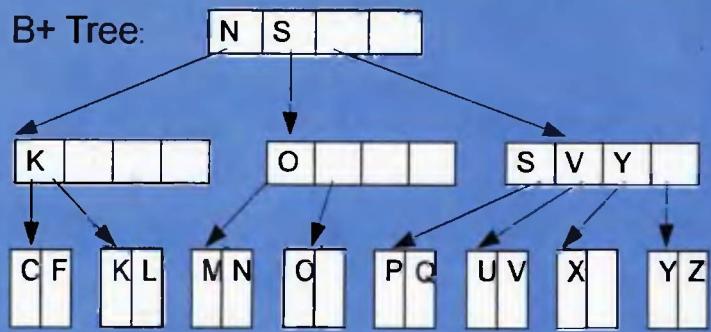


$$\pi \approx 4 \times 22/(22 + 6) \approx 3.142$$



```

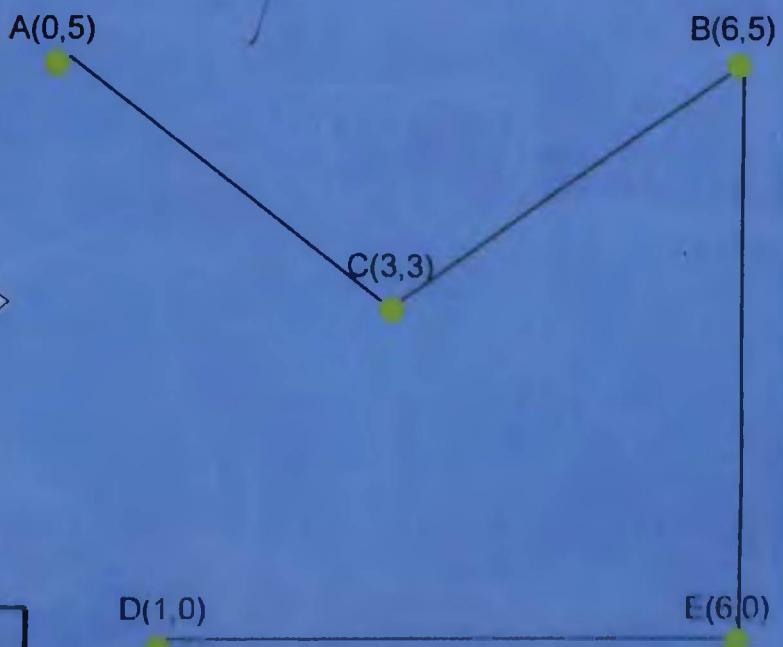
unsigned int randomNumber()
{
    unsigned int x = time(0);
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return x;
}
  
```

Implementing Useful Algorithms in C++

Next	C	D	E
Cumulative Cost	6	361	51
MST	ED + ED	ED + EB	CB + EC
MST Cost	8.61	10	7.85
Connect edge	CB	DC	DC + CB
Connect edge cost	3.61	3.61	5
Lowerbound	17.21	16.55	17.21
Pruned	Pruned	Pruned	Pruned

Next	C	F	
Cumulative Cost	12.17	8.7	10.1
MST	EC	EB	CB
MST Cost	4.24	6	3.61
Connect edge	EC	EC	DC
Connect edge cost	3.61	3.61	4.24
Lowerbound	17.31	17.31	15.82
Pruned	Pruned	Pruned	Pruned

Next	E		
Cumulative Cost	12.31	12.93	
Connect edge	EB	EB	
Connect edge cost	5	4	
Lowerbound	17.31	17.31	
Pruned	Pruned	Pruned	



Dmytro Kedyk

© 2020 Dmytro Kedyk. All rights reserved

ISBN: 9798605325307

Library of Congress Control Number: 2014901334

Independently Published: Bronxville, NY

Implementing Useful Algorithms in C++

Dmytro Kedyk

Contents

Preface	xiii
1 Background	1
1.1 Introduction.....	1
1.2 Algorithm Desiderata.....	1
1.3 Logics of Reasoning.....	2
1.4 Proving Basic Correctness.....	3
1.5 Asymptotic Notation.....	4
1.6 Machine Models.....	4
1.7 Randomized Algorithms.....	5
1.8 Measuring Efficiency.....	5
1.9 Data Types.....	6
1.10 Algorithm Experiments.....	6
1.11 Memory Management.....	7
1.12 Code Optimization.....	7
1.13 Recursion.....	9
1.14 Computation Strategies.....	9
1.15 Choosing among Several Algorithms.....	9
1.16 Making Algorithms Parallel.....	10
1.17 How to Implement an Algorithm.....	10
1.18 Recommended Classes for Computer Science Students.....	11
1.19 Some Study Strategies.....	11
1.20 Projects for the Whole Book.....	12
1.21 References.....	13
2 Software Engineering Essentials	14
2.1 Introduction.....	14
2.2 Overview of the Development Cycle.....	14
2.3 Requirements.....	15
2.4 Design of Component Structure.....	15
2.5 Design of Individual Components and Coding.....	15
2.6 Patterns.....	16
2.7 Error Management.....	18
2.8 Testing.....	19
2.9 Code Reviews.....	20
2.10 Release.....	20
2.11 Maintenance.....	21
2.12 Estimation.....	21
2.13 Following a Formal Process.....	21
2.14 Managing User Data.....	21
2.15 References.....	22
3 Career Advice and Interviews	23
3.1 Introduction.....	23
3.2 Applying for Jobs.....	23
3.3 The Resume.....	23
3.4 Behavioral Questions and Interviews.....	24
3.5 Technical Interviews.....	24
3.6 More Difficult Questions.....	27
3.7 System Design Interviews.....	27
3.8 Offer Negotiation.....	28
3.9 How to Leave Your Current Job Nicely.....	28
3.10 Fight Complacency.....	29
3.11 Projects.....	29
3.12 References.....	29

4 Introduction to Computer Law	30
4.1 Introduction.....	30
4.2 Intellectual Property.....	30
4.3 Patents.....	30
4.4 Trade Secrets.....	31
4.5 Copyrights.....	31
4.6 Trademarks.....	32
4.7 Intellectual Property Management.....	32
4.8 Contracts.....	33
4.9 Licenses.....	33
4.10 Employment Agreements.....	34
4.11 Privacy.....	34
4.12 Computer Crime.....	34
4.13 Serving as an Expert Witness.....	34
4.14 Projects.....	35
4.15 References.....	35
5 Fundamental Data Structures	36
5.1 Introduction.....	36
5.2 Utility Functions.....	36
5.3 Vector.....	40
5.4 Block Array.....	44
5.5 Linked List.....	44
5.6 Garbage-collecting Free List.....	46
5.7 Stack.....	50
5.8 Queue.....	51
5.9 Trees.....	53
5.10 Bit Algorithms.....	54
5.11 Bit Set.....	56
5.12 Union-find.....	60
5.13 Implementation Notes.....	61
5.14 Comments.....	61
5.15 Projects.....	61
5.16 References.....	62
6 Random Number Generation	63
6.1 Introduction.....	63
6.2 A Quick Review of Probability.....	63
6.3 Pseudorandom Number Generation.....	64
6.4 Xorshift.....	65
6.5 Mersenne Twister.....	66
6.6 MRG32k3a.....	67
6.7 RC4.....	68
6.8 Picking a Generator.....	68
6.9 Using a Generator.....	69
6.10 Generating Samples from Distributions.....	69
6.11 Generating Samples from Bounded-range Discrete Distributions.....	70
6.12 Generating Samples from Specific Distributions.....	73
6.13 Generating Random Objects.....	76
6.14 Generating Samples from Multidimensional Distributions.....	77
6.15 The Monte Carlo Method.....	78
6.16 Implementation Notes.....	83
6.17 Comments.....	83
6.18 Projects.....	83
6.19 References.....	84
7 Sorting	85
7.1 Introduction.....	85
7.2 Insertion Sort.....	85
7.3 Quicksort.....	85
7.4 Mergesort.....	87

7.5 Integer Sorting.....	88
7.6 Vector Sorting.....	89
7.7 Permutation Sort.....	91
7.8 Selection.....	92
7.9 Multiple Selection.....	92
7.10 Searching.....	93
7.11 Implementation Notes.....	93
7.12 Comments.....	93
7.13 Projects.....	94
7.14 References.....	94
8 Dynamic Sorted Sequences	95
8.1 Introduction.....	95
8.2 Requirements.....	95
8.3 Skip List.....	95
8.4 Treap.....	99
8.5 Tree Iterators.....	104
8.6 Augmentations and API Variants.....	105
8.7 Vector Keys.....	106
8.8 LCP Augmentation for Trees.....	106
8.9 Tries.....	111
8.10 Ternary Treap Trie.....	112
8.11 Performance Comparisons.....	117
8.12 Implementation Notes.....	117
8.13 Comments.....	117
8.14 Projects.....	118
8.15 References.....	118
9 Hashing	120
9.1 Introduction.....	120
9.2 Hash Functions.....	120
9.3 Universal Hash Functions.....	122
9.4 Nonuniversal Hash Functions.....	124
9.5 Rolling Hash Functions.....	126
9.6 Collection of Hash Functions.....	127
9.7 Hash Tables.....	127
9.8 Chaining Hash Table.....	127
9.9 Linear Probing Hash Table.....	131
9.10 Timings.....	134
9.11 Bloom Filter.....	135
9.12 Implementation Notes.....	136
9.13 Comments.....	136
9.14 Projects.....	137
9.15 References.....	137
10 Priority Queues	138
10.1 Introduction.....	138
10.2 The API.....	138
10.3 Binary Heap.....	138
10.4 Indexed Heaps.....	140
10.5 Implementation Notes.....	143
10.6 Comments.....	143
10.7 References.....	144
11 Graph Algorithms	145
11.1 Introduction.....	145
11.2 Basics.....	145
11.3 Graph Representation.....	145
11.4 Search.....	147
11.5 Some Applications of Search.....	150
11.6 Minimum Spanning Tree.....	151

11.7 Shortest Paths.....	152
11.8 Flow Algorithms.....	154
11.9 Bipartite Matching.....	156
11.10 Stable Matching.....	157
11.11 Assignment Problem.....	158
11.12 Generating Random Graphs.....	159
11.13 Implementations Notes.....	160
11.14 Comments.....	160
11.15 Projects.....	160
11.16 References.....	160
12 Miscellaneous Algorithms and Techniques	162
12.1 Introduction.....	162
12.2 Making Static Data Structures Dynamic.....	162
12.3 Making Data Structures Persistent.....	162
12.4 Maintaining a Cache.....	163
12.5 k-bit-word Vector.....	166
12.6 Set Union on Intervals.....	166
12.7 Generating the First N Primes.....	166
12.8 Generating All Permutations.....	167
12.9 Generating All Combinations.....	168
12.10 Generating All Subsets.....	168
12.11 Generating All Partitions.....	168
12.12 Generating All Constrained Objects.....	169
12.13 Implementation Notes.....	169
12.14 Comments.....	169
12.15 Projects.....	169
12.16 References.....	169
13 External Memory Algorithms	171
13.1 Introduction.....	171
13.2 Disks and Files.....	171
13.3 File Layout.....	173
13.4 Working with CSV Files.....	173
13.5 The I/O Model.....	176
13.6 External Memory Vector.....	179
13.7 Sorting.....	181
13.8 Vector-based Data Structures.....	183
13.9 B+ Tree.....	184
13.10 Comments.....	190
13.11 Projects.....	190
13.12 References.....	190
14 String Algorithms	191
14.1 Introduction.....	191
14.2 Single-pattern Search.....	191
14.3 Multiple Patterns.....	192
14.4 Regular Expressions.....	194
14.5 Extended Patterns.....	195
14.6 String Distance Algorithms.....	196
14.7 Inverted Index.....	200
14.8 Suffix Index.....	200
14.9 Syntax Tree.....	203
14.10 Introduction to Succinct Data Structures.....	203
14.11 Implementation Notes.....	206
14.12 Comments.....	206
14.13 Projects.....	206
14.14 References.....	207
15 Compression	208
15.1 Introduction.....	208

15.2	Fundamental Limits.....	208
15.3	Entropy.....	208
15.4	Bit Stream.....	208
15.5	Codes.....	210
15.6	Static Codes.....	210
15.7	Huffman Codes.....	213
15.8	Dictionary Compression.....	216
15.9	Run-length Encoding.....	218
15.10	Move-to-front Transform.....	219
15.11	Burrows-Wheeler Transform.....	220
15.12	Implementation Notes.....	221
15.13	Comments.....	222
15.14	Projects.....	222
15.15	References.....	222
16	Combinatorial Optimization	223
16.1	Introduction.....	223
16.2	Complexity Theory.....	223
16.3	Typical Hard Problems.....	224
16.4	Approximation Algorithms.....	227
16.5	Greedy Construction Heuristics.....	228
16.6	Branch and Bound.....	228
16.7	State Space Shortest Path Search with Lower Bounds.....	234
16.8	Local Search.....	239
16.9	Applying Local Search to Some Problems.....	243
16.10	Simulated Annealing.....	247
16.11	Iterated Local Search.....	250
16.12	Genetic Algorithms.....	251
16.13	Some Performance Results.....	255
16.14	Problem-specific Preprocessing.....	256
16.15	Multiobjective Optimization.....	256
16.16	Constraint Processing.....	257
16.17	Stochastic Problems.....	260
16.18	General Advice.....	261
16.19	Implementation Notes.....	261
16.20	Comments.....	261
16.21	Projects.....	263
16.22	References.....	264
17	Large Numbers	265
17.1	Introduction.....	265
17.2	Representation.....	265
17.3	Addition and Subtraction.....	266
17.4	Shifts.....	267
17.5	Multiplication.....	268
17.6	Division.....	269
17.7	Conversion to Decimal.....	270
17.8	Exponentiation.....	271
17.9	Lg.....	271
17.10	Integer Square Root.....	271
17.11	Greatest Common Divisor.....	272
17.12	Modular Inverse.....	272
17.13	Primality Testing.....	272
17.14	Rationals.....	273
17.15	Implementation Notes.....	275
17.16	Comments.....	275
17.17	Projects.....	275
17.18	References.....	275
18	Computational Geometry	276
18.1	Introduction.....	276

18.2 Distances.....	276
18.3 VP Tree.....	276
18.4 k-d Tree.....	281
18.5 Problems in High Dimensions.....	286
18.6 Data Structures for Geometric Objects.....	286
18.7 Points.....	286
18.8 Geometric Primitives.....	287
18.9 Convex Hull.....	287
18.10 Plane Sweep.....	288
18.11 Comments.....	290
18.12 Projects.....	290
18.13 References.....	290
19 Error Detection and Correction	291
19.1 Introduction.....	291
19.2 Binary Polynomials.....	291
19.3 Polynomials Over Finite Fields.....	291
19.4 Error Detection.....	291
19.5 Channels and Codes.....	293
19.6 Finite Field Computation.....	294
19.7 Polynomials over Galois Field Elements.....	295
19.8 Reed-Solomon Codes.....	297
19.9 Bounds on Fixed-alphabet Minimum-distance Codes.....	300
19.10 Boolean Matrices.....	300
19.11 Low-density Parity-check Codes.....	301
19.12 Implementation Notes.....	305
19.13 Comments.....	305
19.14 Projects.....	306
19.15 References.....	306
20 Cryptography	307
20.1 Introduction.....	307
20.2 File Encryption.....	307
20.3 Key Length.....	308
20.4 Key Storage.....	309
20.5 Cryptographic Hashing.....	309
20.6 Key Exchange.....	309
20.7 Other Protocols.....	309
20.8 Implementations Notes.....	310
20.9 Comments.....	310
20.10 Projects.....	310
20.11 References.....	311
21 Computational Statistics	312
21.1 Introduction.....	312
21.2 Estimands.....	312
21.3 Estimators.....	314
21.4 Finding Most Efficient Estimators.....	317
21.5 Some Peculiarities of Asymptotics.....	319
21.6 Evaluating the Normal CDF.....	320
21.7 Evaluating the T-Distribution CDF.....	320
21.8 More on Confidence Intervals.....	321
21.9 Finite-sample Bounds for the Mean.....	323
21.10 Confidence Intervals for Common Location Measures.....	324
21.11 Outliers and Robust Inference.....	327
21.12 Functions of Estimates.....	327
21.13 Measuring Algorithm Runtime.....	328
21.14 Correlation Analysis.....	328
21.15 The Bootstrap.....	330
21.16 When Does Bootstrap Work?.....	339
21.17 Hypothesis Tests.....	340

21.18 Comparing Tests.....	341
21.19 Using Tests Effectively.....	342
21.20 Validation of Studies.....	345
21.21 Comparing Matched Pairs.....	346
21.22 Multiple Comparisons.....	347
21.23 Comparing Matched Tuples.....	348
21.24 Comparing Independent Samples.....	349
21.25 Permutation Tests.....	350
21.26 Comparing Many Alternatives on Multiple Domains.....	353
21.27 Working with Count Data.....	354
21.28 Testing Distribution Differences.....	356
21.29 Comparing Data to a Distribution.....	356
21.30 Comparing Distributions of Two Samples.....	357
21.31 Sensitivity Analysis.....	358
21.32 Sobol Sequence.....	360
21.33 Design of Experiments—Main Ideas.....	362
21.34 Markov Chain Monte Carlo.....	364
21.35 Bayesian Methods.....	366
21.36 Finding Best Alternative via Simulation.....	367
21.37 Sample Size Calculations.....	369
21.38 Time Series Analysis.....	369
21.39 Using Statistics in Practice.....	378
21.40 Analysis of Decisions.....	379
21.41 Implementation Notes.....	380
21.42 Comments.....	380
21.43 Projects.....	384
21.44 References.....	385

22 Numerical Algorithms—Introduction and Matrix Algebra 388

22.1 Introduction.....	388
22.2 Floating Point Arithmetic.....	388
22.3 Errors from Using Floating Point Arithmetic.....	389
22.4 Approximation Error.....	391
22.5 Stability and Condition Numbers.....	392
22.6 Optimization Error.....	393
22.7 Other Common Themes.....	394
22.8 Developing Robust Numerical Software.....	395
22.9 Matrix Algebra.....	397
22.10 Matrix Norms.....	400
22.11 LUP Decomposition.....	400
22.12 Cholesky Decomposition.....	403
22.13 Band Matrices.....	404
22.14 Solving Tridiagonal Matrix Equations.....	406
22.15 Orthogonal Transformations.....	406
22.16 QR Decomposition.....	407
22.17 Symmetric Matrix Eigenvalues and Eigenvectors.....	409
22.18 Singular Value Decomposition.....	411
22.19 Asymmetric Matrix Eigenvalues and Eigenvectors.....	413
22.20 Sparse Matrices.....	417
22.21 Iterative Methods for Sparse Matrices.....	421
22.22 Iterative Methods for Eigenvalues	422
22.23 Introduction to Interval Arithmetic.....	423
22.24 Implementation Notes.....	425
22.25 Comments.....	425
22.26 Projects.....	427
22.27 References.....	427

23 Numerical Algorithms—Working with Functions 429

23.1 Introduction.....	429
23.2 Fast Fourier Transform.....	429
23.3 Interpolation—General Ideas.....	432

23.4	Polynomial Interpolation from Existing Data.....	433
23.5	Chebyshev Polynomials.....	436
23.6	Piecewise Interpolation.....	440
23.7	Splines—For Existing Data.....	445
23.8	Comparison of Interpolation Methods.....	448
23.9	Integration.....	449
23.10	Multidimensional Integration.....	454
23.11	Function Evaluation.....	456
23.12	Estimating Derivatives.....	457
23.13	Solving Nonlinear Equations and Systems.....	460
23.14	Finding All Roots in 1D.....	471
23.15	Ordinary Differential Equations.....	472
23.16	Solving Stiff ODEs.....	476
23.17	Boundary Value Problems for ODE.....	478
23.18	Partial Differential Equations—Some Thoughts.....	479
23.19	Some Conclusions.....	480
23.20	Implementation Notes.....	480
23.21	Comments.....	481
23.22	Projects.....	484
23.23	References.....	484

24 Numerical Optimization 487

24.1	Introduction.....	487
24.2	Some General Ideas.....	487
24.3	Single-variable Unimodal Function Minimization.....	487
24.4	Multidimensional Function Minimization—Introduction and Coordinate Descent.....	489
24.5	Line Search.....	491
24.6	Line-search-based Algorithms.....	494
24.7	Some Derivative-free Methods.....	498
24.8	Nonsmooth Minimization.....	502
24.9	Global Minimization.....	504
24.10	Discrete Set Optimization.....	514
24.11	Stochastic Optimization.....	516
24.12	Stochastic Approximation Algorithms.....	516
24.13	Linear Programming.....	518
24.14	Some Thoughts on Nonlinear Programming.....	520
24.15	Implementation Notes.....	521
24.16	Comments.....	521
24.17	Projects.....	524
24.18	References.....	525

25 General Machine Learning 527

25.1	Introduction.....	527
25.2	What Machine Learning Is.....	527
25.3	Mathematical Learning.....	527
25.4	Predictive vs Structural Inference.....	531
25.5	Risk Estimation of a Predictor.....	531
25.6	Sources of Risk.....	533
25.7	Complexity Control.....	534
25.8	Approximation Error.....	535
25.9	Stability.....	537
25.10	Risk Estimation of a Learning Strategy.....	537
25.11	Making Choices and Model Selection.....	540
25.12	Some General Model Selection Strategies.....	540
25.13	Bias, Variance, and Bagging.....	541
25.14	Design Patterns.....	543
25.15	Data Preparation.....	544
25.16	Scaling.....	545
25.17	Handling Missing Values.....	546
25.18	Feature Selection.....	546
25.19	Kernels.....	550

25.20 Online Training.....	551
25.21 Dealing with Nonvector Data.....	552
25.22 Large-scale Learning.....	552
25.23 Conclusions.....	553
25.24 Implementation Notes.....	553
25.25 Comments.....	554
25.26 Projects.....	557
25.27 References.....	557
26 Machine Learning—Classification	559
26.1 Introduction.....	559
26.2 Stratification by Class Label.....	559
26.3 Risk Estimation.....	560
26.4 Reducing Multiclass to Binary.....	563
26.5 Complexity Control.....	565
26.6 Naive Bayes.....	567
26.7 Nearest Neighbor.....	569
26.8 Decision Tree.....	571
26.9 Support Vector Machine.....	577
26.10 Linear SVM.....	578
26.11 Kernel SVM.....	582
26.12 Multiclass Nonlinear SVM.....	585
26.13 Neural Network.....	587
26.14 Randomization Ensembles.....	594
26.15 Online Training.....	595
26.16 Boosting.....	597
26.17 Large-scale Learning.....	599
26.18 Cost-sensitive Learning.....	600
26.19 Imbalanced Learning.....	603
26.20 Feature Selection.....	605
26.21 Comparing Classifiers.....	605
26.22 Implementation Notes.....	607
26.23 Comments.....	607
26.24 Projects.....	612
26.25 References.....	613
27 Machine Learning—Regression	615
27.1 Introduction.....	615
27.2 Risk Estimation.....	615
27.3 Complexity Control.....	616
27.4 Linear Regression.....	617
27.5 Lasso Regression.....	617
27.6 Nearest Neighbor Regression.....	620
27.7 Regression Tree.....	621
27.8 Random Forest Regression.....	623
27.9 Neural Network.....	624
27.10 Feature Selection.....	625
27.11 Comparing Performance.....	625
27.12 Implementation Notes.....	626
27.13 Comments.....	626
27.14 Projects.....	628
27.15 References.....	628
28 Machine Learning—Clustering	630
28.1 Introduction.....	630
28.2 Setup.....	630
28.3 External Evaluation.....	631
28.4 Internal Evaluation and Selecting the Number of Clusters.....	633
28.5 Calculating Stability.....	635
28.6 Clustering in Euclidean Space.....	637
28.7 Clustering in a Metric Space.....	640

28.8 Spectral Clustering.....	642
28.9 Experiments.....	645
28.10 Implementation Notes.....	645
28.11 Comments.....	645
28.12 Projects.....	648
28.13 References.....	648
29 Machine Learning—Other Tasks	650
29.1 Introduction.....	650
29.2 Reinforcement Learning.....	650
29.3 Value Function Temporal Difference Learning.....	650
29.4 Finding Frequent Item Combinations.....	651
29.5 Semi-supervised Learning.....	652
29.6 Density Estimation.....	653
29.7 Outlier Detection	653
29.8 Implementation Notes.....	654
29.9 Comments.....	654
29.10 References.....	654
30 Scrap—Not Useful Algorithms and Data Structures	655
30.1 Introduction.....	655
30.2 Sorting a Linked List.....	655
30.3 Partial Sort.....	656
30.4 Patricia Trie.....	657
30.5 Cuckoo Hashing.....	660
30.6 A Few Priority Queues.....	663
30.7 Lowest Common Ancestor and Range-minimum Query.....	667
30.8 Wilcoxon Signed Rank Test for Two Samples.....	668
30.9 Friedman Test for Matched Samples.....	669
30.10 A MADS-like Optimization Algorithm.....	670
30.11 SAMME Boosting Classification Algorithm.....	670
30.12 Boosting for Regression.....	671
30.13 Probabilistic Clustering.....	672
30.14 Hierarchical Clustering.....	675
30.15 Density-based Clustering.....	677
30.16 Implementations not Presented.....	679
30.17 Projects.....	680
30.18 References.....	680
31 Appendix—C++ Notes	681
31.1 Introduction.....	681
31.2 A Guide to C++ Literature.....	681
31.3 Projects.....	681
31.4 References.....	681
Index	682

Preface

"Of course this book is error-free—but any you find please report to the author" – Forman Acton

This book explains how to properly implement algorithms and data structures worth implementing (yes, many aren't!), mostly assuming familiarity with the basic ideas on the level of an introductory algorithms class. So it:

- Covers numerous specialized topics.
- For the standard ones gives substantial extra material needed for implementation and complete understanding. E.g., how to implement a priority queue that allows keeping track of items for shortest path algorithms?
- Makes unapologetic decisions about which algorithms and data structures to use. For some more advanced chapters this is perhaps the main takeaway for the reader. Though understanding reasoning behind the choices is still more important.

In some cases the known algorithms weren't good enough, so I invented my own. These aren't published elsewhere. Some are:

- A garbage-collected free list (see the "Fundamental Algorithms" chapter)
- A sum heap (see the "Random Number Generation" chapter)
- A xorshift-based implementation of a universal hash function for arrays (see the "Hashing" chapter)
- Use of the sign test for decision and regression tree pruning (see the "Machine Learning—Classification" chapter)
- An efficient k -medoid implementation (see the "Machine Learning—Clustering" chapter)

Ideally for an algorithm every important design choice has been extensively researched and validated and is common knowledge. My inventions need validation by others, but there is likely scientific value in at least some of them. So if you are a researcher, feel free to email me with any questions.

As building blocks for more complicated algorithms and data structures, I use those presented in the book and not the ones from the C++ STL. This allows better testing of the code and has certain usability advantages—e.g., I lost count of caught bugs due to checking bounds in `operator[]` of my own vector implementation, which also allows to work with the last element conveniently. Of course for real-world development, established libraries are preferred due to being standard and more robust. My codes, though reasonably tested, are only starters for good implementations.

For simplicity many latest C++ features are avoided. E.g., move semantics might improve efficiency when objects have efficient destructors, but otherwise compiler optimization is enough. This is an advanced book though, so need some computational and mathematical maturity. The chapters on specialized topics such as numerical algorithms need specific math knowledge such as linear algebra. Some later chapters rely on some earlier ones. There are no exercises, but some chapters have projects for further extension. These usually need research and careful implementation and are mostly things I haven't gotten to.

This book is best used in companion to those that teach the basic ideas, particularly for the later chapters. I don't claim that my implementations are perfect, only a step in the right direction. I optimized the references, and if some popular books on a topic are missing, this is likely because I am very selective. Overall, other books tend to give more examples and mathematics but far fewer implementation details, and need to read several different expositions to get comfortable with the material.

The preliminary versions of this book were titled "Commodity Algorithms and Data Structures in C++", updated every two years. I made substantial effort to fix all errors and to address all the feedback to create a "stable version". So no new edition is expected for the next several years, though I am still actively curious about many of the topics. Please e-mail any feedback to igmdk@msn.com or post a review on Amazon—I am eager to hear about what you liked and what needs to be added or improved. **Expect to learn something new in every chapter!**

All the code and some slides are available at
<https://github.com/dkedyk/ImplementingUsefulAlgorithms>.

1 Background

"I wish my wish would not be granted" – Douglas Hofstadter

"If you don't know how to do something, you don't know how to do it on a computer" – Anonymous

1.1 Introduction

Nontrivial algorithms are rarely implemented due to the availability from various APIs. Most programming work is spent on logical modeling of systems with domain experts. But occasionally need to understand what an API does in principle, and be able to modify or extend the implementation. Also someone needs to develop and maintain the APIs, and for various embedded devices this is commonly done from scratch or by adapting similar APIs.

This book covers only algorithms that are expected to be implemented and used, with some exceptions. Theoretical algorithms, which seem to exist only because of intellectual curiosity, are summarized well in Kao (2016).

This chapter reviews some recurring themes.

1.2 Algorithm Desiderata

An **algorithm** is a well-defined input-to-output transformation. A **data structure** allows efficient access and updates to the stored data. To support software development effectively, both must be:

- **Correct** by giving a satisfactory solution to the exact problem. A formal proof, extensive testing, or intuition can deem an algorithm and its implementation correct. Can get only correctness with high certainty because formal proofs can be unobtainable, intuition wrong, and rarely invoked logic untested. Correctness can be randomized, so that an algorithm that's wrong less often than the computer is hit by lightning is effectively correct. An algorithm should report failures such as the inability to allocate memory.
- **Extendable** by being applicable to general problems with only minor changes. Real-world problems sometimes have extra constraints that require augmenting existing algorithms. An algorithm that applies to a variety of problems is more valuable than a collection of top-performing custom algorithms for every problem. An implementation is extendable if the code applies to many problems unchanged. Usually prefer reusable code, even at the cost of reasonable losses in simplicity and efficiency. But exploiting specific problem structure often allows a better solution, so customized algorithms have their place.
- **Relatively simple** to understand, implement in different environments, and use as an abstract building block. Simplicity of understanding is measured by how long it takes to understand and debug what the algorithm does in all cases, of implementation by the number of lines of properly structured code, and of use by how clean the API is in terms of the number of functions, their parameters, and the availability of good defaults. An important part of simplicity of understanding is that everything useful about the algorithm is known, including theorems describing its behavior and details needed for a robust, efficient implementation.
- **Sufficiently efficient** with valuable resources, at least on practical problems. CPU time and memory are often the most important. Can't use more memory than time if accessing every used bit, but memory is more important because can wait for another hour but not another RAM chip, and, unlike time, it accumulates. Other resources such as concurrency locks, file handles, network ports, requests to another process, and electricity also matter. Paid development time and time waiting for the system to respond are often the most costly. Humans don't differentiate between GUI response times < 150 milliseconds, so don't need to optimize below that.
- **Legal to use.** Many aspects of algorithms are intellectual property. Patents and trade secrets protect ideas, trademarks names, and copyrights code. Using protected software, including open source, requires a special contract called a license (see the "Introduction to Computer Law" chapter).

A correct, efficient, and legal algorithm is **useful**. Simplicity and extendability reduce development and maintenance time. An algorithm is ideally useful, simple, and extendable, though it's unclear how to name this concept. Names such as "commodity" (used in the draft editions) or "toolbox-worthy" seem to confuse more than enlighten. A system composed of such algorithms handles users' requests correctly and efficiently and allows quick feature addition and debugging.

Many useful algorithms giving solutions of substantial economic value have very complicated implementations, and domain experts design libraries of them for years. Such **algorithm engineering** produced impressive results in linear programming, finding shortest paths in continental road networks, computational geometry, etc. But can't fix library bugs yourself, don't know exactly what the code does, and have no guarantee that the sublibraries are legal. E.g., can't quickly resolve user bug reports for issues originating from libraries. But all these issues are rare. So, even for simple algorithms, it's best to use the best available library and not something programmed yourself. In fact, it's a good investment to replace homebrew by library code. Structure libraries in layers to maximize reuse and allow finding reusable components efficiently.

Similarly, at a somewhat further complexity level, some domains need algorithms that are **too technical** in a sense that simple algorithms don't lead to useful solutions. Here useful algorithms are too complicated and need considerable time from domain experts (perhaps months of team effort) to develop—e.g., cryptographic protocols, compilers, sparse matrix algebra, deep networks, GPS road routing, numerical calculations of elementary functions, etc. Such algorithms are sometimes discussed in detail in specialized monographs, and usually aren't developed in this book, except for brief mentions in the comments section of the corresponding chapters. Such algorithms needs thousands of code lines to handle many special cases and take extreme care to be correct and robust.

Algorithms are just problem-solving tools. What matters in real-life is how well a particular problem is solved, and a catalog of good black-box algorithms may not be enough. Usually need some thinking to convert a problem into a form suitable for black-box solutions, and often must choose between several options. Also the black box you want and the best one you can get can be very different, so the adaptation effort is occasionally very large.

1.3 Logics of Reasoning

Can model functionality requirements and many properties of algorithms using various logics, and this section summarizes the most important ones. Their detailed knowledge isn't useful to most developers because most proofs of correctness for algorithms are done by specialists. So it's only important to be familiar with the basics.

E.g., consider “Does this loop terminate?”—you might say “obviously”, but would you put it in a pacemaker? A **logic** assumes **axioms** to decide truth of **statements** in a language. So to decide loop termination need to know at least the effect of each code line on the termination conditions. A logic is **complete** if every true statement is provable. In general it can be impossible to prove that a loop terminates—e.g., what if loop over even numbers until find one that's not a sum of two primes. That this is an infinite loop is Goldbach conjecture (as an irrelevant technicality, need arbitrary-precision arithmetic and unlimited memory).

Propositional logic has true/false variables and statements formed using Boolean operators *and* (“ $\&$ ”), *not* (“ $!$ ”), *or* (“ $|$ ”), *implies* (“ \rightarrow ”), *xor* (“ \wedge ”), *nand* (“ $\#$ ”), etc. It helps to simplify your if statements using common **reasoning rules**—e.g., $(x \& (x \rightarrow y)) \rightarrow y$.

A **truth table** evaluates a statement for every value assignment. Each above operator is defined by a truth table of size four. A **tableau system** is usually more efficient. Two **clauses** connected by an operator must evaluate to particular values for the operator to evaluate to a particular value. Let $T(x)/F(x)$ mean that x is respectively true/false. Then have expansion rules $T(x \& y) \leftrightarrow T_x \& T_y$, $F(x \& y) \leftrightarrow F_x | F_y$, $T!x \leftrightarrow F_x$, and $F!x \leftrightarrow T_x$. Using such rules eventually get a contradiction (i.e., prove false) or run out of clauses to expand (i.e., prove true). See Smullyan (2014) for details and example proofs.

First-order logic is propositional logic with “ \forall ” (for each/any) and “ \exists ” (there exists). It's much better for reasoning about loops but not every false statement is provable (because can't enumerate all cases), which usually isn't too much of a problem. Adding these operators brings extra axioms and tableau rules (again, see Smullyan 2014—I don't wish to go into technical details here). Logic programming language **Prolog** automates tableau reasoning, and developers at best use intuitive reasoning rules (such as **modus ponens** from your discrete math class) that are proven correct by tableau.

Second-order logic allows quantification over sets and is the most natural logic. It's incomplete because \exists statements that lead to contradictions if provable. So often use simpler logics.

Logic of knowledge is first order logic where agent i may know the truth of statement x . Additional axioms:

- $K_i x \rightarrow x$ —if an agent knows something, it's true
- $K_i x \rightarrow K_i K_i x$ —an agent is aware of what it knows
- $!K_i x \rightarrow K_i !K_i x$ —an agent is aware of what it doesn't know
- $K_i(x \rightarrow y) \rightarrow (K_i x \rightarrow K_i y)$ —allows an agent to deduce new knowledge

Note that $x \rightarrow K_x$ is false.

Other logics are also useful for expressing various properties of algorithms:

- **Belief**—an agent can believe something false.
- **Temporal**—a statement's truth depends on discrete time, and have additional operators *always* and *eventually*. E.g., *power failure & always traffic light isn't green on both sides* is a requirement for traffic lights.

1.4 Proving Basic Correctness

Any function can be viewed from the caller's point of view first—as a black box transforming input with some properties into output with some other properties. These are respectively **preconditions/postconditions**. They form an **algorithm's contract**, which is a guarantee that an input satisfying preconditions leads to an output satisfying postconditions. Correctness proofs are feasible for many algorithms. They show that the sequence of program statements transforms the overall precondition so that at the end of the execution the overall postcondition is true. But the practical goal is semiformal reasoning that allows to intuitively deem a program correct to some degree—formal reasoning was proposed in the 70's and never gained popularity outside of special cases such as hardware verification.

Inputs that don't satisfy preconditions usually lead to **undefined behavior**. Good APIs check inputs and assert correctness. Only when the input checks are more expensive than the algorithm itself, it makes sense to only comment about this. Also, some inputs can't have precondition checks—e.g., how to test a user-passed function or a comparator? Even for a trivial test need at least the ability to create the items to which these apply, which may not be possible in template-based, generic C++ code.

Arguably exception-throwing is done more flexibly by allowing the caller code to recover, but assertions are adequate in that the caller must check them first. A common view is that it's always reasonable to expect preconditions to hold exactly. One potential exception to this policy that I have seen is if pass a randomized function to an equation solver based on binary search (see the "Numerical Algorithms—Working with Functions" chapter), but here using standard equation solving is dubious. Also a particular implementation can work with minor precondition violations due more defensive coding than mandated, but another conforming implementation can fail.

Some postconditions that aren't actually so are **nonessential behavior**. E.g., the exact format of outputting an object to a text stream is usually subject to change and shouldn't be relied on. They are also usually not specified in the documentation.

An **invariant** is a property that holds during a computation, particularly loops. Though not part of the contract, it helps to think about correctness after a particular stage of computation. E.g., consider finding the smallest and the second smallest elements in an integer array, returned by array indices (this is an interview question, discussed more later in the chapter). A general solution is to keep two index variables and incrementally update them in a loop. Trying to solve this without an invariant usually leads to issues because it's hard to correct poor initialization in the loop. A simple invariant is that after seeing k out of n elements the minimum is correct, and second minimum is -1 or correct. The corresponding solution is to initialize the minimum to 0 , the second minimum to -1 , and loop from index 1.

A function calling another assumes that the latter satisfies its contract. Preconditions of the callee propagate to the caller, and for a high-level function, the exact contract can be hard to specify in some cases. E.g., correct video streaming produces an error message if networking fails. The best policy is for the high-level function to take responsibility for the preconditions of the functions it calls, and meaningfully propagate these to the caller only if needed, often augmented with more context. Often the API designer needs to account for this beforehand, while also staying close to logical building blocks that the users might expect.

Reasoning in terms of preconditions, postconditions, and invariants is probably as far as practical formal reasoning goes. Algorithms should be designed to be correct from the start and not, e.g., coded until all the tests pass. Ad hoc development is still useful, e.g., if think through the main logic, code and test, and finally think through the special cases. But an algorithm is incomplete without understanding of its properties, and lack of it should cause some mental discomfort. Still, some algorithms (such as the bootstrap—see the "Computational Statistics" chapter) are often used without complete understanding of preconditions and postconditions because these haven't been discovered yet.

Using various logics allows formal proofs of various algorithm properties, but for anything nontrivial need automated tools, and even they can handle only relatively small problems. Some useful tools include **static analysis** (such as compiler warnings about potential problems such as discovery of unused or uninitialized variable) and **model checking** (trying to find a counterexample for a particular statement by enumerating all possible examples).

For further reading on semiformal reasoning see Gries (1981).

1.5 Asymptotic Notation

An algorithm working on an input of n items uses $r(n)$ resources, which for a large enough n , any constant b , and some constant c is:

- $O(f(n))$ iff $r(n)/f(n) \leq c$
- $o(f(n))$ iff $r(n)/f(n) < b$
- $\Theta(f(n))$ iff $r(n)/f(n) = c$
- $\omega(f(n))$ iff $r(n)/f(n) > b$
- $\Omega(f(n))$ iff $r(n)/f(n) \geq c$

Only the first two are regularly used. Common $f(n)$ include $1, \lg(n), \sqrt{n}, n, n\lg(n), n^2, n^3, 2^n$, and $n!$. Use the slowest growing one that applies. To check if a function grows faster than another, can use the L'Hospital's rule.

Asymptotic notation simplifies analysis by ignoring constant factors. Doing $O(f(n))$ computation $O(g(n))$ times is $O(f(n)g(n))$. E.g., if the body of a loop over the input takes $O(1)$ time, the runtime is $O(n)$.

