binary Search Tree 246
17.2.5 Tree Glossary 247
17.3 GRAPHS 248
17.3.1 Computations for Graphs 248
17.3.2 Graphs as Relations 249
17.3.3 Quantifying an Undirected Graph 251
17.3.4 Graph as Adjacency Matrix 252
17.3.5 Graph as Adjacency List 253
17.3.6 Graph Traversal 255
17.3.7 Graph Paths 256
17.3.8 Computing Distance 257
17.3.9 Generating Graph Objects 259
17.3.10 Graph Glossary 260

SECTION VI Data Analysis

CHAPTER 18 ■ Counting: Permutations & Combinations 267
18.1 BASIC COUNTING 267
18.2 FACTORIALS 269
18.3 PERMUTATIONS 269
18.4 COMBINATIONS 270
18.5 MULTISETS 272

CHAPTER 19 ■ Probability & Statistics 275
19.1 PROBABILITY 275
19.1.1 Sample Space & Events 276
19.1.2 Mutually Exclusive Events 277
19.1.3 Independent Events 278
19.1.4 Joint Probability 279
19.1.5 Conditional Probability 280

19.2 STATISTICS 281
19.2.1 Measures of Central Tendency 281
19.2.2 Measures of Variability 282
19.2.3 Measures of Position 284
19.2.4 Weighted Mean 285

CHAPTER 20 ■ Multivariate Analysis 289

20.1 THE PEARSON CORRELATION COEFFICIENT 289
20.2 REGRESSION 291
20.3 LINE FITTING 293
20.3.1 Perspective on b_1 295
20.3.2 More Sums of Squares 296
20.4 COEFFICIENT OF DETERMINATION, R^2 296
20.5 MULTIPLE LINEAR REGRESSION 297
20.6 TESTING THE SIGNIFICANCE OF REGRESSION VARIABLES 298
20.6.1 Analysis of Variance Table 300

CHAPTER 21 ■ Resampling 302

21.1 RANDOM SAMPLES 302
21.2 RESAMPLING 306
21.3 EVALUATING ESTIMATES 307

CHAPTER 22 ■ Information Theory 310

22.1 INFORMATION THEORY 310
22.2 ENTROPY 313
22.3 MESSAGES OF MULTIPLE SYMBOLS 314
22.4 SYMBOL PROBABILITIES 315
22.5 JOINT ENTROPY 316
22.6 CONDITIONAL ENTROPY 317
22.7 MUTUAL INFORMATION 317
22.8 NOISE, ERROR DETECTION & CORRECTION 318

22.9 CHANNEL CAPACITY 319
22.10 DATA COMPRESSION & HUFFMAN ENCODING 320

CHAPTER 23 ■ Data Dimensions 325

23.1 HIGHLY DIMENSIONAL DATA 325
23.2 MULTICOLLINEARITY 327
23.3 INVESTMENT PORTFOLIO EXAMPLE 329
23.4 SAMPLE DENSITY 331
23.5 GENERALIZATION VERSUS OVER-FITTING 334

SECTION VII Appendix

APPENDIX A ■ Coding Conventions 341

A.1 *epop* 341
A.2 Postfix 342
A.3 Postfix Operations 342
A.4 Function Definitions 343
A.5 Data 343

APPENDIX B ■ epop's General Syntax 346

B.1 epop's General Syntax 346
B.2 Data Containers 347
B.3 Postfix 347
B.4 Words As Arguments 347
B.5 Defining New Words 351
B.6 Tick Words 351
B.7 Inline Words 353
B.8 Word Overrides 353
B.9 Deep Copy vs Shallow Copy 354
B.10 Stack Checking & Error Prevention 354
B.11 Local Namespaces 355
B.12 epop Word Summary 356

APPENDIX C ▪ Resources 357

C.1 Online Resources 357

C.2 Source Code - Online 357

Bibliography 359

Index 363

List of Figures

1	Interdisciplinary data science contexts.	xxiv
2	Discrete versus continuous information.	xxv
3	A discrete snapshot of a continuous reality.	xxvi
4	The synthesizer built by Jack Sr.	xxx
1.1	Your mind: A programming environment.	4
3.1	Number classifications.	27
3.2	A typical hierarchy of data types for a programming language.	28
3.3	A compiled program's machine code file in hexadecimal format.	31
3.4	Program output representing ASCII codes.	33
4.1	Using the Unix bc command to change the number base.	49
7.1	(a) An intersection of sets. (b) A union of sets.	102
7.2	(a) A difference of sets. (b) A difference of sets.	102
7.3	(a) The complement of intersecting sets. (b) The complement of a union of sets.	103
9.1	A parse tree for a simple function.	123
9.2	Information transmission, reception, and processing.	132
9.3	State diagram for a machine that matches a word.	133
10.1	Function versus non-function.	146
10.2	Types of functions.	148

11.1 A call tree of the conditions of the Fibonacci sequence. 167

12.1 Abstract-syntax tree: λ is the abstraction operator. eval is the application operator. 173
12.2 Abstract-syntax tree for two successive applications. 174

13.1 *Big-O* time complexity comparisons. 190

14.1 A set of unconnected objects. 206
14.2 A connected list of objects. 207
14.3 (a) A tree. (b) A graph. 208

15.1 A uniform probability distribution of 6 random samples. 214
15.2 The "uniform" distribution of output from a pseudo-random number generator. 216

16.1 A connected list of objects. 219
16.2 A connected list of characters. 219
16.3 An array of characters. 223
16.4 A list of lists. 225
16.5 A sequence of currency symbols to represent abstract objects. 226
16.6 Three separate stacks, before operations. 228
16.7 Three separate stacks, after one operation. 228
16.8 Three separate stacks, after two operations. 229
16.9 A combined stack (before & after one operation). 229
16.10 A hybrid circular/priority queue of p data and q indices. 232

17.1 A list of objects. 237
17.2 A list of objects. 237
17.3 A binary tree. 237
17.4 (a) A binary tree. (b) A graph. 238
17.5 Isomorphic graphs. 239
17.6 A file system. 240
17.7 Binary tree traversal patterns. 242