Superlinear memory and superquadratic runtime aren't scalable for typical tasks. But don't be afraid of algorithms that need more if they are the only good ones around. E.g., most useful operations with matrices (see the "Numerical Algorithms—Introduction and Matrix Algebra" chapter) need $O(n^2)$ space to store the matrix and $O(n^3)$ time to do something useful with it, and these are subtasks in many optimization and machine learning algorithms. They scale to medium input sizes ($n \approx 1000$), which is enough for many practical problems that would otherwise have no effective solutions at all.

1.6 Machine Models

Want highly simplified interfaces that highlight the main features of the hardware. A **machine model** is a set of operations with known contracts and efficiency. Some are mostly theoretical, but all are important. All programs consist of assembly language instructions directly supported by the CPU and other hardware. The CPU operates on integer words of size $w = 32$ or 64 , allowing 2^w memory locations. A `double` provides limited-precision real arithmetic and has 1 bit for sign s , 52 for word w , and 11 for exponent e to approximate real numbers as $\pm(1+m)2^e$ (see the "Numerical Algorithms—Introduction and Matrix Algebra" chapter for more details). The CPU accesses caches, memory, disks, and other resources to get data and simultaneously executes many programs, which compete for resources. Simplified models assume different operation costs. For modern computers:

- Arithmetic, logical, and bit operations on `ints` and `doubles` and function calls take $O(1)$ CPU cycles, with all except divisions taking about the same time and conditionals slowing CPU pipelining.
- Reading and writing from cache and memory take $O(1)$ cycles, but CPU registers are ≈ 100 times faster than memory and 10 than cache. Sometimes this $> O(\lg(n))$ factor in complexity.
- Dynamic memory allocation and deallocation are done in software and take large $O(1)$ time even for large arrays, without the cost of constructors and destructors.
- Accessing devices such as disks, external drives, screens, and network routers takes thousands of cycles, though $O(1)$ if the exchanged data size is bounded.
- Waiting for a lock can take a long time, and managing it takes hundreds of cycles.

Models designed to simplify analysis ignore small constant factors and consider only particular bottleneck operations:

- **Word RAM**—all operations are on words of size $w > \lg(\text{input size})$ and take $O(1)$ time. Memory is unlimited. Don't have other resources. Most algorithms assume this.
- **Real RAM**—word RAM with infinite-precision `double`. Avoids dealing with rounding errors.
- **Representative operations**—all operations except the assumed bottlenecks cost 0. Helps to focus on the relevant operations.
- **I/O**—accessing various devices, particularly disks, takes large $O(1)$ time when exchanged data size $\leq B$. Useful when device access is the bottleneck.
- **PRAM**—word RAM with unlimited parallel processors and shared memory; communication between processors costs 0. Shows the limits of scalability.

Some models assume upper bounds instead of assigning costs and technically aren't machine models:

- **Real-time**—computing answers has deadlines, missing which is too costly.
- **Stream**—have $O(1)$ memory, and inputs arrive continuously. Processing each may be real-time, e.g.,

- for a network router. Generalizes the I/O model.
- **Massive data**—processing trillions of inputs stored on disk or streamed. Requires scalable processing times of maybe $O(n \lg(n))$.

1.7 Randomized Algorithms

Randomness is very useful in computation but unfortunately not well understood among programmers. Let's first consider common games. \forall game \exists an optimal randomized strategy even if \nexists an optimal deterministic one. This is the **Nash equilibrium**. E.g., for rock-paper-scissors, the random choice draws on average, but a deterministic strategy is learnable. Randomization may allow processing inputs unpredictably with good expected efficiency despite the much worse deterministic worst case. Randomized algorithms use a pseudorandom number generator such as `rand()` to make decisions.

A **Monte Carlo algorithm** gives wrong answers with some small probability. A special case is a decision algorithm whose "yes" is correct and "no" wrong with probability p . Running it k times and getting "no" lowers the failure probability to p^k . A more general strategy is to combine answers from different runs by a majority vote, but it's harder to analyze.

A **Las Vegas algorithm** has expected efficiency. Repeating a Monte Carlo algorithm that has an answer verifier until the answer is correct creates a Las Vegas algorithm. The probability of a randomized algorithm's error or excessive inefficiency is often negligibly small. To analyze a Las Vegas algorithm consider its luck:

- The worst luck and input give the worst case
- The best luck and input give the best case
- The best luck and the worst input give the lower bound

Randomized algorithms are often the most efficient ones, e.g., for finding strings in a large text (see the "String Algorithms" chapter), testing if a number is prime (see the "Large Numbers" chapter), etc.

A minor issue in applying Las Vegas algorithms is their variance in metrics other than the runtime. In some cases this influences the results in an unpredictable way—e.g., with equal values, different random permutations result in different minimum calculations. For algorithms that approximate the answer, the quality of the result can differ from run to run.

1.8 Measuring Efficiency

To measure resource use one of the:

- Maximum/worst case—gives complete confidence, but can be much higher than the average.
- Average over the algorithm's random choices or input randomness—with the former the result is input-independent, but sometimes only the latter is available. In some cases the median makes more sense but is rarely used.
- $\Pr(\text{use} > \text{some number})$ —acts as worst case for Las Vegas algorithms when negligibly small.
- **Amortized**—the average worst case for a sequence of operations—allows occasionally expensive ones.
- **Competitive ratio**—the maximum multiplicative factor by which a perfectly lucky algorithm beats an **online algorithm** when receiving inputs one by one and maintaining the best answer. E.g., LRU cache replacement is 2-competitive. This is mathematically challenging to calculate and is done by researchers for particular algorithms.
- **Smoothed**—the average of randomly perturbed worst case input. It's between worst case and random by input, designed for algorithms with poor worst case and good practical performance (Müller-Hannemann & Schirra 2010). This too is best left to researchers to calculate.

Each can be:

- Exact—the complexity expression has $O(1)$ coefficients and relevant lower-order terms. Makes sense only if primitive operations have known exact costs but is complicated and rarely done. At best get approximate results because modern compilers do many optimizations.
- Asymptotic—efficient solutions are timeless, machine-independent, and easy to compare but allow hiding huge $O(1)$ factors and justifying algorithms superior only for unrealistic inputs.
- Experimental—estimated from running the algorithm on different problems.

Maximum, average, and amortized are more informative in this order when of the same complexity, though amortized is better than average in some cases. E.g., an algorithm calling $O(\lg(n))$, amortized $O(1)$, and expected $O(1)$ algorithms is $O(\lg(n))$ expected amortized. Worst case propagates to the depending problems, but others don't.

An algorithm's contract defines an information-theoretic lower bound on the efficiency of any algorithm for the task. E.g., must read every input bit.

1.9 Data Types

Some theoretical properties of data types that help write better code. C++ STL was conceived with these in mind. See Stepanov & MacJones (2019) for more details.

A **data type** is a set of values. It's:

- **Well-formable** if any bit sequence assigned to an object of its type represents a valid value. E.g., if a Boolean is a char with 1 = true and 0 = false, what is 2?
- **Partially formable** if the default constructor creates destructible objects that can be assigned to. But can't have contracts if correctness needs construction from passed data.
- **Ambiguous** if the same bit sequence represents many logical values. E.g., dates with two-digit years don't distinguish centuries.
- **Uniquely represented** if every value corresponds to a unique bit sequence. E.g., for a Boolean represented by a byte, 0 is false, and anything else true.
- **Copyable** if it has a destructor, a copy constructor, and an assignment operator. The minimum requirement for placing objects into data structures.

The **value** of an object is what is obtainable from its public interface and is a subset of the object's state. A function is **regular** if replacing inputs with those of equal value doesn't change the output's value. E.g., random number generators, data readers, and timers aren't regular. Any function that uses properties of inputs not looked at by equality, such as memory addresses, isn't regular. Irregular functions are hard to test, so refactor them into regular and irregular ones.

Representation of a type is one of a:

- Word
- Pointer to a type
- Array of types
- Structure of types

Functions defined on a type are its **computational basis**. A basis is:

- **Minimal** if using it can do all useful operations with negligible efficiency loss. For simplicity prefer smaller computational bases, so resist the temptation to add extra functionality.
- **Expressive** if it includes convenience functions implemented in terms of the minimal basis. E.g., my vector implementation (see the "Fundamental Algorithms" chapter) can do vector-space arithmetic and append vectors of items.

Define classes with minimal bases and use nonmember functions or layers and facade patterns (see the "Software Engineering Essentials" chapter) to give more functionality. For a complex type, a complete computational basis may be unknown, particularly when it's extended with augmentations. So libraries provide a reasonably complete set of operations to satisfy almost all users, while forcing some to implement customized functionality.

A type without pointers is **plain old data (POD)**. A data structure provides **address persistence** if only deletions invalidate item addresses. A **concept** is a collection of data types with a common computational basis. E.g., copyable types define a containable item. This is like abstract algebra, where compute with structures of limited ability. In C++, templates are similar because can only rely on the subtypes and the functions defined within a type.

1.10 Algorithm Experiments

To study the behavior of an algorithm:

1. Pick factors to study. In statistics a **factor** is any property, such as input size or a problem-specific input difficulty category, that influences the measurements. Pick input sizes by doubling to the maximum values that patience allows. Algorithm and its parameter choices are also useful factors.
2. Pick metrics. Include resource use and operation counts for expensive or interesting operations and specialized metrics, such as solution quality for optimization problems. Operation counts is portable, but machine and compiler optimization differences affect runtime and memory use.
3. Pick inputs. Try to represent every factor value, with reasonable discretization for continuous or large-range integer inputs. Common input types and their minor problems:
 - Real-world from industry—usually unavailable
 - From publicly available test libraries—can have biased use cases

- Randomly generated—can have special properties that make algorithms behave differently than with real-world inputs
4. Run tests in a controlled environment with the same light system load so that as much as possible only the picked factors influence the results. Run deterministic algorithms on all inputs once. If the algorithms or the environment are random, run enough times for reasonably accurate estimates. Many issues can come up—e.g., for my experiments Windows 7 gave 1/6 of the CPU to the processes running the algorithms, and Windows 10 1/10. Also compiler optimizations cut runtime by about a factor of four for a process that ran for several days.
 5. Analyze the results using statistical techniques:
 - Linear regression (or lasso; see the “Machine Learning—Regression” chapter) on operation counts and resource use may predict the latter as a function of the former.
 - Regression with resource use or operation counts and input-size functions including 1 , $\lg(n)$, \sqrt{n} , n , $n\lg(n)$, and n^2 may accurately show asymptotic performance. This is exploratory analysis, so, after discover the best fit, to avoid data peeking and multiple testing (see the “Computational Statistics” chapter) fit the chosen function with new data to confirm.
 - If determined input sizes by doubling, regression with \lg of resource use or operation counts might guess the polynomial performance exponent when applicable.
 6. Optionally publish your results. They must be **newsworthy**, i.e., lead to valuable, new, and reproducible conclusions. Be sure to specify inputs and implementations. Many papers aren't newsworthy by reporting:
 - Already published conclusions. This is generally OK only if do a better or a sufficiently different experiment.
 - Nonportable metrics for specialized machines. Generally not a problem if easy to compare, but showing relative and not absolute values and reporting machine and compiler settings sometimes helps.
 - Results with unsupported statistical analysis.
 - Results that don't lead to useful insights.

This basic template isn't enough to compare performance of algorithms in a statistically meaningful way. E.g., using a couple of inputs and saying that one algorithm is better than the other based on a result of a significance test on the averages is the most common mistake. See the “Computational Statistics” chapter for what must work out correctly for such a comparison to be meaningful. Also, should base choices on reasons other than experimental performance comparisons. E.g., for sorting algorithms would estimate and compare the constants behind the $O(n\lg(n))$ runtime.

1.11 Memory Management

The operating system hands out memory in large byte arrays of size 512KB–4MB, called **pages**. Any executable uses ≥ 1 . C++ `new` and `delete` partition pages into smaller blocks. An application needing more memory requests another page (or several if request > page size). Unused pages are returned.

On some machines, x -byte variables must be in an address divisible by x . The compiler does alignment unless variables are cast and reinterpreted. E.g., allocating `char x[8]`, placing a `double` there, and accessing it as a `double` may cause a hardware exception. Sizes for:

- `struct Aligned{double d; char c; } —16`
- `struct NotAligned{char c[9]; } —9.`

A memory manager adds padding and bookkeeping to each allocated item. It satisfies alignment without knowing the type, and supports raw memory requests `malloc` and `free` used by constructor-based `new` and `delete`. \exists theoretical limits on the worst case memory efficiency of any memory manager, but these don't occur in practice even in long-running systems. For well-implemented memory managers such as combinations of **first fit** and **segregated lists** (Bryant 2015):

- Operations take large $O(1)$ time
- Wasted memory per allocation is 4–8 bytes
- Can't use a fraction of memory consisting of small disconnected chunks

Most applications allocate many small items. See the “Fundamental Data Structures” chapter for such an allocator that also collects garbage.

1.12 Code Optimization

Many programmers over- or under-optimize. The justifications behind many such decisions are also often

wrong. The compiler's assembly output ultimately determines the efficiency. Unless need debugging, enable the maximum optimization level that produces a portable executable and doesn't change behavior (during a presentation I attended an IBM supercomputer expert claimed that from his experience too high of an optimization level can change runtime behavior). The compiler can, among other things:

- Evaluate constant expressions
- Reorder instructions
- Inline a function by replacing its call with its body
- Convert a `switch` into an `if` sequence or an array of jump pointers indexed by case constants

An excellent way to explore assembly generation is interactive compilation at godbolt.org.

Assist the compiler by making the code:

- Generic—express common functionality once, and reuse it. Don't copy-paste or manually do bloating optimizations such as inlining, loop unrolling, or replacing an expensive operation with a sequence of cheap ones. Don't overgeneralize by handling unneeded cases. Delete any unused code.
- Simple—short and self-documenting. Use consistent style, shortest descriptive names, and comment only high-level actions. Comments don't affect runtime behavior, so maintaining them is more difficult. Also programmers who don't write self-documenting code are unlikely to be able to explain it in the comments.
- Maximally restricted—use `const` when applicable, minimize variable scope, use minimal number of return statements, etc.

Efficiency should come mostly from better algorithms and domain logic. But sometimes manual optimizations help. Some warnings and pointers:

- Beware of machine-specific, undefined behavior. E.g., on mine $-5 \% 2 = -1$.
- Profile the code to see where it spends most resources. Try to cut out that activity or optimize it by improving the logic. E.g., disabling assertions by defining `NDEBUG` gains almost nothing but loses substantial debugging.
- Using `void*` instead of templates prevents inlining and optimizations that use information about the function. Use it only to hide implementation, compiling the `.cpp` file as a library, with the header using a template wrapper for type safety.
- Optimize alignment for objects in arrays—the compiler inserts dummy bytes for alignment, and `sizeof(structure) ≥ Σ sizeof(its elements)`. Declare bigger objects first; array size = its element size, and structure size = its `sizeof`; e.g.:
 - `struct Wasteful{char a; double d; char b;}` —size 24
 - `struct Smart{double d; char a; char b;}` —size 16
- The compiler might not optimize floating point arithmetic or code that depends on its result because doing so usually changes semantics. E.g., addition isn't associative due to round-off, so it can't rearrange computations.
- A loop evaluates its termination condition at every iteration, and the compiler moves it outside only if it can prove that doing so doesn't change semantics. So move expensive conditions outside just in case.
- Loading a large block of data from memory, processing it, and moving on to another one improves cache efficiency.
- Try making nested-loop and other simple-bottleneck computations not depend on conditionals—the compiler guesses which instruction will go into the pipeline next, and, if wrong, the CPU redoes the computations.
- Return large objects by reference to avoid extra copying unless the interface becomes clumsy, or the compiler can optimize the copy.
- Don't use **sentinels**—many algorithms assume ∞ or null items to simplify the presentation or avoid checks, but:
 - Branch prediction reduces the cost of saved comparisons
 - A general type doesn't have a logical ∞
 - A sentinel may need extra space that's unavailable
 - The computation is usually more difficult to understand
- Compute as much as possible at compile time, but don't over-complicate. Reduce the amount of code under a template. E.g., C++ pointer vectors have a `void*` specialization. If only a method of a class needs to be a template, don't make the class a template.
- Allocating memory in blocks of powers of two is efficient with all allocators. Beware that the OS

doesn't allow arrays larger than some size.

- Avoid static and global variables because their total memory use could be large. Put variables on the stack with maximally restricted scope.
- Converting an array of Booleans to a bit set saves space but slows down access.

See Hyde (2020) for more details and other tips.

1.13 Recursion

A recursive function puts local variables on the stack, calls itself, and waits for the return. For efficiency, put variables and parameters that don't change or appear before the recursive call inside `{}`, or make them class members.

Recursion depth is limited by the OS stack size. This also affects local variables and isn't portable (e.g., on my 2011 computer allocating an `int` array of size $> 2^{19}$ on the stack crashes). Make any recursive algorithm that needs a large stack nonrecursive. This also optimizes function call overhead and enables inlining.

Tail recursion is when the recursive call is the function's last statement. To convert it to iteration:

1. Put a loop around the function's body
2. Put any remembered variables outside the loop
3. At the loop's end, set the values of the parameters passed to the recursive call, and remove it

This simplifies the code unless it uses references. To remove general recursion, keep a stack of records to allow the algorithm to resume its work after a `return`, and put a loop around the recursive work that breaks when the stack is empty. This usually results in some simplifications, i.e., some variables passed as arguments don't need to be on the stack. To ease thinking, formulate the algorithm nonrecursively by mentally executing it.

Can analyze many recursive algorithms quickly using the **master theorem** (Wikipedia 2013): Let resource use $R(n) = f(n) + aR(n/b)$ for $n >$ constant C and $O(1)$ otherwise, and $k = \log_b(a)$. Then $R(n)$ is

- $\Theta(n^k)$ if $f(n) = \Theta(n^c)$ for $c < k$. The total work at the next level of recursion shrinks geometrically.
- $\Theta(n^c)\lg(n)^{+1}$ if $f(n) = \Theta(n^c)\lg(n)$ for $c = k$. The total work on all $O(\lg(n))$ levels is the same.
- $\Theta(n^c)$ if $f(n) = \Theta(n^c)$ for $c > k$. The total work at the next level increases geometrically.

1.14 Computation Strategies

Several strategies can solve many problems:

- **Divide and conquer**—divide the input into parts, and combine answers from each. E.g., quicksort and mergesort (see the "Sorting" chapter) split an array and work on each part separately.
- **Greedy**—at every step take the most promising action. E.g., Prim's algorithm (see the "Graph Algorithms" chapter) iteratively adds the smallest-cost vertex to the current MST.
- **Dynamic programming**—when the solution is a sequence of n actions, and $E[\text{the total cost}] = \sum E[\text{cost of each action}]$, to find the best action to step i , pick the best actions for steps 0 to $i - 1$ and then for step i . E.g., a path from A to B passing through C is optimal if the paths from A to C and C to B are optimal, so from B find the best path to each nearby C , and check which leads to the best overall path. Dynamic programming generally applies when can solve the problem by recursion much less efficiently. E.g., when calculating the n^{th} Fibonacci number, the recursion recalculates subresults such as the $(n - 2)^{\text{th}}$ number, which dynamic programming remembers.
- **Algorithm mixing**—if two algorithms are best in different cases, run each when it's best. E.g., STL sort uses deterministic median-of-three-pivot quicksort and switches to heapsort if the stack is too deep, giving fast average runtime and the $O(n\lg(n))$ guarantee.

1.15 Choosing among Several Algorithms

For a programmer the choice is usually simple—use whatever the chosen API provides. So this section applies only to when write the API yourself.

Most algorithms are legal and correct, but differ in extendability, simplicity, and efficiency. Simplicity gain leads to economic gain, but efficiency gain does so only if the algorithm is on the critical path, and the extra efficiency matters to the user. So **choose the simplest reasonably efficient algorithm unless a slightly more complicated one is much more efficient**.

Small-constant-factor efficiencies usually matter for widely used libraries. Implementing all algorithms where each is the most efficient for a particular case and deciding among them at runtime is a maintenance nightmare. It usually costs more in development time than saves in resource use, except when using a few

simple algorithms. But this strategy can be useful—e.g., when one algorithm is used for initialization and another for iterative refinement. In such cases should combine algorithms with similar resource use, i.e., don't want any to be much slower asymptotically or use much more memory than the rest.

Ideally want to avoid thinking and automate as much as possible. For things that can't be fully automated, it's important to know what to consider when choosing. In particular, in terms of any important algorithm property, it's better to avoid a bad choice than to make the best one.

For many domains choosing the algorithm isn't easy. E.g., for global numerical optimization (see the “Numerical Optimization” chapter) it's still far from clear even how to pick algorithms for a hybrid approach that tries a bit of everything. In such domains, with new algorithms being developed and old ones becoming increasingly more established, there is a temptation to use only whatever is believed to be the standard. While it's usually a good idea to do so for the first attempt, don't hesitate to experiment with new ideas or combinations. I made useful tweaks to many standard algorithms while writing this book.

1.16 Making Algorithms Parallel

Concurrency is useful when the program computes something that takes too long or interacts with several resources. E.g., many computations use clusters of computers, and a browser displays parts of a web page before it downloads fully. But scalable computations that don't use IOs usually don't need concurrency. Can decompose a problem by:

- Input—each processor handles a part. Applies to divide-and-conquer algorithms, randomized algorithms rerun with many random seeds, and systems that process independent events. E.g., in the PRAM model, unsorted array search takes $O(n)$ time and $O(\lg(n))$ parallel time by recursively giving each subarray half to a separate processor.
- Functionality—each processor handles a particular transformation (e.g., washer and dryer).

Can implement concurrency by:

- Separate processes—communication uses **message passing**, either directly or through a router. The processes can run on different machines. Message costs follow the IO model, and the OS need extra resources \forall process.
- Separate threads—a threads costs < a process and uses shared memory, as assumed by PRAM. Locks ensure correct concurrent access. But programming with locks is difficult and inefficient if many threads wait for a lock. Sometimes use the **actor model**, which is an abstraction of message passing.

If locking public methods, a data structure is atomic but not completely safe. E.g., a **race condition** occurs when two threads check if a stack is empty and pop, even when these methods are locked. Can lock, in order of increasing complexity and performance but decreasing applicability:

1. The code range containing calls to the data structure
2. Public methods of the data structure
3. To allow a single writer or an unlimited number of readers
4. Parts of the data structure accessed at the moment

Another issue is **deadlocks**. If two threads are waiting for two locks, and each gets a different one, none will make progress. A deadlock is impossible if all threads get locks in the same order.

Should parallelize the highest level of the program and not lower-level functionality. This is usually both more efficient and easier to maintain.

1.17 How to Implement an Algorithm

Though this book describes my implementations of common algorithms, it's almost never straightforward to create an implementation from a typical textbook description. The process is much more involved:

1. Read an easy introduction to get the basic idea
2. Code the basic functionality
3. Test it on some simple examples
4. Read other sources to figure out various choices and analysis
5. Experiment with promising choices if several
6. Code the final version with all choices decided
7. Clean up the code
8. Create comprehensive, preferably automated tests
9. Be on the lookout for improvements from new sources and research

The biggest issue is that almost always no single source is enough for complete understanding, this book included. But in many cases it + a basic introduction with many examples are enough for all the details.

A common mistake is not doing (8) right away. For many implementations I did this years later and had to reacquaint myself with all the details needed to process the examples and fix any found bugs.

No implementation is ever final. There is always more work with regard to (9), fixing bugs major and minor, adding more tests, and making improvements at least in using better subcomponents. It's better to have a finished implementation than a perfect one to have something useful.

I included an "Implementation Notes" section in most chapters to mention some specific difficulties in implementing specific algorithms and data structures and the steps I took to resolve them.

1.18 Recommended Classes for Computer Science Students

In college I took 6–9 classes per semester, and below lists the most useful classes from various departments.

Mathematics:

- Calculus—essential background
- Linear algebra—for numerical algorithms, optimization, machine learning, etc.
- Probability—the basis of randomized algorithms and much of mathematical modeling
- Mathematical statistics—probabilistic reasoning behind statistical inference

Optional:

- Stochastic processes—for simulation and understanding of some algorithms
- Abstract algebra—theoretical concepts relevant to design of many discrete systems
- Mathematical analysis—for algorithm analysis, familiarity with proofs, and general mathematical maturity
- Numerical methods—how computers solve equations, evaluate functions, etc.
- Complex analysis—for analysis of some numerical methods

Economics (optional):

- Microeconomics—decisions for an individual, including utility theory
- Game theory—making decisions given information about other stakeholders

Computer science (many of these are often electives):

- Algorithms and data structures—the core of computing, take as many classes as you can
- Computability and complexity—limits of computation
- Computer architecture—how hardware works
- Operating systems—how the OS works, important for general understanding
- Networking—basic communication protocols
- Databases—storing data on disk and accessing it with SQL
- Distributed systems—the next step after networking and OS, important for understanding real-world system structure, and essential for system design interviews
- Artificial intelligence—solving games and puzzles, logic, and probabilistic computing
- Machine learning—how computers do statistics

Optional:

- Computational geometry—strong focus on algorithms and data structures
- Optimization—linear programming, approximation algorithms, and metaheuristics
- Programming in a special environment—using various APIs to create real-world applications

The math classes are the most useful because the topics are much harder to learn on your own and give intuitive insight into various computations. It may be difficult to fit all classes you want in the schedule, particularly because degree requirements are more important, but some strategies can help:

- In U.S. high school: take as many AP classes as possible—many colleges accept these, and some offer summer and regular-term classes for high school students
- Plan for prerequisites—take the core classes as soon as possible because upper-level classes can be offered rarely, and the prerequisite chain can be long
- Take summer and winter classes—knock off general education requirements to make space for others during the regular semester
- Enroll in a combined bachelors/masters—may be able to get a masters in a year instead of the usual two by taking graduate classes early

1.19 Some Study Strategies

- Eager—actively process any new information, and try to understand it immediately. The most effi-

cient way to learn, unless the information is poorly presented or useless.

- Lazy—read about the topic in many sources, without spending too much effort on each. The goal is to use subconscious processing. Also benefit from several different expositions that complement each other.

The lazy strategy had been used by chess champion Tigran Petrossian to prepare for matches by quickly playing through recent grandmaster games and since became adopted by the former Soviet chess schools. It also has an advantage when want to become an expert—any single source can miss something important, particularly if the idea is complex or technical. An expert must have no doubt about their knowledge and be able to answer questions well, and consulting multiple sources helps with both.

A simple pattern to make yourself efficient is to concentrate on one task at a time to get mental locality of reference. E.g., Donald Knuth famously reads all emails once every few months, but this is also useful in a less extreme form. Humans are psychologically unable to multitask—at best context switch like single-CPU OS, with same kind of slowdown. Still, if done thoughtfully, limiting the concentration to a fixed, short time block allows and carefully saving the work allows no-regret breaks and avoids burnout. For large projects this is the only feasible strategy, and can share any intermediate work with others.

Formal education is perhaps most useful by teaching to study on your own, which is the only choice for many more advanced or specialized topics. Typically need to get several books on the topic of interest, and go through them efficiently. Ideally, a book is:

- Clear—well-written, with well-explained concepts and many examples/illustrations
- Relevant—includes only useful material—watch out specifically for the authors' own research that's not validated by others
- Complete—includes all useful material on the subject

Some general patterns in book quality:

- Much older, classic books with many good reviews are generally worse than newer, well-written books—the notation, the available production technologies, and the audiences have changed over the years. A general expectation is that modern readers should be able to grasp the material quickly due to lack of time, but this wasn't a concern for older books. The subject matter is also often advanced by research.
- Books by software writers tend to be more practical, and books by class instructors more readable. But ease of reading shouldn't be taken to an extreme—it's better to assume proper prerequisite knowledge than to dumb down excessively or do an insufficient “quick introduction” or a “hands-on” API tutorial.
- The English-tradition PhD tends to be less mathematical than the European-tradition one. This also reflects on a book's contents and the style of the authors.
- Good books avoid complicated topics for readability; excellent books explain these with enough examples and motivation. With theorems want to recognize when they apply, so motivation and examples help more than proofs.

When several books don't discuss something that you want to do or are curious about, it's usually not because it's obvious but because it's unknown, and it's rational to not mention anything. I have seen this in many different fields, so it's generally best to read a few latest books and papers and not hope that well-reviewed and heavily cited older books have the answers—they usually don't. So accept some seeming contradictions or incomplete understanding, and move on.

To save time must select good sources of material. For papers, read the abstract to decide whether to keep going. For books, check the table of contents, any available previews, such as those on Amazon and Google books, and reader ratings (but beware that out of everything that goes into a book's quality, ratings tend to reflect mostly readability; also beware of **logrolling**—professionals praise each other's work). In many cases research papers have more information than books and are usually free. But it's common for a topic to be developed in sequence of papers, and a subsequent book will organize all the information.

1.20 Projects for the Whole Book

- Every nontrivial piece of code must have an automated test
- For every function, check for const-correctness and reasonable input validation via assertions
- Make sure header file variable definitions are static const to avoid multiple definition when included in several cpp files

1.21 References

- Bryant, R., & David Richard, O. H. (2015). *Computer Systems: a Programmer's Perspective*. Addison-Wesley.
- Gries, D. (1981). *The Science of Programming*. Springer.
- Herlihy, M., & Shavit, N. (2020). *The Art of Multiprocessor Programming*. Elsevier.
- Hyde, R. (2020). *Write Great Code, Vol. 2: Thinking Low-Level, Writing High-Level*. No Starch.
- Kao, M. Y. (Ed.). (2016). *Encyclopedia of Algorithms*. Springer.
- Müller-Hannemann, M., & Schirra, S. (Eds.). (2010). *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*. Springer.
- Smullyan, R. M. (2014). *A Beginner's Guide to Mathematical Logic*. Dover.
- Stepanov, A. A., & MacJones, P. R. (2019). *Elements of Programming*. Semigroup Press.
- Wikipedia (2013). Master theorem. https://en.wikipedia.org/wiki/Master_theorem. Accessed July 30, 2013.

2 Software Engineering Essentials

2.1 Introduction

Creating an algorithm, a data structure, or a large program is in principle not different from creating anything else:

1. Decide what you need
2. Write the code
3. Verify that it does what you want
4. Correct any mistakes

While this works well for class projects, it doesn't for real-world development. Some complications:

- This is just a start. Need to at least deploy the code to clients.
- Making a few mistakes no longer gives you at least A-. Clients will leave and maybe sue along with third parties such as government regulators.
- All work is done by teams. Different people decide what is needed, write code, and maybe even verify and deploy it. Need techniques to make it easy for everyone.

2.2 Overview of the Development Cycle

Need a well-coordinated process for the interaction between the general roles of a **developer**, a **product manager**, and **clients** with contradictory needs. They all try to maximize economic value by working efficiently:

- Clients ask for new systems and new features for existing systems
- The product manager decides which of these are worth implementing and clarifies the requirements
- The developer writes the code and manages expectations regarding feasibility, completion time, performance, and other material factors

The development cycle is upgraded to:

1. The product manager creates requirements, taking feedback from the clients about what is wanted from the developer about what is technologically feasible
2. The developer does a high-level component design, breaking up the work into units of work. This is coordinated with the product manager if the units correspond to requirement milestones.
3. The developer works on one unit at a time.
4. The developer verifies the code, and the product manager whether it confirms to the requirements.
5. The code is released.
6. The client verifies that the functionality confirms to requirements.
7. Any of the above is repeated as needed.
8. The code is maintained forever after in response to additional requirements, newly discovered issues, and technology changes.

These are discussed in more detail in the rest of the chapter, whose content is based on my experience, various expert presentations, and years of reading software engineering literature. No few specific references can be given for everything, but consider the end-of-chapter references further reading.

Because of how the human mind works, unforeseen issues from the involved parties, and quick results from small code changes, want frequent feedback to address any issues ASAP. This is **agile software development**. A popular particular method is **Scrum**, though teams almost always do it in their own way by omitting or adjusting some activities. Typically implement minimal-functionality requirements, and add more features as needed. This is efficient because can pick any two of on time, on budget, and complete functionality. Never give up quality though—only less important features because poor quality of implementation slows down overall work.

Automated tools such as IDEs, profilers, static analysis, version control, debuggers, and code autocomplete bring substantial value for developers. For product managers, the most important tool tends to be a feature/bug tracking system such as Jira.

The most critical developer skill is the ability to model a problem abstractly. This comes to some degree from experience and problem-solving practice. Some working code is much better than other.

2.3 Requirements

The wanted functionality is usually specified informally or with **use cases**—complete user interaction scenarios with the system. Partial use cases are also useful for working on one small feature at a time.

Any clear description of the essentials of a feature suffices. Often this is text with simple drawings, entered into a system such as Jira. Much software engineering literature dwells on specialized notation such as UML for classes or component structure, but following it exactly is wasteful. Minor deviations of functionality are often created by developers and accepted by product managers.

External considerations often influence requirements. E.g., standards for UI design make the user experience consistent and more productive. Developers are naturally concerned with code and not UI, but the system is what the user sees. Producing quality work is easier when having a style guide or other quality requirements for the work; this isn't restricted to UI. This also frees up product managers to focus on other aspects.

Often the only reliable documentation is the user manual and testing instructions or programs, assuming they are kept up to date. Requirements and design decisions are often discussed informally and invalidated by subsequent changes. People who discussed them have vague memories and occasionally switch teams or quit.

2.4 Design of Component Structure

Give each component distinct responsibility, and maximize reuse. The resulting **system architecture** should have a modular, extendable component structure to allow changes without needing substantial modification or increase in complexity. Avoid coupling, and create components that are individually testable.

Team dynamics matter:

- Intrateam communication is easier and more efficient than interteam one, so component structure tends to follow team structure, and different teams should work on independent functionality.
- Code ownership is essential to not run into the tragedy of the commons where nobody wants to clean up or rearchitect the common code.

2.5 Design of Individual Components and Coding

Studies of developers show that they **usually work on one small problem at a time** by understanding it, coming up with an initial solution, testing it on some input, and revising until correct. Instead of following a top-down or bottom-up process, exploit mental “locality of reference” by moving on to a related small problem. So while design of the component structure is necessarily top-down, a flexible process is more important for individual components. Another common approach is risk-first—i.e., work on a potentially problematic API or a seemingly difficult integration piece.

Even before try to design one, consider what can be reused from previous or similar projects or other available APIs. Also most nontrivial projects **need learning** properties of considered technologies, users' needs, and properties of the deployment environment. Even picking an API is usually hard due to many choices. With every new one need to use the basic functionality, understand any nonobvious constraints, debug issues, and know enough of what happens under the hood. Often code some functionality without fully understanding even its API, usually because want a prototype for exploration, have no time, or implementing is the easiest way to learn.

The first step in creating a component is to create its API/public interface. Designing a good API is hard—as a general guideline, design as though it would become part of a language's standard library. In particular, use intuitive names for classes, functions, variables, etc., be consistent, and avoid unjustified magic numbers as parameters. Future programmers tend to follow conventions of the codebase and not their best judgment—e.g., poor variable and function names are rarely corrected. So, e.g., avoid deep nesting by using several returns. APIs should be code-reviewed even more carefully because they are harder to change. Good object orientation is essential, but don't design for the future too much—most such “future” will never happen. Also API writers need to create and maintain a searchable documentation with enough code examples for any component that will be reused by many other teams.

Usually spend much more time reading code than writing it. So investing into readability pays off unconditionally. Even for own code developers need to reconstruct the mental model that lead to it to make further changes. If rely on specific documentation, such as a webpage explaining uncommon use of certain API, put the link to it in a comment. If have many of these, consider creating a cached documentation repository for the project. Introducing new developers also becomes easier. **Disposable code** is very different from

permanent code. The former is for one-time tasks, such scientific experiments or class assignments. The latter must be maintained.

2.6 Patterns

Patterns are recipes for various system design aspects. The basic ideas are simple, and many diverse patterns are discussed succinctly in Buschmann et al. (2007). The literature spends much coverage on implementation in languages that use inheritance, whereas in C++ templates are more efficient unless concerned about code size. Inheritance is also a tricky concept sometimes—e.g., a modifiable square isn't a modifiable rectangle due to the base class incompatibility for modification. Below are the essentials of many useful patterns.

Architectural—for organizing system components:

- **Layers**—assign levels to components, and call lower- or same-level functionality from higher-level ones. Used in every system where a high-level business logic component calls algorithm, network, screen, and other libraries.
- **Pipeline**—transform data by a sequence of components. Each performs a specific work on its input and passes on the result. E.g., washer and dryer.
- **Model-view-controller (MVC)**—separate components handle data, how it's viewed, and responding to actions. The model contains the data and a list of subscribed views, to which it sends updates. The controller manipulates views or changes data to process requests. Organize views into a **transformation pipeline**, and separate data from its presentation. It's also useful to group actions by what they act on. An enhancement to MVC is **flux** (see <https://facebook.github.io/flux/docs/in-depth-overview.html>):
 - The controller is renamed to **dispatcher** and only forwards actions to the subscribed models.
 - The model is called **store** and is fully responsible to reacting to actions sent by the dispatcher.
 - In addition to better object orientation (with the store doing some of the controller's work), enforce direct communication. So with many stores, sending actions to each other through the dispatcher is more scalable than the direct communication of MVC.
- **Blackboard**—several knowledge sources update data in a shared repository. E.g., several people solving a jigsaw puzzle. Used for problems that need a complex solution strategy, mostly for AI.
- **Reflection**—interact with components using an additional interface, hidden from the users. Used to send configuration commands or collect usage statistics, etc.
- **Declarative configuration**—initialize components using editable settings files, making deployment more flexible.
- **Replicated components**—replicate very critical components, possibly using a different design. A manager component hides this from the caller, sending a request to all and returning the first or the majority answer. Allows parallel processing and taking components offline for maintenance.
- **Context object**—instead of maintaining state during component interaction, pass around an object with the state. Allows to not store state.
- **Safe interface**—public/private methods trust nothing/everything. E.g., in case of concurrency, public methods ask for locks, and private ones don't. For security, public methods check authorization, and private ones don't.
- **Data normalization**—avoid redundancy in data to minimize the risk of wrong and inconsistent data. E.g., prefer cheap recomputation over duplication.

Design—for higher-level code issues:

- **Factory**—to create a complex object with desired properties, encapsulate the creation in a separate function to isolate the complexity.
- **Builder**—if need several steps to construct an object, instead of a constructor with many parameters use an object that allows to construct parts and retrieve only the complete result.
- **Prototype**—to copy a complex object, create a cloning method that copies. C++ uses the copy constructor and the assignment operator (the move constructor is optional).
- **Singleton**—to restrict the number of object instances to one, create a static function that returns it, constructing on first call. Concurrent use needs a lock. Beware that this is only slightly better than a global variable, and this pattern rightly has a bad reputation. In many cases it's better to have a single instance owned by a component and give access to it to others—i.e., decide on the ownership.

- **Command**—actions are objects with an `execute()` method. C++ uses `operator()`. Used to give actions state and make them generic.
- **Strategy**—to change runtime behavior, assign a member object that causes the change. For behaviors determined statically, use a code, and for dynamically, a pointer to a command object.
- **Composite**—an object holds a list of subobjects from which it's composed, and an operation on it applies to its own data and every subobject. E.g., to `draw()` an object, `draw()` each subobject it contains.
- **Facade**—create an interface to join several related systems, possibly with a modified API. E.g., if you must call several functions in specific order, create a single function with the fewest parameters that makes the calls.
- **Proxy**—represent one object by another. E.g., `operator[]` for a STL bit set returns a reference object that's constructible from and assignable to `bool`.
- **Encapsulated implementation**—separate interface and implementation. Keeps the API implementation secret and allows shared library updates without recompiling the caller. In C++ use the **pointer to implementation idiom (PIMPL)**.
- **Iterator**—to go through all items in a data structure, create an object that can efficiently access an item, go to the next one in the wanted order.

Security:

- **Authorization**—pass an object holding access rights to anything that checks authorization. E.g., if a user is logged in, the object contains the login data.
- **Firewall**—check requests for safety before forwarding to applications. A firewall component validates things not considered by an application's input checks. E.g., it can drop requests that come too often from a single client.
- **Role-based access**—create roles, each with only the needed privileges. Simplifies privilege management, and allows secure access based on need.

Concurrency and distribution:

- **Broker**—each component communicates to others through a proxy. E.g., applications needing network access use a router. Reduces the number of connections, and decouples networking functionality.
- **Reactor**—a thread waits for and puts events into one of several queues, while other threads process events in each queue sequentially. Allows flexible concurrent processing.
- **Proactor**—requests are asynchronous, with a handler waiting for the results, identified by **unique completion tokens**.

Resource management:

- **Resource acquisition is initialization (RAII)**—get resources in constructors, and release in destructors. Ensures release in case of exceptions and multiple returns. In C++, use with {} for control.
- **Lookup**—a directory service tracks available services. E.g., can register services with network routers to make routing requests more efficient.
- **Resource pool**—keep a cache of resources, possibly leasing some if not using many. Related to the below patterns.
- **Lazy acquisition**—get resources at the last moment, and release as soon as possible. Maximizes resource availability to other applications.
- **Eager acquisition**—get all resources at the start-up. Improves reliability because secure everything needed or let others do so.
- **Partial acquisition**—get resources in stages. E.g., online video buffers and plays before it's fully downloaded.
- **Leasing**—give resources for a limited time, and take them back after it elapses. E.g., don't renew authorization for inactive users.
- **Reference counting**—A resource keep a count, and release if it = 0. Useful if accessing a resource from several places and using dynamic memory. In C++ this is done using a **smart pointer**.

Coding idioms

are language-specific patterns. E.g., consider:

- Looping in reverse given begin and end iterators (C++ arrays allow to point to the one-after-the-last element but not to the one-before-the-first):

1. **From iter = end to iter ≠ begin**
2. **--iter**

3. Loop work

- To declare a global constant such as π , in most languages a clean solution is to have a function return it
- In C++, to define operators that take template arguments of same type, make them `friend` functions of that type—used often in this book
- In C++, pass arrays using C syntax for generality—this reduces type safety though, so perhaps is best avoided
- Public methods use public interfaces—e.g., no private type declarations
- In a language without references, like Java, pass the result object as argument, and have the function change its data fields
- For exception safety, during a logical unit of operation, don't change state until secure the results of a potentially problematic code, such as resource allocation or some parallel calculation
- For a constant function called by both constant and nonconstant functions, constant-cast the result as needed

Patterns show up in many domains. E.g., for working with time:

- Can't convert a future wall time to a timestamp—the process depends, e.g., on a government's decision to abandon DST, so use UTC time
- Jobs scheduled near DST transition time can run twice or not at all

2.7 Error Management

A component can go into a **bad state** due to a:

- Bug
- Failure of a dependent component such as a database, a network connection, or a memory allocator
- Lag due to serving too many requests
- **Model-of-computation error** by memory corruption or loss of power

To avoid or limit loss of value for a critical system, consider:

- Limiting impact of errors
- Reporting errors to callers
- Correcting errors
- Putting the system into a good state

For critical components develop management procedures \forall likely error type. Patterns include (Buschmann et al. 2007):

- **Sanity checks**—check all preconditions, possibly invariants, and postconditions. Minimizes error effect on a component's state and helps catch bugs. For complex algorithms, some checks may be infeasible due to needing recomputation.
- **Execution trace**—critical computations record a summary of inputs, invariants, and used resources. The most useful trace is usually about utilization (how much of a resource is used), saturation (how much capacity remains), and errors. This allows to detect sources of errors and configure components dynamically. E.g., a manager component may restart a component that produced the trace for starting a computation but didn't produce the trace for finishing it after some time.
- **Known good state**—have a way to put the system into it. A restart fixes most issues in most systems. Keep a previous version of the component ready to be deployed in case the new one fails.
- **Redundant data**—avoids model-of-computation errors with high probability. E.g., if a bank loses electricity after debiting a source account and before crediting the destination, without redundancy someone loses money.
- **Dedicated components**—have several component copies, each serving a specific part of the functionality. A copy's failure affects only its part.
- **Multiple components**—maintains service if several components become unresponsive. The components should be separate processes at different computers and locations. This also helps with deployment because can transfer the users from the old version to the new one by gradually increasing the percentage of requests to the new version.
- **Replicated components pattern**—for avoiding random errors. Used in systems that can't be wrong, such as airplane control. With three independently developed components picking the majority answer substantially reduces $\text{Pr}(\text{error})$ due to handling a single unpredictable failure.
- **Functionality reduction**—put an erring component in a reduced-functionality mode where it does only critical operations. Avoids future errors, and is effective for performance issues.

A component is **robust** if it effectively handles precondition violations.

For testing robustness a common pattern is **failure injection** (also called **chaos engineering**)—bring down inessential or redundant parts of a system to see whether the main functionality is maintained, and if not then whether the existing trace, alarms, and recovery mechanisms behave as designed in dealing with failures. E.g., for some financial trading functionality this is required by government regulations to be done twice a year. Netflix's **chaos monkey** has been designed to bring down parts of the network randomly and is active all the time. But usually want a scientific approach where manually analyze a system for potential problematic dependencies, and bring down components that are inessential but may have been coded as essential by mistake. Also want to ensure independent and not cascading failure of components. Even if no issues are detected, this is a good way to train the developers to deal with potential issues.

Instead of bringing down components, can also slow down some responses to see the impact on overall system speed. Particularly, get an idea of what is likely to cause bottlenecks under increased request volumes. This usually needs specialized tools. In all cases need to find a failure model set that is guarded against because no clever logic will guard from complete failure of all computational resources.

2.8 Testing

Untested code is often wrong. Testing runs an algorithm with a subset of possible inputs and checks if the outputs satisfy the contract. If the subset \neq all inputs, testing can't prove absence of bugs but increases confidence that none exist and checks for performance issues.

Partition inputs into classes so that \forall contract condition \exists a representative class, and within a class pick inputs randomly. If this is difficult or inefficient, use random inputs for unbiased coverage; missing rare inputs during the finite testing time is OK.

Unit testing tests individual components, and **integration testing** the whole system. Theoretically, the latter is enough to eventually cover all cases, but the former discovers bugs faster because system test cases can miss a component's hard-toInvoke logic. **Automated regression** runs a system against many inputs. Such a test set usually evolves, starting from a simple **development test set** that checks that the basic functionality is working, and transitioning to a full regression test set to check many use cases. Also want the ability to verify certain properties of the results such as correctness and estimated efficiency automatically and get a reliable pass/fail.

It's best to add tests shortly after writing the code because can take the tried-on examples and convert them into automated tests before forget about the functionality due to moving on to something else. For existing code without tests should start slowly by adding one test at a time—even a trivial test is better than no test, at least for being a step in the right direction.

Black-box testing tests a system without the knowledge of the implementation, and **white-box testing** uses it to select inputs. For large systems, the latter is infeasible because much complex code can implement simple business logic. For small algorithms usually use white box testing to check the border cases, such as the smallest and the largest inputs, which developers tend to neglect.

To debug an algorithm, consider using automated tools to see stack traces and print out variable values. Also often use these common strategies:

- Binary search—print “hello” and variable values at certain points in the code until narrow down to the line that leads to the crash.
- Continuation—when have a correct program and a wrong one that are similar and have the same test cases, make a sequence of changes to turn the correct one into the wrong one and test after each. The first change that breaks usually contains the bug.

Testing can prove absence of bugs with high certainty. If executing a use case gives loss 1 if anything fails and 0 otherwise, after n correct use cases, the chance of seeing a problem $\leq 3/n$ with 95% confidence (see the “Computational Statistics” chapter). Testing algorithms that output real numbers, such as statistical and numerical ones, is more difficult because the answers change from run to run, but are nonetheless correct. Here should define a safe range for a correct answer as acceptable output, e.g., matching the known answer to a few decimal places. For something like random generators that fail statistical tests with some probability, and fail the test only if the failure frequency of the mechanism is exponentially-unlikely high.

Extensive testing is generally what distinguishes **production code**. It's not enough to check all border cases and put exception-throwing checks on inputs and other unpredictable code sections. Production code has been used by many callers and no critical bugs have been reported. It also has hooks for debugging (the reflection pattern), such as the ability to log some computational details for potential future debugging. It's almost never fully completed because eventually discover enhancements. For deployment to production need to make an is-it-good-enough judgment call after certain quality standards have been met, such as

completing testing and code reviews.

Another useful technique is **fuzzing**. It consists of testing with random inputs with only a limited result verifier. E.g., at least know that no input causes a crash. So want the code to assert as much as possible. Using specialized tools also allows to detect conditions such as out-of-memory accesses. Most such tools also save problematic inputs and generate bug reports with them.

For legacy code start with **characterization testing**—first run the system to create a set of results. Then check that any changes such as refactoring and component upgrades lead to equivalent results. Eventually such tests evolve to automated regression tests, at which point the code is potentially no longer legacy.

Test cases are also a part of the documentation. Ideally, they are executed after every change to the code base, and so are always up to date. So convert any comments and documentation that gives examples into tests that check the expected response. The tests can then be automatically included in any documentation.

2.9 Code Reviews

First, a developer is expected to test the code on their own. Often some routine or new automated tests are run. After this is successful, the code is checked into a version control system such as git. To be incorporated into the project, the code must be externally validated.

Code reviews have proven very effective at validation. Here another developer looks at the difference between new and existing code in the project and decides whether the changes are OK as is. They can be rejected for many reasons, including a:

- Bug not caught by testing
- Lack of clarity in the code or the comments
- Better way to do something
- Style or coding standard inconsistency

I believe a single code review is enough, though in many projects request several.

Often the comments are “nits”—i.e., changes that are desired by not blocking such as comment spell checks. Useful code review types include:

- Comprehensive review—review everything, understand the logic, and give complete advice with nits if needed
- Safety review—for experimental changes only check if anything important will break, and accept if convinced that not even in presence of flaws in proposed experimental logic
- Stamp—approve very quickly in case of urgency to facilitate recovery from discovered bugs

2.10 Release

Releasing new functionality is critical and must be done right—it's far beyond handing in a homework project in a class. After developer testing and code reviews, the software should be deployed from the main repository to a **test environment** and tested significantly by developers, dedicated testers (if any), product managers, and sometimes even clients. Code reviews and frequent testing of all functionality substantially reduce bugs. If notice a bug, scan the codebase for similar ones—likely to find some due to at least “reference” copy-paste.

The next step is releasing to the clients. Make all stakeholders aware of the release, and give explicit instructions for how to undo it in case of issues. In some companies this process may be audited when the software is part of a government-regulated workflow, such as for trading. Any breaking changes are problematic because they are costly to the users and free or even beneficial to the developers.

An important pattern is not to release everything at once. At first release a little bit, maybe to a percentage of clients or to dedicated clients. Then to a few more etc. Can also release by physical computer servers in a cluster where the software runs. This helps in many ways:

- Only a small percentage of users will be affected by a bug that slipped through, which happens often, so this will reduce reputation risk.
- Regulations might require certain quality of service to be maintained, e.g., if have local power outages, etc. So any crashes due to bugs will be limited in impact.
- Many systems have failover mechanisms where a message to a service fails over to another server if not responded to. So a bug that causes a server to crash on a bad message will bring down the entire cluster if released everywhere at once.

For software that the users download and run locally, slow release is more complicated but can be done by one of:

- Creating stable and latest versions—many open-source projects do this

- Deciding the version randomly, based on the user's public information such as an IP address
- Having both old and new functionality in the software, and allowing it to decide randomly which one is used, maybe with the choice made during the installation for consistency.

Many companies created **site reliability engineering (SRE)** teams who manage the release process instead of the developers. With good automatic bug and crash reporting, some of this can even be done automatically by deploying to more clients every few days if no issues are reported in the previous release. See Beyer et al. (2016) for details on many aspects of SRE.

2.11 Maintenance

Developer turnover, lack of documentation, and legacy technologies make parts of a system difficult to maintain. Systems with such characteristics and simple business logic are often completely rewritten. Typical systems are large after years of development. Usually things are the way they are because they got that way, and not because of some well-thought-out design. Code changes in response to new knowledge by replacing current implemented components with better ones. Infrastructure improvement features and requirement revisions often make future development more efficient and must be invested in for its sustainability. Upgrades of functionality, particularly APIs are difficult because of **Hyrum's law**—any observable behavior will be depended on by somebody. So often can't avoid making a few users unhappy because it's not cost-effective to support their favored functionality. Companies commonly make their low-level libraries open source to cut maintenance costs.

System rewrites are problematic because **some important use cases are expressed only in code** and invoked too rarely to be noticed. Migrating databases is particularly problematic because the data can be modified or entered incorrectly, common-sense-encoded to bypass system constraints, not updated to reflect reality, or viewed through a special, unknown error-correcting logic. **Automated regression** of all use cases is the best way to avoid mistakes, though it can still miss hard-to-test use cases.

2.12 Estimation

Accurate estimates on how much time work will take are impossible due to unique aspects of each project, discovered only during design or implementation. To get a rough estimate, it's usually best to take the median of the estimates of several informed developers (see bias-variance decomposition in the "General Machine Learning" chapter for a mathematical justification). To improve accuracy, have a discussion to find out the reasons behind any differences. Sometimes prefer **story points**—a common strategy is to use a less precise "T-shirt size" category (small/medium/large/very large) as a more accurate and less binding estimate.

New features are scheduled depending on business priorities such as the number of users benefiting and outside requirements. Even with agile development this often needs long-term planning because, e.g., legal requirements don't allow deadline extensions. Though the functionality benefit is often misestimated, cost-benefit ratio based on it and resource estimates from developers is the best available guidance about which feature to develop next.

2.13 Following a Formal Process

Useful development activities produce value. **Eliminate wasteful ones**. Usually these are easy to distinguish because the useful ones have been validated—e.g., testing, code reviews, documentation, automation tools, etc. But sometimes it's hard to tell whether an activity is useful because this may be project-specific. So a development process should be suggestive and not prescriptive. Still some regularity improves productivity, so agree a process, follow it, and opportunistically look for ways to improve it. At least want to automate all common routines such as the build and the test environment deployment.

2.14 Managing User Data

Many applications, particularly those on the phone, offer personalized functionality that requires using the some data of the clients. So must consider natural and legal privacy issues. The main idea is to use as little data as possible:

- Don't ask for unneeded data
- Don't store data that the user can easily reenter
- Make it clear to the user about what data is collected, how it's stored, etc.

It's useful to classify user data into:

- **Sensitive**—e.g., passwords and credits card numbers—don't collect until needed, and store safely if

must remember

- **Personal**—e.g., name, GPS location, certain personal preferences—give the user an option to opt into having these used for functionality
- **Inensitive**—e.g., IP address, application settings—can use freely, but still keep private

2.15 References

- Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly.
- Buschmann, F., Henney, K., & Schimdt, D. (2007). *Pattern-oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing*. Wiley.
- Hoffman, D. M., & Weiss, D. M. (Eds.). (2001). *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley.
- Humble, J., & Kim, G. (2018). *Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations*. IT Revolution.
- Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley.
- Winters, T., Mansreck, T., & Wright, H. (Eds.). (2020). *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly.

3 Career Advice and Interviews

"The best interview question is to ask the candidate to write a letter to their grandmother" – Alexander Stepanov, from presentation "Advice to a Young Programmer"

3.1 Introduction

Will reading this book help you do better at interviews? If you have a few months to study the chapters through "Graph Algorithms", then definitely yes. If you have a few weeks or days, go directly to McDowell (2019) and read it at least three times.

What I cover in this chapter is mostly from my own experience, though quite a bit is from the references and numerous online blog posts.

3.2 Applying for Jobs

You can be a student or someone with experience—the process is the same. Don't make the mistake of applying to only one company—pick several, do some research about them, i.e., look at least their website (for basic info), glassdoor.com and teamblind.com (for employee happiness), levels.fyi (for salary statistics), and Wikipedia (for unbiased info). Ideally, a company has a promising and interesting to you main product and treats its employees well in terms of compensation and management. Generally large companies can afford to train new hires, and startups want very quick learners or those already familiar with specific technologies. In terms of reputation FAANG+, hedge funds, and unicorns (startups with $\geq 1B$ value) are all considered to be the top. Almost always a company has a careers section on their website to allow applications, though some, particularly hedge funds ask for a direct email to their HR.

About 20-30 such companies isn't too many to find—some won't respond, at others you will fail interviews, and some won't give you appealing offers, but in the end there want be a couple of good offers to choose from. Also, your first interviews will be a good preparation for the rest. Finally, any offers you reject in favor of the best one are sometimes valid for up to a year, so they serve as an insurance policy to some degree.

To do some math imagine that you have 80% chance of passing any given single interview, and that a typical interview set consists of 5 interviews, of which need to pass all to get an offer. Then the chance of an offer at a single company $\approx 33\%$, so expect an offer for every 3 interview sets, and the probability of getting somewhere among n interview sets $= 1 - (1 - 0.33)^n$. E.g., with 10 interview sets have $\approx 98\%$ chance of getting an offer.

3.3 The Resume

Assume that you found a company you would like to work for, applied, and got called for an interview after talking to their recruiter. For a company, the goal is to distinguish a suitable candidate from an unsuitable one and sell the position. Interviewers consider problem solving, coding ability, communication, general motivation, and interest in the company's work.

The process starts with a resume (a recruiter usually calls after reviewing it). Some guidelines for writing effective resumes:

- Check for online examples of good and bad resumes to get a general idea. If your college may have resume workshop or service, check it out (mine was pretty bad but still somewhat useful).
- Follow the standard template—contact header, education, internships, special projects, skills. Interviewers and resume reviewers want to see a good GPA and some coding projects to discuss during interviews. For senior developers experience comes before education.
- Don't include an objective—it's easy to get wrong and never useful.
- Phrase items in terms of business impact instead of what was done—e.g., "increased user base by 20% by releasing feature x".
- Triple-check spelling and grammar—if your resume has bugs, your code will too. Keep the fonts and the formatting clean—I once saw a resume in a combination of Times New Roman (the old MS Word default) and Cambria (the new default). Don't get creative with the layout—go for ease of readability.
- Don't emphasize languages or technologies—companies want problem solvers, and you can get in trouble claiming to be an expert in something that you aren't. Most candidates with a long list of programming languages have working knowledge of one or two and used the rest for a single class

project.

- Remove any irrelevant experience such as that restaurant job in college. For education/previous employment don't include any classes/projects that aren't special enough to talk about for a few minutes.
- Keep it concise—most impressive essentials only.
- Meet with friends and review each other's resumes. Feedback is always useful. Professional resume writing services may be worth the fee to create a first version that you then update yourself.

3.4 Behavioral Questions and Interviews

Typically an interview starts with a discussion of the candidate's experience or education. For experienced candidates the focus is on the former and for recent graduates on the latter. For recent graduates, it helps to have had an internship because it gives something to discuss during the interview and may prepare for questions that concern team software development practices. Must clearly communicate any previous experience on the resume. It's OK to not know some things, but must be able to explain everything. So remove from your resume experiences that are irrelevant or can't be discussed reasonably. I usually ask to describe a challenging part of "your favorite" project, but many interviewers pick an arbitrary one. If not a native speaker, remember that communication is about the ability to explain things to another person in a logical way, and not perfect language mastery.

A popular experience-question technique is **STAR**—describe a situation and a task within it, the taken actions, and the results. The candidate should understand and have used the solution technology at least as a black box. Be careful to not disclose proprietary aspects of such projects because doing so is illegal and gives the impression of carelessness—sometimes this is difficult when pressed for details.

Many companies are doing **behavioral interviews** for experienced candidates. This is the next step over simple, open-ended project questions. Typically get asked "describe a time when you faced situation x". Usually x is something to highlight your leadership such as reaching out to another team. But it can also be a something less comfortable such as handling a negative feedback. To do well on such questions:

- Have a list of your most impressive projects over the years, for which you can give reasonable non-proprietary details
- Be ready to admit minor mistakes and explain how you learned from them
- Be honest and stand your ground when challenged—e.g., somebody once asked me why I did some minor work instead of my main project work, to which I responded that when feasible I take small percentage of time off main projects to not stall important minor ones.

It makes sense to structure your career at your current company to be ready to answer behavioral questions in future. E.g., try to maximize impact by working on most important projects—those that bring maximum benefit with least work. It also makes sense to prefer bigger projects over smaller ones because they are easier to talk about. This all has an extra benefit of helping on performance reviews in the company you work for.

Some basic knowledge questions may follow. These are typically reasonable and depend on what the candidate claims to or should know. E.g., everyone can be asked about stack vs. heap memory allocation. Those who took operating systems or have experience with multithreaded programming can be asked about deadlocks. Some other typical questions:

- Compare any balanced tree and any hash table from a user's point of view, such as for Python or Javascript built-in dictionary implementation.
- Describe what happens when running a recursive function that allocates an integer on the heap and calls itself with no base case.

3.5 Technical Interviews

Most companies prefer strong critical thinking and basic knowledge over extensive knowledge and medium critical thinking because software development needs constant learning. So the main part of the interview concentrates on technical questions. The main question types:

- Coding—implement a simple algorithm. A correct answer demonstrates coding skills and basic problem-solving.
- Puzzles—good ones are purely mathematical and not outside-the-box. These test more complicated problem solving.

Lately the trend is toward exclusively coding questions, and some companies even require this. Also the questions often come from a company-wide repository, so company-specific research often helps.

The communication aspect is much more important than it seems. Interviewers look for the ability to discuss the problem and ask clarifying questions, sometimes even deliberately omitting crucial details. Many candidates jump into solving the problem without fully understanding all stated requirements. It's crucial to load the problem in your mind first. Pay attention to all mentioned information because usually need all of it to solve the problem.

A good psychological approach is to have the "soldier/scientist" attitude. Military organizations evolved a standard for profession conduct—be honest, follow directions to the best of your ability, report failures, and be confident. Same for scientists—collaborate to solve problems by considering other people's work (interviewers' suggestions here) and abandoning own ideas in favor of currently best ones. Contrast this with a salesperson who defends their product (problem solution here) no matter what.

All it takes to solve most questions is to have a clear invariant \forall part of the solution. Don't present rough draft code as a finished solution without making an effort to check it—mentally test it with a few use cases. Also pay attention to the API, and try to handle corner cases. If this is problematic, ask the interviewer about what they want instead of ignoring the issue. Modern interviews are mostly done in an executable environment, so always offer to run the code on a simple test case.

Interviewers reuse questions or take them from various compilations. Study such sources to know what to expect. Candidates who prepare generally do better. Interview preparation is like SAT preparation in high school—it helps to get a better score. Don't expect to ever be asked a question you saw—out of many interviews I did, this has rarely happened. But similar questions to which can apply learned solution techniques will be seen.

Effective questions have decision power like statistical tests (see the "Computational Statistics" chapter). I.e., a bad/good response should respectively mean unsuitable/suitable candidate. Some effective questions from my experience:

- Simpler type—to test basic coding ability:
 - Reverse a string where consecutive numbers represent single "logical" characters—e.g., $f("123hi45") = "45ih123"$.
 - Calculate the n^{th} Fibonacci number—e.g., $f(7) = 21$.
 - Find the smallest and the second smallest item indices in an array, which are different even in the presence of equal values—e.g., $[20, -40, -10]$ leads to $\{1, 2\}$ and $[50, 40, 30, 30, 30]$ to $\{2, 3\}$ or any other combination of the indices of the 30s.
 - Describe how to find the intersection of two arrays, and analyze the solution's performance
 - Print a matrix in diagonal order—i.e., top-right to bottom-left.
- More complex—to test the ability to use CS knowledge:
 - Describe how to implement a LRU cache (hint—use a hash table with item = a pointer into a linked list).
 - Given a marathon track with n runners and m sensors that report passing runners, design a data structure to keep track of leading $k \leq n$ runners. Hint—use an indexed heap of k current best.
 - Describe how to maintain 5th and 95th order statistics in a stream. Hint—use two priority queues, and don't forget about edge cases.

Many other questions are sprinkled throughout the book to add to the discussion of the topics.

Interviewers are usually working programmers, so their ultimate impression of a candidate is about whether they want him/her on their team. So candidates should approach problems as they would if such problems were to be encountered during work. In particular, if you are stuck on a problem, make an effort to solve it, and, if still stuck after a short while, ask for a hint. Another strategy is to first try a simple, inefficient solution, and then make it better. This at least confirms a basic understanding of the problem. Present yourself as a confident, motivated problem solver who can be independent but collaborates. A candidate is usually expected to solve all easy problems and show good thinking on some hard ones. They are evaluated relative to others, which eventually converges to a specific good enough performance (see the "Computational Statistics" chapter for a discussion of estimands and estimators). So not finishing a hard question or not getting the perfect solution is usually OK. In terms of feedback companies rarely tell anything beyond offer/no offer to reduce minor legal risks to themselves, but finishing a question early is almost always a good sign that the interview went well.

A problem can have several good solutions, and the interviewer usually expects a particular one. General possibilities:

- The engineering solution—simplest in terms of the functionality available in common libraries and efficient enough for the stated use case. This is what you usually do in real work.
- The textbook solution—O-optimal, reasonably simple, uses one or more commonly taught algo-

rithms and data structures. This is how someone with good CS knowledge would answer.

- A clever, optimized solution—usually the most efficient way to solve the problem, but sometimes not obvious, and perhaps only implemented in library code.

Most interviewers are most happy with the last solution, accepting of the first as a starter, and OK with the second. So, given that may need to discover more than one solution, don't lock yourself in a particular way of thinking. Many candidates make this mistake, and, once locked, no amount of hinting helps.

One of my favorite questions is finding indices of two smallest elements in an array. This question was derived from a more complicated subproblem of Nelder-Mead optimization (where also need the maximum—see the “Numerical Optimization” chapter). As stated, the question is incomplete because it's not specified what should happen if the array fails to have ≥ 2 elements. Most candidates ask eventually, and I say that the answer is undefined, so need input checking, e.g., with an exception.

First decide on the function declaration. For two smallest indices (if asked I confirm that no generalization to more is needed), STL `pair` is a good way to return the answer. Some candidates are confused by needing to return more than a single value, which is what most languages allow. Others try to return arrays or vectors, in which case I press for details. Often the array is incorrectly allocated, or one piece of code allocates it, and another frees. With a C++ `vector`, a follow up question is to compare its total memory use with that of `pair`. Because `vector` uses dynamic memory and has some bookkeeping variables on the stack, it uses about five words of memory (see the “Fundamental Algorithms” chapter). Some candidates get confused about needing to separate stack and heap memory.

Next solve the problem. A typical initial answer is to sort the array and get the answer from that. This destroys the association between values and indices, but most candidates adjust to use pairs of values and indices. Though a good engineering solution, it's not optimal, so I ask to solve without extra memory use and library help.

The next attempt of many is to reduce the problem to finding the minimum by doing it twice—ignore the minimum when looking for the 2nd minimum (I asked to code this many times, and it's not trivial to get the 2nd loop right). A follow-up is to solve with a single pass and explain which approach is more efficient from memory cache point of view. Those who studied operating systems know that a single pass results in fewer cache misses (assuming the array is too large to fit entirely in the cache) and is preferable, despite the same optimal $O(n)$ runtime.

In this one-loop solution keep two index variables corresponding to the two smallest values seen so far (and in-order), and update them when looping over a new value in the array. A common mistake is initializing both indices to a fixed value such as 0. This usually results in 2nd value's remaining at 0 and is discovered when testing with increasing value sequence such as $[-40, -10, 20]$. The -1 initialization and the 0/1 one based on the values of the first two elements are correct. Interestingly enough, so is the accidental starting at $\text{min} = 0$ and $2^{\text{nd}} \text{ min} = 1$, and looping from index 0. The rest of the code poses no particular challenges, except for forgetting to shift the indices properly when see a value smaller than both.

The next step is extending to return $k > 2$ smallest indices for $1 \leq k \leq n$. After working on the $k = 2$ case, the obvious solution is to use an array of k currently smallest values and a loop instead of the if statements. But, as the interviewer would quickly point out, this has $O(nk)$ runtime. A good intermediate step to a better approach is to create an array of value-index pairs and sort it. The $O(n \lg(n))$ runtime isn't too far from the optimal, but the $O(n)$ extra memory use is. For optimizing only runtime, the optimal expected $O(n + k \lg(k))$ solution is to use quickselect (see the “Sorting” chapter) to partition the value-pair array on the k^{th} element and then sort the left part. But this also has $O(n)$ extra memory use and is almost never mentioned.

Constrained to $O(k)$ memory, the optimal solution is to use the $O(nk)$ brute force solution, but with a maximum heap as the storage. The trick is to realize that all the brute force array loop is doing is ejecting the maximum of k current indices and i . Many candidates need a hint to replace the array with a dynamic sorted sequence. Some go for a balanced tree, which also gives $O(n \lg(k))$ runtime, but with worse constant factors.

For the implementation I ask to use a min heap from the candidate's language's standard library. In some cases the candidates don't know how to write a comparator that the min heap expects. In C++ make a heap of indices only, and write a class with

`operator<(int a, int b) const {return array[b] < array[a];}`, where `array` is a member-variable reference to the input vector (see the “Fundamental Data Structures” chapter for more on writing comparators; can also do this with a lambda). In other languages may need to make a heap of value-index pairs explicitly.

After the loop get the k smallest indices by repeatedly dequeuing into a result array. Most candidates don't realize that this returns the indices in reverse order. After being pointed out, this is quickly corrected by placing the indices to the end of the array instead.

The above must have made the false impression that the interview process is mostly mechanical, and good preparation guarantees a satisfactory performance. Many aspects are arbitrary:

- You might not feel your best due to not enough sleep or being anxious; other common issues like headaches happen unpredictably
- Some interviewers are more lenient than others and ask questions with fewer logical clicks (I am one of them!)
- You might be a good programmer but not solve the problem well within the limited time or have difficulty thinking out loud, as is usually requested

So the interviewing process is imperfect, but the best available, just like college admissions based on SAT or GRE tests. In my experience with problem solving, even a simple asked problem usually needs a day of thinking to work out all the details, and the first implemented solution, even if well-reasoned, is likely to allow for a number of improvements. The evaluation mostly depends on whether the interviewer feels they can have you on their team, so take a deep breath and try to do your best. If you show that you can work with them, they will work with you.

3.6 More Difficult Questions

A typical interview set has a technical screening interview and, once passed, a couple of main interviews. I have always been asked and asked at least one easy question such as the one discussed at length in the previous section. At very selective companies this can be taken a step further:

- Can have a time limit. E.g., I once had a automated screening interview which required to solve four medium-easy questions in 70 minutes. At another place I was asked an easy coding question to be done in 10 minutes.
- Can have a lot of time with a difficult question. E.g., I had questions whose optimal solution required knowledge of topological sort and the ability to modify it in such a way that the simple DFS-based version I cover in the “Graph Algorithms” chapter can't handle (only the more complicated, constant-factor-less-efficient queue-based version can).

During preparation using McDowell (2019) or other resources it helps to focus on common solution techniques. For problems with:

- Arrays—consider working from the back—helps manage space.
- Linked lists—use the 2nd pointer technique (called runner), where several pointers loop through with different increments and eventually meet. I only had one list question where this was needed because interviewers in generally don't have much experience working with lists.
- Trees—know how to implement pre-, post-, and in-order iteration. For level questions (e.g., compute sum of each level) the technique is DFS with an array or a hash table to keep track of level-specific data.
- Graphs—most asked problems can be solved by DFS. Don't use more complicated primitives such as union-find—they are too complicated to implement correctly in an interview—I tried.
- Dynamic programming—at some companies these are banned, but at least be comfortable to solve by recursion with memoization.
- Combinatorial enumeration—use recursive backtracking with a state object passed by reference; it need to have ability to generate possible moves, make a move, and undo a move. If only need a count of all possibilities then dynamics programming is often the answer.

3.7 System Design Interviews

These are usually about distributed systems (e.g., design a distributed hash table) and sometimes about product (e.g., design a book suggestion system). McDowell (2019) has a brief chapter on these, but it's not enough. For the former it helps to read Kleppmann (2017), and for the latter something like Cervantes & Kazman (2016), but it's also not enough. There are also a couple of good self-published “system design interview” books that are slowly getting better (search Amazon for an updated list).

The only way to do well is to have experience—infrastructure engineers will do well on the former, and full-stack, application engineers on the latter. Take this into account when planning your career.

A general outline in my experience is something like this:

1. Will get a semi-specified problem description such as design video upload and streaming for 1B daily users
2. Ask some clarifying questions about the scope and the APIs
3. First don't worry about scale and design a system that works for 1 user

4. After interviewer OKs will scale to the 1B users
5. May need to do some rough capacity calculations (usually in terms of requests per second) to estimate the number of machines or instances

Almost always the solution is a standard pull architecture with a UI, a service, and a database, with sharding for scaling. Occasionally need a cache and a message queue. Replication is also useful to discuss. And don't worry about silly things such as SQL vs NoSQL databases.

3.8 Offer Negotiation

If you did well on interviews in several companies, you will get several offers. The most common mistake is to accept the best offer right away without negotiation. It's a very human thing to do to not want to argue about money or assume that what is offered is somehow related to what you deserve. Recruiters give you offers depending on your interview performance and special aspects of your resume such as years of experience, advanced education, or knowledge of wanted technologies. Some tips for negotiation:

- Have no fear—read online articles, watch practice videos, and don't worry about having the offer rescinded after an attempt to negotiate or asking what seems like too much. But be polite because rudeness will rescind the offer. The company is happy with your interview performance and expects some negotiation—they will usually say “final offer” when not willing to concede further. It's expensive for them to find a great candidate, and they will suffer substantial reputation loss for rescinding offers on a whim.
- If you have many offers, tell the recruiter about the other offers. They are most likely willing to match them to some degree. But it's best to not say they gave x so please do better unless they ask for details. Say that I will accept your offer if I get y or ask for their best offer and you will sleep on it.
- Look at statistics at levels.fyi, and show those to the recruiter if your offer is worse than the average. Though offers of other employees have no relation to yours at some companies, it's a clean way to ask for more, i.e., “How can I accept x if others get y ?”
- Ask for a specific, realistic increase. A specific hard minimum (based on your current compensation or financial needs) or 10–20% more of the initial offer is a good start because recruiters will be willing to meet you at least part of the way. Even if not, they are likely to at least give you something small such as more vacation days or a relocation bonus boost.
- The offer is a package and not a single number, consisting of salary, bonuses, stock bonus, sign-on bonuses, etc. The flexibility of a company to give more on each of these is essentially in this order—e.g., often salaries are capped. So first ask for largest salary, then more stock, and finally settle on the sign-on bonus. This also shows a clear measure of progress in the negotiation to the company and buys you time if you expect other offers soon. Note that this advice contradict some negotiation literature that recommends negotiating the whole package at a time.
- Beware that equity at pre-IPO startups or unicorns usually isn't tradable—eventually it usually multiplies or goes to 0, so need higher premium to compensate for the risk.
- Beware that any promises by the recruiter such as future expected raises or promotions are void unless in the employment contract in writing.

Remember that any economic deal depends on supply and demand, so your negotiating position is related to the number of good offers you have, including your current position. Recruiters are usually nice and pass on your requests to a compensation manager, but, depending on the company, they can also be trained negotiators who have a fiduciary duty to use any tactic they can to get acceptance at the lowest total.

There is a large literature on negotiation, and it's best to be familiar with it. I recommend Fisher et al. (2011), Voss & Raz (2016), and Malhotra & Bazerman (2007), in this order.

3.9 How to Leave Your Current Job Nicely

If you accept an offer, give a two-week notice (or whatever is the standard), and thank the current company for the experience during an exit interview. But be careful with dates to collect any stock vests or bonuses or initial grants—companies use the resignation date instead of last-work-day date for these. Eventually they might want to hire you back, or you might want a reference. I once saw someone send a parting email with a list of frustrations about the department—while it was somewhat amusing, it contained no constructive feedback, and nobody would probably want to work with that person in future.

Don't feel bad about abandoning your current team. Everyone understands that moving on is business decision, and often former coworkers stay in touch and occasionally gather socially. In your last few days do

your best to train your replacement, but don't extend your stay excessively just to make this happen—they can choose to let you go earlier, including on the day of notice. So pick a timeline that does least harm to both you and them.

3.10 Fight Complacency

Plan to interview about once every 1-2 years even if you love your current position (more often if you don't!). You don't have to accept the offers you get if any, but the process keeps your problem solving in a good shape, tells you about what to improve about yourself, and removes complacency from being too comfortable in the current position. Even better, become an interviewer yourself—the experience is invaluable.

Anecdotal evidence is that many developers stay in a comfortable position with reduced learning and growth opportunities. This comfort is bad for career growth, so when in such a situation, look for interviews with other teams (if working for a large company) or companies ASAP—don't let the problem get worse. Even when you apply to another company, someone with 10 years of experience at different teams or companies looks better than someone at the same place, as long as they don't jump to often (i.e., typically stay for 2 years at the same company and 1 year at the same team). It's also easier to get promoted with diverse experience.

If you feel that you are still growing and learning well, still look for more opportunities to do so. Always prefer learning fundamental ideas over specific technologies—given understanding, tools are easy to learn, but not the other way around. I have seen this in presentations of where someone new to machine learning got their hands on some popular Python toolkits.

3.11 Projects

- A popular website for practicing interview question is leetcode.com. Many programmers try to solve one problem a day there. My opinion is that this is an unnecessary time investment when you are an interviewer but necessary when not. Do you agree?
- Solve a number of interview questions in other programming languages such as Python or Javascript. Does this give a better understanding of the question or how others might solve it or grade the solution?

3.12 References

- Cervantes, H., & Kazman, R. (2016). *Designing Software Architectures: A Practical Approach*. Addison-Wesley Professional.
- Fisher, R., Ury, W. L., & Patton, B. (2011). *Getting to Yes: Negotiating Agreement Without Giving in*. Penguin.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly.
- Malhotra, D., & Bazerman, M. (2007). *Negotiation Genius: How to Overcome Obstacles and Achieve Brilliant Results at the Bargaining Table and Beyond*. Bantam.
- McDowell, G. L. (2019). *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup.
- Sonmez, J. (2017). *The Complete Developer's Career Guide*. Simple Programmer.
- Voss, C., & Raz, T. (2016). *Never Split the Difference: Negotiating as if Your Life Depended on it*. Random House.

4 Introduction to Computer Law

4.1 Introduction

Law is a huge subject and is different in other countries despite ongoing standardization through various treaties. This chapter summarizes some useful general information about the US law, which continually changes. **To get a legal opinion, talk to a lawyer about a particular issue in detail.**

For developers the main goal is to avoid getting sued for innocent mistakes, so it's best to be familiar with some danger zones. Usually, even if a party has an actionable case, a suit happens only if it's expected to win enough damages to pay for the legal costs or the emotional factor is very high, and even before that a middle-ground settlement is likely. But this is only a minor insurance.

4.2 Intellectual Property

Knowledge is costly to create and most beneficial when secret. To encourage sharing, the law makes some shared knowledge property. Legally, a **property** is a set of owners' rights and using it in any infringing way without permission is illegal. For intellectual property, the permission is a **license**.

Property	Owned through	Software example
Idea	Patent or trade secret	Algorithm
Expression	Copyright	Source code
Association	Trademark	Program name

Figure 4.1: Types of intellectual property

4.3 Patents

A patent:

- Covers “**new and useful**” processes, machines, manufactures, and compositions, which respectively are a sequence of steps to transform some physical object, a device, items made by a human or a device, and a chemical formula.
- Gives exclusive right to make, use, or sell the invention or anything containing it for a limited time, usually 20 years.
- Goes into the public domain after expiration.
- Can be unknowingly infringed. You are liable if calling someone's infringing API, even if can't check for infringement.

Almost anything is “useful” as long as a person with “ordinary skill in the art” can recreate it without unreasonable effort. “New” means not obvious to such person at the time of publication and not published before the patent application. Generally, disclosing the invention means publishing it unless the disclosee signs a nondisclosure agreement. E.g., as a new parent you move the stroller back-and-forth and sing some lullaby to help your child fall asleep. You invent a robotic arm to do this—patentable (nobody invented this yet), at least until this sentence is published.

Can't patent things that are:

- Naturally occurring—discoveries aren't inventions.
- Purely abstract—an idea isn't a tangible invention. Algorithms are currently patentable, and what aspects are considered purely abstract is subject of many recent Supreme Court cases. Though the rulings have been mostly against algorithm patentability, increasingly more devices are relying on nontrivial software components. Future laws and cases will clarify.
- Not functionally—e.g., something artistic.

Applying for a patent is simple in concept but complicated in details and costly ($\approx \$10,000$ to apply and $\$20,000$ over the life of the patent as of 2010). An examiner does research and accepts the application or returns it for revision if rejecting some claimed novelties. At first most patents are overly broad and get revised until less broad claims are accepted.