17.8 A parse tree of $(A - B) + (C \times D)$. 243
17.9 A parse tree of $(A - B) + (C \times D)$. 243
17.10 Evaluating a binary tree on a stack in level-order. 245
17.11 A binary search tree: (a) before (b) and after, inserting the new element "11". 246
17.12 A sorted list of numbers (ascending). 247
17.13 A relation of two objects: B depends on A. B is directed to A. 250
17.14 Relations of objects. 250
17.15 A reflexive relation. 250
17.16 An undirected graph representing the bordering states of New England. 251
17.17 A complete graph. 251
17.18 An incomplete directed graph. 252
17.19 Euclidean distance. 257

20.1 Positive, none, & negative linear correlation. 292
20.2 A linear fit of marathon times versus temperature. 292

21.1 A uniform distribution. 303
21.2 A normal distribution. 304
21.3 An approximately "normal" distribution of output from a pseudo-random number generator. 305
21.4 More of a "normal" distribution of output from a pseudo-random number generator. 305
21.5 Random sampling of a variable. 306
21.6 Non-random sampling of a variable. 306

22.1 Information transmission, reception, and processing. 311
22.2 Level 1 and leaves of a Huffman tree. 321
22.3 A Huffman tree. 321

23.1 A two dimensional plot, at a single period in time. 332
23.2 Model performance versus dimensions (hypothetical). 334
23.3 The S& P 500 Stock Index. 335

23.4 A hypothetical view of portfolio risk. 337

A.1 Entry sequence (left to right) & corresponding stack sequence (top to bottom). 345

A.2 A stack trace of an operation entry sequence. 345

List of Tables

3.1 Number classifications 27
3.2 Octal file permission combinations in a Unix filesystem 32
3.3 File permissions for Unix's three user types 32

4.1 Numbers in different bases 37
4.2 Place values of base 2 38
4.3 Columnar correspondence of a number in binary 39
4.4 A Binary Coded Decimal lookup table 42

7.1 Relation where a > b 99
7.2 An adjacency matrix 101

11.1 Function calls structured as a stack in memory 162

13.1 Magnitudes of algorithm expansion 188
13.2 Three algorithms that compute a mean. *Big-O* & operation totals 189

14.1 Attributes of data organizations. 208

17.1 Complete binary tree to list mapping 241
17.2 An adjacency matrix 253

20.1 An analysis of variance table (ANOVA) 300

23.1 Two symbols at a snapshot in time, with rate of change (ROC) the only dimension 331

23.2 Sample densities, with constant sample size 333
23.3 Sample densities, with increasing sample size 334

Preface

0.1 THE PURPOSE OF THIS BOOK

Does precision mean relevance? Can reasoning exist without data? For that matter, what is data?

With those questions in mind, this book serves an early course in mathematical thinking for data science. Because data science is interdisciplinary, students of data science come from a variety of disciplines. Interdisciplinary motivations are fueling the emergence of data science initiatives in various academic disciplines, with Business, Statistics, Computer Science, Economics, and Psychology among the departments offering studies in data science. For many students, data science is considered a means of insight into a particular field of interest, with the study of its discrete mathematics not a primary objective.

Courses in introductory data science and discrete mathematics tend to be ambitious. Introductory data science courses are seen requiring students to have prerequisite knowledge of statistics and computer programming, and a first course in discrete mathematics generally resembles a buffet of mathematical topics. Students are often preoccupied with enough abstraction to miss the basic empirical essence of data science. This book aims to make these connections.

In particular, those pursuing data engineering and high-performance computing for data science should be aware of the foundational aspects of data and computation present in this book.

I wrote this book to give interdisciplinary students of data science a solid foundation in discrete mathematics for data science, connecting fundamental and abstract concepts for elementary problem solving.

0.1.1 Context

It is imperative to respect the interdisciplinary backgrounds of data science practitioners, as domain expertise ensures that analysis is kept to relevant contexts.

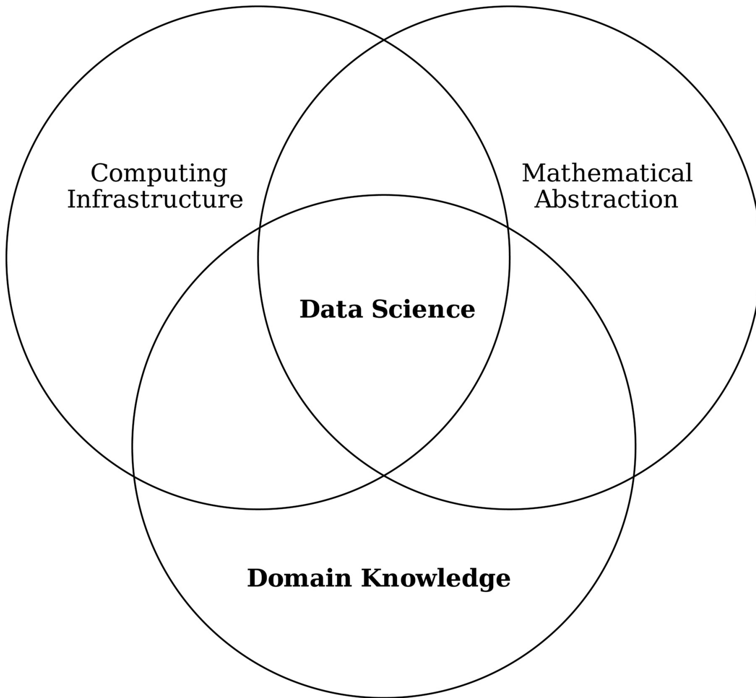


Figure 1 Interdisciplinary data science contexts.

Context is important for the classification of data, for determining which data values belong in a certain class and which do not. In this age of *Big Data*, much irrelevant data and noise can amount to a significant waste of computing resources, and a much higher cost when mistaken for relevant data.

For example, the history data scientist decides to count battlefield casualties that include soldiers too wounded for further combat. The business data scientist counts only those sales transactions where the customer's payment has been received. The data engineer supports the domain

specialists, filling voids and minimizing redundancies among maintained data sets.

The classification of data requires details with respect to its source, type, organization, and sampling rate. These details are largely matters of domain expertise. In the process of data analysis, raw data must be ingested from a source and format, and computational results then generated and encoded for the next stage of analysis and eventually storage.

Data scientists use programming tools for data conditioning, converting data to various formats. When data inconsistencies are encountered that the tools fail to process correctly, the data scientist is forced to become a student of low-level character encoding. Therefore, machine representations of numbers and text are considered important. Machine numbers, data types, and numerical representations are a few examples of discrete mathematics in practice.