A granted patent contains most importantly:

- A list of inventions and inventors for each. Each independently owns the inventions for which they

- are listed unless all assigned their rights to someone else, in which case that entity is the owner.
- Dates—the **issue date** is when the patent becomes enforceable, the **application date** when it's applied for, and the **priority date** when the invention is deemed to be conceived. The twenty years is now counted from the application date. In the past complex rules determined the start date, which was always between the priority and the application dates.
- Written descriptions of examples and illustrative drawings. These must be enough to allow reproducing the invention, or the patent is invalid. The application may not withhold important details.
- **Claims** to what is the inventive part—written broadly to avoid minor changes that produce logical noninfringements. It's an infringement to produce "essentially equivalent" functionality, but this may be hard to prove.

Damages for patent infringement may include:

- Lost profits
- A reasonable royalty
- An injunction against further use of the invention if not against public interest

If an algorithm is patented, even if it's obvious, use another to be safe.

4.4 Trade Secrets

A trade secret:

- Covers information that isn't publicly known, not illegal, derives part of its value from being secret, is kept reasonably secret, and not acquired by fraud. Companies strengthen that definition in employment contracts by considering anything internal a secret unless must share it with outsiders to perform work duty.
- Protects against disclosure by improper means or breach of confidential relationship. **Improper means** is any fraud, such as an employee bribe. You can't disclose anything learned by accident if have a good reason to suspect it's secret unless the disclosure makes it publicly known. Using a trade secret needs explicit or implied permission.
- A **confidential relationship** arises between parties doing business with each other and is breached if one reveals another's secrets. So many businesses don't view sealed offers that don't state that nothing contained is secret.
- Doesn't protect against independent invention and reverse engineering.
- Loses protection if it becomes publicly known, i.e., by sufficiently many people. E.g., if an employee publishes a secret online, it stops being secret, though the company can sue the employee.
- Needn't be uniquely owned. Several companies can own a secret to the same idea, but, if any discloses it, all lose it. In case of an infringement, only the owner from whom it was stolen is entitled to damages.

4.5 Copyrights

A copyright protects the exact expression of original works of authorship fixed in any tangible medium, in particular the rights to make copies, distribute them, and make derived works.

- Any record suffices. E.g., writing a poem on a napkin or a computer both qualify.
- Ideas and facts aren't protected. An expression isn't protected when needed to state an idea. A collection of facts isn't copyrightable unless the selection is creative; only a particular arrangement or presentation is copyrightable if creative. But in future it's likely that data sets will get some legal protection.
- Need very little originality. A combination of several elements is treated as a combination and not a whole if the combining is functional and not creative. E.g., dividing a GUI into components and giving protection only to components protected individually determines the look-and-feel protection.
- Owned by multiple parties if each created the expression without copying. E.g., for reverse engineering, a development team gets complete functional specifications of the product but isn't allowed to see it.
- Automatic and needn't be registered. "This material is copyright" isn't necessary in the U.S. but internationally is enough to own a copyright.
- Generally valid for the lifetime of the author + 70 years. For works before 1978 or **work for hire**, expiration is usually 95 years after publication. Anything before 1923 is in the public domain, but beware of copyright renewals for later works.
- The artist is the owner and not the machine used for creation or its owner. If a tourist asks you to

take a picture with his camera, you own the copyright. But need permission for any copyrighted work used by the machine. E.g., famously a monkey's accidental selfie was ruled to be public domain because the camera placer can't have reasonably intended this.

- If you produce copyrighted content as part of working for someone, they likely own it as work for hire. Because of exceptions, you usually sign an agreement to transfer all rights.
- Joint work by several authors is owned entirely by each author if they intended to combine their individual contributions. Derivative work isn't joint even if permitted.
- The owners' rights are very broad. E.g., downloading an illegal copy of a song violates their right to distribution because of the copy in the download.
- The **fair use doctrine** allows copying all or parts of a work in some cases, particularly small amounts for education or critique. E.g., taking a few pixels from an image or copying a legally acquired CD for personal use isn't infringement. But determining fair use is complicated and often unpredictable—e.g., Google Books was deemed to be fair use by the Supreme Court due to substantial public benefit, among other things. It's best to get an explicit permission.
- The **first-sale doctrine** allows a purchaser of a copyrighted work to resell it, with exceptions. Almost all software disallows resale in its license, which is usually binding.
- Substantial assistance of infringement leads to liability as does the combination of knowing of the infringement, having a financial gain from it, and being able to control it. That's why certain type of file-sharing services eventually get out of business. It's illegal to try or help break copyright protection by creating or selling devices for doing so. Content providers such as YouTube aren't liable for users' infringements if they quickly remove the infringing material when given notice.

4.6 Trademarks

A trademark is any symbol used commercially to identify goods. A **service mark** does that for services. Also, \exists **certification** and **collective marks**. Need to register a trademark to sue for infringement, and should do so to avoid disputes when many parties want it. "<Trademark>®" denotes a registered trademark, and "<Trademark>™" or "<Servicemark>℠" an unregistered one.

Only the owner can use a trademark to mark goods or services so that consumers associate the mark with their reputation. Intending to or causing a reasonable confusion or dilution infringes (e.g., "Yihaa" search engine or "micro-soft" underwear). An exception is artistic work, particularly parody, if it's clear that the trademark owner didn't produce it.

A trademark symbol can't be:

- Functional—useful information, such as a direction sign, can't be a trademark.
- Offensive, generic, or misleading—a random symbol is the strongest possible trademark; e.g., "Igmdk" works, but "Joe's Pizza" is generic. To pass a generic trademark, can make a distinctive spelling alteration; e.g., "Joe's ZzaPi".
- A variation of an existing trademark—to prevent consumer confusion. \exists exceptions for different industries and locations.

A trademark never expires but can be revoked if becomes too common or isn't used commercially for three years, and ! \exists intent to resume use. E.g., "zipper" and "escalator" became too common, and "google" is likely next. Using a trademark means displaying it, so that it's associated with goods or services, and actively defending it by suing all infringers. Registering a domain name is nonuse because registration alone isn't commercial use.

If several parties claim ownership, the one with earliest use wins unless one party has registered and used a trademark for five years, in which case it's **incontestable**.

4.7 Intellectual Property Management

Patents and trade secrets are mutually exclusive. But some trade secrets can become patents, and both work well with trademarks because after patent expiry or secret discovery the consumers used to the product usually don't switch.

A **standard** can be created to combine several intellectual properties to promote commerce. It's guaranteed to be free of intellectual property claims, but royalties are usually paid to the standards body.

It's important to keep good records of various contracts, particularly permissions by others to use their intellectual property. This serves as insurance against various misunderstandings, which happen occasionally.

4.8 Contracts

A contract is an agreement among several parties to have each do something specific. For it to be legally enforceable:

- An **offer** is made and, until rejected or expired, accepted by a reasonable means of communication such as mail, e-mail, or as specified by the offer. Negotiation of terms is a counteroffer and an implicit rejection. Asking questions about the offer isn't negotiation.
- \exists illegal activity. If a contract was made with intent of fraud or a party was forced to sign, it's void.
- All parties must get something in return (**consideration**) that isn't already owed to them. E.g., if you pay someone part of owed money, a promise that they won't seek the rest is void.
- Parties must be **competent**. A contract made with a minor or someone incapacitated is generally void. For a contract with a corporation, the individual making the contract on its behalf must have enough authority.
- All parties must fully understand all the **material terms**. E.g., "you hereby assign me the rights to your house" in small print isn't binding unless you explicitly acknowledge it (and even then in case of a house it wouldn't be). Common contracts such as leasing tend to be standardized, and all parties are assumed to know the common terms; the uncommon ones need specific mention. So it's safe to sign without reading the small print—even lawyers do it regularly.
- It must be in writing in cases such as buying real estate. For provability most contracts of nonnegligible value are in writing, but most oral contracts are valid. An electronic contract with click acceptance is as valid as a written one.

A party's performance is **complete** if everything promised is done with changes that aren't material, i.e., the other parties can't reasonably expect them to be essential, and the contract doesn't forbid them. If a party fails to perform, the other parties can sue but not force performance. The court may award damages or force performance if it's irreplaceable (e.g., sale of unique items) and not too damaging to the performing party.

In some cases, a contract is automatic to avoid injustice or unjust enrichment. E.g., can't refuse to pay a reasonable price after eating in a restaurant because you didn't explicitly agree to it before ordering.

4.9 Licenses

For software, a license is a contract between you and the owner. Without a license, you most likely can't use the property at all.

\exists much freedom in what the owner gets. An author's pride in someone's using his software qualifies. The six pack license asks for beer. An open-source license may ask to make your code available upon request if you use anything covered by it as a component. Another license may ask to show some notice. Most licenses ask for payment and specify that you can't resell or share the software. A combined license is possible, e.g., allowing personal and educational use under one license and commercial under another.

Licensing isn't a sale, and the first-sale doctrine doesn't apply. When a "license inside" label is present, in some cases acceptance of some terms happens at the purchase and not the "I agree" click during installation. But it's legal to reverse-engineer even if the license forbids it if you're not bypassing copyright protection. A license also contains:

- Payment terms.
- Quality guarantees and litigation control. Usually, a company mandates litigation in jurisdiction favorable to it and disclaims all liability in case of improper operation, intellectual property, and other claims by other parties.
- How to update and ask for tech support. New version updates may be mandated.
- Use conditions. An **exclusive license** means the owner may not license the property to anyone else. With a low fee, it can make intellectual property almost useless. Open source licenses may require making any derivatives public.

To license your property, use the existing popular licenses and not your own because you can incorrectly specify too many things.

Open source projects pose some extra challenges:

- The code is usually from multiple contributors, and, unless enforced by the project from the start to agree to a common license, every contribution can potentially be licensed differently. Each of these licenses is binding.
- Attribution is usually required even by the most permissive licenses even for the binary. So optimization that takes out comments might infringe because some comments might be the attribution.

- Accidentally bundling proprietary third-party code with copyleft code can open one to substantial liability because that code can't be shared as required by copyleft.

4.10 Employment Agreements

When you get a programming job, you usually sign a contract with terms of employment, such as duties and compensation, and a **nondisclosure agreement (NDA)**. The contract ensures that you don't reveal any proprietary information even after employment termination and don't claim anything produced for the company as your own. Usually, any such intellectual property is work for hire, but there are exceptions. In particular, you may not be allowed to own any work not produced for the company but related to its business, regardless of whether you produce it during the work hours or use the company's resources.

If you are working during your off hours on a project covered by the agreement, get a written permission from the legal department before starting. Also, periodically show them what you have so far, and get a written statement that they don't have claims to it. Many companies have established workflows for this.

A less common part of an NDA is a **noncompete agreement**, which forbids former employees to work for a competitor or a specific industry for some time. In most cases it's at best enforceable only if reasonable, and often not at all—the answer depends on the state and the specific restrictions. E.g., there should be salary continuation for the noncompete period, and it shouldn't be several years.

A variation is **nonsolicitation agreement**, which disallowed soliciting customers and other employees to go to a former employee's new company. The customer part is usually enforceable, but the employees part usually isn't, and both time out in about a year though this is a gray area. In the latter case the law isn't clear, so it's safer to comply with the terms.

4.11 Privacy

People have a right to privacy, in particular, electronic privacy. **Libel/defamation** laws prevent disclosure of information, even if it's true, if it's both:

- Highly offensive to a reasonable person
- Not of a legitimate concern to the public

Otherwise truth is a complete defense to defamation. A common real-life scenario with defamation potential is a product review. Many are based on opinions—"never do business with them" can be interpreted as libel. It's best to not generalize, i.e., "I will never do business with them again". Beware of giving someone a reference—many companies specifically only verify employment dates and titles and forbid employees from giving personal references due to potential libel exposure.

There is also a lot of data control legislation coming from special municipalities such as European Union and California who give their residents the right to delete their data such as account or search history, and any entity doing business there must comply.

You get privacy when you reasonably expect it. If you catch a coworker watching adult material, and he gets fired, he can't complain. But searching their bag is illegal.

A program may not spy on your actions unless you allowed it. Software must respect users' tracking and data collection preferences. It's illegal to collect data of children of age 13 and younger without specific permission.

4.12 Computer Crime

An e-mail solicitor must allow unsubscribing, and a registration form must allow opting out of receiving advertising.

Unauthorized use of a system and identity theft are illegal. This includes using someone else's password to get in. When assessing unauthorized use, intent and caused damage matter.

4.13 Serving as an Expert Witness

Parties in court are allowed to bring experts to testify opinions (and not facts, like regular witnesses) about the relevant issues. Due to high stakes many issues arise.

Expert witnesses have a duty to the court and are paid by the retained party for their time (in some other countries they are hired by the court directly). But few scientific issues are in universal agreement, and the opposing parties have no difficulty finding witnesses who genuinely believe in the wanted point of view. E.g., I once heard from a doctor who claimed that interpreting MRI imaging isn't based on proof. Inci-

dentially, the lawyers prevented anyone with medical background from serving on that jury. A mixed strategy is also possible—both parties employed a medical expert to testify about the needed treatments and a financial planner to calculate and add up the costs, and of course ended up with different numbers.

The judge makes a pretrial determination of who is allowed to testify as an expert witness and what they can say. The basic rule is that an expert must be reasonably qualified and use a **reasonably accepted methodology** to reach their conclusions. The methodology ranges from included in a peer-reviewed publication to merely useful to the jury, and is somewhat of a gray area. While demanding the former may be too much of a burden on a party, there have been many cases where a particular analysis of an expert has been deemed invalid by the scientific community years later. Both parties try to prevent each other's expert testimonies from being admitted.

A potential expert witness must be prepared for a thorough cross-examination by the other party. Their main goal is getting an expert to admit certain things favorable to them and possibly trip up or lose confidence. It's like an oral exam except that the other party is only asking the hard questions to induce mistakes or beyond-usual hesitation.

4.14 Projects

- Research intellectual property projection of fonts. Distinguish between a font and a typeface (font when printed). The latter isn't copyrightable in the US. But for electronic publication use fonts and not typefaces. Would using a raster image of a typeface bypass this? See <http://www.oddmoxie.com/blog/2016/2/8/everything-you-want-to-know-about-using-fonts-legally-but-didnt-know-to-ask> and https://en.wikipedia.org/wiki/Intellectual_property_protection_of_typefaces as useful resources.

4.15 References

- Kadane, J. B. (2008). *Statistics in the Law: A Practitioner's Guide, Cases, and Materials*. Oxford University Press.
- Landy, G. K. (2008). *The IT/Digital Legal Companion: A Comprehensive Business Guide to Software, Internet, and IP Law*. Syngress.
- Mallor, J. P., Barnes, A. J., Bowers, T., & Langvardt, A. W. (2006). *Business Law: The Ethical, Global, and E-commerce Environment*. McGraw.
- McJohn, S. M., & Graham, L. (2016). *Fundamentals of Intellectual Property Law*. American Bar Association.
- Stim, R. (202*). *Patent, Copyright & Trademark: an Intellectual Property Desk Reference*. Nolo. (A new edition comes out every year).

5 Fundamental Data Structures

"How I want a drink, alcoholic of course, after the heavy lectures involving quantum mechanics" – James Jeans (the word lengths give the first 15 digits of π)

5.1 Introduction

Arrays are intuitive and universal (even a Turing machine is a big array), but to a programmer familiar with only arrays, it's a revelation to discover other data structures. Most of the material in this chapter is discussed in a data structures class. So I expect the reader to have seen the material on dynamic arrays, linked lists, stacks, queues, etc. The emphasis here is on their good implementation in C++.

5.2 Utility Functions

C++ unfortunately doesn't provide some basic frequently used functionality, so I developed my own. E.g., want a macro to print a variable's name and value. My implementation in `<Debug.h>` illustrates good file design (though here `DEBUG` is a macro not affected by namespaces).

```
#ifndef IGMDK_DEBUG_H
#define IGMDK_DEBUG_H
#include <iostream>
using namespace std;
namespace igmdk{
//print the expression, a space, and a new line
#define DEBUG(var) cout << #var " " << (var) << endl;
} //end namespace
#endif
```

In particular, for efficient **large scale code management**:

- The code is organized into logical functionality packages—one per file. A package can have several classes and functions. Typically a data structure is a package with a single class. But algorithms and their helper functions for a common task are also a single package, though this is often debatable. When in doubt, my preference is for fewer but larger packages.
- Headers have **include guards**, include everything they need, don't define variables, and contain only reusable code.
- All functionality ∈ namespace—essential not only for organization but also for correctness. This is because the **at-least-one-definition rule** of C++ allows any matching function to be selected by the linker from separately-compiled units with unpredictable results. E.g., many files can have the output stream operator for string of vectors defined, though it's generally a bad idea to define operators for types not owned.
- Neither `using` a namespace nor implementing functionality in a header is recommended, but both are convenient, and the latter is compatible with templates. With .cpp files include the header as the first include in the file to ensure self-containment. It's also a good style to include more local files first.

Some other useful functionality:

- Ceiling of integer division

```
long long ceiling(unsigned long long n, long long divisor)
{return n/divisor + bool(n % divisor);}
```

- Convenience wrappers around placement `new` and `delete`. Remember that these don't call constructors and destructors and must be used in pairs.

```
template<typename ITEM> ITEM* rawMemory(int n)
{return (ITEM*)::operator new(sizeof(ITEM) * n);}
void rawDelete(void* array){::operator delete(array);}
template<typename ITEM> void rawDestruct(ITEM* array, int size)
{
    for(int i = 0; i < size; ++i) array[i].~ITEM();
    rawDelete(array);
}
```

- A generic assignment operator—destroys the target and copies the value into it using placement

new—but despite popular use this is technically undefined behavior because a destructed object isn't supposed to be used according to the standard

```
template<typename TYPE> TYPE& genericAssign(TYPE& to, TYPE const& rhs)
{//first do a self check
    if(&to != &rhs)
    {
        to.~TYPE();
        new(&to) TYPE(rhs);
    }
    return to;
}
```

- A key-value pair—to avoid using generic first and second in place of more meaningful values, particularly for dictionary data structures (discussed in later chapters)

```
template<typename KEY, typename VALUE> struct KVPair
{
    KEY key;
    VALUE value;
    KVPair(KEY const& theKey = KEY{}, VALUE const& theValue = VALUE{}):
        key(theKey), value(theValue) {}
};
```

- An empty structure—for when a template needs a type but don't have anything meaningful

```
struct EMPTY{};
```

Many algorithms, such as those for sorting and searching, do comparisons. The algorithms in libraries such as STL and the equivalent in other languages expect the user to provide appropriate comparators for the wanted task. To use generic algorithms in STL technically need to define only "<". E.g., can implement "==" in terms of it because $x=y \leftrightarrow !(x < y) \&\& !(y < x)$. So define generic operators which are used by the compiler when not defined on a type:

```
template<typename ITEM> bool operator==(ITEM const& lhs, ITEM const& rhs)
    {return lhs <= rhs && lhs >= rhs;}
template<typename ITEM> bool operator!=(ITEM const& lhs, ITEM const& rhs)
    {return !(lhs == rhs);}
```

Most algorithms use more general **comparators** instead of operators for extra generality. E.g., consider finding a minimum in an array of pointers to objects. An operator compares the pointers, whereas a comparator that dereferences items before comparing them can be specified by the caller. The convention of C++ STL is using operator() for "<", and "==" isn't supported. For efficiency my comparators support "==" to avoid calling "<" twice but are otherwise compatible. By default reduce to the operators:

```
template<typename ITEM> struct DefaultComparator
{
    bool operator()(ITEM const& lhs, ITEM const& rhs) const{return lhs < rhs;}
    bool isEqual(ITEM const& lhs, ITEM const& rhs) const{return lhs == rhs;}
};
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
```

Often want to compare in reverse, i.e., " $a < b$ " returns " $b < a$ ".

```
struct ReverseComparator
{
    COMPARATOR c;
    ReverseComparator(COMPARATOR const& theC = COMPARATOR()): c(theC) {}
    bool operator()(ITEM const& lhs, ITEM const& rhs) const{return c(rhs, lhs);}
    bool isEqual(ITEM const& lhs, ITEM const& rhs) const
        {return c.isEqual(rhs, lhs);}
};
```

Many comparators such as pointer and array index ones are special cases of transform-then-compare. So implement them as such with the appropriate transformations.

```
template<typename ITEM, typename TRANSFORM, typename COMPARATOR>
struct TransformComparator
{
    TRANSFORM t;
    COMPARATOR c;
    TransformComparator(TRANSFORM const& theT = TRANSFORM(),
        COMPARATOR const& theC = COMPARATOR()):
            t(theT), c(theC) {}
```

```

COMPARATOR const& theC = COMPARATOR(): t(theT), c(theC) {}
//needed if transform is default-constructable but comparator isn't
TransformComparator(COMPARATOR const& theC): c(theC) {}
bool operator()(ITEM const& lhs, ITEM const& rhs) const
{
    return c(t(lhs), t(rhs));
}
bool isEqual(ITEM const& lhs, ITEM const& rhs) const
{
    return c.isEqual(t(lhs), t(rhs));
}
};

template<typename ITEM> struct PointerTransform
{
    ITEM const& operator()(ITEM const* item) const{assert(item); return *item;}
};

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>>
using PointerComparator =
    TransformComparator<ITEM const*, PointerTransform<ITEM>, COMPARATOR>;
template<typename ITEM> struct IndexTransform
{
    ITEM const* array;
    IndexTransform(ITEM* theArray): array(theArray) {}
    ITEM const& operator()(int i) const{return array[i];}
};

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>>
using IndexComparator =
    TransformComparator<int, IndexTransform<ITEM>, COMPARATOR>;

```

Use the same approach to compare pairs by the first value:

```

template<typename FIRST, typename SECOND> struct PairFirstTransform
{
    FIRST const& operator()(pair<FIRST, SECOND> const& p) const{return p.first;}
};

template<typename FIRST, typename SECOND, typename COMPARATOR =
    DefaultComparator<FIRST>> using PairFirstComparator = TransformComparator<
    pair<FIRST, SECOND>, PairFirstTransform<FIRST, SECOND>, COMPARATOR>;

```

Variable-length arrays are usually compared **lexicographically**, i.e., using the dictionary order (e.g., "cat" < "mouse"). LexicographicComparator works for types supporting `operator[]` and `getSize`. The items past the last are implicitly null, with `null = null`. For arrays of length k , comparisons take $O(k)$ time but less on average, depending on how long is the common prefix (vectors are discussed later in the chapter). For simplicity the items are compared using operators and not a passed-in comparator.

```

template<typename VECTOR> struct LexicographicComparator
{//the first two functors are for the ith element
    bool operator()(VECTOR const& lhs, VECTOR const& rhs, int i) const
    {
        return i < lhs.getSize() ? i < rhs.getSize() && lhs[i] < rhs[i] :
            i < rhs.getSize();
    }
    bool isEqual(VECTOR const& lhs, VECTOR const& rhs, int i) const
    {
        return i < lhs.getSize() ? i < rhs.getSize() && lhs[i] == rhs[i] :
            i >= rhs.getSize();
    }
    bool isEqual(VECTOR const& lhs, VECTOR const& rhs) const
    {
        for(int i = 0; i < min(lhs.getSize(), rhs.getSize()); ++i)
            if(lhs[i] != rhs[i]) return false;
        return lhs.getSize() == rhs.getSize();
    }
    bool operator()(VECTOR const& lhs, VECTOR const& rhs) const
    {
        for(int i = 0; i < min(lhs.getSize(), rhs.getSize()); ++i)
        {
            if(lhs[i] < rhs[i]) return true;
            if(rhs[i] < lhs[i]) return false;
        }
    }
}
```

```

    return lhs.getSize() < rhs.getSize();
}
int getSize(VECTOR const& value) const{return value.getSize();}
};

```

Must take care to implement “`<`” correctly. E.g., a common difficult case is comparing based on some composite priorities—logic of the type “if condition x then compare variable 1 else variable 2” needn’t give a transitive comparison—can have false x between a, b , and $a < b$, false x between b, c , and $b < c$, and true x between a, c , and $a > c$.

A frequently used operation is finding the minimum or the maximum in an array, possibly relative to a comparator or a function that evaluates the items. The most useful ones are implemented below:

```

template<typename ITEM, typename COMPARATOR> int argMin(ITEM* array,
    int size, COMPARATOR const& c)
{//array minimum with an item comparator
    assert(size > 0);
    int best = 0;
    for(int i = 1; i < size; ++i) if(c(array[i], array[best])) best = i;
    return best;
}
template<typename ITEM> int argMin(ITEM* array, int size)
{
    return argMin(array, size, DefaultComparator<ITEM>());
}
template<typename ITEM> int argMax(ITEM* array, int size)
{
    return argMin(array, size, ReverseComparator<ITEM>());
}
template<typename ITEM> ITEM& valMin(ITEM* array, int size)
{
    int index = argMin(array, size);
    assert(index > -1);
    return array[index];
}
template<typename ITEM> ITEM& valMax(ITEM* array, int size)
{
    int index = argMax(array, size);
    assert(index > -1);
    return array[index];
}
template<typename ITEM, typename FUNCTION> int argMinFunc(ITEM* array,
    int size, FUNCTION const& f)
{//array minimum with a transform function
    assert(size > 0);
    int bestIndex = -1;
    double bestScore;
    for(int i = 0; i < size; ++i)
    {
        double score = f(array[i]);
        if(bestIndex == -1 || score < bestScore)
        {
            bestIndex = i;
            bestScore = score;
        }
    }
    return bestIndex;
}
template<typename ITEM, typename FUNCTION> ITEM& valMinFunc(ITEM* array,
    int size, FUNCTION const& f)
{
    int index = argMinFunc(array, size, f);
    assert(index > -1);
    return array[index];
}

```

Another useful functionality is some common arithmetic operators. In many cases it's standard to implement `operator+` in terms of `operator+=`. A general guideline is to make an operator a nonmember or a friend function if possible or convenient, otherwise a member function. With templates friend is often less clumsy than a nonmember. Here create a type that can be inherited from with type `T = the`

inheritor. This allows the latter to automatically reuse the below operators that are implemented in terms of its custom `*=` versions. Remember that template code is created only when used, so this class is safe to inherit from even if don't have all `*=` operators.

```
template<typename T> struct ArithmeticType
{
    friend T operator+(T const& a, T const& b)
    {
        T result(a);
        return result += b;
    }
    friend T operator-(T const& a, T const& b)
    {
        T result(a);
        return result -= b;
    }
    friend T operator*(T const& a, T const& b)
    {
        T result(a);
        return result *= b;
    }
    friend T operator<<(T const& a, int shift)
    {
        T result(a);
        return result <<= shift;
    }
    friend T operator>>(T const& a, int shift)
    {
        T result(a);
        return result >>= shift;
    }
    friend T operator%(T const& a, T const& b)
    {
        T result(a);
        return result %= b;
    }
    friend T operator/(T const& a, T const& b)
    {
        T result(a);
        return result /= b;
    }
};
```

To estimate π can compute `4 * atan(1)`. But a constant with more precision than needed is good enough:

```
double PI() {return 3.1415926535897932384626433832795;}
```

5.3 Vector

An array of dynamic size is the simplest and the most useful data structure—even in cases where it's inefficient but not the bottleneck. It models a collection such as a shopping list, but without any extra properties that may be useful for efficient operations. E.g., before I knew about other data structures, I implemented a very good maze game using an array of dimension 7 (!) as the only data structure. Even in the applicability of more complex data structures that aren't available from a library and need to be implemented from scratch and properly tested, starting with the less efficient vector is rarely a mistake.

The initial and min sizes are small powers of two to cut the number of memory manager calls and not use excessive memory. In a dynamically allocated array of size `capacity`, the first `size` items are constructed:

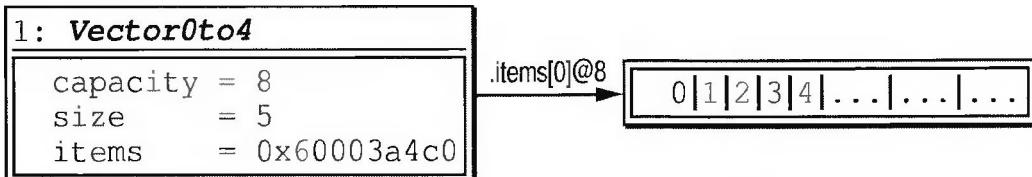


Figure 5.1: Memory layout of a vector containing integer items 0–4. This and other data structure debug figures are created with `ddd`, which has its own syntax for labeling pointer links.

The basic code declares the data members and allows their access. Remember that in C++ `const` function are for reading, and the rest for writing, and the avoided move constructor and assignment operator are deleted when not defined.

```

template<typename ITEM> class Vector: public ArithmeticType<Vector<ITEM>>
{
    enum{MIN_CAPACITY = 8};
    int capacity, size;
    ITEM* items;
public:
    ITEM* getArray() {return items;}
    ITEM* const getArray() const {return items;}
    int getSize() const {return size;}
    ITEM& operator[](int i)
    {
        assert(i >= 0 && i < size);
        return items[i];
    }
    ITEM const& operator[](int i) const
    {
        assert(i >= 0 && i < size);
        return items[i];
    }
};

```

The constructors create a vector of some size from a given item or don't need the item's default constructor. C++ STL additionally uses `reserve()` to avoid unnecessary reallocations, but this is clumsy, needs one reallocation, and in my experience used rarely.

```

explicit Vector(): capacity(MIN_CAPACITY), size(0),
    items(rawMemory<ITEM>(capacity)) {} // no default ITEM constructor needed
explicit Vector(int initialSize, ITEM const& value = ITEM()): size(0),
    capacity(max(initialSize, int(MIN_CAPACITY))),
    items(rawMemory<ITEM>(capacity))
    {for(int i = 0; i < initialSize; ++i) append(value);}
Vector(Vector const& rhs): capacity(max(rhs.size, int(MIN_CAPACITY))),
    size(rhs.size), items(rawMemory<ITEM>(capacity))
    {for(int i = 0; i < size; ++i) new(&items[i]) ITEM(rhs.items[i]);}
Vector& operator=(Vector const& rhs){return genericAssign(*this, rhs);}
~Vector(){rawDestruct(items, size);}

```

The main modification operation is **appending** items to the end. A vector uses **array doubling** (see the “Miscellaneous Algorithms and Techniques” chapter for more details) to create space for extra items when needed:

1. Allocate a new array of capacity = $2 \times$ the size
2. Copy all the items into it
3. Deallocate the old array

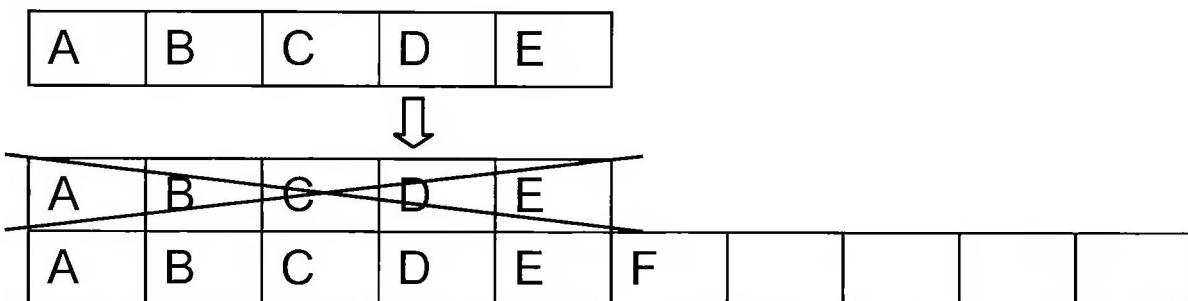


Figure 5.2: Memory layout change during doubling while appending "F"

This makes $O(\lg(n))$ memory manager calls for n appends. Appending is worst-case $O(n)$ and amortized $O(1)$ because $n/2$ items must have been inserted since the last resizing.

```
void resize()
{
    //allocate
    ITEM* oldItems = items;
    capacity = max(2 * size, int(MIN_CAPACITY));
    items = rawMemory<ITEM>(capacity);
    for(int i = 0; i < size; ++i) new(&items[i]) ITEM(oldItems[i]); //copy
    rawDestruct(oldItems, size); //deallocate
}
void append(ITEM const& item)
{
    if(size >= capacity) resize();
    new(&items[size++]) ITEM(item);
}
```

Deletion of the last item resizes if $\text{size} < \frac{1}{4} \times \text{capacity}$ to $2 \times \text{size}$:

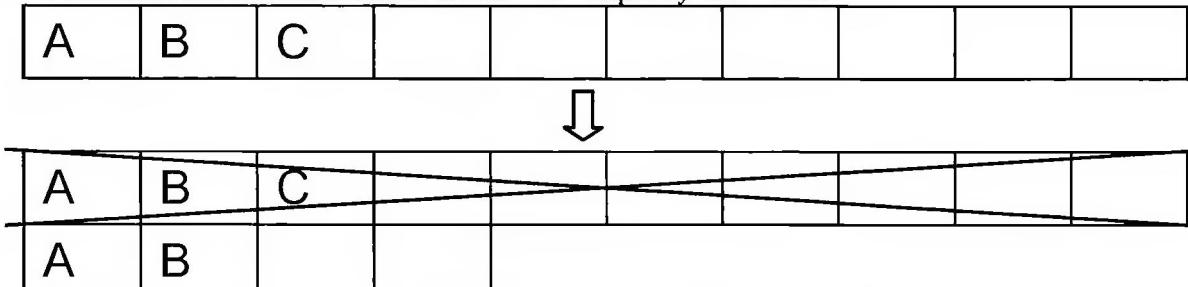


Figure 5.3: Memory layout changes during shrinking while deleting "C"

Using $\frac{1}{4}$ or other values $< \frac{1}{2}$, insertion and deletion are amortized $O(1)$.

```
void removeLast()
{
    assert(size > 0);
    items[--size].~ITEM();
    if(capacity > MIN_CAPACITY && size * 4 < capacity) resize();
}
```

The advantages of vector over other dynamic array data structures:

- Fastest possible random access
- Cache-efficient iteration
- Can pass as a contiguous array to a C API

The disadvantages:

- $O(n)$ insertion (not part of the API), most felt by expensive-to-copy items
- After doubling $\frac{1}{2}$ of the new array is unused
- Can fail to allocate a very large array due to OS restrictions
- Reallocation invalidates item references

The STL vector doesn't shrink on deletion. To do so, swap it with a temporary empty vector, and assign the temporary to the vector. To pass to a C API, use `&vector[0]`. This is wrong when $\text{size} = 0$ (technically not because STL doesn't check bounds), but my `getArray()` is always correct.

Allow $O(1)$ swap:

```
void swapWith(Vector& other)
{
    swap(items, other.items);
```

```

    swap(size, other.size);
    swap(capacity, other.capacity);
}

```

For convenience support working with the last item, reversing, arithmetic operations (note the absence of “+” and “-” which use the generic operators presented before), the 2-norm, and appending vectors:

```

ITEM const& lastItem() const{return items[size - 1];}
ITEM& lastItem(){return items[size - 1];}
void reverse(int left, int right)
{
    while(left < right) swap(items[left++], items[right--]);
}
void reverse(){reverse(0, size - 1);}
void appendVector(Vector const& rhs)
{
    for(int i = 0; i < rhs.getSize(); ++i) append(rhs[i]);
}
Vector& operator+=(Vector const& rhs)
{
    assert(size == rhs.size);
    for(int i = 0; i < size; ++i) items[i] += rhs.items[i];
    return *this;
}
Vector& operator-=(Vector const& rhs)
{
    assert(size == rhs.size);
    for(int i = 0; i < size; ++i) items[i] -= rhs.items[i];
    return *this;
}
template<typename SCALAR> Vector& operator*=(SCALAR const& scalar)
{
    for(int i = 0; i < size; ++i) items[i] *= scalar;
    return *this;
}
friend Vector operator*(Vector const& a, ITEM const& scalar)
{
    Vector result(a);
    result *= scalar;
}
friend ITEM dotProduct(Vector const& a, Vector const& b)
{
    assert(a.size == b.size);
    ITEM result(0);
    for(int i = 0; i < a.size; ++i) result += a[i] * b[i];
    return result;
}
Vector operator-() const{return *this * -1;}
bool operator==(Vector const& rhs) const
{
    if(size == rhs.size)
    {
        for(int i = 0; i < size; ++i)
            if(items[i] != rhs[i]) return false;
        return true;
    }
    return false;
}
double norm(Vector<double> const& x){return sqrt(dotProduct(x, x));}

```

The 2-norm isn't implemented in the most stable way (see the “Numerical Algorithms—Introduction and Matrix Algebra” chapter), but this is good enough.

An interesting interview question is to implement C++ STL `remove_if`, that given an array and a filter criteria moves matching elements to the end of the array. It returns the new end pointer and preserves the order of the kept elements. This is best asked for numbers with a “keep_if” filter. E.g., given [1, 2, 3, 4, 5] and $f(x)=x \% 2$, the result is [1, 3, 5, |4, 2|]. The solution is to keep a rolling group of “no” items, which is updated by a left-to-right loop that puts the first group item after a new “yes” item. E.g., [|1, 2, 3, 4, 5] \rightarrow [1, |2|, 3, 4, 5] \rightarrow [1, 3, |2|, 4, 5] \rightarrow [1, 3, |2, 4|, 5] \rightarrow [1, 3, 5, |4, 2|].

5.4 Block Array

Use many arrays of fixed size k , indexed by a vector of pointers:

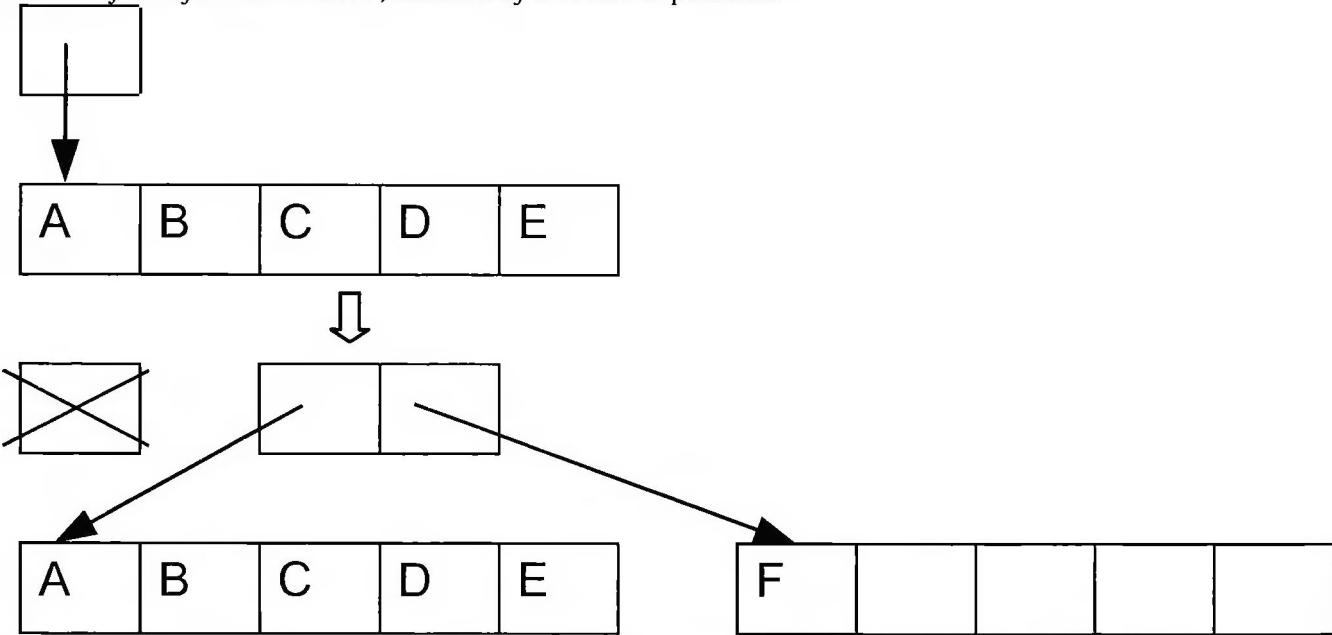


Figure 5.4: Inserting into a block array resizes the pointer vector only when needed

Appending is $O(n/k)$ and amortized $O(1)$ because doubling happens after inserting nk items and copies n/k pointers. The i^{th} item = $\text{vector}[i/k][i \% k]$, and k is maybe 64 or another not too large/small value. The advantages:

- The wasted space after a resizing is $O(n/k + k)$
- Don't need very large arrays
- Items aren't copied
- Iteration is reasonably cache-efficient
- Item references are valid after reallocation

The disadvantages:

- Slower random access
- Can't pass to a C API
- Insertion and deletion are $O(n/k)$

Prefer a vector due to faster random access and the ability to pass to a C API. Would consider a block array for a compressed representation where arrays containing 0's aren't allocated or when need a very large vector. But such use cases are specific and rare, so no implementation is provided.