Computers get information from the outside world in discrete periodic samples. These samples are usually incomplete representations of *real world* data, being limited to a sampling rate per second of continuous data or a data set sampled from a much larger population of data. Figure 2 depicts discrete samples of a continuous data waveform, such as from audio or video input. Each of those discrete samples in Figure 2 may be considered a snapshot in time, as conveyed by Figure 3.

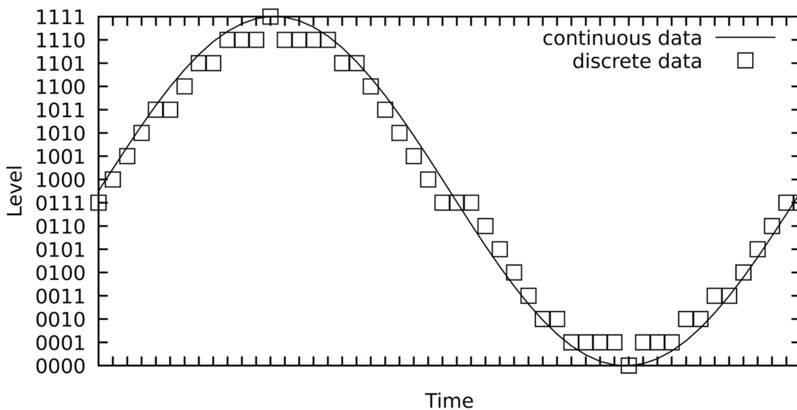


Figure 2 Discrete versus continuous information.

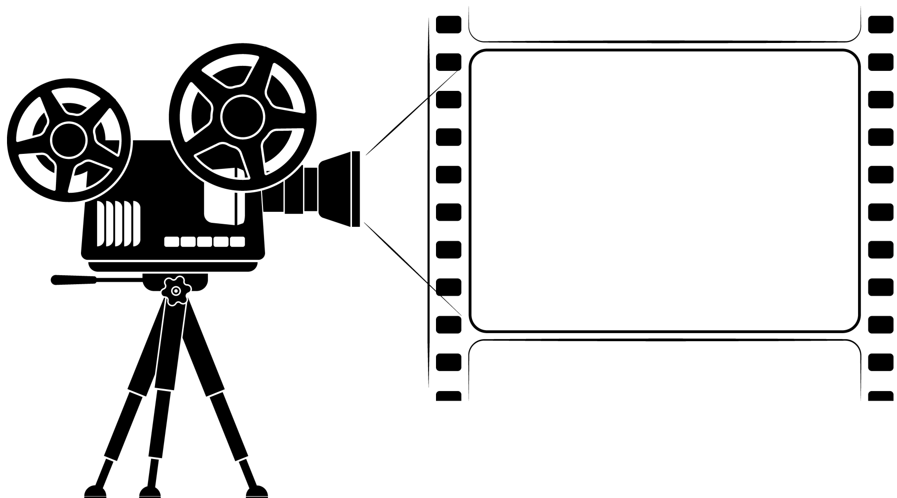


Figure 3 A discrete snapshot of a continuous reality.

0.1.2 History & Human Nature

Formal courses in discrete mathematics for computer science first appeared in the U.S. university curricula in the late 1970s. The rationale was that students of computer science needed exercise in mathematics related to applications and architectures of digital computers. The curriculum was later codified by the Association for Computing Machinery in a joint effort with the Mathematical Association of America.[1] Early topics generally included analysis of algorithms, counting, propositional logic, and set theory.

Over the past forty years, topics in discrete mathematics textbooks have expanded as computing topics have burgeoned, with college textbooks for discrete math now including several more subjects than were initially codified by ACM.[2]

Expanded coverage includes: automata theory, theory of computation, cryptography, information theory, and graph theory with various heuristics.

[1] B. Marion, "Discrete mathematics: Support of and preparation for the study of computer science," *Journal of Computing Sciences in Colleges*, vol. 16, pp. 190–199, 2000. DOI: https://dl.acm.org/doi/10.5555/357697.357730.

[2] D. Stanat and D. McAllister, *Discrete Mathematics in Computer Science*. Prentice Hall Englewood Cliffs, N.J., 1977, vol. 16, pp. 190–199 (Stanat & McAllister's text may be considered the first definitive textbook on discrete mathematics.)

The situation for students new to data science is, perhaps, mired by introductory courses that expose advanced topics too early. For example, it is not uncommon to see in an introductory data science text such sophisticated topics as *machine learning*, including *cluster analysis* and *neural networks*. This presupposes that the underlying principles and algorithms are understood by students, who may have just had their first exposure to statistical regression. Keep in mind that students of data science coming from varied interdisciplinary backgrounds often do not have adequate experience with computational systems to boldly go about such problem solving.

0.2 PREREQUISITE KNOWLEDGE

This book assumes no prior background in computer programming or statistics. The student should be comfortable with algebra at the college level. Some prior knowledge of the Linux command-line terminal would be helpful as some example programs may be run from the command-line.

0.3 UNIX

This book refers from time to time to the Unix operating system. Internet servers hosting *Big Data* are overwhelmingly Linux/Unix systems. These, in turn, are accessed by analysts remotely through various types of application interfaces, including database connections and Internet browsers. Derived from Unix is Linux, the most ubiquitous operating system in the world today. Of the top 500 supercomputers in the world, all run the Linux operating system.[3] Most Internet servers are also hosted on Unix or Linux.

More data is being both generated and stored in a Linux/Unix environment than people realize. But it is easy to take for granted the architecture of our programming environment. When “properly” coded, we expect our programs to work smoothly, often oblivious to the extraordinary yet beautiful complexity of the underlying system. Some low-level aspects of such system complexity are illuminated by this book, which should be of particular interest to students of data science planning to work in data engineering.

[3] See `https://top500.org`

0.3.1 Coding Conventions

Coding is used in this book as a vehicle for learning the subject matter. You can learn the streets of the big city by studying a map. You can additionally walk the streets yourself and learn your way around through experience. To program is to walk the streets. Likewise, programming a statistics routine is a *learning-by-doing* approach that seems to be effective for many students.

Coding allows us to condense otherwise verbose descriptions of concepts into a relatively compact, albeit abstract, perspective. For example, code that tests conditions is used to augment concepts of mathematical proof.

To an extent, this book uses a Forth-like code called *epop* for its compact, yet natural language structure. *epop* is designed as a teaching tool to help students learn discrete math for data science, with data processing in mind. *epop*'s postfix notation allows fairly concise mathematical expressions, which leads to relatively compact syntax. More about this coding convention is presented in Appendix A, as well as in the online resources noted in Appendix C.

0.4 AUTHOR BIOGRAPHY

Jack Pope has wrangled financial data since Big Data meant a big pile of floppy disks. He works at Investment Economics (aka, System Goats) providing system configuration, guidance, and training for organizations interested in data science infrastructure. He is also department coordinator for Computer Science and Data Science at North Hennepin Community College and chairman of the Twin Cities IEEE Computer Society.

Acknowledgments

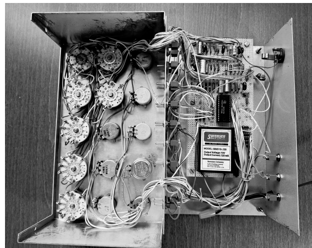
I am thankful to my colleagues and friends who contributed to this book in some way, whether by suggesting content, editing, or with encouragement.

In particular, the following individuals (in alphabetical order) deserve acknowledgment for their significant input: John Augustine, Phil Colangelo, Bruce Hanson, Yectli Huerta, Siva Jasthi, Chris Koehnen, Stephan Kolassa, Gary Lynch, Stephen Maresh, Ahmed Naumaan, Dipankar Saha, Chris Sahr, and Ted Volk.

This book has benefited from feedback in one form or another over the years from eager students of Computer Science and Data Science at North Hennepin Community College. My associates in the Twin Cities IEEE Computer Society have also been influential and encouraging.

Finally, I want to remember my father, Jack Sr., who introduced me to the importance of mitigating complexity. With an interest in guitar and electronics, in the early 1980s he constructed an audio synthesizer. This device, pictured in Figure 4, has fifteen knobs (potentiometers) on the top, eighteen switches on one side, and a few more on the opposite side. All told, the knobs and switches allow for a large number of sound effects considering the mixing of the knobs at various levels. Such was the extent of his quest for sound. Wiring such complex circuits was a challenge that required ongoing testing and calibration. In the midst of it all was this refrain from the basement workbench: "Keep it simple!" Translation: Break challenging problems into small do-able tasks.

(a)



(b)

Figure 4 The synthesizer built by Jack Sr. (a) and its electronics (b).

I

Problem Solving

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

CHAPTER 1

Your Mind: A Programming Environment

Abstract

Why don't you eat your book? That would be one way to absorb information. Seriously, problem-solving skills along with understanding of facts in context (knowledge) come with working out problems in various contexts. Otherwise, we can *thoughtlessly possess knowledge* not much differently than possessing a reference source. This is a trend these days, as solutions to simple programming tasks are easily available on the Internet. The student need not be so introspective about what they actually know when information is so easily and freely available.

However, when tasks become more challenging, only those experienced in problem solving will be able to handle the situations. This chapter is meant to prepare the student for the subject matter by introducing fundamental concepts in problem solving and coding.

1.1 PSEUDO CODE

Framing problems in our natural language is a first step in problem solving. We can solve problems or complete tasks following a set of instructions such as a cooking recipe. These instructions can be translated

DOI: 10.1201/9781032687759-1

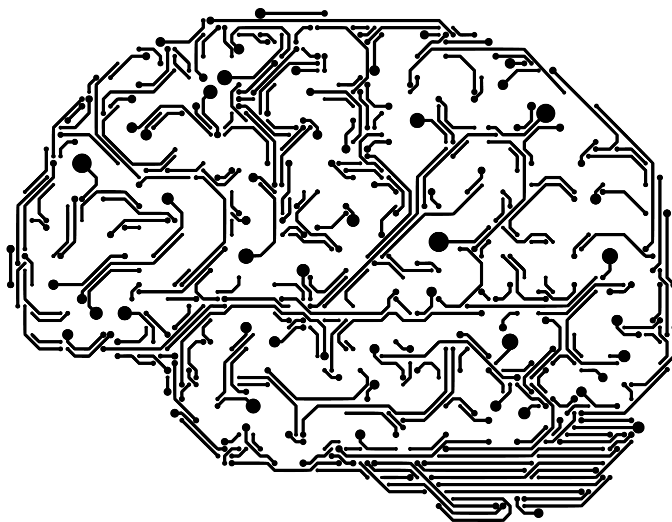


Figure 1.1 Your mind: A programming environment.

into a form that is acceptable to a machine in order to automate the task at hand. In the computer context, recipes are called algorithms and its words are codes. A general recipe as such can be referred to as *pseudo-code*. Pseudo-code can be defined as a general expression of an algorithm, minus the idioms of a programming language that are unnecessary to convey the algorithm's meaning.

There are a variety of pseudo-code styles, paralleling various programming languages. For example, the Forth programming language dovetails with its style of pseudo-code because Forth already has a natural language structure. Depending on the names that you give to words that you define in Forth, the syntax can read more like a spoken sentence than machine code. (As mentioned earlier, this book uses a Forth-like code called *epop*.)

A good overview of the pseudo-code aspects of Forth is presented by Leo Brodie, in chapters two and eight of his book *Thinking Forth*, which he made freely available online at https://thinking-forth.sourceforge.net.[1]

[1]L. Brodie, *Thinking Forth*. Punchy Publishing, 2004, ISBN: 0976458705. DOI: https://thinking-forth.sourceforge.net.

In *Thinking Forth*, chapter eight has a section on minimizing control structures. Imagine if you were reading a sentence in an article and upon coming to a certain word, *summation*, you then needed to refer to another paragraph somewhere else in the article about *add*. Such detachment disrupts the flow of reading. Ideally, we would come into the context of *summation* more sequentially. Accordingly, word naming should also facilitate meaning. For example, in Forth as well as in *epop*, we can define the words *add* and *summation* to add three numbers as follows:

```
: add + ;
: summation add add ;
```

As you can see, the word *summation* is defined in terms of the word *add*. In *epop*, these words can alternatively be defined in a more postfix manner as follows, with { braces } containing a particular expression of a word's definition.

```
{ + } add WORD
{ add add } summation WORD
```

Either way, the invocation of *summation* is after the numeric arguments (3, 5, and 7), in the postfix form seen here:

3 5 7 summation (→ 15)

In the field of linguistics, it can be argued that postfix expressions are most efficient. Consider languages that use *object-subject-verb* grammar, similar to the pattern in American Sign Language. That order makes sense for sign language, given the need to economize the effort of translating vocal sounds into hand signs.