More complicated methods using blocks of variable sizes have the optimal $O(1)$ random access, append, and remove-last and $O(\sqrt{n})$ wasted space. Experimentally, among several dynamic array structures, vector performs best and block array second best, and block array is best when vector approaches the memory limit (Joannou & Raman 2011).

5.5 Linked List

To represent a sequence of items, instead of an array can use a sequence of items linked by pointers:



Figure 5.5: A simple singly-linked list

Such a list can be **doubly-linked**, with bidirectional pointers, or **circular**, with the last item pointing to the first. It supports $O(1)$ node movement, but:

- No random access unless store node pointers externally—to access the i^{th} item need $O(i)$ time
- Waste space on navigation pointers and bookkeeping of the node memory allocator
- Iteration isn't cache-inefficient because of pointer jumping

With few items, when don't need random access, a list may be better than a vector. Such implementations for efficiency usually use own linked lists and don't reuse a generic one—e.g., see the chaining hash table implementation (in the “Hashing” chapter). An interesting technique is to implement data structures consisting of nodes linked by pointers on top of an array where each array item is a node. This avoids

dynamic memory allocation but removes most advantages of a list, so it's only useful for special cases.

The below doubly-linked list implementation allows appending, **prepend**ing (adding to front), removing, and moving nodes. It also allows STL-like bidirectional iteration and is encapsulated to use iterators for almost all operations. Such operations are useful for some other data structures. The main difficulty is relinking pointers correctly for the affected node, the previous node, and the next node. The constructor is templated and takes a single argument that is either `ITEM` or something that allows to implicitly construct one in-place. This is a useful general idiom. It also uses C++ new/delete and not the garbage-collecting free list (discussed later).

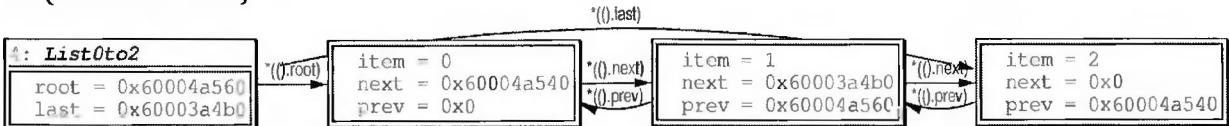


Figure 5.6: Memory layout of a doubly-linked list with integer items 0-2

```

template<typename ITEM> class SimpleDoublyLinkedList
{
    struct Node
    {//note the templated constructor
        ITEM item;
        Node *next, *prev;
        template<typename ARGUMENT>
        Node(ARGUMENT const& a): item(a), next(0), prev(0) {}
    } *root, *last;
    void cut(Node* n)//unlink a node from the list
    {//join prev and next
        assert(n);
        (n == last ? last : n->next->prev) = n->prev;
        (n == root ? root : n->prev->next) = n->next;
    }
public:
    SimpleDoublyLinkedList(): root(0), last(0){}
    template<typename ARGUMENT> void append(ARGUMENT const& a)
    {//note the templated constructor
        Node* n = new Node(a);
        n->prev = last;
        if(last) last->next = n;
        last = n;
        if(!root) root = n;
    }
    class Iterator
    {
        Node* current;
    public:
        Iterator(Node* n): current(n){}
        typedef Node* Handle;
        Handle getHandle(){return current;
        Iterator& operator++()
        {//to next item
            assert(current);
            current = current->next;
            return *this;
        };
        Iterator& operator--()
        {//to prev item
            assert(current);
            current = current->prev;
            return *this;
        };
        ITEM& operator*() const{assert(current); return current->item;}
        ITEM* operator->() const{assert(current); return &current->item;}
        bool operator==(Iterator const& rhs) const
            {return current == rhs.current;}
    };
}

```

```

Iterator begin(){return Iterator(root);}
Iterator rBegin(){return Iterator(last);}
Iterator end(){return Iterator(0);}
Iterator rEnd(){return Iterator(0);}
void moveBefore(Iterator what, Iterator where)
{//where is allowed to be 0 which means move to end
    assert(what != end());
    if(what != where) //first check for self-reference
    {
        Node *n = what.getHandle(), *w = where.getHandle();
        cut(n);
        n->next = w;
        if(w)
        {
            n->prev = w->prev;
            w->prev = n;
        }
        else
        {
            n->prev = last;
            last = n;
        }
        if(n->prev) n->prev->next = n;
        if(w == root) root = n;
    }
}
template<typename ARGUMENT> void prepend(ARGUMENT const& a)
//append and move to front
    append(a);
    moveBefore(rBegin(), begin());
}
void remove(Iterator what)
//unlink and deallocate
    assert(what != end());
    cut(what.getHandle());
    delete what.getHandle();
}
SimpleDoublyLinkedList(SimpleDoublyLinkedList const& rhs)
    for(Node* n = rhs.root; n; n = n->next) {append(n->item);}
SimpleDoublyLinkedList& operator=(SimpleDoublyLinkedList const&rhs)
    return genericAssign(*this, rhs);
~SimpleDoublyLinkedList()
{
    while(root)
    {
        Node* toBeDeleted = root;
        root = root->next;
        delete toBeDeleted;
    }
}
};

```

Linked lists aren't used often in day-to-day programming due to preference for vectors. But they tend to show up in interview questions. A common one: two improper lists eventually merge at some node and continue as a single list. Given the head nodes, find the merge node. A simple $O(n^2)$ solution is checking every node for being the merge point. Putting the nodes of one list in a hash table, and checking it against the other is expected $O(n)$ but uses extra memory. The clever solution is to go through both lists to find their length, then through the longer one by the length difference, and finally step by one in each list until merge.

5.6 Garbage-collecting Free List

A **free list** is a collection of blocks of memory, allocated once and cut into smaller objects. This avoids allo-

cating many small objects but doesn't win much against modern memory managers. A free list can also collect garbage for a data structure, which is more useful. Though the deallocated memory isn't available to other parts of the program, and can't merge or swap data structures faster than by making a complete copy, these are rarely used.

A free list gives back `ITEM*` on allocation and takes it on deallocation. `ITEM*` is a cast `Item*`, where `Item` is a structure containing a raw-memory `ITEM` as the first member and some bookkeeping (remember that in C++ the address of a struct = that of its first member). A building block is a single-block **static free list** without garbage collection. It consists of an array of k `Items`. Have pointers to the next `Item` in the block (index `maxSize`) and the head of the returned list. The latter is defined implicitly by the returned pointers. This is an example of two lists stored in a single array.

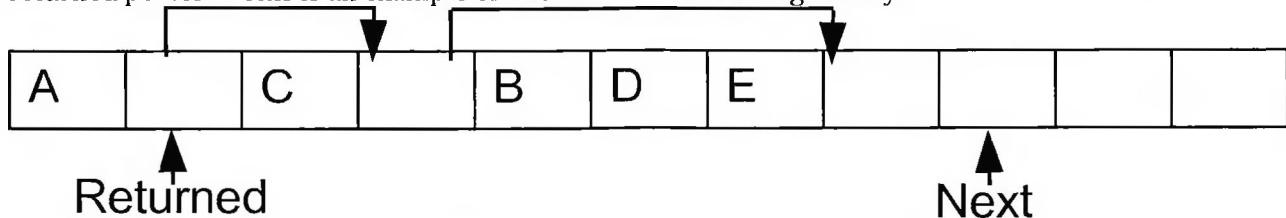


Figure 5.7: Structure of a static free list

```
template<typename ITEM> struct StaticFreelist
//size is current size and maxSize is largest allocation
    int capacity, size, maxSize;//always size <= maxSize <= capacity
    struct Item
    {
        ITEM item;
        union
        {
            Item* next;/used when empty cell
            void* userData;/used when allocated
        };
    } *nodes, *returned;
    StaticFreelist(int fixedSize): capacity(fixedSize), size(0), maxSize(0),
        returned(0), nodes(rawMemory<Item>(fixedSize)) {}
    bool isFull(){return size == capacity;}
    bool isEmpty(){return size <= 0;}
};
```

Allocation:

1. If the returned list isn't empty, use its first node
2. Else if the block is full, the caller must create another
3. Use the next item

```
Item* allocate()
{//must handle full blocks externally - check using pointer arithmetic
    assert(!isFull());
    Item* result = returned;
    if(result) returned = returned->next;
    else result = &nodes[maxSize++];
    ++size;
    return result;
}
```

The cost is $O(\text{constructor call} + \text{memory manager call})$. The latter is amortized due by the block size.
Deallocation:

1. Destruct the item
2. Mark it destructed
3. Prepend the cell to the returned list—this does (2) actually

```
void remove(Item* item)
{//nodes must come from this list
    assert(item - nodes >= 0 && item - nodes < maxSize);
    item->item.~ITEM();
    item->next = returned;
    returned = item;
    --size;
```

}

The space use is $O(\text{the maximum number of items allocated at any time})$. The destructor collects garbage by destructing any unreturned items, which takes $O(1)$ if all items are returned, $O(\maxSize)$ otherwise.

```

~StaticFreelist()
{
    if (!isEmpty())
        //mark allocated nodes, unmark returned ones, destruct marked ones
        Vector<bool> toDestruct(maxSize, true);
        while (returned)
            //go through the return list to unmark
            toDestruct[returned - nodes] = false;
            returned = returned->next;
        }
        for (int i = 0; i < maxSize; ++i)
            if (toDestruct[i]) nodes[i].item.~ITEM();
    }
    rawDelete(nodes);
}

```

For a general free list need a doubly-linked list of static free lists and items to know from which static list they came. Each static list forms a block. This is similar to a block array. The list is partitioned so that full blocks are in the back and nonfull ones in the front.

Root

Last

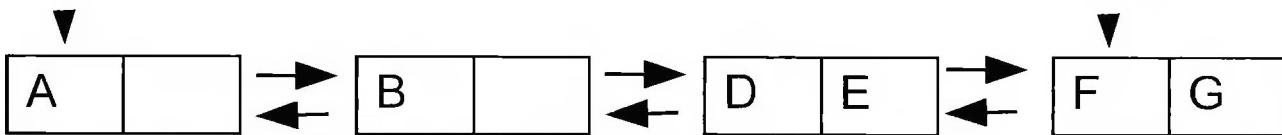


Figure 5.8: Structure of a dynamic free list

The blocks are in sizes increasing by a factor of 2, from 32 to 8192, to be efficient and useful for small sizes. The unlikely worst-case space use $O(\min(\max \text{ number of items allocated at any time} \times (\text{sizeof(ITEM)} + \text{sizeof(pointer)}), \text{the number of allocated items} \times \text{the size of a block}))$ occurs when each block has a single item. For garbage collection the total cost is $O(\text{the number of live items} \times \text{the destructor cost} + \text{the total size of the remaining blocks})$.

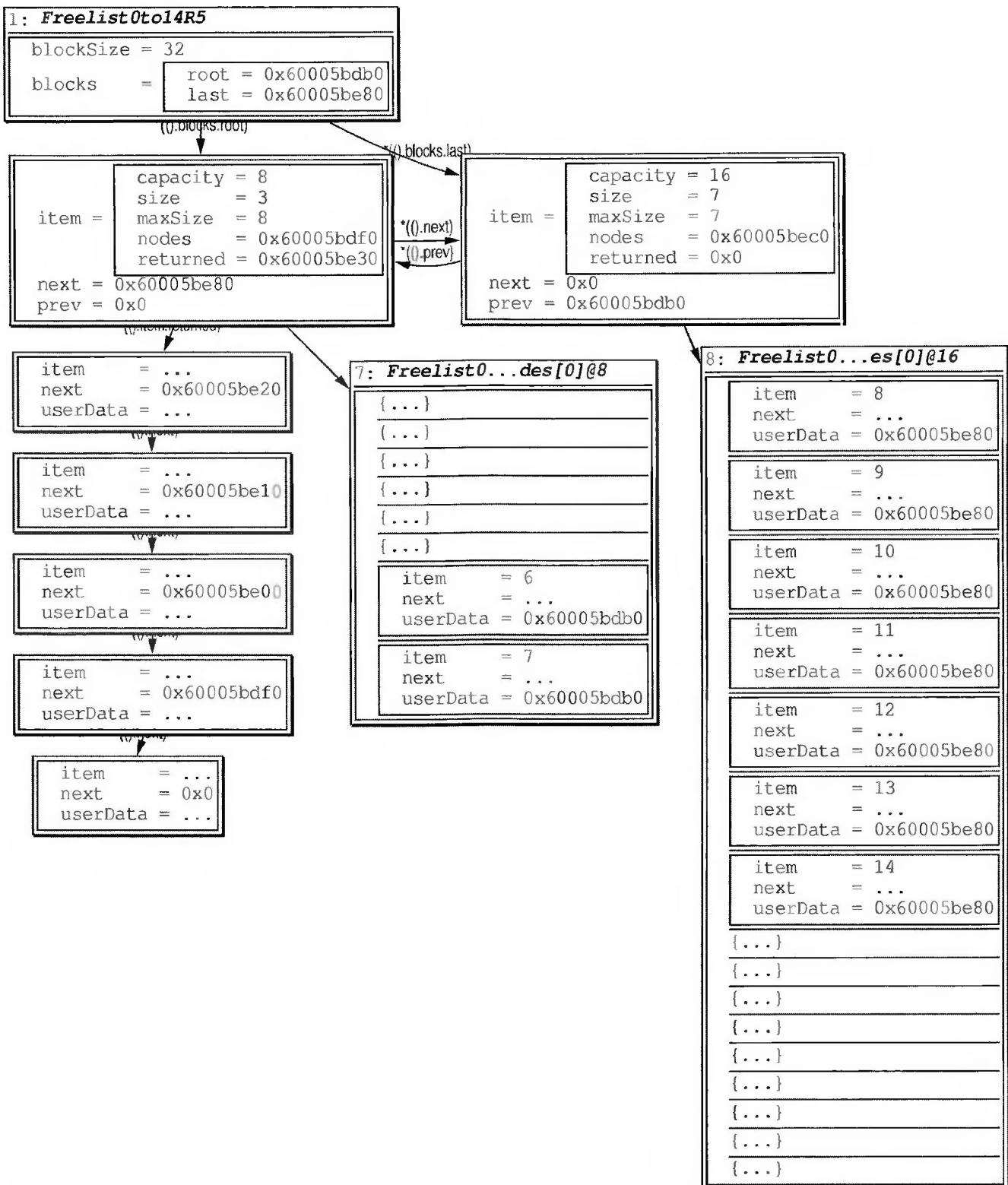


Figure 5.9: Memory layout of a free list with integer items 0–14 allocated and 0–5 deallocated

```

template<typename ITEM> class Freelist
{
    enum{MAX_BLOCK_SIZE = 8192, MIN_BLOCK_SIZE = 8, DEFAULT_SIZE = 32};
    int blockSize;
    typedef SimpleDoublyLinkedList<StaticFreelist<ITEM>> ListType;
    typedef typename StaticFreelist<ITEM>::Item Item;
    typedef typename ListType::Iterator I;
    ListType blocks;
    //disallow copying
    Freelist(Freelist const&);
    Freelist& operator=(Freelist const&);
public:
  
```

```

Freelist(int initialSize = DEFAULT_SIZE): blockSize(max<int>(
    MIN_BLOCK_SIZE, min<int>(initialSize, MAX_BLOCK_SIZE))) {}
};

```

Allocation:

1. Use the first block, creating one if needed
2. Take space for the item from there
3. If the block becomes full, move it to the back

```

ITEM* allocate()
{
    I first = blocks.begin();
    if(first == blocks.end() || first->isFull())
        //make new first block if needed
        blocks.prepend(blockSize);
        first = blocks.begin();
        blockSize = min<int>(blockSize * 2, MAX_BLOCK_SIZE);
    //allocate item
    Item* result = first->allocate();
    result->userData = (void*)first.getHandle(); //block list pointer
    //move full blocks to the end
    if(first->isFull()) blocks.moveBefore(first, blocks.end());
    return (ITEM*)result; //cast works because of first member rule
}

```

Deallocation:

1. Return the item to the block from which it came
2. If the block became empty, deallocate it
3. Else move it to the front

```

void remove(ITEM* item)
//undefined behavior if item not from this list
if(!item) return; //handle null pointer
Item* node = (Item*)(item); //cast back from first member
I cameFrom(typename I::Handle) node->userData;
cameFrom->remove(node);
if(cameFrom->isEmpty())
    //delete block if empty, else reduce its size
    //beware that block boundary delete/remove thrashes, but unlikely
    blockSize = max<int>(MIN_BLOCK_SIZE,
        blockSize - cameFrom->capacity);
    blocks.remove(cameFrom);
} //move available blocks to the front
else blocks.moveBefore(cameFrom, blocks.begin());
}

```

5.7 Stack

Implement a **first-in-last-out** sequence where can only **push** (add) items to the **top** and access or **pop** (remove) the top item.

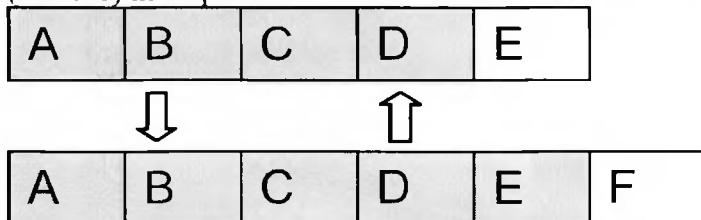


Figure 5.10: Stack push and pop

To implement, use a sequence where work with the last item. So can use one of a:

- Fixed-sized array with a top counter, if the maximum size is known and small, and the item has a default constructor—fast, simple, and avoids dynamic memory allocation
- Vector—fast, takes little code, and results in few memory allocations
- Block array—same code as for vector and efficient for very large sizes or items, but a repetitive push/pop sequence on the block boundary causes repetitive allocation/deallocation

- Linked list—not competitive because allocating single nodes is inefficient
- Vector and block array lead to amortized $O(1)$ push/pop due to resizing and also support random access, which is useful in some cases (but not implemented here). The former is chosen here:

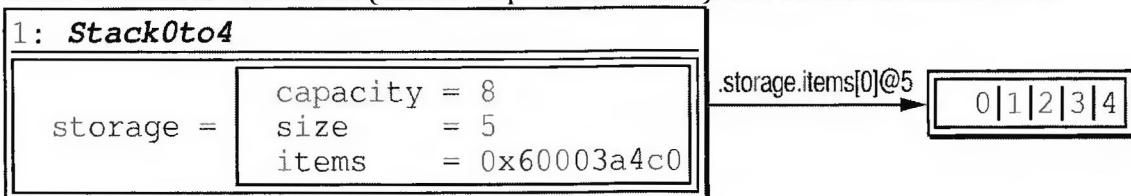


Figure 5.11: Memory layout of a vector-based stack containing integer items 0–4

```

template<typename ITEM, typename VECTOR = Vector<ITEM>> struct Stack
{
    VECTOR storage;
    void push(ITEM const& item) {storage.append(item);}
    ITEM pop()
    {
        assert(!isEmpty());
        ITEM result = storage.lastItem();
        storage.removeLast();
        return result;
    }
    ITEM& getTop()
    {
        assert(!isEmpty());
        return storage.lastItem();
    }
    bool isEmpty() {return !storage.getSize();}
};
  
```

It's unlikely that stack operations are the bottleneck, and developers should treat it as an abstract concept with any library implementation as good enough. Stack logic is usually presented with examples for parsing simple expressions such as for calculator, but in the code you are likely to write yourself it's essentially only useful for recursion removal.

5.8 Queue

A queue implements a **first-in-first-out** sequence allowing **enqueueing** (adding) items to the back and accessing or **dequeuing** (removing) them from the front:

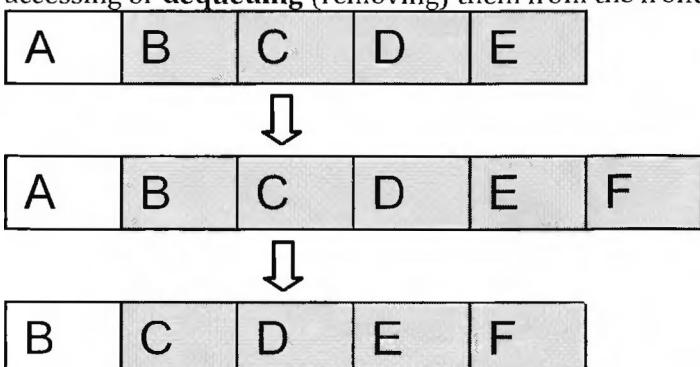


Figure 5.12: Queue enqueue and dequeue

To implement, need a sequence where can work efficiently with the first and the last elements. So can use one of a:

- Linked list
- **Circular queue** with doubling (explained later)—the simplest reasonably efficient choice
- Circular queue over a block array—somewhat clumsy but probably best for sizes near the memory limit
- Linked list of blocks, similar to a block array—more clumsy than a linked list and doesn't seem to have any advantages over the other choices
- Array with extra space at both ends, doubling it when either runs out— $O(1)$ amortized push/pop, but too clumsy

Similar considerations as for stack apply. Can extend to a **double-ended queue**, where allow to

enqueue/dequeue from both ends, but this isn't done here for simplicity.

A circular queue with doubling uses a dynamic array but items start at index *front* to allow efficient operations at the beginning. So $\text{index}(i) = (\text{front} + i) \% \text{capacity}$, and $\text{back} = \text{index}(\text{the size} - 1)$. Push and pop at both ends resize as for vector; their runtime is amortized O(1). Random access is also supported.

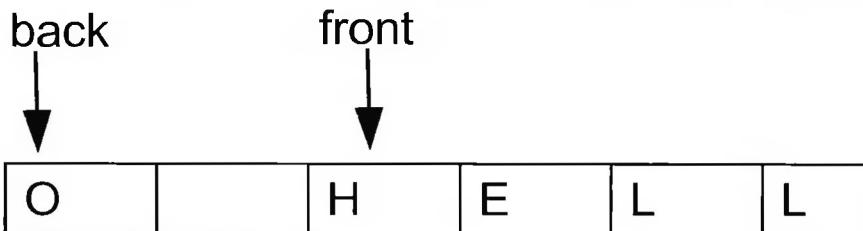


Figure 5.13: Queue pointers

For simplicity the implementation only supports a regular queue.

1: Queue1to4

capacity = 8	.items[0]@8	... 1 2 3 4
front = 1		
size = 4		
items = 0x60003a4c0		

Figure 5.14: Memory layout of a queue with integer items 0-4 inserted and 0 removed

```
template<typename ITEM> class Queue
{
    enum{MIN_CAPACITY = 8}; //same as for vector
    int capacity, front, size;
    ITEM* items; //must be declared after capacity
    int offset(int i) const{return (front + i) % capacity;}
    void resize()
    {
        ITEM* oldArray = items;
        int newCapacity = max(int(MIN_CAPACITY), size * 2);
        items = rawMemory<ITEM>(newCapacity);
        for(int i = 0; i < size; ++i) new(&items[i]) ITEM(oldArray[offset(i)]);
        deleteArray(oldArray);
        front = 0;
        capacity = newCapacity;
    }
    void deleteArray(ITEM* array)
    { //destruct only allocated items
        for(int i = 0; i < size; ++i) array[offset(i)].~ITEM();
        rawDelete(array);
    }
public:
    bool isEmpty() const{return size == 0;}
    int getSize() const{return size;}
    ITEM& operator[](int i)
    {
        assert(i >= 0 && i < size);
        return items[offset(i)];
    }
    ITEM const& operator[](int i) const
    {
        assert(i >= 0 && i < size);
        return items[offset(i)];
    }
    Queue(int theCapacity = MIN_CAPACITY): capacity(max(int(MIN_CAPACITY),
        theCapacity)), front(0), size(0), items(rawMemory<ITEM>(capacity)) {}
    Queue(Queue const& rhs): capacity(max(int(MIN_CAPACITY), rhs.size())),
        size(rhs.size), front(0), items(rawMemory<ITEM>(capacity))
        {for(int i = 0; i < size; ++i) push(rhs[i]);}
    Queue& operator=(Queue const& rhs){return genericAssign(*this, rhs);}
}
```

```

~Queue() {deleteArray(items);}
void push(ITEM const& item)
{
    if(size == capacity) resize();
    new(&items[offset(size++)]) ITEM(item);
}
ITEM pop()
{
    assert(!isEmpty());
    ITEM result = items[front];
    items[front].~ITEM();
    front = offset(1);
    if(capacity > 4 * --size && capacity > MIN_CAPACITY) resize();
    return result;
}
ITEM& top() const
{
    assert(!isEmpty());
    return items[front];
}
};

```

Can simplify a circular queue to a fixed-size circular buffer for special, mostly hardware, applications, but queue operations aren't usually the bottleneck.

A queue is a useful building block of some algorithms, such as sorting on disk (see the “External Memory Algorithms” chapter). But mostly use it when a logical model of something calls for it, such as simulating waiting times in a line. In many cases the more general priority queue (see the “Heaps” chapter) is more appropriate.

An interesting interview question is how to implement a queue API with two stacks. The best solution pushes enqueued items onto stack₁. To dequeue take an item from stack₂ if any, otherwise move over all stack₁ items to stack₂. This takes O(1) amortized time.

5.9 Trees

A **tree** is similar to a linked list, but a node has pointers to all its **children** and can't have circular connections. A node that doesn't have any children is **external/leaf** and **internal** otherwise. A tree with n nodes has $n - 1$ child pointers. A tree is **binary** if each node has at most two children. Binary trees are easy to represent in code due to having a bounded number of children and form many data structures. For an **ordered tree**, the data in a left node < the data in the parent node ≤ the data in the right node.

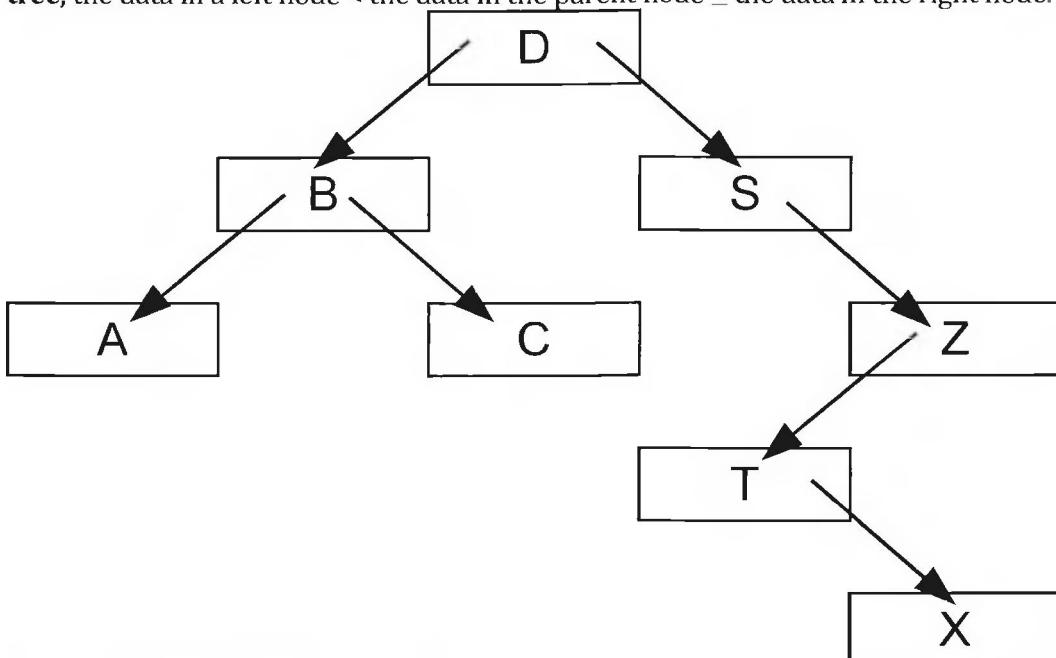


Figure 5.15: An ordered binary tree

A general tree is isomorphic to a binary tree using the mapping (first child, next sibling) ↔ (left child, right child). To represent it directly need a variable-length list of child pointers, which is usually clumsy,

depending on the wanted operations. So in general it's useful when needed by the domain logic.

A node may have a pointer to its parent. Unlike linked lists, trees are useful only with extra logic, and many specific implementations are discussed in later chapters.

5.10 Bit Algorithms

C++ doesn't have instructions for \lg (binary logarithm), **pop count** (the number of set bits in a word) and many other operations doable in $O(1)$ time in assembly. Remember:

- Shifts by < 0 or \geq the number of bits in the word are undefined.
- Signed word right shift fills up with 1 and not 0.
- `numeric_limits<WORD>::digits` is 1 less for signed words.
- `1 = sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`, and `unsigned long long` has ≥ 64 bits. E.g., depending on computer architecture, `int` can have 32 or 64 bits, `long long` 64 or 128, and either pointer or `int` has more bits than the other.
- `<cstdint>` defines `intX_t` and `uintX_t` for $X = 8, 16, 32$, and 64 , but an implementation may omit X for which \exists word type. So in general these types may not be portable, though in practice this is rare, and the below alternative is clumsy.
- `int_leastX_t` and `uint_leastX_t` for $X = 8, 16, 32$, and 64 have at least the specified number of bits. Use them with `0xFF...ull` masks to simulate any unavailable fixed-width types.

When relying on word sizes, use portable fixed-width types. It's more error-prone to use a common word type and check for the wanted number of bits. With any unusual architectures must deal separately in many other ways.

Bit operations allow working efficiently with powers of two for positive integers:

- $[\lg(x)]$ = the position of the highest set bit.
- If x is a power of two, $x - 1$ has all lower bits = 1. For both use **bit-parallelism**, i.e., work on several bits at once with a single word operation.

```
unsigned long long twoPower(int x){return 1ull << x;}
//e.g., for 8 have !(0100 & 0011); careful: returns true for 0
bool isPowerOfTwo(unsigned long long x){return !(x & (x - 1));}
int lgFloor(unsigned long long x)
{//shift until 0 and count; e.g., lgFloor(2) = lgFloor(3) = 1
    assert(x); //log of 0 is undefined
    int result = 0;
    while(x >= 1) ++result;
    return result;
}
```

On top of these can efficiently implement $[\lg(x)]$ and the next power of two of x .

```
//E.g., lgCeiling(3) = 2
int lgCeiling(unsigned long long x){return lgFloor(x) + !isPowerOfTwo(x);}
//E.g., nextPowerOfTwo(7) = nextPowerOfTwo(8) = 8
unsigned long long nextPowerOfTwo(unsigned long long x)
    {return isPowerOfTwo(x) ? x : twoPower(lgFloor(x) + 1);}
```

A **bit mask** is a word such that applying a bit operation to it and x gives the wanted result. To get, set, or flip a single bit at position i , the mask for the i^{th} bit = 2^i .

```
namespace Bits
{//below useful due to type because literals are int
unsigned long long const ZERO = 0, FULL = ~ZERO;
bool get(unsigned long long x, int i){return x & twoPower(i);} //and
bool flip(unsigned long long x, int i){return x ^ twoPower(i);} //xor
template<typename WORD> void set(WORD& x, int i, bool value)
{
    assert(!numeric_limits<WORD>::is_signed);
    if(value) x |= twoPower(i);
    else x &= ~twoPower(i);
}
}//end namespace
```

For manipulating bits in word x :

- `getValue` produces the word formed by the n bits starting from position i of x

- `setValue` sets the n bits of x starting from position i to the first n bits of word `value`
- Can mask bit ranges

The main work for these is computing the masks. The below operations are generic \forall unsigned type, but prefer casting to `unsigned long long` over templates where possible. They are also part of the `Bits` namespace.

```
unsigned long long upperMask(int n){return FULL << n;}//11110000
unsigned long long lowerMask(int n){return ~upperMask(n);}//00001111
unsigned long long middleMask(int i, int n){return lowerMask(n)<<i;}//0011i000
unsigned long long getValue(unsigned long long x, int i, int n)
{
    {return (x >> i) & lowerMask(n);} //shift down and mask
template<typename WORD>
void setValue(WORD& x, unsigned long long value, int i, int n)
{
    assert(!numeric_limits<WORD>::is_signed);
    WORD mask = middleMask(i, n); //note the cast
    x &= ~mask; //erase
    x |= mask & (value << i); //set
}
```

To compute pop count, i.e., the number of set bits in a word, break it up into bytes, and use a table of pre-computed bit counts \forall byte.

```
class PopCount8
{
    char table[256];
public:
    static int popCountBruteForce(unsigned long long x)
    {//for computing the table
        int count = 0;
        while(x)
            {//count rightmost bits one-by-one
                count += x & 1;
                x >>= 1;
            }
        return count;
    }
    PopCount8(){for(int i = 0; i < 256; ++i) table[i] = popCountBruteForce(i);}
    int operator()(unsigned char x) const{return table[x];}
};
int popCountWord(unsigned long long x)//loses no efficiency over template
{//initialization on first call
    static PopCount8 p8;
    int result = 0;
    for(; x; x >>= 8) result += p8(x); //equal performance for every word type
    return result;
}
```

Mixing arithmetic and bit operations enables efficient manipulation of the right bits. E.g., to count rightmost 0's before the first 1 in a word, change the word so that they become 1 and the other bits 0, and count the 1's:

```
int rightmost0Count(unsigned long long x){return popCount(~x & (x - 1));}
```

Using a table is also a fast, simple method to reverse rightmost bits. This is used, e.g., in FFT (see the “Numerical Algorithms—Working with Functions” chapter). Compose the result one byte at a time from the MSD byte first, which is the bit-reversed LSD byte of the original word.

```
class ReverseBits8
{
    unsigned char table[256];
public:
    template<typename WORD> static WORD reverseBitsBruteForce(WORD x)
    {
        assert(!numeric_limits<WORD>::is_signed);
        WORD result = 0;
        for(int i = 0; i < numeric_limits<WORD>::digits; ++i)
        {
```

```

        result = (result << 1) + (x & 1);
        x >>= 1;
    }
    return result;
}
ReverseBits8()
{
    for(int i = 0; i < 256; ++i)
        table[i] = reverseBitsBruteForce<unsigned char>(i);
}
unsigned char operator()(unsigned char x) const{return table[x];}
};

template<typename WORD> WORD reverseBits(WORD x)
{
    assert(!numeric_limits<WORD>::is_signed);
    static ReverseBits8 r8;
    WORD result = 0;
    for(int i = 0; i < sizeof(x); ++i, x >>= 8) result = (result << 8) + r8(x);
    return result;
}

```

Reversing the lower n bits reduces to reversing the whole word with the appropriate shifts. An optimization is to reduce the number of byte shifts by zeroing higher bits—don't even need to mask off the result.

```

template<typename WORD> WORD reverseBits(WORD x, int n)
{
    int shift = sizeof(x) * 8 - n;
    assert(!numeric_limits<WORD>::is_signed && n > 0 && shift >= 0);
    return reverseBits<WORD>(x & Bits::lowerMask(n)) >> shift;
}

```

5.11 Bit Set

A bit set behaves like a vector of Booleans, but each of which `operator[]` returns by value and not by reference because each is represented by a bit in a vector of words.

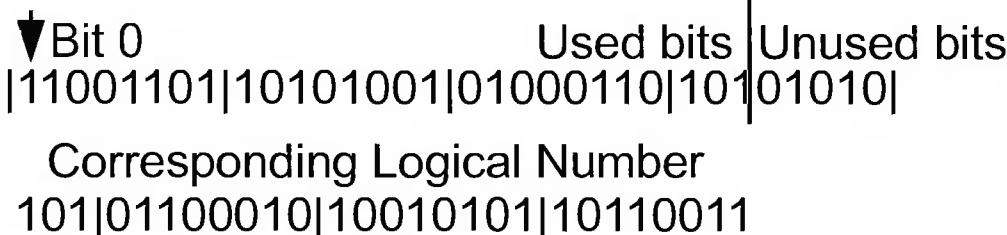


Figure 5.16: Byte layout of bits in a bit set

A bit set supports set operators "&", "|", "^", and "~", which behave as if it were an integer with the least significant bit at 0. For efficiency use word operations and words of the largest possible size. So `unsigned long long` is the default, though on 32-bit machines `unsigned int` may be faster. For portable storage, `unsigned char` avoids endianness incompatibility, so use that when needed.

The extra bits in the last word are garbage, so keep them zeroed out. This has many benefits, e.g., compare for equality by comparing the storage vectors directly. Let B be the number of bits in the word. The logical bit i corresponds to the bit $i \% B$ in the word i / B . So a single-word bit set matches the corresponding word, and the little-endian word order allows easy appending. Get/set find the word and the bit and apply the corresponding bit operations. Due to using vector, the cost of append is amortized $O(1)$.

1: BitsetChar19Every4

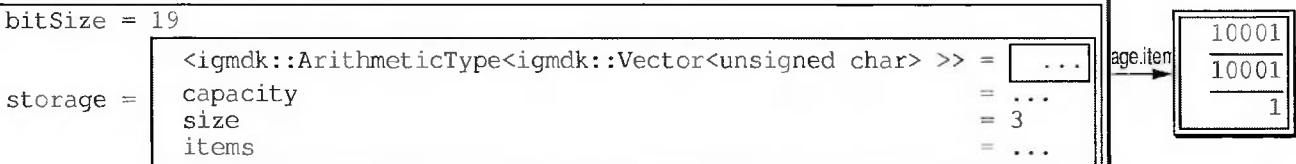


Figure 5.17: Memory layout of a bit set with 19 bits and every 4th bit set. Word printout is a bad way to see the bits due to the jumps.

```
template<typename WORD = unsigned long long> class Bitset
```

```

{
    enum{B = numeric_limits<WORD>::digits};
    unsigned long long bitSize; //must come before storage
    Vector<WORD> storage;
    void zeroOutRemainder()
        {if(bitSize > 0)storage.lastItem() &= Bits::lowerMask(lastWordBits());}
    bool get(int i) const
    {
        assert(i >= 0 && i < bitSize);
        return Bits::get(storage[i/B], i % B);
    }
    unsigned long long wordsNeeded() const{return ceiling(bitSize, B);}
public:
    Bitset(unsigned long long initialSize = 0): bitSize(initialSize),
           storage(wordsNeeded(), 0){}//set all bits to 0
    Bitset(Vector<WORD> const& vector): bitSize(B * vector.getSize()),
          storage(vector) {}//direct construction from storage vector is useful
    int lastWordBits() const
    { //1 to B if > 1 bit
        assert(bitSize > 0);
        int result = bitSize % B;
        if(result == 0) result = B;
        return result;
    }
    int garbageBits() const{return bitSize > 0 ? B - lastWordBits() : 0;}
    Vector<WORD> const& getStorage() const{return storage;}
    unsigned long long getSize() const{return bitSize;}
    unsigned long long wordSize() const{return storage.getSize();}
    bool operator[](int i) const{return get(i);}
    void set(int i, bool value = true)
    {
        assert(i >= 0 && i < bitSize);
        Bits::set(storage[i/B], i % B, value);
    }
    void append(bool value)
    { //increase size if needed, and get last bit
        ++bitSize;
        if(wordSize() < wordsNeeded()) storage.append(0);
        set(bitSize - 1, value);
    }
    void removeLast()
    {
        assert(bitSize > 0);
        if(lastWordBits() == 1) storage.removeLast(); //shrink if can
        --bitSize;
        zeroOutRemainder(); //removed bit might have been 1
    }
    bool operator==(Bitset const& rhs) const{return storage == rhs.storage;}
    Bitset& operator&=(Bitset const& rhs)
    { //only makes sense for equal sizes
        assert(bitSize == rhs.bitSize);
        for(int i = 0; i < wordSize(); ++i) storage[i] &= rhs.storage[i];
        return *this;
    }
    void flip()
    {
        for(int i = 0; i < wordSize(); ++i) storage[i] = ~storage[i];
        zeroOutRemainder();
    }
};

```

Replacing `&=` with `|=` or `^=` gives the codes for these operations. Can implement other set operations efficiently on top of the above.