Consider these expressions for object-subject-verb (OSV) versus subject-verb-object (SVO):

OSV: "Straw goats eat." ↔ SVO: "Goats eat straw."

OSV presents the object and subject as arguments for the verb, emphasizing the object first.

In this manner, Forth's postfix notation allows fairly concise mathematical expressions, translating into compact syntax. Word (function) sequences are self-documenting, resembling a natural written language. This frees the mind to focus on problem solving. In contrast, coding

in C-like languages requires the mind to interpret connections between separate scopes of syntax. Consider the following C code:

```
int add(int a, int b) {
  int result = a + b;
  return result;
}

int summation(int a, int b, int c) {
  int result = add(a, b);
  return add(result, c);
}

int main(int argc, char *argv[ ]) {
  int a = atoi(argv[1]);
  int b = atoi(argv[2]);
  int c = atoi(argv[3]);
  print(summation(a, b, c);
  return 0;
}
```

As can be seen, these C functions for *summation* and *add* are relatively verbose and do not convey a sequential flow of logic. Such are the traits of procedural programming languages, such as C, relative to functional programming languages.

Arguably, Forth has characteristics of the *functional programming* paradigm. Functional programming emphasizes *what* task is to be accomplished, in contrast to *procedural programming*, which deals more with *how* a task is to be accomplished. Long before the fad of functional programming, Forth was *functional.* Being a work of Charles (Chuck) Moore in the late 1960s, Forth may have taken inspiration from what might be considered the first functional programming language, Lisp, which was originated by John McCarthy at the Massachusetts Institute of Technology in 1958. Chuck Moore was a student of John McCarthy.[2]

Why not use Python or R? With Python and R, the student can possess the capabilities of built-in abstract functions, without necessarily understanding the problem at hand. Because Python and R libraries exist for just about any problem, these libraries are recommended for use only after fundamentals are understood by wording them in a more natural language, such as Forth.

[2] J. McCarthy, "History of lisp," *Special Interest Group on Programming Languages*, vol. 13, issue 8, pp. 173–185, 1978, ISSN: 0362-1340. DOI: https://dl.acm.org/doi/10.1145/960118.808387.

1.2 SYSTEMATIZING PROBLEMS VIA CODE

If you understand the problem at hand, then you can begin coding to automate its solution. The data might change, but the variables and functions will remain the same.

Knowing the pattern of a solution, one can work backwards to inductively build the logic that generates the solution.

For example, suppose that the problem is to compute the sum of the squares of a list of numbers. In terms of mathematical symbols of abstraction, a sum of squares can be expressed as ΣN_i^2. If you are uncertain of what all that means, then you may assume that the task is complicated. Putting the task into words makes it somewhat easier, and if those same words amount to a runnable program, then wonderful!

In the *epop* programming language, we can work out a sum of squares routine with word definitions as follows.

Square a number by duplicating it in order to multiply it by itself:

```
{ DUP * } square WORD
```

Square a number, swap it with the next number on the list, square that next number, and then add the two squared numbers together:

```
{ square SWAP square + } sumsquares WORD
```

Invoke the sumsquares operation using input numbers 2 and 3 and display the result:

```
2 3 sumsquares   ( → 13 )
```

To verify that the results from a program are correct, we can devise an assertion function to check algorithm results against expected results. For example:

```
{ 2 3 sumsquares 13 EQ } assert WORD
assert   ( → 1, true )
```

The expression of EQ is a test of equivalence.

Invariably, problem solving in data science entails functions that initially take input data and then output some data as a result. Within testing frameworks common to computer science, input and output are often referred to as *preconditions* and *postconditions*, respectively.

A function's precondition includes function arguments and the initial values of any internal variables. A function's postcondition includes the final values of internal variables and any related output.

What follows are the steps for a coded assertion to test a function, $f(n) = n + 1$, that takes an input variable n that has a value of 3, the precondition, and makes it 4, the postcondition.

Set a variable n to the value of 3:

$$3\ n\ !$$

The "!" symbol sets a variable to a value. The "@" symbol will be used to get the value of the variable n. The order of operations for ! and @ is: "value variable !" and then "variable @".

Define a function f that increments (adds 1 to) the variable n:

$$\{\ \mathrm{n}\ @\ 1 + \mathrm{n}\ !\ \}\ \mathrm{f\ WORD}$$

Define function *assert* that tests whether or not the postcondition of n is satisfied:

$$\{\ \mathrm{f\ n}\ @\ 4\ \mathrm{EQ}\ \}\ \mathrm{assert\ WORD}$$

Invoke assert. An output of 1 denotes *true* and an output of 0 denotes *false*:

$$\mathrm{assert} \quad (\rightarrow 1,\ \textit{true}\ )$$

When defining words, we should heed the problem-solving concept of breaking big problems into small problems. In that way, words that we define will have a more specific and identifiable purpose. In programming talk, this is known as *factoring*.

When we factor code, such as a word or function, we break it into other words that specifically deal with a component task of the general problem. Factoring function definitions becomes more important as the definitions become extensive and complex.

Such factoring is reminiscent of algebraic factoring, such as:

$$x^2 + 3x + 2 \rightarrow (x + 2)(x + 1)$$

where the factors are: $(x + 2)$ and $(x + 1)$.

For example, to process any number of inputs, a complicated unfactored version of sumsquares might contain the following definition: (*Alert! The following code is intentionally complicated.*)

```
{ { DUP * >R } GSC LREPEAT WAIT { R> + } GSC LREPEAT }
```

Such complex word definitions make execution control and problem isolation more difficult. However, the extra challenge can be lessened by factoring operations into smaller word definitions. For example:

```
{ DUP * >R GSC 0 > squares ? } squares WORD
{ R> + GRC 0 > sums ? } sums WORD
{ squares sums } sumsquares WORD
0 1 3 5 7 9 sumsquares   ( → 165 )
```

Note that some of these terms are predefined in *epop*. GSC and GRC are counters. >R and R> are operations that move data to and from a container called the *return stack* (we cover stacks in the *Linear Structures* chapter). The squares function repeats until GSC is 0, as determined by the *?* operator (symbolizing an *if* statement). Similarly, the sums function repeats until GRC is 0.