Shifts ensure that the shift amount > 0 and mod `bitSize`, split it into word and bit shifts, and apply each separately. The word shift part moves the words and fills up the end with 0's. The bit shift part keeps track of carries to split and recombine words correctly. E.g., for the left shift start with the higher words, and carry their lower bits to the lower words. Both take $O(n/B)$ time.

```

Bitset& operator>>=(int shift)
{
    //shift by 0 no-op
    if(shift < 0) return operator<<=(-shift);
    int normalShift = shift % bitSize, wordShift = normalShift/B,
        bitShift = normalShift % B;
    if(wordShift > 0)//shift words
        for(int i = 0; i + wordShift < wordSize(); ++i)
        {
            storage[i] = storage[i + wordShift];
            storage[i + wordShift] = 0;
        }
    if(bitShift > 0)//shift bits
    {//for word layout 00000101|00111000 >> 4 -> 10000000|00000011
        WORD carry = 0;
        for(int i = wordSize() - 1 - wordShift; i >= 0; --i)
        {
            WORD tempCarry = storage[i] << (B - bitShift);
            storage[i] >>= bitShift;
            storage[i] |= carry;
            carry = tempCarry;
        }
    }
    return *this;
}
Bitset& operator<<=(int shift)
{
    if(shift < 0) return operator>>=(-shift);
    int normalShift = shift % bitSize, wordShift = normalShift/B,
        bitShift = normalShift % B;
    if(wordShift > 0)//shift words
        for(int i = wordSize() - 1; i - wordShift >= 0; --i)
        {
            storage[i] = storage[i - wordShift];
            storage[i - wordShift] = 0;
        }
    if(bitShift > 0)//shift bits
    {//for word layout 10000000|00000011 <<= 4 -> 00000000|00111000
        WORD carry = 0;
        for(int i = wordShift; i < wordSize(); ++i)
        {
            WORD tempCarry = storage[i] >> (B - bitShift);
            storage[i] <=> bitShift;
            storage[i] |= carry;
            carry = tempCarry;
        }
    }
    //some 1 bits could have shifted into the remainder
    zeroOutRemainder();
    return *this;
}

```

To check if all bits = 0 or set all bits work with words:

```

void setAll(bool value = true)
{
    for(int i = 0; i < wordSize(); ++i)
        storage[i] = value ? Bits::FULL : Bits::ZERO;
    zeroOutRemainder();
}
bool isZero() const
{

```

```

    for(int i = 0; i < wordSize(); ++i) if(storage[i]) return false;
    return true;
}

```

To manipulate a bit sequence of size n at position i , find the affected words, and work with their affected bits. In case of several words, the affected bits will be the left ones for the first word, all for the middle words, and the right ones for the last word. E.g., for word layout $yyyyxxxx/aaaaaaaa/xxxxxxxxy$, y are the affected bits. For get, for the first word start at $bit = 0$, get $\min(n, B-bit)$ bits, and put in the result from $shift = 0$. For later words $bit = 0$ and $shift$ increases by the number of read bits. Same for set. The cost is $O(\text{the number of affected words})$.

```

unsigned long long getValue(int i, int n) const
{
    assert(n <= numeric_limits<unsigned long long>::digits && i >= 0 &&
           i + n <= bitSize && n > 0);
    unsigned long long result = 0;
    for(int word = i/B, bit = i % B, shift = 0; n > 0; bit = 0)
        {//get lower bits first
            int m = min(n, B - bit); //all bits or as much as the word has
            result |= Bits::getValue(storage[word++], bit, m) << shift;
            shift += m;
            n -= m;
        }
    return result;
}

void setValue(unsigned long long value, int i, int n)
{
    assert(n <= numeric_limits<unsigned long long>::digits && i >= 0 &&
           i + n <= bitSize && n > 0);
    for(int word = i/B, bit = i % B, shift = 0; n > 0; bit = 0)
        {//set lower bits first
            int m = min(n, B - bit); //all bits or as much as the word has
            Bits::setValue(storage[word++], value >> shift, bit, m);
            shift += m;
            n -= m;
        }
}

```

The append operations are implemented in terms of `setValue` by increasing capacity if needed and setting the last bits to the wanted value:

```

void appendValue(unsigned long long value, int n)
{
    int start = bitSize;
    bitSize += n;
    int k = wordsNeeded() - wordSize();
    for(int i = 0; i < k; ++i) storage.append(0);
    setValue(value, start, n);
}

```

It's also useful to append a bit set directly:

```

void appendBitset(Bitset const& rhs)
{//append storage words and remove extra words if any
    if(rhs.getSize() > 0)
    {
        for(int i = 0; i < rhs.wordSize(); ++i)
            appendValue(rhs.storage[i], B);
        bitSize -= B - rhs.lastWordBits();
        if(wordSize() > wordsNeeded()) storage.removeLast();
    }
}

```

Reversing uses word operations, which is usually more efficient than a vector-like swap of bits. First shift to cover up the garbage bits, and then reverse the storage words. The 0's from the shift end up as the new garbage bits.

```

void reverse()
{//fill up garbage bits
}

```

```

int nFill = garbageBits();
bitSize += nFill;
(*this)<<=(nFill);
//reverse storage words
storage.reverse();
for(int i = 0; i < wordSize(); ++i)
    storage[i] = reverseBits(storage[i]);
//delete the garbage
bitSize -= nFill;
zeroOutRemainder();
}

```

Pop count = \sum pop counts of individual words.

```

int popCount() const
{
    int sum = 0;
    for(int i = 0; i < wordSize(); ++i) sum += popCountWord(storage[i]);
    return sum;
}

```

5.12 Union-find

Solve the **disjoint set problem** of maintaining an equivalence relation on subsets of items. Supported operations:

- Add a subset
- **Join** two subsets
- Check if two items are in the same subset
- Get the size of the subset containing an item

Array indices represent all items. The data structure is a vector of integers, initially -1, with $vector[i]$ representing node i . The integers represent negative subset sizes for root nodes and parent indices otherwise. Two items are in the same subset if their root indices are equal.

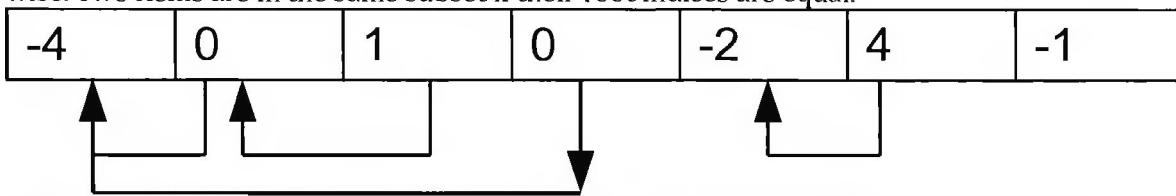


Figure 5.18: Pointer structure of a union-find with some items joined

Finding a root follows the chain of parent pointers to it, compresses the path by setting the parent of every visited node to the root, and returns its index. Getting the subset size returns -the value at $findRoot$.

```

class UnionFind
{
    mutable Vector<int> parent; //parent or negated size of the tree
public:
    UnionFind(int size): parent(size, -1){}
    bool isRoot(int n) const {return parent[n] < 0;}
    int find(int n) const
        {return isRoot(n) ? n : (parent[n] = find(parent[n]));}

    bool areEquivalent(int i, int j) const {return find(i) == find(j);}
    int subsetSize(int i) const {return -parent[find(i)];}
    void addSubset() {parent.append(-1);}
}

```

To join two roots:

1. Make the smaller subset a child of the larger
2. Change its root to the root of the other
3. Set the root of the other to -their total size

```

void join(int i, int j)
{
    int parentI = find(i), parentJ = find(j);
}

```

```

if (parentI != parentJ)
    // parent[parentI] and parent[parentJ] are negative sizes
    if (parent[parentI] > parent[parentJ]) swap(parentI, parentJ);
    parent[parentI] += parent[parentJ];
    parent[parentJ] = parentI;
}
}

```

Find and join are $O(\lg(n))$ and amortized $O(1)$ \forall practical n . This implementation is called **union-by-size with compression**.

5.13 Implementation Notes

My vector differs substantially in the API compared to the STL one. E.g., I included common functionality of a mathematical vector, which is perhaps an overconceptualization but a useful one.

The free list implementation idea is original and is actually the inspiration for the whole book. I was excited by the idea of data structure based on a sequence of blocks of doubling size, discovering this as a general strategy (see the “Miscellaneous Algorithms” chapter) a few years later. I considered the same structure for the stack and the queue implementations, but the slightly reduced memory manager calls of these don’t justify the clumsiness among other things, as discussed. The garbage collection allows to not write destructor for many pointer-based structures.

The bit algorithms were the hardest to select because there are usually many different ways to do something simple such as a pop count or a bit reversal. The table-based algorithms put the tables in static memory, which is best avoided, and can have cache misses, but smarter brute-force algorithms seem worse.

For a bit set one of the main decisions is what brute-force functionality to replace with word-based operations. The reverse function is the main example—for small words as storage the brute-force bit swapping is probably faster.

For union-find, subset size isn’t a common operation but included here because of usefulness and easy implementation.

5.14 Comments

A garbage-collecting free list is original to the best of my knowledge. Various smart pointers of C++ would have a similar convenience effect, but with extra overhead and without making memory allocation more efficient.

For pop count, a common warning is that on some hardware, if the table isn’t in a cache, shifting and counting could be faster despite using many more instructions. But from my tests the table version is an order-of-magnitude faster than the brute-force version. Alternative nonportable magic-number algorithms aren’t considered here. For these and many other bit algorithms see Warren (2012) and Arndt (2010).

For union-find, the classical alternative **union-by-height with compression** leads to the same asymptotic and practical performance but needs extra space for the heights. A slightly experimentally faster algorithm (Patwary et al. 2010) is more complicated, negligibly asymptotically slower, and doesn’t allow checking subset sizes.

5.15 Projects

- Does it make sense to have a square helper function? The built-in `pow` is very slow due to working with floating-point arguments and needing to give many bits of precision (see the “Numerical Algorithms—Working with Functions” chapter). Even the floating-point standard recommends that languages libraries offer a `pow` function for integer exponents (see the “Large Numbers” chapter for how to implement this O -efficiently). But a simpler, more efficient option for very small exponents such as 2 or 3 is to use template metaprogramming to simply expand the multiplications. Write such a function. The benefit is that don’t need to create temporary variables for common squaring or cubing.
- For vector does it make sense to add functions to insert/delete an item not at the end? This can be implemented on top of the minimal computational basis. Same for appending an array or a vector of items instead of a single one.
- Make pop count and bit reversing portable by not assuming 8 bits in a byte.
- For bit reversing study whether brute force is less efficient and with what word sizes.

5.16 References

- Arndt, J. (2010). *Matters Computational: Ideas, Algorithms, Source Code*. Springer.
- Joannou, S., & Raman, R. (2011). An empirical evaluation of extendible arrays. In *Experimental Algorithms* (pp. 447–458). Springer.
- Patwary, M. M. A., Blair, J., & Manne, F. (2010). Experiments on union-find algorithms for the disjoint-set data structure. In *Experimental Algorithms* (pp. 411–423). Springer.
- Warren, H. S. (2012). *Hacker's Delight*. Addison-Wesley.

6 Random Number Generation

"Lottery is a tax on those who don't know math" – Ambrose Bierce

"Anyone who considers arithmetic methods of producing random digits is, of course, in a state of sin" – John von Neumann

6.1 Introduction

Randomization is an important part of the algorithmic toolkit but is rarely discussed in an algorithms class. Some understanding of probability theory is assumed. This chapter is more advanced than the rest through "Graph Algorithms", which ends the standard algorithms class topics.

6.2 A Quick Review of Probability

Probability is a function that measures uncertainty of sets E_i of events, which are subsets of some **sample space** Ω . Typically Ω = the real numbers or the integers, and the sets are intervals—continuous or discrete. These are naturally extended to multidimensional Ω . The E_i can be composed using set operations such as intersections, unions, and complements. A probability satisfies **Kolmogorov axioms**:

- $P(\Omega) = 1$
- For disjoint E_i , $P(\cup E_i) = \sum P(E_i)$
- $P(E^c) = 1 - P(E)$

A **probability distribution** uses a **probability density function (PDF)** f to assign values to $x \in \Omega$ such that $\forall E \Pr(E) = \int_{x \in E} f(x)$. The **cumulative distribution function (CDF)** $F(x) = \int_{\infty < t \leq x} f(t)$ is often more convenient to work with. With these, can use intervals as elementary events, and calculate probabilities of more complicated events using the above axioms.

Some important distributions:

- **Normal(μ, σ)**— μ is the mean and σ the standard deviation. It models many events and is heavily used.

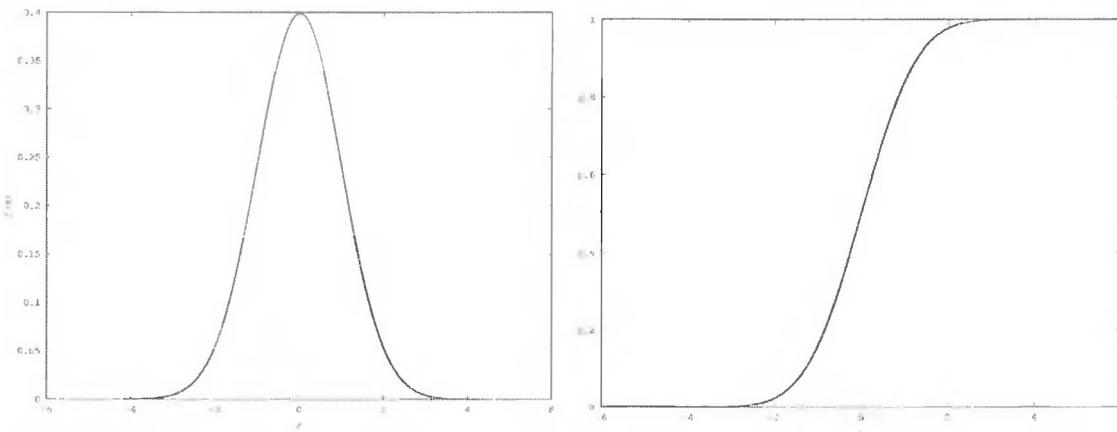


Figure 6.1: Normal($0, 1$) PDF and CDF

- **Geometric(p)**— $f(x) = (1-p)^{x-1} p$ —for discrete x .

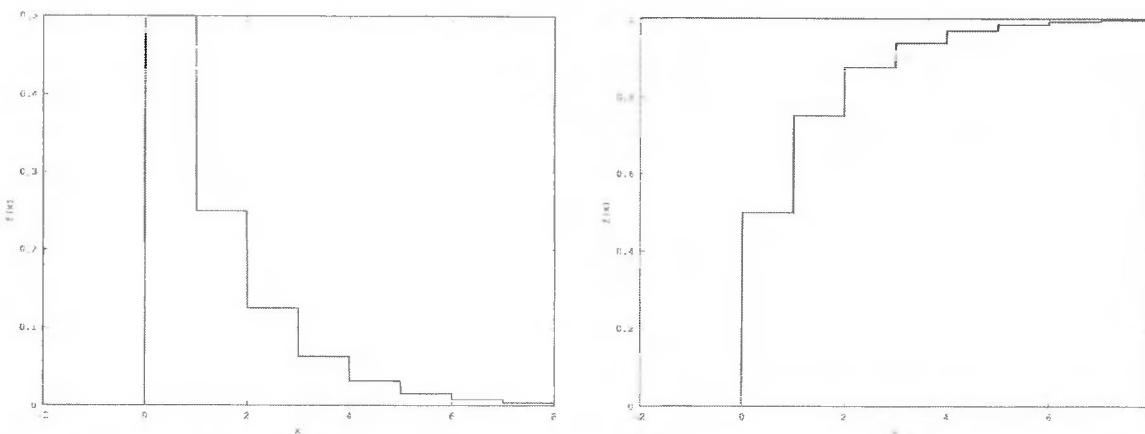


Figure 6.2: Geometric(0.5) PDF and CDF

Certain naturally occurring events don't have known probabilities, so assign them approximately using distributions. Many different distributions are defined for various cases and overall do a good job, but some of their assumptions don't hold in practice. E.g., for a continuous distribution $\Pr(\text{a particular } x) = 0$, but if $x = 0$ or some other logical value, a continuous distribution is technically a wrong model because $x = 0$ can occur reasonably often. This has resulted in patches to some statistical procedures, such as handling assumption-contradicting ties in nonparametric tests (see the "Computational Statistics" chapter).

Many events follow the normal distribution due to statistical mechanics—random interactions of many particles tend to be normal by the CLT (discussed later in the chapter). But many others are incorrectly considered normal when they aren't. Every distribution is a good approximate model for some events and a bad one for others. E.g., the normal is a bad model for a flying bird's avoiding a pole because the bird will swerve left or right, producing a two-peaked distribution:

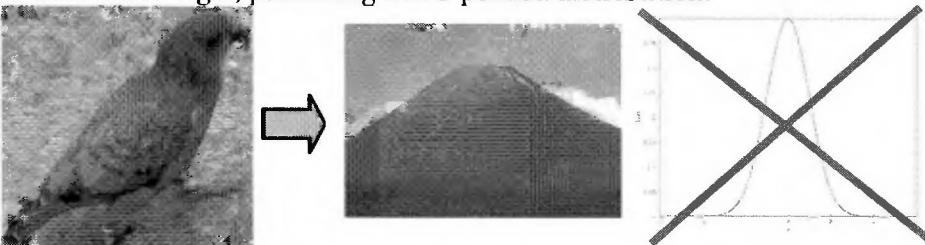


Figure 6.3: Smart birds don't fly into obstacles normally

Multimodal or skewed distributions are often important counter-examples to assumptions of particular statistical procedures. In many cases, the needed information about the data isn't the full probability distribution but a summary parameter such as **expected value** $\mu = \int_{x \in S} xf(x)$ or **variance** $\sigma^2 = E[(x - \mu)^2]$.

6.3 Pseudorandom Number Generation



Figure 6.4: Will it be heads or tails?

Games, randomized algorithms, and simulations use random numbers, from which can create more complex random objects. A number sequence is **random** if observing its past outputs doesn't help predict the next one.

Uniformly distributed physical phenomena such as atmospheric noise or radioactive decay give such random numbers and should be used in lotteries or jury selection, where need provable randomness. E.g., see www.random.org. Special generator devices are available, and operating systems have a random stream generated from the CPU or hard drive activity. These are used for cryptographic algorithms but are slow and nonportable.

Fast **pseudo-random generators** generate a random sequence starting from some **initial state**. The output and the next state are respectively the output (g) and the **transition** (f) functions of the current state. Usually, need $O(1)$ time for both and $O(1)$ space for state.

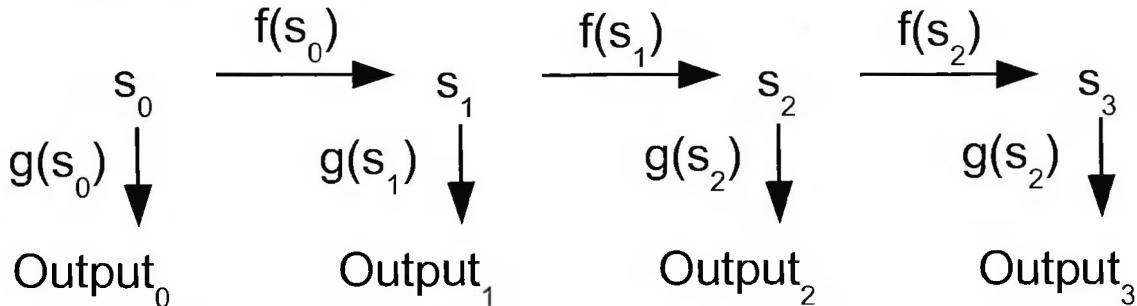


Figure 6.5: Generic pseudo-random generator state transitions and outputs

Is this random enough even if the initial state has many random bits? Technically not because by combinatorial arguments not all output sequences are possible. But almost always assume that pseudorandom is random for a given application. Any worst-case behavior that is deliberately trying to exploit the mechanics of a particular good generator tends to be possible only if intended. In practice the information stream of the generator and that of the domain are independent.

For simplicity and portability the initial state can be a function of the system time and a password. Use other, not portably accessible sources if need more random bits. For complete independence of generator runs, save its state to a file, and restore for the next simulation, but this is rarely worth the effort. The simplest generators use a single-word state and the identity output function. The output usually has 32 or 64 bits. Can combine two consecutive 32-bit outputs to get 64-bit outputs if needed.

Good generators:

- Pass most tests of extensive statistical suites, such as TestU01 (L'Ecuyer & Simard 2007), that try to reject the hypothesis (see the “Computational Statistics” chapter for hypotheses tests) that the sequence is random and have **period** (after how many transitions the state = the initial state) $\geq 2^{64}$ and high **equidistribution** (the max k for which a sequence of k outputs can be any sequence). Due to multiple testing (see the “Computational statistics” chapter), passing all tests isn't necessary.
- Are efficient, simple, portable, and easy to initialize.
- Return a random `double` $u \in (0, 1)$, and not $[0, 1]$, to avoid undefined results for transformations such as $\log(u)$. No generator produces 2^w , so a normalization constant $\geq 2^w$ takes care of the 1. The `double` normalization constants for full-range 32- and 64-bit values are respectively 2.32830643653869629E-10 and 5.42101086242752217E-20 (Press et al. 2007). Some generators also don't produce 0. Otherwise use $\max(1, u)$ or generate until $u \neq 0$. The former is biased, and the latter has no worst-case bound, but both are fine in practice.
- Optionally generate **independent streams** for parallel computations. This needs very long period and the ability to skip forward efficiently.

The historically popular **linear congruential generator** with single-word state s uses transition $s = (as + c) \% m$ (e.g., with $a = 69069$, $c = 1234567$, and $m = 2^{32}$). But it has poor quality:

- The period of the lower k bits = 2^k , which is problematic for typical `% n` use case
- It fails many tests
- To avoid overflow, the multiplication needs double-word precision or breaking up into parts

The C++ `rand()` is likely to be based on a LCG or another poor-quality generator, so it's best to avoid it unless the standard changes to demand at least certain quality.

6.4 Xorshift

The transition multiplies the bit vector represented by word `state` by a sparse Boolean matrix and is implemented by a sequence of shifts and xors. The sequence used here has good test suite results; mathematical justification of Xorshift properties is complicated (Press et al. 2007). For 64-bit state the period is $2^{64} - 1$ (0 is never generated), which is large enough. Below is the 32-bit transition function, with period $2^{32} - 1$ and different coefficients, which is useful for implementing universal hashing (see the “Hashing” chapter):

```

uint32_t xorshiftTransform(uint32_t x)
{
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
}
  
```

```

x ^= x >> 17;
x ^= x << 5;
return x;
}

```

Only the 64-bit version is considered for pseudo-random generation due to superior test performance (discussed later in the chapter). Still, the bits of successive numbers have some correlation due to linearity of matrix multiplication. So **QualityXorshift64** outputs the LCG successor of the Xorshift result.

```

class QualityXorshift64
{
    uint64_t state;
    enum{PASSWORD = 19870804};

public:
    QualityXorshift64(uint64_t seed = time(0) ^ PASSWORD)
        {state = seed ? seed : PASSWORD;}
    static uint64_t transform(uint64_t x)
    {
        x ^= x << 21;
        x ^= x >> 35;
        x ^= x << 4;
        return x * 2685821657736338717ull;
    }
    uint64_t next(){return state = transform(state);}
    unsigned long long maxNextValue(){return numeric_limits<uint64_t>::max();}
    double uniform01(){return 5.42101086242752217E-20 * next();}
};

```

It doesn't generate 0 because the multiplication constant is prime, and for a prime p , $0 = xp \% 2^{64} \rightarrow xp = c2^{64}$. Because $c > p$, c divides p , implying a contradiction that $x = \frac{p}{c}2^{64}$, while x is less.

The Xorshift transitions are one-to-one, and the LCG ones aren't.

6.5 Mersenne Twister

Mersenne Twister is popular, fast, and high quality in either 32 or 64 bits. (Matsumoto and Nishimura 1998) presents a short algorithm, while <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html> has unnecessarily complicated C implementations, which however contain updated parameters and initialization procedures. The 64-bit version has an extra & with a constant, but otherwise is only different in parameters. The generator is as fast as QualityXorshift64 but much more complicated and uses an array of 623 integers as state.

So I simplified the codes and confirmed that the simplifications are equally fast on my machine.

```

class MersenneTwister64
{
    enum{N = 312, PASSWORD = 19870804};
    uint64_t state[N];
    int i;

public:
    MersenneTwister64(uint64_t seed = time(0) ^ PASSWORD)
    {
        state[0] = seed;
        for(i = 1; i < N; ++i) state[i] = 6364136223846793005ULL *
            (state[i - 1] ^ (state[i - 1] >> 62)) + i;
        i = 0;
    }
    uint64_t next()
    {
        int j = (i + 1) % N, k = (i + 126) % N;
        uint64_t y = (state[i] & 0xFFFFFFFF80000000ULL) |
            (state[j] & 0x7fffffffULL);
        y = state[i] =
            state[k] ^ (y >> 1) ^ (y & 1 ? 0xB5026F5AA96619E9ULL : 0);
        i = j;
        y ^= (y >> 29) & 0x5555555555555555ULL;
    }
};

```

```

    y ^= (y << 17) & 0x71D67FFFEDA60000ULL;
    y ^= (y << 37) & 0xFFFF7EEE0000000000ULL;
    y ^= (y >> 43);
    return y;
}
double uniform01() {return 5.42101086242752217E-20 * next(); }
};

```

6.6 MRG32k3a

Multiple recursive generator (MRG) generalizes LCG by being a linear combination of several last states mod m . MRG32k3a combines output of two three-state generators, which use the transition $s_{i,next} = c_i \cdot s_i T_i \% m_i$, with suitably chosen transition matrices T_i and moduli m_i . Both parameters were picked by an exhaustive search, and the suggested T_i are sparse for efficiency (L'Ecuyer 1999). The output is a combination of the c_i .

```

struct MRG32k3a
{
    enum{PASSWORD = 19870804};
    static long long const m1 = 429496708711, m2 = 429494444311;
    long long s10, s11, s12, s20, s21, s22;
    void reduceAndUpdate(long long c1, long long c2)
    {//helper for transition and jumping
        if(c1 < 0) c1 = m1 - (-c1 % m1);
        else c1 %= m1;
        if(c2 < 0) c2 = m2 - (-c2 % m2);
        else c2 %= m2;
        s10 = s11; s11 = s12; s12 = c1;
        s20 = s21; s21 = s22; s22 = c2;
    }
public:
    unsigned long long next()
    {//doesn't return 0
        long long c1 = (1403580 * s11 - 810728 * s10),
               c2 = (527612 * s22 - 1370589 * s20);
        reduceAndUpdate(c1, c2);
        return (c1 <= c2 ? m1 : 0) + c1 - c2;
    }
    unsigned long long maxNextValue(){return m1;}
    //s1(0-2) and s2(0-2) must be respectively < m1 and m2 and not all 0
    MRG32k3a(): s10(max(time(0) ^ PASSWORD, 11) % m2), s11(0), s12(0),
               s20(s10), s21(0), s22(0){}
    double uniform01(){return next()/(m1 + 1.0);}//ensures u in (0, 1)
};

```

Can skip forward using efficient exponentiation mod m (see the “Large Numbers” chapter) on T_i . The recommended jump size = 2^{76} , and the resulting matrices are already computed (L'Ecuyer et al. 2002).

```

void jumpAhead()
{
    const long long A1p76[3][3] = {
        { 82758667u, 1871391091u, 4127413238u},//for s10
        {3672831523u, 69195019u, 1871391091u},//for s11
        {3672091415u, 3528743235u, 69195019u}},//for s12
    A2p76[3][3] = {
        {1511326704u, 3759209742u, 1610795712u},//for s20
        {4292754251u, 1511326704u, 3889917532u},//for s21
        {3859662829u, 4292754251u, 3708466080u}};//for s22
    long long s1[3] = {s10, s11, s12}, s2[3] = {s20, s21, s22};
    for(int i = 0; i < 3; ++i)
    {
        long long c1 = 0, c2 = 0;
        for(int j = 0; j < 3; ++j)
        {
            c1 += s1[j] * A1p76[i][j];
            c2 += s2[j] * A2p76[i][j];
        }
        reduceAndUpdate(c1, c2);
    }
}

```

```

        c2 += s2[j] * A2p76[i][j];
    }
    reduceAndUpdate(c1, c2);
}
}

```

A minor issue with this generator is that the range of the output isn't full 64-bit, but is capped at m_1 . A 64-bit version hasn't been developed. So, e.g.:

- u , which has ≈ 16 digits of precision, only gets ≈ 12
 - Anything that relies on the log of the variate (or u) may be too bounded in range (u can't be too close to 0 due to precision limit)

6.7 RC4

Cryptographically secure random output isn't predictable by any efficient method whose result is more than negligibly better than a guess. RC4 securely generates a byte at a time from a random enough initial key. To avoid some known attacks, it drops the first 1024 bytes. "RC4" is trademarked, and implementations call it "ARC4" (with "A" for "alleged"), though the trademark is probably lost due to nondefense. The logic behind the algorithm is complicated, and its security still isn't well-understood (see Wikipedia 2013 for details). It tries to generate and maintain something like a random permutation.

```

struct ARC4
{
    unsigned char sBox[256], i, j;
    enum{PASSWORD = 19870804};
    void construct(unsigned char* seed, int length)
    {
        j = 0;
        for(int k = 0; k < 256; ++k) sBox[k] = k;
        for(int k = 0; k < 256; ++k)
            //different from the random permutation algorithm
            j += sBox[k] + seed[k % length];
            swap(sBox[k], sBox[j]);
    }
    i = j = 0;
    for(int dropN = 1024; dropN > 0; dropN--) nextByte();
}

ARC4(unsigned long long seed = time(0) ^ PASSWORD)
    {construct((unsigned char*)&seed, sizeof(seed));}
//for cryptographic initialization from a long seed
ARC4(unsigned char* seed, int length){construct(seed, length);}
unsigned char nextByte()
{
    j += sBox[++i];
    swap(sBox[i], sBox[j]);
    return sBox[(sBox[i] + sBox[j]) % 256];
}

unsigned long long next()
{
    unsigned long long result = 0;
    for(int k = 0; k < sizeof(result); ++k)
        result |= ((unsigned long long)nextByte()) << (8 * k);
    return result;
}

unsigned long long maxNextValue()
    {return numeric_limits<unsigned long long>::max();}
double uniform01(){return 5.42101086242752217E-20 * max(lull, no);
}

```

It still uses $O(1)$ time and space due to 256's being a constant.

6.8 Picking a Generator

Generator	Period, Memory	Number of	Seconds for 2^{30}	Special considerations
-----------	----------------	-----------	----------------------	------------------------

	for state	TestU01 failures	next() calls	
Xorshift	$2^{32} - 1$, 4 bytes	59	3	Simplest, fastest, good as a 32-bit hash function
Xorshift64	$2^{64} - 1$, 8 bytes	16, with different shifts	6.3	Useful as a building block and as a hash function
QualityXorshift64	$2^{64} - 1$, 8 bytes	N/A; 0 for smaller test suite Diehard	6.6	Simple, good as a 64-bit hash function
MGR32k3a	$\approx 2^{182}$, 24 bytes	0	25.6	Supports skipping
Mersenne Twister 64	$2^{19937} - 1$, 2504 bytes	N/A (4 for the 32-bit version)	9.7	Popular, but difficultly escaping states with many 0's
RC4	Huge, 256 bytes	0	47.8	Cryptographic

Simulations aim to not lose significance due to nonrandomness, and randomized algorithms to be efficient overall. All generators are fast enough to make the use of the result the bottleneck in most cases. So use:

- QualityXorshift64 (my choice) or Mersenne Twister (popular choice) as default
- MGR32k3a if need independent stream support
- RC4 for cryptography

6.9 Using a Generator

A global object does generation. It also generates numbers mod n and $\in [a, b]$, and most generators of random objects and samples from probability distributions are its members.

```
template<typename GENERATOR = QualityXorshift64> struct Random
{
    GENERATOR g;
    enum{PASSWORD = 19870804};
    Random(unsigned long long seed = time(0) ^ PASSWORD): g(seed) {}
    unsigned long long next(){return g.next();}
    unsigned long long maxNextValue(){return g.maxNextValue();}
    unsigned long long mod(unsigned long long n)
    {
        assert(n > 0);
        return next() % n;
    }
    int sign(){return mod(2) ? 1 : -1;}
    long long inRange(long long a, long long b)
    {
        assert(b >= a);
        return a + mod(b - a + 1);
    }
    double uniform01(){return g.uniform01();}
};

Random<>& GlobalRNG()
{
    static Random<> r; // runs only once
    return r;
}
```

6.10 Generating Samples from Distributions

Most generation algorithms assume the ability to generate u , a sample from $\text{uniform}(0, 1)$. Several general methods work with it for continuous distributions:

- **Inverse**—a cumulative probability distribution F is a function mapping $x \rightarrow [0, 1]$, so $F^{-1}(u)$ is a random variate. E.g., see the exponential distribution generator later in the chapter. This works well when F^{-1} is easy to calculate. Numerically solving $F(x) = u$ (see the “Numerical Algorithms—Working with Functions” chapter) works but is slow. For the latter find a range containing the x correspond-

ing to u using exponential search starting from $x = 0$:

```
template<typename CDF> double invertCDF(CDF const& c, double u,
    double guess = 0, double step0 = 1, double prec = 0.0001)
{
    assert(u > 0 && u < 1);
    auto f = [c, u](double x){return u - c(x);};
    pair<double, double> bracket = findInterval0(f, guess, step0, 100);
    return solveFor0(f, bracket.first, bracket.second, prec).first;
}
```

Exponential search is guaranteed to work on monotonic functions, so the above implementation is robust. Inversion also works for discrete distributions but not for multidimensional ones.

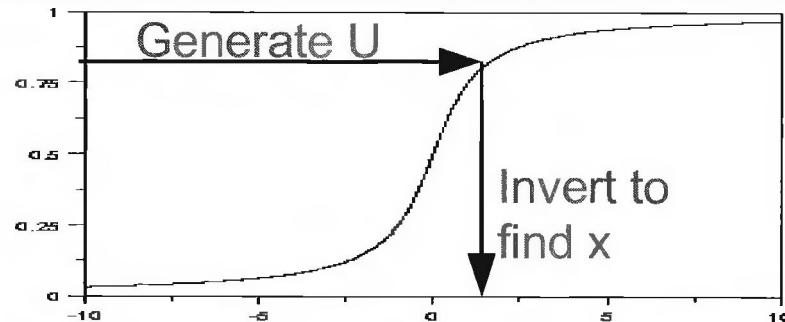


Figure 6.6: Inversion method strategy

- **Accept/reject**—If $x \in A$ is uniformly distributed in A , $B \subseteq A$, and $x \in B$, then x is uniformly distributed in B . Let f, g be PDFs such that for some constant $c > 0 \ \forall x \in \mathbb{R}^D f(x) < cg(x)$. If x is generated from g , u from uniform(0, $cg(x)$), and $u \leq f(x)$, then x is distributed according to f (Devroye 1986). So generate x and u until $u \leq f(x)$, and return x . This works for discrete and multidimensional distributions, but for large D is inefficient because $E[\text{the number of rejections}]$ is exponential in D . Need only an inside/outside membership tester, and can speed up generation by checking if a sample is inside a fast-to-test-membership **squeeze function** contained by f —e.g., see the gamma generator later in this chapter. A technical issue is that the worst-case number of generations = ∞ , even though it's exponentially unlikely to be larger than expected.

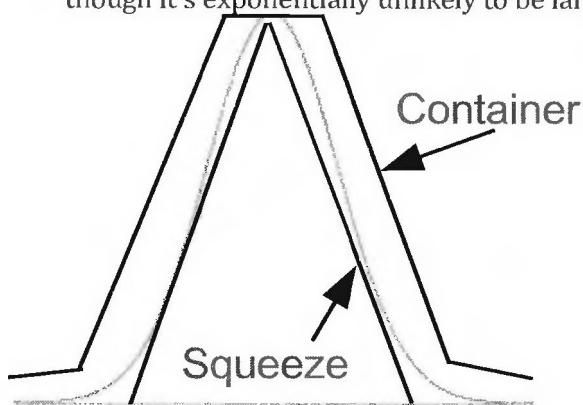


Figure 6.7: Accept/reject method strategy

- Relationships among distributions—can use variates from some distributions to generate from some related distributions. E.g., the generators for beta and many others reduce to those for gamma.
- **Composition method**—partition a distribution defined on (a, b) into two distributions defined on (a, c) and $[c, b]$, generate a selection variable that defines which of the two is used, and generate from the picked one using different method than for the other. This is most convenient for distribution parameters (where don't pick a region)—e.g., the gamma generator uses different strategies depending on whether the shape parameter < 1.

6.11 Generating Samples from Bounded-range Discrete Distributions

Assume the wanted distribution is represented as an array p of n probabilities. If need to generate only $O(1)$ values, the simplest and fastest method is to generate u and do a linear search to find the first index i such that $\sum p_i \geq u$. But even here use the below method from an API.

The **alias method** uses $O(n)$ memory and supports $O(1)$ generation. In the easy-generation case of equal probabilities, the chance of each value is $1/n$. In general, some values will have more probability (**rich**) and others less (**poor**), but the average is still $1/n$. So reduce to the easy case by pairing every poor with a single

rich that gives it probability. A rich can get poor by giving too much and will be paired with another rich. To simplify, instead of the probabilities use $wealth_i = np_i$. After the pairing completes, $\forall i \in wealth_i +$ the net taken-and-given amount. An **alias** is the index of the donor.

To sample:

1. Generate uniformly random i
2. If i has no donor, return i
3. Else generate uniform(0, 1/n), and return i with probability proportional its original wealth, else its donor

All probability is accounted for. The pairing is greedy:

1. Put every index i in the poor list if $wealth_i < 1$ and the rich list otherwise
2. Until either list is empty
3. Pick a poor and a rich, take from the rich $1 - wealth_{poor}$
4. Move the rich to the poor list if it became poor

To remember the original poor wealth, don't update $wealth_i$ to make it 1. The combined list size shrinks even if a rich becomes poor, so have $O(n)$ runtime. Can use any type of list, but a stack is simple and common. Both lists should become empty at the same time, but not always due to round-off, which isn't an issue because only a negligible amount of wealth will be unaccounted for. So pair until one list is empty.

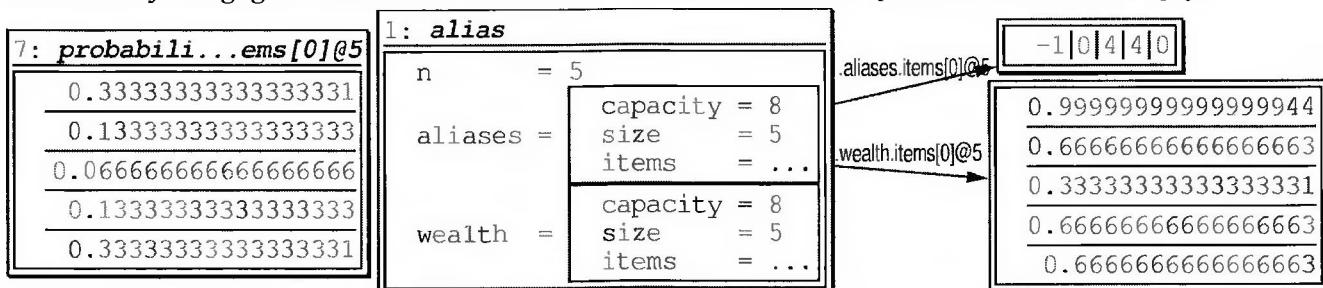


Figure 6.8: Alias method memory for 5 items and some unequal probabilities

```
class AliasMethod
{
    int n;
    Vector<int> aliases; // -1 means no donor
    Vector<double> wealth; // original or after donation
public:
    AliasMethod(Vector<double> const& probabilities):
        n(probabilities.getSize()), aliases(n, -1), wealth(n, 0)
    {
        Stack<int> smaller, greater;
        for(int i = 0; i < n; ++i)
            // separate into poor and rich
            (wealth[i] = n * probabilities[i]) < 1 ?
                smaller.push(i) : greater.push(i);
        while(!smaller.isEmpty() && !greater.isEmpty())
            // reassign wealth until no poor remain
            int rich = greater.pop(), poor = smaller.pop();
            aliases[poor] = rich;
            wealth[rich] -= 1 - wealth[poor];
            if(wealth[rich] < 1) smaller.push(greater.pop());
    }
    int next()
    // -1 check handles wealth round-off accumulation
    int x = GlobalRNG().mod(n);
    return aliases[x] == -1 || GlobalRNG().uniform01() < wealth[x] ?
        x : aliases[x];
};
```

A sum heap uses no extra space and allows efficient generation, updates, retrieval, and cumulative probability retrieval. It's a tree where the value of parent = \sum the values of its children and itself:

$6 = 1 + 2 + 3$, so the root's value = 3

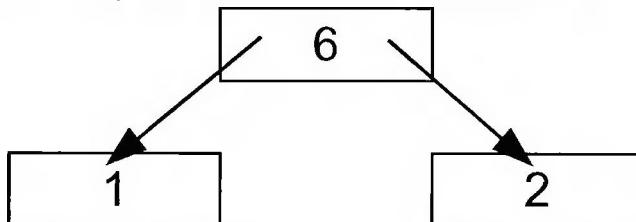


Figure 6.9: Sum heap parent node sum logic

For a probability distribution, the root's value = 1; if the distribution isn't normalized, the value = the normalization constant. The heap represents $[0, 1]$ recursively by $[\text{left CDF}] \cup [\text{right CDF}] \cup [\text{parent PDF}]$, where the value of parent PDF = $\text{parent} - \text{left} - \text{right}$. As for the alias method, the nodes are indexed as array positions. Most operations take $O(\lg(n))$ time. The values are set using incremental updates. An update adds the increment amount to the node and its every ancestor. As for binary heap (see the "Priority Queues" chapter), the implementation uses a vector for storage.