The above code will make more sense as we move through the text. In any case, you may wish to refer to the coding conventions discussed in Appendices A and B.

In programming we want instructions that when executed only affect the parameters required by the instructions, and not those of other instructions. The term for this property in the context of programming is *orthogonality*. A more orthogonal instruction set better avoids *side effects* or unintended results, including output that was expected from different instructions and other mysterious errors.

When we define multiple complex words with similar functionality, we leave the door open to side effects. The situation is compounded when operations involve a complex nesting of data structures. Then we are more likely to see resource conflicts, such as when multiple operations simultaneously vie for the use of a single data stack. In such cases, it would be better to give each function instance its own internal stack.

1.3 DATA CONFUSION

Is Big Data always necessary, or are we merely amusing ourselves by collecting more and more data? Perhaps it is human nature to be fascinated

by mystery, and for that matter, complexity. This is reminiscent of the gold miners in the American west in the 1800s who spent their lives digging for gold, ofttimes to no avail. They had gold fever. Nowadays, data scientists have *data fever*.

The motives for data mining are not always about finding nuggets of information. There are often institutional motives as well. The marshaling of resources and labor to deal with great amounts of data may both influence and depend on institutional budgets, business models, and other managerial motives. This means that the rationale for an acceptable data feature set may not be clear.

In situations where the data is huge and complex, it is easier to camouflage mistakes, including:

- Discarding subsets of data that contradict an objective (aka, *cherry picking*).
- Inflating data complexity/redundancy to amuse/confuse/impress stakeholders.
- Mistaking precision and process detail for relevance.

In the topic of *highly dimensional data* (which we will cover in more detail in the *Data Dimensions* chapter), data transformations have the potential to multiply the number of predictors (factors). Consider:

$$\text{total number of factors} = \text{number of factors} \times \text{number of transformations}$$

If we additionally want to include lags for each time-series record and transformation, then our factor count becomes:

$$\text{total number of factors} = \text{number of factors} \times \text{number of transformations} \times \text{number of lags}$$

One way or another we end up cursed by this exponential expansion of dimensionality, whereby the number of factors exceeds the number of observations, making analysis statistically unsound (the infamous *curse of dimensionality*[3]).

[3] R. Bellman, R. Corporation, and K. M. R. Collection, *Dynamic Programming*, ser. Rand Corporation research study. Princeton University Press, 1957. (The curse-of-dimensionality term was perhaps originated by Bellman in the context of dynamic programming.), ISBN: 9780691079516. [Online]. Available: https://books.google.com/books?id=wdtoPwAACAAJ.

Despite redundant data, there is a certain *fatal conceit* that comes with having computational power, or thinking that we will find the results we seek if we only have more resources. Despite hoards of information, or perhaps because of it, unforeseen phenomena emerge amid complexity against the best efforts of organizations.

For example, in my book review of *Keeping Up with the Quants* by Davenport and Jinho Kim (2013), I make the following points:[4]

> Problem solving is not the core of analytical thinking. A solution in a vacuum solves nothing. Problems must be framed correctly to reflect a relevant context. Then the results need to be communicated in a manner which facilitates correct interpretation.
>
> In any organization, analytics is not plainly mechanical, but a communications process… you will want to maintain a communications framework to keep analytical work relevant to project goals. As such, there must be effective interaction between analysts and non-quantitative stakeholders, who might include managers, customers, suppliers, and support personnel.
>
> …
>
> Consider also the description of the famous Black-Scholes options-pricing model. While the model was derived with rigorous statistical tests, it rests on flawed assumptions such as risk-free rates, volatility based on arbitrary periods, and a normal movement of prices that is small and random, with rare events considered irrelevant.
>
> As we know with investment systems, 99% accuracy is meaningless if annihilation occurs 1% of the time. The practitioners of Black-Scholes at Long Term Capital Management found this out the hard way, at best applying the formula and with excessive leverage, and at worst applying even more dubious methods behind the marketing veil of the famous formula.

[4] J. Pope, "Keeping up with the quants (book review)," *Foresight: The International Journal of Applied Forecasting*, vol. 38, Fall issue, pp. 38–40, 2013. DOI: https://ideas.repec.org/a/for/ijafaa/y2013i31p38-40.html.

> Maybe the authors should have used Black-Scholes as an example of what not to do, say next to their discussion of AIG's [American International Group's] mistaken analysis of credit default swaps. Such discussion would exemplify the danger of ignoring context and of applying erroneous assumptions.
>
> ...
>
> Human nature is subject to a fatal conceit from possessing things that are rare, elegant or impressively complex, like some quantitative methods. Such conceit can lull us to disaster. From Enron to LTCM, from Lehman to AIG, forecasting precision took precedence over forecast relevance, as some successes with impressive quantitative methods fueled hubris and recklessness.

1.4 ERRONEOUS PROBLEMS & EXPENSIVE SOLUTIONS

Apart from seeking an elusive solution to a well-understood problem, the problem at hand may not be understood and should be reconsidered.

Both difficult problems and erroneous questions can masquerade as non-deterministic problems. Examples of erroneous endeavors include attempting to prove that all scores are above average, isolating patterns among the decimal digits in irrational numbers, and other tasks that attempt solutions to problems that are not understood. These are vexing issues that are explored in more depth in the *Computability* chapter.

The alchemists of ancient times sought to synthesize gold via chemistry. Such a solution is far from economical, even though the atomic structure of gold is known. It is not just a matter of having the right tools. When a problem is understood, one can then fashion an appropriate tool or practical algorithm. For example, in competitive industries such as Electronics, valuable devices are regularly reverse-engineered.

For the sake of progress, it is best to identify unsolvable problems and erroneous assumptions early.

1.5 PROBLEM-SOLVING AMID COMPLEXITY

Only when a problem is understood can its solution be succinct or elegant. Conversely, when a problem is merely known to exist, but not understood, the work toward a solution may entail numerous failed attempts of experimentation. Sometimes it takes such experimentation in order to intellectually isolate the true nature of a problem and then to provide an appropriate solution. This may be undesirable when experimentation is costly. In computer programming, usually the most significant cost is your time.