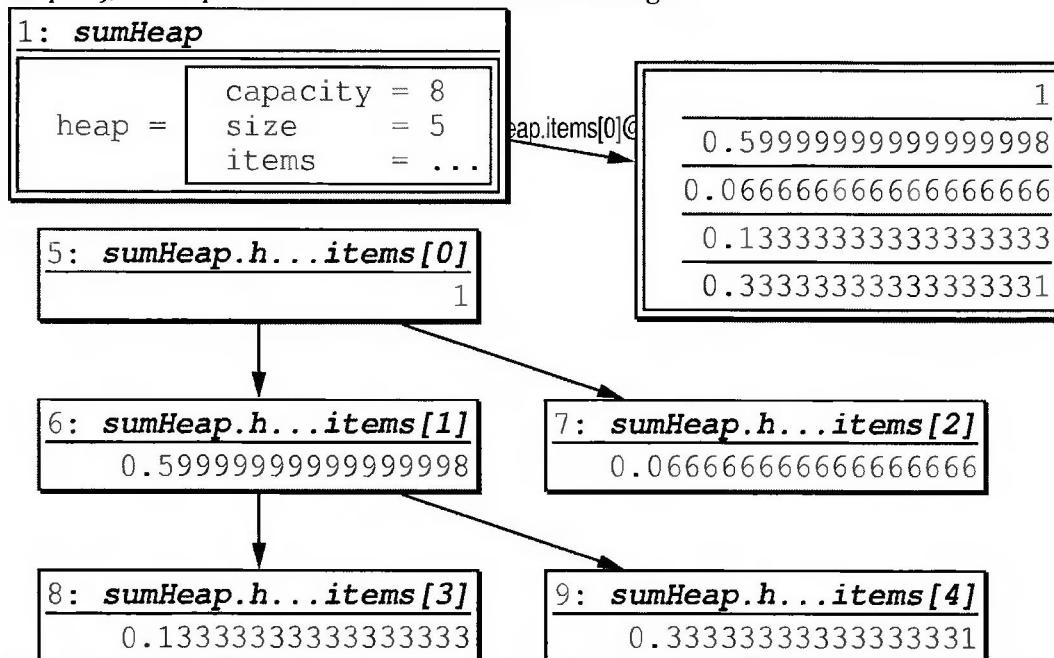


Figure 6.10: Sum heap memory logic for the same data

```

template<typename ITEM> class SumHeap
{
    Vector<ITEM> heap;
    int parent(int i){return (i - 1)/2;}
    int leftChild(int i){return 2 * i + 1;}
    int rightChild(int i){return 2 * i + 2;}
public:
    ITEM total(){return heap[0];}
};
  
```

To get the value of a node subtract its children's values:

```

ITEM get(int i)
{
    ITEM result = heap[i];
    int c = leftChild(i);
    if(c < heap.getSize())
    {
        result -= heap[c];
        c = rightChild(i);
        if(c < heap.getSize()) result -= heap[c];
    }
    return result;
}
  
```

Adding value to a node adds it to the node and to every parent on the path to the root.

```

void add(ITEM value, int i = -1)
{ // -1 means no nodes yet
    if(i == -1)
    {
        i = heap.getSize();
        heap.append(0);
    }
    for(;; i = parent(i))
    {
        heap[i] += value;
        if(i < 1) break;
    }
}

```

Generation uses the interval relationship. Given a value $\in [0, \text{the root's value}]$, find the corresponding node by checking where it lands in the interval of the current node and returning it or continuing to one of its children. Accumulate the left values until add up to the wanted value.

```

int find(ITEM value)
{
    assert(0 <= value && value <= total());
    ITEM totalLeftValue = 0;
    for(int i = 0, c;; i = c)
    {
        c = leftChild(i);
        if(c >= heap.getSize()) return i; // leaf node
        if(value > totalLeftValue + heap[c])
        {
            totalLeftValue += heap[c];
            c = rightChild(i);
            if(c >= heap.getSize() || // check if value in parent
                value > totalLeftValue + heap[c]) return i;
        }
    }
    int next() {return find(GlobalRNG().uniform01() * total());}

```

CDF of a node = its value + \sum the values of all its left siblings on the path to the root.

```

ITEM cumulative(int i)
{
    ITEM sum = heap[i];
    while(i > 0)
        { // add value of every left sibling if
            int last = i;
            i = parent(i);
            int l = leftChild(i);
            if(l != last) sum += heap[l];
        }
    return sum;
}

```

6.12 Generating Samples from Specific Distributions

Some of the presented generators are simple but the others mathematically complicated. The latter are discussed in more detail in Kroese et al. (2011) or the specific references. Those for some important continuous distributions:

- **Uniform(a, b)** with $a < b$. $f(x) = \begin{cases} 1/(b-a), & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$. $F(x) = 0$ if $x < a$, $\frac{x-a}{b-a}$ if $a \leq x \leq b$, and 1 if $x > b$.

b. The generator scales u :

```
double uniform(double a, double b) {return a + (b - a) * uniform01();}
```

- **Exponential(a)** models times between independent events, such as times between traffic accidents for a particular driver. $f(x) = ae^{-ax}$, and $F(x) = 1 - e^{-ax}$. To use the inverse method, let $F(x) = u$, and solve to get $x = -\ln(1 - u)/a$, which simplifies to $-\ln(u)/a$ because u and $1 - u$ have the same distribution (note that `<cmath> log` uses base e):

```
double exponential(double a) {return -log(uniform01())/a;}
```

- **Cauchy(μ, σ)**—fat tails and undefined expectation. Intuitively, the latter happens because a single observation can account for > 99% of the sample variance. $f(x) = \frac{1}{\pi + ((x-\mu)/\sigma)^2}$, and $F(x) = \frac{1}{\pi} \arctan\left(\frac{x-\mu}{\sigma}\right) + \frac{1}{2}$. Generation uses the inverse method by solving the above:

```
double cauchy(double m, double q)
    {return (tan((uniform01() - 0.5) * PI()) + m) * q;}
```

- **Normal(μ, σ)**—widely used and needed by other generators. $f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$, and $F(x) = \frac{1}{2} \left(1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right)$, where erf is the error function. Generation uses the **polar method**, specific to the normal distribution (Press et al. 2007). $E[\text{the runtime}] = O(1)$.

```
double normal01()
{
    for(; ;)
    {
        double a = 2 * uniform01() - 1, b = 2 * uniform01() - 1,
               c = a * a + b * b;
        if(c < 1)
        {
            double temp = sqrt(-2 * log(c)/c);
            return a * temp; //can return b * temp as 2nd iid sample
        }
    }
    double normal(double m, double q) {return m + q * normal01();}
}
```

- **Gamma($1, b$)**—used by other generators. The generator is complicated, but the basic idea is to accept/reject with a fast polynomial squeeze function (Marsaglia & Tsang 2000). $E[\text{the runtime}] = O(1)$.

```
double gamma1(double b)
{
    assert(b > 0);
    if(b >= 1)
    {
        for(double third = 1.0/3, d = b - third, x, v, u, xs;;)
        {
            do
            {
                x = normal01();
                v = 1 + x * third/sqrt(d);
            }while(v <= 0);
            v *= v * v; u = uniform01(), xs = x * x;
            if(u > 0.0331 * xs * xs || log(u) < xs/2 +
               d * (1 - v + log(v))) return d * v;
        }
    }
    else return pow(uniform01(), 1/b) * gamma1(b + 1);
}
```

- Derived from gamma using relationships between distributions:

```
double erlang(double m, int k){return gamma1(k) * m/k; }
double chiSquared(int k){return 2 * gamma1(k/2.0); }
double t(int v){return sqrt(v/chiSquared(v)) * normal01(); }
double beta(double p, double q)
{
    double G1 = gamma1(p);
    return G1/(G1 + gamma1(q));
}
```

```
double F(int v1, int v2)
    {return v2 * chiSquared(v1) / (v1 * chiSquared(v2));}
```

- **Triangular(*middle*)**—PDF is the triangle formed by the points (0,0), (*middle*, 1), and (1, 0). Generator is obtained by integrating to get CDF and using the inverse method (Wikipedia 2015):

```
double triangular01(double middle)
{
    assert(0 < middle && middle < 1);
    double u = uniform01();
    return sqrt(u <= middle ? middle * u : (1 - middle) * (1 - u));
}
double triangular(double a, double b, double middle)
    {return a + (b - a) * triangular01((middle - a)/(b - a));}
```

- **Levy(*c*)**—fatter tails than Cauchy, asymptotically $O(1/x^{1.5})$ (Wikipedia 2017).

$f(x) = \sqrt{\frac{c}{2\pi}} \exp\left(-\frac{c}{2x}\right) \frac{1}{x^{1.5}}$. Generation uses the fact that if N is a sample from $\text{normal}(0, 1/\sqrt{c})^2$,

then $1/N \sim \text{Levy}(c)$. Using $c = 0.455$ gives median ≈ 1 . Unlike Cauchy, Levy is one-sided, but can use the composition method to sample from a “symmetrized Levy” distribution by sampling $-\text{Levy}(c)$ with probability 0.5.

```
double Levy(double scale = 0.455)
{
    double temp = normal(0, 1/sqrt(scale));
    return 1/(temp * temp);
}
double symmetrizedLevy(double scale = 0.455) {return sign() * Levy(scale);}
```

For discrete distributions using relationships between distributions often gives good generators. Some important ones:

- **Bernoulli(*p*)**—1 with probability *p* and 0 with $1 - p$:

```
bool bernoulli(double p) {return uniform01() <= p;}
```

- **Binomial(*p, n*)**—a sum of *n* Bernoulli(*p*). The runtime is $O(n)$.

```
int binomial(double p, int n)
{
    int result = 0;
    for(int i = 0; i < n; ++i) result += bernoulli(p);
    return result;
}
```

- **Geometric(*p*)**—the number of times Bernoulli(*p*) is generated to make it = 1. $E[\text{the runtime}] = O(1/p)$. For *p* = 0.5 use random integer bits for efficiency.

```
int geometric(double p)
{
    assert(p > 0);
    int result = 1;
    while(!bernoulli(p)) ++result;
    return result;
}
int geometric05() {return rightmost0Count(next()) + 1;}
```

- **Poisson(*l*)**—the number of independent events in a given time interval with *l* average. Intervals between events \sim exponential, so generate intervals between events, and count how many fit into the *l* interval. To avoid logs in the exponential generator, work with the exponentiated values directly. The runtime is $O(l)$.

```
int poisson(double l)
{
    assert(l > 0);
    int result = -1;
    for(double p = 1; p > exp(-l); p *= uniform01()) ++result;
    return result;
}
```

```

GlobalRNG().uniform01() 0.60228108650398515
GlobalRNG().uniform(10, 20) 18.376161340514045
GlobalRNG().normal01() 1.7189974518331508
GlobalRNG().normal(10, 20) 15.259720132769834
GlobalRNG().exponential(1) 0.54213976708769562
GlobalRNG().gamma1(0.5) 0.023344422176525669
GlobalRNG().gamma1(1.5) 3.0564205547857468
GlobalRNG().weibull1(20) 0.95422285166630827
GlobalRNG().erlang(10, 2) 19.44737361504022
GlobalRNG().chiSquared(10) 13.561992437780951
GlobalRNG().t(10) -0.51654593773277468
GlobalRNG().logNormal(10, 20) 3.9190681459327817e+023
GlobalRNG().beta(0.5, 0.5) 0.058201077636632362
GlobalRNG().F(10, 20) 0.3580019590627147
GlobalRNG().cauchy(0, 1) 4.700499584114346
GlobalRNG().Levy() 8.4811381466439535
GlobalRNG().symmetrizedLevy() -0.11641736404696633
GlobalRNG().binomial(0.7, 20) 14
GlobalRNG().geometric(0.7) 1
GlobalRNG().poisson(0.7) 1

```

Figure 6.11: Sample output of the discussed generators

6.13 Generating Random Objects

Technically can't properly generate some objects because the number of bits needed to specify an object > that in the generator state, but in practice this isn't a problem.

In mathematical calculations it's often useful to generate a random unit vector. To allow normalization, want to avoid all-0 components. So disallow 0 by generating each component from $\pm u$.

```

Vector<double> randomUnitVector(int n)
{
    Vector<double> result(n);
    for(int i = 0; i < n; ++i) result[i] = uniform01() * sign();
    result *= 1/norm(result);
    return result;
}

```

To randomly permute n items, swap the first with a random one, and randomly permute the remaining $n - 1$:

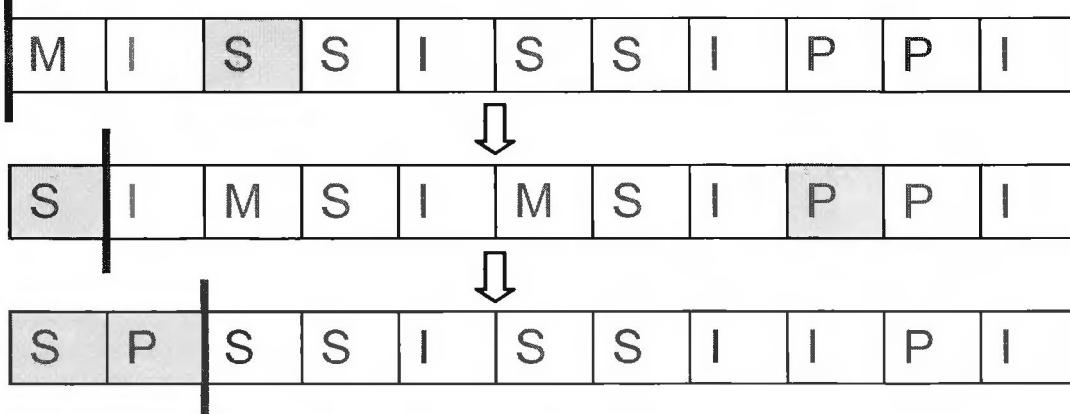


Figure 6.12: Random permutation recursive logic: here pick "S" for position 0 and "P" for 1

```

template<typename ITEM> void randomPermutation(ITEM* numbers, int size)
{
    for(int i = 0; i < size; ++i)
        swap(numbers[i], numbers[inRange(i, size - 1)]);
}

```

The runtime is $O(n)$, which is faster than sorting with random priorities. But the latter is faster for external memory (see the "External Memory Algorithms" chapter).

To generate an **ordered sample** of k integers $\in [0, n - 1]$, select each with probability $\frac{k - nSelected}{n - nConsidered}$.

Then 0 is included with probability $\frac{k}{n}$, and 1 with probability $\frac{k-1}{n-1}$ if 0 is selected and $\frac{k}{n-1}$ if not, with total probability $\frac{k-1}{n-1} \frac{k}{n} + \frac{k}{n-1} \left(1 - \frac{k}{n}\right) = \frac{k}{n}$, etc. So each item is selected with probability $\frac{k}{n}$, and k items are selected because if $nSelected = k$, probability = 0 and, if probability = 1, all subsequent items are selected.

```
Vector<int> sortedSample(int k, int n)
{
    assert(k <= n && k > 0);
    Vector<int> result;
    for(int considered = 0, selected = 0; selected < k; ++considered)
        if(bernoulli(double(k - selected)/(n - considered)))
            //select
            result.append(considered);
            ++selected;
    }
    return result;
}
```

The runtime is $O(n)$, and the algorithm can produce samples online, though this isn't implemented here. It also efficiently produces combinations of integers $\in [0, n - 1]$ in sorted order, which is easy to randomize:

```
Vector<int> randomCombination(int k, int n)
{
    Vector<int> result = sortedSample(k, n);
    randomPermutation(result.getArray(), k);
    return result;
}
```

Reservoir sampling finds a sample of size k from a stream of unknown size in one pass with $O(1)$ work per item:

1. Select the first k items
2. For $n > k$, when the n^{th} item arrives, let $r = \text{random number \% } n$
3. If $r < k$, replace the r^{th} item with the new one

```
template<typename ITEM> struct ReservoirSampler
{
    int k, nProcessed;
    Vector<ITEM> selected;
    void processItem(ITEM const& item)
    {
        ++nProcessed;
        if(selected.getSize() < k) append(item); //select first k
        else
            //replace random
            int kickedOut = GlobalRNG().mod(nProcessed);
            if(kickedOut < k) selected[kickedOut] = item;
    }
}
ReservoirSampler(int wantedSize): k(wantedSize), nProcessed(0) {}
```

Sorting (or multiple-selecting; see the "Sorting" chapter) n random variates from a given distribution gives **order statistics**. For only i^{th} order statistic out of n , it's usually more efficient to generate a uniform i^{th} (starting from 1) order statistic, $\sim \text{beta}(i, n - i + 1)$ (Devroye 1986), and apply the inverse method to it.

```
double uniformOrderStatistic(int i, int n) {return beta(i, n - i + 1);}
```

6.14 Generating Samples from Multidimensional Distributions

Accept/reject, transformations between distributions, and composition work for $D > 1$, but the inverse method doesn't. Accept/reject is ineffective for large D , and sometimes have no good alternatives at all. Some distributions have efficient specific generators. MCMC methods (more advanced; discussed in the "Computational Statistics" chapter) may be more efficient.

An example of accept/reject is generating a point in a unit circle: generate a point $p \in (-1, 1)^2$ until dis-

tance($p, (0, 0)$) ≤ 1 :

```

pair<double, double> pointInUnitCircle()
{
    double x = uniform(-1, 1), y = uniform(-1, 1);
    while(x * x + y * y > 1)
        // regenerate if repeated
        x = uniform(-1, 1);
        y = uniform(-1, 1);
    }
    return make_pair(x, y);
}

```

For multidimensional normal with mean μ and covariance matrix Σ , generate vector V of $\text{normal}(0, 1)$ variates, and return $\mu + L \times V$, where L is computed by Cholesky factorization of Σ (see the “Numerical Algorithms—Introduction and Matrix Algebra” chapter). This works because the resulting variate has mean μ and variance $LL^T = \Sigma$.

```

class MultivariateNormal
{
    Vector<double> means;
    Matrix<double> L;
    static Matrix<double> makeL(Matrix<double> const& covariance)
    {
        Cholesky<double> c(covariance);
        assert(!c.failed); // covariance might be wrong or have numerical issues
        return c.L;
    }
public:
    MultivariateNormal(Vector<double> const& theMeans, Matrix<double> const& covariance): means(theMeans), L(makeL(covariance)) {}
    Vector<double> next() const
    {
        Vector<double> normals;
        for(int i = 0; i < means.getSize(); ++i)
            normals.append(GlobalRNG().normal01());
        return means + L * normals;
    }
};

```

Testing the result is also difficult because statistical distribution tests such as Kolmogorov-Smirnov (see the “Computational Statistics” chapter) only work for 1D.

6.15 The Monte Carlo Method

Can deduce much from calculating averages and standard deviations. First, some theory:

- The **law of large numbers (LLN)**: Given n iid samples x_i such that $E[x_i] = \mu$, for the **sample mean** $\bar{x} = \frac{\sum x_i}{n} \rightarrow \mu$ for $n \rightarrow \infty$.
- The **sample variance** $s^2 = \frac{\sum (x_i - \bar{x})^2}{n-1}$ → the variance σ^2 . Use $n-1$ instead of n to correct for bias; intuitively, the **plug-in estimator** $E[(x_i - \bar{x})^2]$ doesn't work because the x_i affect \bar{x} .
- The **central limit theorem (CLT)**: Given n iid samples y_i from a distribution with finite σ^2 , for $n \rightarrow \infty$, $\bar{x} \sim \text{normal}(\mu, \sigma^2/n)$.
- The LLN and **Slutsky's theorem** (see Wasserman 2004) allow using s^2 instead of σ^2 (to apply the theorem use the fact that $s^2/\sigma^2 \rightarrow 1$).
- So \bar{x} estimates μ with variance s^2/n . Don't confuse this with the variance of the data—this is variance of the average as estimator.

For the resulting error estimate of $2s/\sqrt{n}$ with 95% confidence need:

- $n \geq 30$. Otherwise the t -distribution (see the “Computational Statistics” chapter) gives a better approximation
- The average to converge to normal for that n —usually unpredictable

E.g., for a particular sample and the corresponding interval $\bar{x} \pm \epsilon$, a 95% confidence interval means 5%

chance of false enclosure, i.e., for some samples the calculated interval, which is fixed after the calculation, won't enclose μ . Get intervals of slightly different length and center with every simulation, but expect large differences only occasionally. See the "Computational Statistics" chapter for more discussion.

Monte Carlo applies the CLT to compute a quantity of interest μ if:

- \exists an event generator function f producing iid events with value y_i such that $E[y_i] = \mu$
- All important events are generated often enough to ensure that μ exists and n needn't be too large for the CLT to kick in (this is an intuitive and not a formal requirement)

1. Come up with μ and f

2. Until out of patience or the error small enough

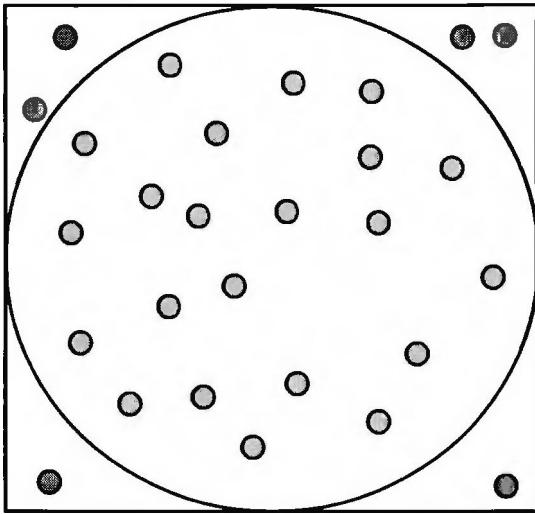
3. $y_i \leftarrow f(x_i)$

4. Incrementally update \bar{x} and s^2 with y_i

5. Return $\mu \leftarrow \bar{x}$, with 95% probability asymptotic error estimate $2s/\sqrt{n}$

Assuming event generation takes $O(1)$ time and space, a simulation needs $O(n)$ time and $O(1)$ space due to not needing to store the x_i . To get $|\text{error}| < \epsilon$, need $n = O\left(\frac{1}{\epsilon}\right)^2$. This is often too large, so use the available computation budget, and accept the resulting accuracy as the best available. Another strategy is to repeat the simulation with doubled n until the answer converges to the desired precision—then can even recalculate the final answer from all the simulations, and need to start from $n \geq 100$.

E.g., consider computing π . The area of a circle with radius r is πr^2 and of its enclosing square $4r^2$, so $\mu = \frac{\pi}{4} = \frac{\text{area}(\text{the circle})}{\text{area}(\text{the square})}$. Let f generate a random point $p \in [-1, 1]^2$ and return $y_i = (\text{distance}(p, (0, 0)) \leq 1)$. Both $y_i = 1$ and $y_i = 0$ should happen often, so small n is enough. After 10^8 particular variates $\pi = 3.14182 \pm 0.000493$. Other runs will give other estimates, but with equally valid error estimates.



$$\pi \approx 4 \times \frac{22}{22 + 6} \approx 3.142$$

Figure 6.13: Estimating π by the number of points in the circle/the number of points in the square

Some tasks for which Monte Carlo fails:

- Estimating μ of samples from a Cauchy distribution—it doesn't exist. The algorithm will return an arbitrary estimate, depending on the used samples.
- Estimating the average income of the county where a billionaire lives—the result is very different, depending on whether he/she is included in the sample, and meaningless in both cases. This is perhaps the best illustration of the "with 95% confidence"—every data set is different, and occasionally by much.

Can extend Monte Carlo in several ways:

- For the error calculation 95% confidence actually needs 1.96 multiplier, but use 2 anyway and get 95.45%. 95% is standard in statistics because higher is often deceptive and lower isn't enough. Given a multiplier, can evaluate the normal CDF to get the associated confidence (see the "Computational Statistics" chapter). This is based on asymptotic normality and is at best approximately accurate, so the exact % isn't important.
- A simulated event can produce k values. This effectively performs k related simulations at the cost of one. For simultaneous error estimates beware of multiple testing (see the "Computational Statistics" chapter).

tics" chapter).

At the end of the simulation, compute a normal distribution summary data structure to hold the mean and the variance of the mean. The normal distribution allows adding and subtracting independent normal distributions and scaling by a constant. Scaling by 2 is different from adding to itself because addition only applies to independent distributions. Can also maintain the minimum and the maximum incrementally, but beware the these don't estimate anything meaningful (see the "Computational Statistics" chapter; probably want a tolerance interval instead of these) and are computed only as diagnostics.

```
struct NormalSummary: public ArithmeticType<NormalSummary>
{
    double mean, variance;
    double stddev() const {return sqrt(variance);}
    double error95() const {return 2 * stddev();}
    explicit NormalSummary(double theMean = 0, double theVariance = 0):
        mean(theMean), variance(theVariance){assert(variance >= 0);}
    NormalSummary operator+=(NormalSummary const& b)
    { //for sum add means and variances
        mean += b.mean;
        variance += b.variance;
        return *this;
    }
    NormalSummary operator-=(NormalSummary const& b)
    { //for difference subtract means but add variances
        mean -= b.mean;
        variance += b.variance;
        return *this;
    }
    NormalSummary operator*=(double a)
    { //scale both mean and variance
        mean *= a;
        variance *= a * a;
        return *this;
    }
};
```

The calculation uses $s^2 = \frac{\sum y_i^2 - \sum y_i^2/n}{n-1}$, which is less numerically stable due to possible cancellation in

the subtraction (see the "Numerical Algorithms—Introduction and Matrix Algebra" chapter). But it's simple and allows incrementally updating $\sum y_i$ and $\sum y_i^2$. Cancellation isn't a problem with double-word precision unless the error is already extremely small relative to the numerator quantities. So it's safe to assume that the error is 0 in case of a negative numerator.

```
struct IncrementalStatistics
{
    double sum, squaredSum, minimum, maximum;
    long long n;
    IncrementalStatistics(): n(0), sum(0), squaredSum(0),
        minimum(numeric_limits<double>::infinity()), maximum(-minimum) {}
    double getMean() const {return sum/n;}
    double getVariance() const {return n < 2 ? 0 :
        max(0.0, (squaredSum - sum * getMean())/(n - 1.0));}
    double stdev() const {return sqrt(getVariance());}
    void addValue(double x)
    { //update incremental variables
        ++n;
        maximum = max(maximum, x);
        minimum = min(minimum, x);
        sum += x;
        squaredSum += x * x;
    }
    NormalSummary getStandardErrorSummary() const
    {return NormalSummary(getMean(), getVariance()/n);}
};
```

template<typename FUNCTION> IncrementalStatistics MonteCarloSimulate(

```

FUNCTION const& f, long long nSimulations = 1000)
{
    IncrementalStatistics s;
    for(long long i = 0; i < nSimulations; ++i) s.addValue(f());
    return s;
}

```

A **tail inequality (Mill's inequality)** shows that the probability of very large error in the estimate of the mean is exponentially small: Let $z = \frac{x - \bar{x}}{s}$; then $\Pr(Z > z) < \frac{2}{\sqrt{\pi}} \frac{f(z)}{z}$, where f is the normal PDF (Wasserman 2004). This applies to simulated data only asymptotically though.

Can use Monte Carlo to find $E[\text{the resource use}]$ of a randomized algorithm. For speed testing use:

```

template<typename FUNCTION> struct SpeedTester
{
    FUNCTION f;
    SpeedTester(FUNCTION const& theFunction = FUNCTION()): f(theFunction) {}
    double operator()() const
    {
        int now = clock();
        f();
        return (clock() - now) * 1.0/CLOCKS_PER_SEC;
    }
};

```

Can use Monte Carlo to compare average performances of randomized algorithms. Because $\text{normal}(a, b) - \text{normal}(c, d) = \text{normal}(a - c, b + d)$, can conclude with 95% probability that $a > c$ if $0 \notin [a - c \pm 2\sqrt{b + d}]$. But to avoid multiple testing this must be the only conclusion—i.e., can't report this interval and the ones for the averages.

Some problems with Monte Carlo:

- Events may be correlated or form correlated sequences. Then can average correlated subsequences, and assume that the resulting **batch averages** are independent.
- Convergence is too slow because the error bound is $O(1/\sqrt{n})$. When x_i come from a high-dimensional space, CLT seems to overcome the curse of dimensionality in that the convergence rate doesn't depend on the dimension. But the variance of the y_i can depend on the dimension of the x_i , and affect the constant behind the "O".
- Very rare events need arbitrarily large n for the CLT to kick in. E.g., an event that needs a fair coin be tails 100 times in row won't happen.

Variance reduction techniques somewhat speed up the CLT convergence by performing a related simulation but with less variance. **Common random numbers** fixes everything that isn't simulated. E.g., when comparing performances of randomized algorithms, run all simulations on a fixed set of inputs instead of generating a new one for each run. To test on various inputs use each several times. The technique seems to be useful only for such paired comparisons—inappropriate use can introduce bias due to nonrandomness, possibly changing the conclusions.

To simulate a request-processing system, use a priority queue (see the “Priority Queues” chapter) of events scheduled for some absolute or relative times. Until it's empty, dequeue an event, execute it, and possibly enqueue some others.

An interview question that I invented and asked many times is how to find the approximate area of a collection of possibly overlapping circles specified by centers and radii, given unlimited but finite computational resources. E.g., it's hopeless to try to find formulas for intersections of several circles in a “flower” case. But Monte Carlo works:

1. Calculate a bounding box for the circles
2. Write a function to decide if a point is inside a circle (and avoid the square root in the distance)
3. Use the $O(1/\sqrt{n})$ bound to estimate the number of samples n , given some approximate target precision such as 0.001
4. Generate n points inside the box, and for each decide if it's inside any circle
5. Return the proportion of points that landed on a circle \times the box's area

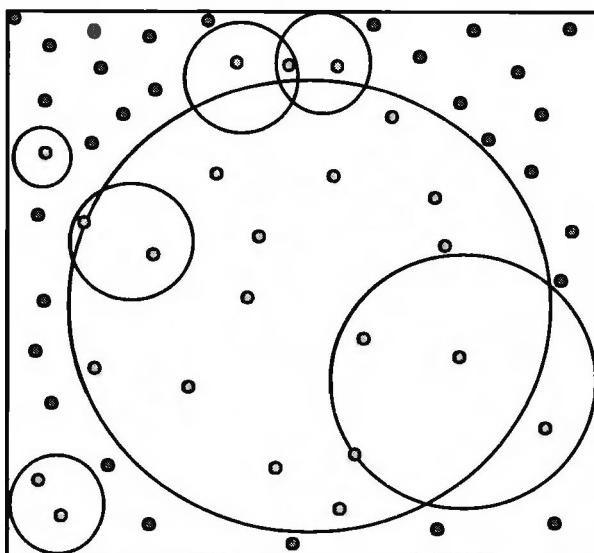


Figure 6.14: Monte Carlo solution to the circle area problem

A slightly more difficult but fundamentally similar technique is to count pixels in an artificial image, which is the hint I give to those not familiar with Monte Carlo. Here is the simulation solution coded in Python:

```

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
class Circle:
    def __init__(self, p, r):
        self.p = p
        self.r = r
import math
def distance(p1, p2):
    return math.sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)
def isInCircle(p, circles):
    for c in circles:
        if distance(p, c.p) <= c.r:
            return True
    return False
def findBoundingBox(circles):
    inf = float("inf")#no math.inf in my python
    left = inf
    right = -inf
    down = inf
    up = -inf
    for c in circles:
        if c.p.x - c.r < left:
            left = c.p.x - c.r
        if c.p.x + c.r > right:
            right = c.p.x + c.r
        if c.p.y - c.r < down:
            down = c.p.y - c.r
        if c.p.y + c.r > up:
            up = c.p.y + c.r
    return [Point(left, up), Point(right, down)]
import random
def findCoveredArea(circles, nSimulations):
    box = findBoundingBox(circles)
    left = box[0].x
    right = box[1].x
    down = box[1].y
    up = box[0].y
    inCount = 0
    for i in range(0, nSimulations):
        p = Point(random.uniform(left, right), random.uniform(down, up))
        if isInCircle(p, circles):
            inCount += 1
    return inCount/nSimulations

```

```

if isInCircle(Point(random.uniform(left, right),
    random.uniform(down, up)), circles):
    inCount += 1
return float(inCount)/nSimulations * (up - down) * (right - left)
print findCoveredArea([Circle(Point(0, 0), 1)], 1000000) #expect Pi

```

6.16 Implementation Notes

Perhaps too many generators have been presented. But all have some flaws.

For distribution generation the question is what to include—some of the distributions presented here aren't used often except for special purposes. Many others have been excluded. Only the simplest algorithms have been picked.

The sum heap implementation took the most time because of all the research to find at least a single source for the rather obvious idea.

The main decision for basic simulation functionality is what to include. So focusing on getting an estimate with a normal-based measure of accuracy seems to be a good choice. A separate normal summary is designed to allow easy manipulation of the results. The minimum and the maximum statistics are easy to compute and potentially useful for debugging and not for statistical purposes. It wouldn't be wrong to remove that code.

6.17 Comments

If a binary source generates heads with probability $p \neq 0.5$, can get unbiased bits:

1. Flip twice until get HT or TH
2. If HT return H
3. Else return T

This way heads and tails occur with equal probability. $E[\text{number of flips}] = \frac{1}{p(1-p)}$, and though technically have no finite worst-case bound, large values are exponentially unlikely. But this method is just an intellectual curiosity—properly random or pseudo-random numbers are easy to get.

The idea of a sum heap with ternary node structure is original (to the best of my knowledge). E.g., the tree of Wong & Easton (1980) has a different structure.

The CLT generalizes to vector-function outputs, in which case have a limiting correlation matrix instead of a variance, but this is less useful.

Other methods for variance reduction, including **antithetic variates**, **control variates**, and **importance sampling**, aren't useful for black-box inference because they need some knowledge about the system and give only $O(1)$ error reduction, while simulation is most useful when know nothing. But importance sampling can be useful for rare event sampling where have analytical knowledge.

6.18 Projects

- Improve 32-bit Xorshift to transform the output using an LCG with multiplier 1099087573 from L'Ecuyer & Simard (2007). This should make it a better hash function.
- Extend the normal generator to return a pair of variates. Make a convenience wrapper around it to select one at a time.
- Change the geometric generator to run in $O(1)$ time using a special formula from, e.g., Kroese et al. (2011). Does the number of bits in the generator output limits the useful range of the variates, unlike for the presented brute-force implementation?
- Research numerically stable formulas for incrementally calculating an average.
- Does it makes sense to time-out potential “forever” generators such as the one for a point in a circle to avoid infinite loops? One strategy is throwing an exception after some unreasonably larger number of tries.
- Convert the Python solution of the circle area problem to C++. Implement the pixel-counting solution, and compare the two. If you are interviewing, ask this question several times, and note how candidates approach it. Do they need an initial hint to go with one approach or the other? Which one is usually easier to get right? Think of some other things to observe.

6.19 References

- Devroye, L. (1986). *Non-uniform Random Variate Generation*. Springer.
- Kroese, D. P., Taimre, T., & Botev, Z. I. (2011). *Handbook of Monte Carlo Methods*. Wiley.
- L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47: 159–164.
- L'Ecuyer, P. & Simard, R. (2007). TestU01: a C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS)*, 33(4), 22.
- L'Ecuyer, P., Simard, R., Chen, E. J., & Kelton, W. D. (2002). An object-oriented random number package with many long streams and substreams. *Operations Research*, 50(6), 1073–1075.
- Marsaglia, G., & Tsang, W. W. (2000). A simple method for generating gamma variables. *ACM Transactions on Mathematical Software (TOMS)*, 26(3), 363–372.
- Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1), 3-30.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press.
- Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer.
- Wikipedia (2013). RC4. <http://en.wikipedia.org/wiki/RC4>. Accessed May 12, 2013.
- Wikipedia (2015). Triangular distribution. https://en.wikipedia.org/wiki/Triangular_distribution. Accessed November 1, 2015.
- Wikipedia (2017). Lévy distribution. https://en.wikipedia.org/wiki/L%C3%A9vy_distribution. Accessed December 10, 2017.
- Wong, C. K., & Easton, M. C. (1980). An efficient method for weighted sampling without replacement. *SIAM Journal on Computing*, 9(1), 111-113.

7 Sorting

7.1 Introduction

Sorting is well-discussed in algorithms classes, and you should almost always use the functionality of the standard library, even if a special, more efficient algorithm applies. The emphasis of this chapter is on the implementation details of the most useful algorithms.

Some theory:

- A collection of items satisfying a **weak order relation** " $<$ " is sorted. Beware that " \leq " isn't weak.
- For n items $\exists n!$ orders, and k binary comparisons decide between $\geq 2^k$ of them, so to sort using only comparisons need $k = \Theta(n \lg(n))$.
- Sorting is **stable** if equal items keep their original relative order. Using item location as a **secondary key** ensures stability but needs more memory and is clumsy for the caller
- Item sorting analyses usually assume $O(1)$ comparisons. For expensive-to-copy items, sort an array of pointers to them for efficiency.

7.2 Insertion Sort

For small arrays insertion sort is stable and the fastest, so it makes a good helper function for some smarter algorithms. It mimics sorting a hand of cards. Given a sorted array, initially with the first item, iteratively insert the next item into the correct place.

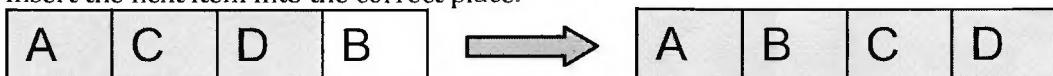


Figure 7.1: The next unsorted item logic of insertion sort

```
template<typename ITEM, typename COMPARATOR>
void insertionSort(ITEM* vector, int left, int right, COMPARATOR const& c)
{//allow more general left != 0
    for(int i = left + 1; i <= right; ++i)
    {
        ITEM e = vector[i];
        int j = i;
        for(;j > left && c(e, vector[j - 1]); --j) vector[j] = vector[j - 1];
        vector[j] = e;
    }
}
```

The runtime is $O(n^2)$ with very low constant factors, and $O(\text{the number of reversed pairs called inversions}) = O(n)$ for almost sorted input.

7.3 Quicksort

If don't need stability, quicksort is the fastest and almost universally the "sort" algorithm of any good API. The basic version:

1. **Pick a pivot item**
2. **Partition the array so that items $\leq \geq$ the pivot are respectively on the left/right**
3. **Sort the two halves recursively**

The item order within each subarray after a partition doesn't matter.