Code bloat in a program is a sign that either a single problem is not understood or that the program is actually addressing various distinct problems.

The recurrence of a somewhat-the-same section of code throughout a program suggests opportunity to condense those sections into a single abstract/generalized module, in one place, that can be used (imported, called, extended, ...) in other places. Rather than exhausting all possible solutions for all possible problems, trust that abstract solutions exist for a general set of problems.

Abstraction as such coincides with understanding the problem, the context, and the tools available for the task. This allows a more efficient use of your time and resources.

Problem solving involves recognizing the abstract aspects of a general solution for a problem, and then tailoring a solution for the specifics of the problem at hand. The process requires research as well as trial-and-error. Hence, work proceeds from abstract generalization to concrete implementation.

Abstraction allows programmers to focus energies on problem solving rather than on the internal architecture or composition of a given function or object. This also allows for a concept known as *information hiding*, as when the internal architecture is off limits to outside inspection.

Consider that the words of a spoken language are abstractions of meaning that embody other words. When combined with other words, they generally describe a context.

Abstraction coincides with generalization. A broad framework can be honed for a specific solution at the programmer's discretion, and one

solution can differ from another. Programmers can extend a programming language's abstract facilities, perhaps overriding or wrapping a function with a new one for previously unforeseen capabilities.

For example, one need not understand the mechanical aspects of an automobile engine to drive a car. The driver just operates a few controls. If the problem is to retrieve produce from the market, the solution is not to build an engine, but to drive the car. Similarly, one can use a programming language's built-in functions without needing to understand the algorithms comprising these functions.

This is not to say that abstraction should be taken for granted or that one need only memorize procedures. For the sake of applying analytical tools in a relevant manner, it is important to be aware of context and, for data science, to understand linkages of data and function abstractions to their primitive roots.

Scoping out the deeper implications of analytics is common in other fields as well. It reminds me of when in my college days my friend Mike told me about his dental school lab assignment. After he detailed the extensive dissection of his assigned cadaver, I asked, "Mike, what does this have to do with becoming a dentist!?" He replied, "Muscle and nerve roots throughout the head and upper torso connect everything."

See, even dentists have to realize when seemingly peripheral matters are actually integral to a system of knowledge.

1.6 CHALLENGES WITH PROBLEM SOLVING

An understanding of the problem must exist before writing a program to deal with the problem. Otherwise, the program will be irrelevant and the nature of the problem will likely be clouded by redundant complexities.

When a problem is translated into programming syntax for the purposes of writing a program to solve a problem, there are usually two dimensions of complexity: The parameters of the problem at hand and the programming language syntax.

However, when a student attempts to solve a problem by first translating example code, they actually face a third dimension of complexity: the logic of the example code. Example code as such may be a crutch that

does not help the student strengthen problem-solving skills. Therefore, it is often better to write a program from scratch to solve a problem.

Understanding the problem requires asking questions about the problem. Asking questions can help you effectively break a problem into smaller problems. The questions need not actually be directed to anyone but ourselves, for once you have isolated a question you are partway to the answer, and to further questions.

So, propose a solution and test the solution incrementally as you build your solution. If you have a reasonably good grasp of the problem, you can start coding its solution, revisiting details as you go, and verifying program output against known correct values.

1.7 MENTAL CONDITIONING FOR PROBLEM SOLVING

If when faced with analytical puzzles you are prone to panic, for whatever reason, then your problem-solving abilities will be restricted. For example, students who start assignments the night before a due date tend to panic and find themselves unable to calmly work out problems. They may instead be tempted to seek turn-key solutions from others, which will not develop problem-solving muscle.

Regardless of whether a student is apt to excel under pressure, the proper logistics for organizing assignment resources may be impossible given a late start. There may not be enough hours to perform research, contemplate ideas, and to follow up with research about one's own questions. Additionally, if the most energetic hours of your day are dedicated to non-assignment activities, then you have to question your priorities.

For many students, the key is to start assignments well in advance of due dates and at times of the day when the mind is rested and alert.

> *Late start* → *panic*:
> *Non-analytical area of brain in physical-survival mode.*
> *Early start* → *relaxed*:
> *Analytical area of brain in abstract problem-solving mode.*

Sometimes even the most diligent students fail at managing their schedules. This may be because they have not yet learned how to efficiently allocate an ambitious schedule.

When starting a task early you have time to explore your thoughts. That also provides time for *free association journaling* about problem solutions/coding, much the way that writers use journaling to break *writer's block*. After writing for a while, once removed from the outer world, you may well enter a stream of thought with a focus on the problem at hand.

1.8 BOOKS FOR PROBLEM SOLVING

- G. Polya, *How To Solve It: A New Aspect of Mathematical Method.*[5]
- V. Anton Spraul, *Think Like a Programmer: An Introduction to Creative Problem Solving.*[6]
- Paul Vickers, *How to Think Like a Programmer: Problem Solving for the Bewildered.*[7]

EXERCISES

1. Do some research about the concept of the *specialization of labor* as popularized by Adam Smith, the 18th-century economist. Discuss ways that the *specialization of labor* pertains to generalization in problem solving and abstraction.

2. Suppose that within a large program there exists ten specific lines of code that are seen repeated many times throughout the program. Explain what could or should be done about it.

[5] G. Polya, *How To Solve It: A New Aspect of Mathematical Method.* Princeton University Press (Reprint edition), 2014, ISBN: 069116407X.

[6] V. A. Spraul, *Think Like a Programmer: An Introduction to Creative Problem Solving.* No Starch Press, 2012, ISBN: 1593274246.

[7] P. Vickers, *How to Think Like a Programmer: Problem Solving for the Bewildered.* Cengage Learning, 2008, ISBN: 1408065827.

References

B. Marion , “Discrete mathematics: Support of and preparation for the study of computer science,” Journal of Computing Sciences in Colleges, vol. 16, pp. 190–199, 2000. doi: https://dl.acm.org/doi/10.5555/357697.357730.

D. Stanat and D. McAllister , Discrete Mathematics in Computer Science. Prentice Hall Englewood Cliffs, N.J., 1977, vol. 16, pp. 190–199 (Stanat & McAllister's text may be considered the first definitive textbook on discrete mathematics.)

L. Brodie , Thinking Forth. Punchy Publishing, 2004, isbn: 0976458705. doi: https://thinking-forth.sourceforge.net.

J. McCarthy , “History of lisp,” Special Interest Group on Programming Languages, vol. 13, issue 8, pp. 173–185, 1978, issn: 0362-1340. doi: https://dl.acm.org/doi/10.1145/960118.808387.

R. Bellman , R. Corporation , and K. M. R. Collection , Dynamic Programming, ser. Rand Corporation research study. Princeton University Press, 1957. (The curse-of-dimensionality term was perhaps originated by Bellman in the context of dynamic programming.), isbn: 9780691079516. [Online]. Available: https://books.google.com/books?id=wdtoPwAACAAJ.

J. Pope , “Keeping up with the quants (book review),” Foresight: The International Journal of Applied Forecasting, vol. 38, Fall issue, pp. 38–40, 2013. doi: https://ideas.repec.org/a/for/ijafaa/y2013i31p38-40.html.

G. Polya , How To Solve It: A New Aspect of Mathematical Method. Princeton University Press (Reprint edition), 2014, isbn: 069116407X.

V. A. Spraul , Think Like a Programmer: An Introduction to Creative Problem Solving. No Starch Press, 2012, isbn: 1593274246.

P. Vickers , How to Think Like a Programmer: Problem Solving for the Bewildered. Cengage Learning, 2008, isbn: 1408065827.

R. Chassell , An Introduction to Programming in Emacs Lisp. GNU Press, 2004. (See section 1.1.1 on page 5.) isbn: 1-882114-43-4. doi: https://www.gnu.org/software/emacs/manual/html_node/eintr/Lisp-Atoms.html.

FreeBSD Handbook. The FreeBSD Documentation Project, 2021. (Details for setting Unix file permissions are found in chapter section 3.4.) doi: https://www.freebsdhandbook.com/basics/#permissions.

ARM , *Getting started with arm assembly language*, 2023. doi: https://documentation-service.arm.com/static/64e7245d04d0d65e67136806.

B. Kahanwal , “Abstraction level taxonomy of programming language frameworks,” International Journal of Programming Languages and Applications, vol. 3.4, Oct. 2013. doi: https://www.researchgate.net/publication/258499542_Abstraction_Level_Taxonomy_of_Programming_Language_Frameworks.

A. Ohori , “A curry-howard isomorphism for compilation and program execution,” in Typed Lambda Calculi and Applications, J.-Y. Girard , Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 280–294, isbn: 978-3-540-48959-7.

S. F. Bacon , *The Advancement of Learning*. 1605, vol. 8. (This quote of Sir Francis Bacon may describe a recursive function, wherein its self-reference occurs only until there is doubt.)

R. Bertand , Introduction to Mathematical Philosophy. Macmillan, 1919, p. 6. doi: https://archive.org/details/cu31924001598188.

G. Peano , Arithmetices Principia, Nova Methodo Exposita. unknown, 1889, pp. 23–29. doi: https://github.com/mdnahas/Peano_Book/blob/master/Peano.pdf.

A. Church and J. Rosser , “Some properties of conversion,” Transactions of the American Mathematical Society, vol. 39, pp. 472–482, 1936. doi: https://www.ams.org/journals/tran/1936-039-03/S0002-9947-1936-1501858-0/S0002-9947-1936-1501858-0.pdf.

D. Hoyt , *Let Over Lambda—50 Years of Lisp*. Lulu, 2008. (A good discussion about let-over-lambda can be found in chapter 2 of Hoyt's book.) isbn: 978-1-4357-1275-1, isbn: 978-1-4357-1275-1. doi: https://letoverlambda.com.

G. Michaelson , An Introduction To Functional Programming Through Lambda Calculus. Dover, 2011. (A comprehensive coverage of lambda calculus, including syntax in Backus-Naur Form and programming language implementations in LISP and Standard ML.) isbn: 978-0-486-47883-8.

R. Giegerich , C. Meyer , and P. Steffen , “A discipline of dynamic programming over sequence data,” Science of Computer Programming, vol. 51, no. 3, pp. 215–263, 2004, issn: 0167-6423.

doi: https://doi.org/10.1016/j.scico.2003.12.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642304000176.

C. Okasaki , “Purely functional random-access lists,” in *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* , ser. FPCA ‘95. (Perhaps the first use of Random Access List was by Okasaki in this paper.) La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 86–95, isbn: 0897917197. doi: 10.1145/224164.224187. [Online]. Available: https://doi.org/10.1145/224164.224187.

S. Smith , The Scientist & Engineer's Guide to Digital Signal Processing, First. California Technical Pub, 1997. (A good take on hardware applications of the circular queue in found in chapter 28.) isbn: 0966017633. doi: https://dspguide.com/ch28.htm.

C. E. Shannon , “A mathematical theory of communication,” The Bell System Technical Journal, vol. 27, pp. 379–423, 623–656, 1948. doi: https://archive.org/details/ost-engineering-shannon1948.

P. Norvig , *English Letter Frequency Counts: Mayzner Revisited*. (In practice we would account for many other types of characters, including those representing numbers, as well as lower case characters. For example, according to Peter Norvig on norvig.com, E is the most used letter, with 12.49% of the total letter count, and Z is the least used letter, with 0.09% usage.) 2012. [Online]. Available: http://www.norvig.com/mayzner.html.

H. Nyquist , “Certain topics in telegraph transmission theory,” Transactions of the American Institute of Electrical Engineers, vol. 47, no. 2, pp. 617–644, 1928. (The Nyquist Rate addresses a similar concept called aliasing with respect to signal processing.) doi: 10.1109/T-AIEE.1928.5055024.

J. Neter , W. Wasserman , and M. Kutner , Applied Linear Statistical Models, Third. Irwin, 1990, p. 300, isbn: 025608338X.

H. Markowitz , “Portfolio selection,” The Journal of Finance, vol. 7, no. 1, pp. 77–91, Mar. 1952.

T. Hastie , R. Tibshirani , and J. Friedman , The Elements of Statistical Learning. Springer, New York, 2009, pp. 22–23. (This text provides a good examination of sample density in section 2.5.) doi: 10.1007/978-0-387-84858-7.

A. Majdara and S. Nooshabadi , “Efficient density estimation for high-dimensional data,” IEEE Access, vol. 10, pp. 16592–16608, 2022. (This article provides a comprehensive examination of density estimation.) doi: 10.1109/ACCESS.2022.3149280.