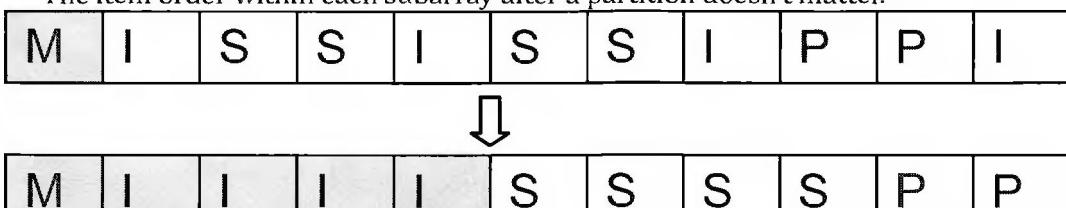


Figure 7.2: A possible result of quicksort pivoting with pivot 'M'

The most practical pivot selection rule is the **median of three** random items:

- Deterministic picks can give $O(n^2)$ runtime
- A single random pivot is slightly slower

- Using five or more is negligibly faster but more complex

```
template<typename ITEM, typename COMPARATOR>
int pickPivot(ITEM* vector, int left, int right, COMPARATOR c)
{//OK if pivots are same by chance occasionally
    int i = GlobalRNG().inRange(left, right), j =
        GlobalRNG().inRange(left, right), k = GlobalRNG().inRange(left, right);
    if(c(vector[j], vector[i])) swap(i, j);
    //i <= j, decide where k goes
    return c(vector[k], vector[i]) ? i : c(vector[k], vector[j]) ? k : j;
}
```

Partitioning actually groups items into < pivot, = pivot, and > pivot, which is more useful (Sedgewick 1999). Unlike for a complete sort, the items in "<" and ">" sections will be in an arbitrary order. As an intermediate step, move equal items to the sides:

=	<	?	>	=
---	---	---	---	---

Figure 7.3: Quicksort partitioning work strategy

Use a left and a right pointer for a bidirectional scan of the array. If a scanned item doesn't belong to corresponding "<" or ">" section, mark it for swapping. At every iteration the pointers stop at such items. Stop when the pointers cross. Then swap the side "=" sections to the middle.

```
template<typename ITEM, typename COMPARATOR> void partition3(ITEM* vector,
    int left, int right, int i, int j, COMPARATOR const& c)
{//i/j are the current left/right pointers
    ITEM p = vector[pickPivot(vector, left, right, c)];
    int lastLeftEqual = i = left - 1, firstRightEqual = j = right + 1;
    for(;;)//the pivot is the sentinel for the first pass
    {//after one swap swapped items act as sentinels
        while(c(vector[++i], p));
        while(c(p, vector[--j]));
        if(i >= j) break;//pointers crossed
        swap(vector[i], vector[j]);//both pointers found swappable items
        //swap equal items to the sides
        if(c isEqual(vector[i], p))//i to the left
            swap(vector[++lastLeftEqual], vector[i]);
        if(c isEqual(vector[j], p))//j to the right
            swap(vector[--firstRightEqual], vector[j]);
    }
    //invariant: i == j if they stop at an item = pivot
    //and this can happen at both left and right item
    //or they cross over and i = j + 1
    if(i == j){++i; --j;}//don't touch pivot in the middle
    //swap side items to the middle; left with "<" section and right with ">"
    for(int k = left; k <= lastLeftEqual; ++k) swap(vector[k], vector[j--]);
    for(int k = right; k >= firstRightEqual; --k)
        swap(vector[k], vector[i++]);
}
```

Either i or j may be out of bounds. The postcondition:

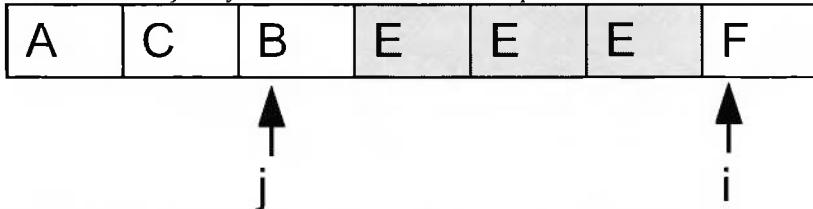


Figure 7.4: Quicksort pointers after a three-partition

Must be very careful with the code because it's easy to get it wrong, particularly with sentinels for the while loops. The slightly less complicated, basic " \leq/\geq " partitioning pays no special attention to equal items and is obtained by just removing the code for swapping to the sides and back. The $i-j$ invariant remains the same, but it defines the postcondition. Three-partitioning also applies to vector sorting and is faster for arrays with many equal items. With few extra items its equality checks and swaps are also few.

Optimizations for the main algorithm:

- Sorting smaller subarrays first ensures $O(\lg(n))$ extra memory, which practically guarantees that the recursion stack won't run out.
- Use insertion sort for small subarrays of size 16 (per studies 5–25 works well). Due to caching, recursing to insertion sort is faster than a single insertion sort over the whole array in the end, despite using more instructions (Mehlhorn et al. 2019).
- Remove the tail recursion. Removing the other one complicates the algorithm with little gain.

```
template<typename ITEM, typename COMPARATOR>
void quickSort(ITEM* vector, int left, int right, COMPARATOR const& c)
{
    while(right - left > 16)
    {
        int i, j;
        partition3(vector, left, right, i, j, c);
        if(j - left < right - i) //pick smaller
        {
            quickSort(vector, left, j, c);
            left = i;
        }
        else
        {
            quickSort(vector, i, right, c);
            right = j;
        }
    }
    insertionSort(vector, left, right, c);
}
template<typename ITEM> void quickSort(ITEM* vector, int n)
{quickSort(vector, 0, n - 1, DefaultComparator<ITEM>());}
```

With basic partitioning $E[\text{the runtime}] = O(n\lg(n))$. Proof (Cormen et al. 2009): Suppose the pivot is random, and all items are unique. Let X_{ij} be the number of times that the item at i was compared to the item at j in the sorted array with $j > i$. $E[X_{ij}] = \Pr(i \text{ or } j \text{ was a pivot})$ because i and j were compared at most once and only if one of them was a pivot in a subarray containing the other. Else, if an item at $>j$ or $<i$ was a pivot, i and j go into the same subarray, else into separate ones. Because $\exists j - i + 1$ separating pivots, $\Pr(i \text{ or } j \text{ was a pivot}) = \frac{2}{j - i + 1}$. So the $E[\text{the total number of comparisons}] = E\left(\sum_{0 \leq i < n} \sum_{i+1 \leq j < n} X_{ij}\right) < 2 \sum_{0 \leq i < n} \sum_{1 \leq k < n} \frac{1}{k} < 2n\lg(n)$. \square

The unlikely worst case is $O(n^2)$ if bad pivots are picked every time. The same analysis extends to the three-partitioning because the only difference is in the equality comparisons, which are a nondominant operation.

7.4 Mergesort

Mergesort is the most efficient stable sort:

1. Split the array in half
2. Mergesort each recursively
3. Merge the halves in $O(n)$ time

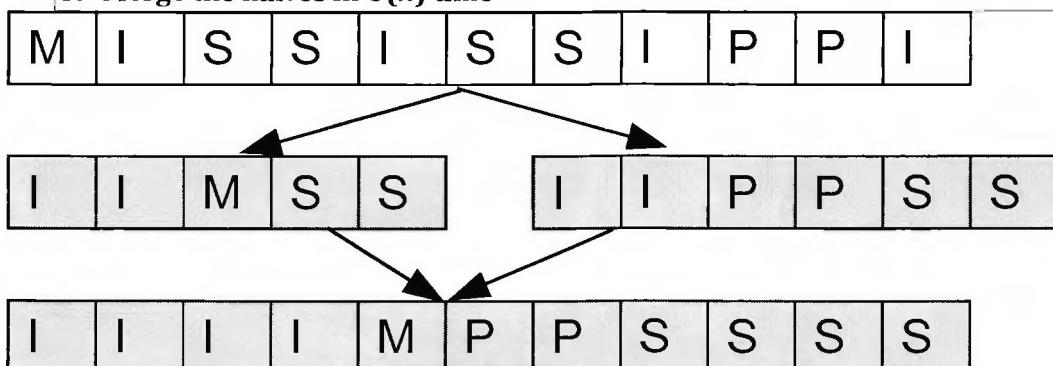


Figure 7.5: Mergesort splitting and merging

The runtime satisfies $R(n) = O(n) + 2R(n/2)$, so by the master theorem $R(n) = O(n\lg(n))$. Optimizations for the main algorithm:

- Alternate the data and the temporary storage arrays to avoid unnecessary copies
 - Use insertion sort for arrays of size ≤ 16 as for quicksort

Merging is most of the work. It iteratively moves the smallest leftmost item of both arrays to the result array. The rightmost index of the left array is `middle`. The recursive call and the merge assume that for a range $[left, right]$ the items are in the temporary storage array, so that the original $[left, right]$ slice is overwritten with the sorted result.

```

template<typename ITEM, typename COMPARATOR> void merge(ITEM* vector,
    int left, int middle, int right, COMPARATOR const& c, ITEM* storage)
{ // i for the left half, j for the right, merge until fill up vector
    for(int i = left, j = middle + 1; left <= right; ++left)
        { // either i or j can get out of bounds
            bool useRight = i > middle || (j <= right &&
                c(storage[j], storage[i]));
            vector[left] = storage[(useRight ? j : i)++];
        }
}
template<typename ITEM, typename COMPARATOR> void mergeSortHelper(
    ITEM* vector, int left, int right, COMPARATOR const& c, ITEM* storage)
{
    if(right - left > 16)
        { // sort storage using vector as storage, then merge into vector
            int middle = (right + left)/2;
            mergeSortHelper(storage, left, middle, c, vector);
            mergeSortHelper(storage, middle + 1, right, c, vector);
            merge(vector, left, middle, right, c, storage);
        }
    else insertionSort(vector, left, right, c);
}
template<typename ITEM, typename COMPARATOR>
void mergeSort(ITEM* vector, int n, COMPARATOR const& c)
{ // copy vector to storage first
    if(n <= 1) return;
    Vector<ITEM> storage(n, vector[0]); // reserve space for n with 1st item
    for(int i = 1; i < n; ++i) storage[i] = vector[i];
    mergeSortHelper(vector, 0, n - 1, c, storage.getArray());
}

```

The use case relies on stability because would use quicksort otherwise, e.g.:

- When sort a table by a column, can sort on multiple columns implicitly if use a stable sort, with other columns as nonprimary key
 - In some algorithms may want to lose dependence on the randomness of quicksort if have secondary keys that influence the overall computation

Perhaps mergesort is mostly useful for illustrating its work pattern, which is a general divide-and-conquer strategy.

7.5 Integer Sorting

Can sort integers in $O(n)$ time by not using " $<$ ". For integers mod N , **counting sort** counts how many times each occurs and creates a sorted array from the counts in $O(n + N)$ time. It's stable.

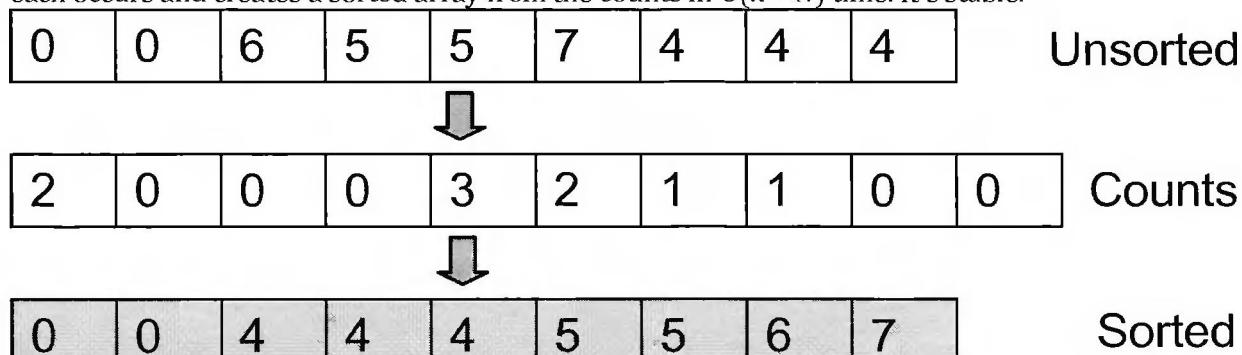


Figure 7.6: Counting sort counting and sorted array creation for $N = 10$

```

void countingSort(int* vector, int n, int N)
{
    Vector<int> counter(N, 0);
    for(int i = 0; i < n; ++i) ++counter[vector[i]]; //count
    for(int i = 0, index = 0; i < N; ++i) //create in order
        while(counter[i]-- > 0) vector[index++] = i;
}

```

For items with integer-mod- N keys, **key-indexed counting sort (KSort)** works similarly but needs temporary storage because can't create items from counts. Use cumulative counts because they tell how many smaller-value items precede the one with a particular key. Nothing precedes the smallest item, so create a sentinel for 0. As items are placed, precedence counts are incremented to move to the next location. So KSort:

1. Counts how many items have a particular key
2. Uses the cumulative counts to create a temporary sorted array
3. Copies it into the original

The implementation needs a functor `ORDERED_HASH` that extracts items' keys—e.g., it can get byte 0 from an integer. Beware that such functors' behavior depends on endianness.

```

template<typename ITEM, typename ORDERED_HASH> void KSort(ITEM* a, int n,
    int N, ORDERED_HASH const& h)
{
    ITEM* temp = rawMemory<ITEM>(n);
    Vector<int> prec(N + 1, 0);
    for(int i = 0; i < n; ++i) ++prec[h(a[i]) + 1];
    for(int i = 0; i < N; ++i) prec[i + 1] += prec[i]; //accumulate counts
    //rearrange items
    for(int i = 0; i < n; ++i) new(&temp[prec[h(a[i])]++]) ITEM(a[i]);
    for(int i = 0; i < n; ++i) a[i] = temp[i];
    rawDelete(temp);
}

```

It's stable and takes $O(n + N)$ time.

7.6 Vector Sorting

Sorting n vectors of size k as items takes $O(knlg(n))$ time. But can sort as vectors, which makes sense for indirect access where sort an index array to avoid copying vectors. For quicksort on n random vectors of length ∞ , $E[\text{the runtime}] = O(nlg(n)^2)$ (Vallee et al. 2009). This is intuitive because for two random sequences in a collection of n with items from an alphabet of size A , $E[\text{lcp (least common prefix)}] = \log(n)$, requiring that time for a comparison (see the "String Algorithms" chapter for more on lcp).

Multikey quicksort three-partitions on the first letter and recurses on each subarray, going to the next letter for the equal part:

Sort Left on 0			Sort Middle on 1			Sort right on 0	
A	C	B	E	E	E	F	
D	A	A	A	E	A	O	
T	T	R	L	R	R		
		L				D	

Figure 7.7: Multikey quicksort recursive three-partitioning on consecutive characters of words

The user-provided comparator, such as the one below for vectors, keeps track of the current depth, starting with 0, which allows sorting arbitrary tuples. For code reuse it assumes a vector accessor and an item comparator. Also have an indexed accessor.

```

template<typename VECTOR, typename ITEM> struct DefaultVectorAccessor
//for sorting vectors
ITEM const& getItem(VECTOR const& v, int i) const{return v[i];}

```

```

int getSize(VECTOR const& v) const{return v.getSize();}
};

struct StringAccessor
{//for sorting strings
    char getItem(string const& s, int i) const{return s[i];}
    int getSize(string const& s) const{return s.length();}
};
template<typename VECTOR, typename ITEM, typename ACCESSOR =
DefaultVectorAccessor<VECTOR, ITEM>> struct IndexedAccessor
{
    VECTOR const*const v;
    ACCESSOR a;
    IndexedAccessor(VECTOR const& const theV, ACCESSOR const& theA = ACCESSOR())
        : v(theV), a(theA){}
    ITEM getItem(int i, int j) const{return a.getItem(v[i], j);}
    int getSize(int i) const{return a.getSize(v[i]);}
};

template<typename VECTOR, typename ITEM, typename ACCESSOR =
DefaultVectorAccessor<VECTOR, ITEM>, typename COMPARATOR =
DefaultComparator<ITEM>> struct MultikeyQuicksortVectorComparator
{
    ACCESSOR s;
    COMPARATOR c;
    mutable int depth;
    MultikeyQuicksortVectorComparator(ACCESSOR const& theS = ACCESSOR(),
        COMPARATOR const& theC = COMPARATOR()): s(theS), c(theC), depth(0){}
    bool operator()(VECTOR const& lhs, VECTOR const& rhs) const
    {
        return depth < s.getSize(lhs) ?
            depth < s.getSize(rhs) && c(s.getItem(lhs, depth),
                s.getItem(rhs, depth)) : depth < s.getSize(rhs);
    }
    bool isEqual(VECTOR const& lhs, VECTOR const& rhs) const
    {
        return depth < s.getSize(lhs) ?
            depth < s.getSize(rhs) && c.isEqual(s.getItem(lhs, depth),
                s.getItem(rhs, depth)) : depth >= s.getSize(rhs);
    }
};

```

Remove the recursion to not run out of stack for long vectors with high lcp. The algorithm is driven from a stack that contains a set of intervals and depths to process. Start it with $(left, right, 0)$. No longer:

- Use insertion sort for small subarrays due to its inability to work efficiently with vectors
- Process the smaller subarray first—the equal is the longest in the worst case, and don't worry about the memory use of recursion—so process the middle piece last

```

template<typename VECTOR, typename COMPARATOR> void multikeyQuicksortNR(
    VECTOR* vector, int left, int right, COMPARATOR const& c,
    int maxDepth = numeric_limits<int>::max()
)
{
    Stack<int> stack;
    stack.push(left);
    stack.push(right);
    stack.push(0);
    while(!stack.isEmpty())
    {
        c.depth = stack.pop();
        right = stack.pop();
        left = stack.pop();
        if(right - left > 0 && c.depth < maxDepth)
        {
            int i, j;
            partition3(vector, left, right, i, j, comparator);
            //left
        }
    }
}

```

```

    stack.push(left);
    stack.push(j);
    stack.push(c.depth);
    //right
    stack.push(i);
    stack.push(right);
    stack.push(c.depth);
    //middle
    stack.push(j + 1);
    stack.push(i - 1);
    stack.push(c.depth + 1);
}
}

```

$E[\text{the runtime}] = O(n \lg(n))$, and the expected runtime with respect to the length is the optimal $O(n(\text{the length} + \lg(n)))$ (Sedgewick 1999). The unlikely worst case is $O(n(\text{length} + n))$.

For vectors of small integers of fixed length k , **LSD sort** is stable and the most efficient. It sorts k times using `vector[k - i]` as the key to KSort in pass i , with the overall runtime $O(nk)$. This works because KSort is stable. Because the use case is so specific, no implementation is provided, and it's up to the caller to setup the KSort calls—e.g., see suffix array construction in the “String Algorithms” chapter. Note that it's wrong to use LSD to sort integers one byte at a time because of endianness unless take special case to access the correct bytes in all cases.

7.7 Permutation Sort

To sort an array a according to a permutation p defined by an array of sorted indices, can copy the items to a temporary array and populate the original from it according to p .

But can avoid the temporary (Flamig 1995). Any permutation is composed of independent subpermutations (i.e., a **product of disjoint cycles** in abstract algebra terminology). Think of $f = p[i]$ as specifying from where to take the item for position i . A loop over the array gets all cycles in $O(n)$ time. Start an unprocessed cycle if $p[i] \neq i$.

1. Remember the first item in the cycle, and set $to =$ its index i
 2. While $p[to] \neq to$
 3. $from = p[to]$
 4. $a[to] = a[from]$
 5. Mark to processed with $p[to] = to$
 6. $to = from$
 7. End cycle with $a[to] =$ the first item

E.g., consider the permutation 3210 applied to $abcd$. Start with 0. Remember $v[0] = a$, from position $p[0] = 3$ take d , and put it into position 0. Check $p[3] = 0$ to discover the end of cycle due to $p[0] = 0$, and put stored a into position 3. Move to position 1. The same logic swaps b and c . After this, all positions are marked identity, and moving to 2 and 3 changes nothing. Beware that the code always resets the permutation to identity, so the caller must copy it if the goal is to apply this algorithm to multiple arrays.

```

template<typename ITEM> void permutationSort(ITEM* a, int* permutation, int n)
{//need permutation to be valid, else get an infinite loop
    for(int i = 0; i < n; ++i) if(permutation[i] != i)
        {//start cycle
            ITEM temp = a[i];
            int to = i;
            do
            {
                a[to] = a[permutation[to]];//put element in right place
                swap(permutation[to], to);//mark to done, and advance cycle
            }while(permutation[to] != i);//until find what goes to i
            a[to] = temp;//complete cycle
            permutation[to] = to;//becomes identity
        }
}

```

The use case is rare though because can access the sorted order through the permutation directly, for either in-order iteration or binary search.

7.8 Selection

Want to arrange items so that the specified item is in the correct place, e.g., to find the median. **Quickselect** is like quicksort but doesn't sort the subarray that can't contain the item. So it avoids one recursive call and can be implemented iteratively.

```
template<typename ITEM, typename COMPARATOR> ITEM quickSelect(ITEM* vector,
    int left, int right, int k, COMPARATOR c)
{
    assert(k >= left && k <= right);
    for(int i, j; left < right;)
    {
        partition3(vector, left, right, i, j, c);
        if(k >= i) left = i;
        else if(k <= j) right = j;
        else break;
    }
    return vector[k];
}
```

$E[\text{the runtime}] = O(n)$. The unlikely worst case is $O(n^2)$. For random vector items somewhat unintuitively $E[\text{the runtime}] = O(n)$ (Vallee et al. 2009), but can extend multikey quicksort to **multikey quickselect**. Same for multiple select (discussed later in the chapter; neither extension is implemented).

```
template<typename VECTOR, typename COMPARATOR> void multikeyQuickselect(
    VECTOR* vector, int left, int right, int k, COMPARATOR const& c)
{
    assert(k >= left && k <= right);
    for(int d = 0, i, j; right - left >= 1;)
    {
        partition3(vector, left, right, i, j, c);
        if(k <= j) right = j;
        else if (k < i) // equal case j < k < i
        {
            left = j + 1;
            right = i - 1;
            ++c.depth;
        }
        else left = i;
    }
}
```

7.9 Multiple Selection

To sort only the first k items, an optimal $O(n + k \lg(k))$ solution is to run quicksort($0, k - 1$) on the result of quickselect(k).

A more general problem is to output an array with only k specified items in correct places. Specify them with a Boolean array, and have quicksort not recurse into subarrays without any selected items. E.g., can compute statistical quantiles this way.

```
template<typename ITEM, typename COMPARATOR> void multipleQuickSelect(ITEM*
    vector, bool* selected, int left, int right, COMPARATOR const& c)
{
    while(right - left > 16)
    {
        int i, j;
        for(i = left; i <= right && !selected[i]; ++i);
        if(i == right + 1) return; // none are selected
        partition3(vector, left, right, i, j, c);
        if(j - left < right - i) // smaller first
        {
            multipleQuickSelect(vector, selected, left, j, c);
            left = i;
        }
        else
        {

```

```

        multipleQuickSelect(vector, selected, i, right, c);
        right = j;
    }
}
insertionSort(vector, left, right, c);
}

```

✗ selection E[the runtime] is optimal but depends on the number and the positions of the specified items (Kaligosi et al. 2005). The unlikely worst case is $O(n^2)$.

7.10 Searching

Sequential search is the fastest for few items despite the $O(n)$ runtime and the only choice if items aren't sorted. For sorted items **binary search** is worst-case optimal, taking $O(\lg(n))$ time. It starts in the middle, and if query \neq item, goes left if query $<$ item and right otherwise.

```

template<typename ITEM, typename COMPARATOR> int binarySearch(ITEM const*
vector, int left, int right, ITEM const& key, COMPARATOR const& c)
{
    while(left <= right)
    { //careful to avoid overflow in middle calculation
        int middle = left + (right - left)/2;
        if(c isEqual(key, vector[middle])) return middle;
        c(key, vector[middle]) ? right = middle - 1 : left = middle + 1;
    }
    return -1; //not found
}

```

The overflow avoidance in the calculation of the middle may seem silly, but due to virtual memory large arrays are possible. Though this use case is unlikely, in general it's best to protect against issues and not think about it again.

Before using **binary search** need prior knowledge of sorting. It's easy to check in $O(n)$ time:

```

template<typename ITEM, typename COMPARATOR> bool isSorted(ITEM const*
vector, int left, int right, COMPARATOR const& c)
{
    for(int i = left + 1; i <= right; ++i)
        if(c(vector[i], vector[i - 1])) return false;
    return true;
}

```

Exponential search is useful when searching over an implicit unbounded "array". Assume that the bound 1, then 2, 4, 8, etc. After it's found, do binary search between bound/2 and bound. E.g., can guess a positive number that someone thinks of but discloses only comparison results of it to other numbers. The runtime is $O(\lg(\text{the wanted value}))$.

7.11 Implementation Notes

Despite many presented algorithms, my selection is actually very restricted compared to many other sources. The most useful sorting algorithms are quicksort and KSort—both are actually used in other parts of the book. Mergesort, counting sort, and permutation sort are useful for special cases, but it wouldn't be wrong to exclude them.

An interesting observation is that it takes much research to figure out how to implement even something basic such as quicksort—every book in the references had something to contribute toward the final implementation or analysis.

For the selection algorithms it's also strange that I had to go through primary literature to find good algorithms and analysis. For such a basic tasks books should be enough.

7.12 Comments

Unlike insertion sort, other $O(n^2)$ sorts aren't useful. E.g.:

- **Selection sort**—swap the minimum item with the first item, then repeat this for the rest of the array—makes the minimal possible number of item moves
- **Bubble sort**—exchange adjacent items until every item is in correct order—how people in a group sort themselves by height

Three-partitioning is a solution to the **Dutch national flag problem**, which is implicitly defined by it. The classic algorithm for it by Dijkstra uses fewer instructions and is a bit simpler but has higher constant factors with few equal items, which is often the case for sorting (Sedgewick 1999).

An interesting idea is using several pivots, resulting in particular in **dual-pivot quicksort**. It's slightly faster than regular quicksort but needs twice more code and is more complicated. The analysis is still ongoing, but currently the conclusion is that have fewer cache misses, which more than compensates for more instructions (Kushagra et al. 2013).

C++ STL uses quicksort with deterministic median-of-three pivot but switches to a slower, safer in-place **heapsort** (see the "Priority Queues" chapter) on reaching a high enough depth. Though this strategy ensures $O(n \lg(n))$ runtime, it too benefits from using random pivots. Due to $E[\text{the runtime}]$ guarantees, the switch seems unnecessary and doesn't generalize to other situations such as vector sorting. But must do it because the C++ standard requires worst-case performance guarantees, which is perhaps unduly restrictive but satisfiable. **Shellsort** is suboptimal but empirically slightly faster than heapsort (Sedgewick 1999); despite that it has no use case.

An interesting search algorithm for sorted numeric items is **interpolation search** (Wikipedia 2015). It's like binary search but, instead of using the average index, uses the index based on the item values. $E[\text{the runtime for items } \sim \text{uniform}] = O(\ln(\ln(n)))$, but the use case is limited, and any gain over binary search is negligible in practice.

To verify that an array is sorted can also do several binary searches for random elements. I haven't been able to find a reference for this with a probabilistic correctness guarantee.

An interesting problem is sorting a linked list. Because a list doesn't support random access, the only goal is traversing in sorted order. Can adapt mergesort to sort without using extra memory (Roura 1999).

7.13 Projects

- Add input range checks for counting sort and Ksort.
- Implement a stability test for algorithms that ensure it

7.14 References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Flamig, B. (1995). *Practical Algorithms in C++*. Wiley.
- Kalogi, K., Mehlhorn, K., Munro, J. I., & Sanders, P. (2005). Towards optimal multiple selection. In *Automata, Languages, and Programming* (pp. 103–114). Springer.
- Kushagra, S., López-Ortiz, A., Munro, J. I., & Qiao, A. (2013). Multi-pivot Quicksort: theory and experiments. In *Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM.
- Mehlhorn, K., & Sanders, P., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- Roura, S. (1999). Improving mergesort for linked lists. In *Algorithms-ESA'99* (pp. 267–276). Springer.
- Sedgewick, R. (1999). *Algorithms in C++, Parts 1–4*. Addison-Wesley.
- Vallée, B., Clément, J., Fill, J. A., & Flajolet, P. (2009). The number of symbol comparisons in Quicksort and Quickselect. In *Automata, Languages, and Programming* (pp. 750–763). Springer.
- Wikipedia (2015). Interpolation search. https://en.wikipedia.org/wiki/Interpolation_search. Accessed November 3, 2015.

8 Dynamic Sorted Sequences

8.1 Introduction

A typical algorithms class discusses balanced binary search trees such as a simple but obsolete AVL tree for an easy introduction. A more advanced presentation uses a red-black tree, but with limited discussion of its balancing. Also common is a discussion of a basic trie, but without important implementation details. This is confirmed by my interviewing experience though basic knowledge questions.

My approach is to focus on a more general concept of a dynamic sorted sequence, which isn't attached to a tree implementation. This allows to discuss a skip list as a useful data structure for some occasions. For a balanced tree I present a random treap, which is just as efficient but much simpler than a red-black tree. It's randomized balancing also applies to tries. Finally, a tree augmentation for vector keys that typically outperforms tries but isn't discussed by any existing books is presented.

8.2 Requirements

A **dynamic sorted sequence** maintains a collection of items in sorted order. For a key x it efficiently supports:

- **Map** operations **find**, **insert**, and **remove**.
- Max and min.
- **In-order iteration** between any two elements.
- **Predecessor** and **successor**, which respectively are the previous max item $< x$ and the next min item $> x$. Their **inclusive** versions use " \leq " and " \geq " respectively and always exist for an inserted key. Iteration and predecessor give find and successor. Combining these with iteration enables efficient **range search**.
- **Join** of two sequences such that keys in one $<$ keys in the other.
- **Split** of a sequence such that the first has items $\leq x$, and the second $> x$.

Various augmentations can support other operations, such as finding the k^{th} element. Join and split of two sequences are rarely useful and not efficiently supported by a free-list-based implementation because need to move items between free lists unless use the same one for all instances, which is dummy. Map operations, min, max, predecessor, and successor usually take the same worst case and/or expected $O(\lg(n))$ time, where $n =$ the number of currently maintained items. Iteration over all items needs $O(n)$ time and is faster than finding them one by one. Unlike for sorting, items with nonunique keys aren't supported by most data structures and must be handled by augmenting keys with tiebreak information.

8.3 Skip List

A **skip list** is a collection of linked lists in sorted order, where list i has each item with probability p^i for some constant p . The bottom list has all the items, and higher-level lists outline lower-level ones for efficient operations.

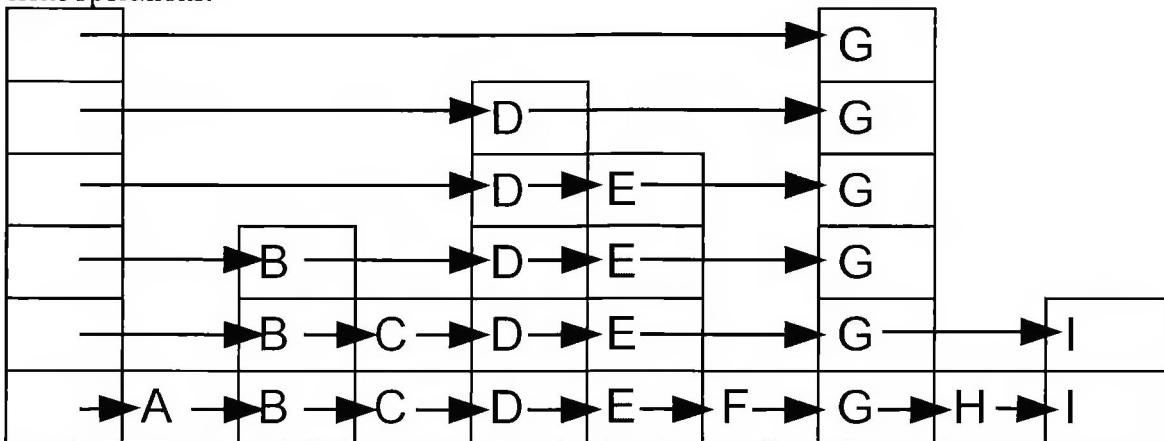


Figure 8.1: Skip list pointer structure

The number of levels $\approx \lg(n)$, and in practice $\lg(n) < 32$, so for implementation simplicity use 32 as the height bound. Most operations support items with nonunique keys.

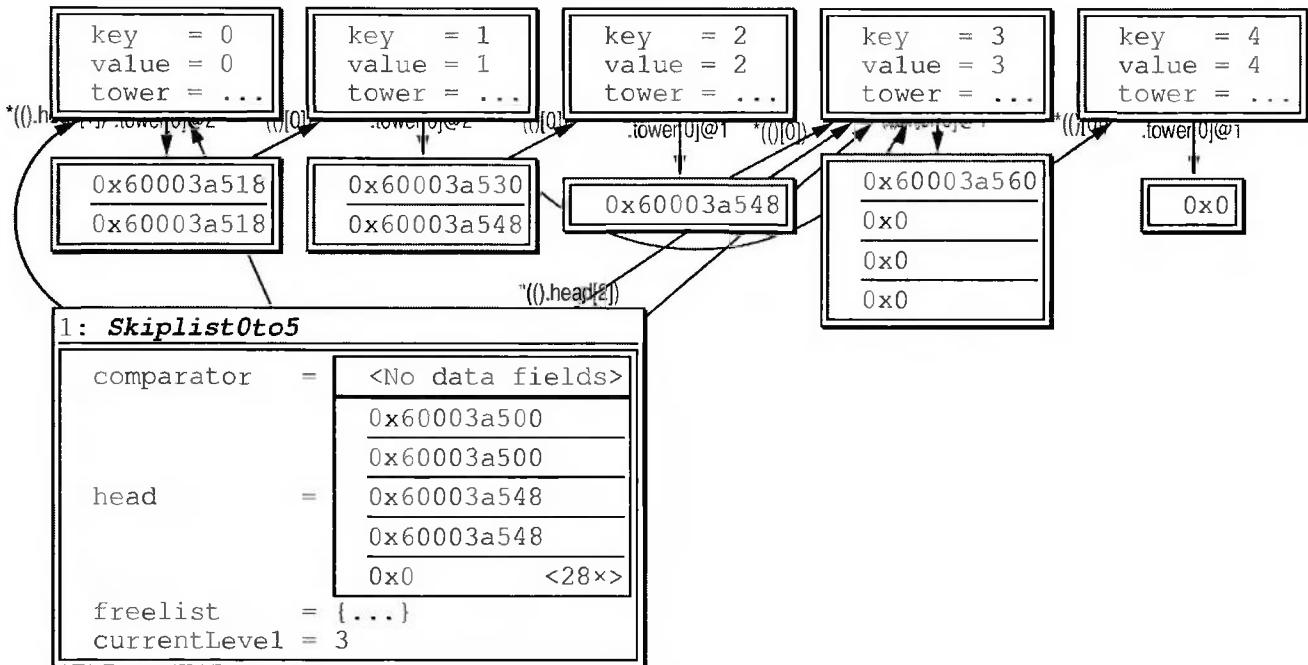


Figure 8.2: Skip list memory layout with integer items 0–4

```

template<typename KEY, typename VALUE, typename COMPARATOR =
DefaultComparator<VALUE>> class SkipList
{
    COMPARATOR c;
    enum{MAX_HEIGHT = 32};
    struct Node
    {
        KEY key;
        VALUE value;
        Node** tower;
        Node(KEY const& theKey, VALUE const& theValue, int height):
            key(theKey), value(theValue), tower(new Node*[height]) {}
        ~Node() {delete[] tower;}
    }* head[MAX_HEIGHT];
    Freelist<Node> f;
    int currentLevel;
public:
    SkipList(COMPARATOR const& theC = COMPARATOR()): currentLevel(0), c(theC)
    {for(int i = 0; i < MAX_HEIGHT; ++i) head[i] = 0;}
    SkipList(SkipList const& rhs): currentLevel(0), c(rhs.c)
    { //order of items with nonunique keys in copy is reversed
        for(int i = 0; i < MAX_HEIGHT; ++i) head[i] = 0;
        for(Node* node = rhs.head[0]; node; node = node->tower[0])
            insert(node->key, node->value, false);
    }
    SkipList& operator=(SkipList const&rhs){return genericAssign(*this, rhs);}
};
  
```

Search:

1. Start with the highest nonempty list
2. Follow the links until reach list 0, going one list down when the next item > query
3. Linearly search list 0

This allows efficient predecessor operation in terms of which some others are implemented. Instead of node pointers they return iterators for type safety.

```

Iterator predecessor(KEY const& key)
{
    Node **tower = head, *pred = 0;
    for(int level = currentLevel; level >= 0; --level)
        for(Node* node; (node = tower[level]) && c(node->key, key);
            tower = node->tower) pred = node;
  
```

```

    return Iterator(pred);
}
Iterator inclusiveSuccessor(KEY const& key)
{ //next(pred) = inc succ
    Iterator pred = predecessor(key);
    assert(pred == end() || c(pred->key, key));
    return pred == end() ? begin() : Iterator(pred->tower[0]);
}
Iterator findNode(KEY const& key)
{ //general pattern - return a pointer for reuse in other operations
    Iterator node = inclusiveSuccessor(key); //match if not larger than key
    assert(node == end() || !c(node->key, key));
    return node == end() || c(key, node->key) ? end() : node;
}
VALUE* find(KEY const& key)
{
    Iterator result = findNode(key);
    return result == end() ? 0 : &result->value;
}
Iterator successor(KEY const& key)
{ //is inclusiveSuccessor with nonunique keys, else loop over equal keys
    Node* node = inclusiveSuccessor(key)->current;
    while(node && !c(node->key, key)) node = node->tower[0];
    return Iterator(node);
}

```

Insert relies on that higher-level lists have fewer nodes than lower ones:

1. Generate item height as geometric($1 - p$)
2. Start at the highest level
3. \forall level
4. Find the place to insert the node so that have sorted order
5. Link the previous node to the inserted node and it to the next node
6. Go down one list
7. Optionally return a handle to the inserted node

$$E[\text{the number of pointers per node}] = \frac{1}{1-p} . E[\text{the number of comparisons for find}] \leq \frac{\log_{1/p}(n)}{p} + \frac{1}{1-p},$$

which for large n is minimized when $p = 1/e$ (Pugh 1990). This is the runtime for most operations.

```

Iterator insert(KEY const& key, VALUE const& value, bool unique = true)
{
    if(unique)
        { //for unique keys check if one already exists
            Iterator result = findNode(key);
            if(result != end())
            {
                result->value = value;
                return result;
            }
        } //level = height - 1
    int newLevel = min<int>(MAX_HEIGHT, GlobalRNG().geometric(0.632)) - 1;
    Node* newNode = new(f.allocate())Node(key, value, newLevel + 1);
    if(currentLevel < newLevel) currentLevel = newLevel;
    Node** tower = head;
    for(int level = currentLevel; level >= 0; --level)
    {
        for(Node* node; (node = tower[level]) && c(node->key, key); tower = node->tower);
        if(level <= newLevel)
            { //relink pointers
                newNode->tower[level] = tower[level];
                tower[level] = newNode;
            }
    }
}

```

```

    return Iterator(newNode);
}

```

Remove removes the item from every list it's in using the same down-then-right search.

```

void remove(KEY const& key)
{ //with nonunique items will remove first found
    Node **prevTower = head, *result = 0;
    for(int level = currentLevel; level >= 0; --level)
        //go down if node->key < key (when result ==0), else keep moving right
        for(Node* node; (node = prevTower[level]) && !c(key, node->key);
            prevTower = node->tower)
            //found if hit remembered node or nothing remembered and ==
            if(node == result || (!result && !c(node->key, key)))
                //unlink the node from current level
                prevTower[level] = node->tower[level];
                node->tower[level] = 0;
                if(!head[currentLevel])--currentLevel; //if removed top node
                result = node; //remember node
                break; //go down
        }
    if(result) f.remove(result);
}

```

The minimum is the first element and needs $O(1)$ time to get to. The maximum is at the bottom list's end, to get to which use higher-level pointers by repeatedly going to rightmost nonzero node and then down.

```

Iterator findMin(){return Iterator(head[0]);}
Iterator findMax()
{
    Node *result = 0, **tower = head;
    for(int level = currentLevel; level >= 0; --level)
        for(Node* node; node = tower[level]; tower = node->tower)
            result = node;
    return Iterator(result);
}

```

The bottom list is easy to iterate due to being a linked list:

```

class Iterator
{
    Node* current;
    Iterator(Node* node): current(node) {}
    friend SkipList;
public:
    Iterator& operator++()
    {
        assert(current);
        current = current->tower[0];
        return *this;
    }
    Node& operator*()
    {
        assert(current);
        return *current;
    }
    Node* operator->()
    {
        assert(current);
        return current;
    }
    bool operator==(Iterator const& rhs) const
        {return current == rhs.current;}
};

Iterator begin(){return Iterator(findMin());}
Iterator end(){return Iterator(0);}

```

$E[\text{the runtime}] = O(\lg(n) + k)$ for range search returning k items, and $E[\text{the runtime for operations other}$

than $\min] = O(\lg(n))$. A skip list is a bit less efficient than a tree-based dynamic sorted sequence because of variable length nodes and larger constant factors but more extendable by:

- Having items in sorted order at the bottom level
- Supporting nonunique items, which allows it to be a multimap or a priority queue as is
- Supporting many augmentations

The worst-case unlikely runtime for most operations is $O(n)$.

8.4 Treap

Tree height = the maximum number of ancestors of any node. An ordered binary tree with height h implements dynamic sorted sequence operations in $O(h)$ time. The first item becomes a single-node tree of height 0. Any subsequent item recursively replaces the current if $=$, is inserted into the left child if $<$, and into the right child if $>$. When items are inserted in sorted or random order, h is respectively n or $O(\lg(n))$ (Seidel & Aragon 1996). A tree is **balanced** if $h = O(\lg(n))$, which ensures $O(\lg(n))$ runtimes for most operations.

Treap is the simplest balanced binary tree. A newly inserted node gets a random priority. The tree is structured so that a node's priority \leq its parent's priority. This is **heap order** (see the "Priority Queues" chapter), maintained in insert and remove.

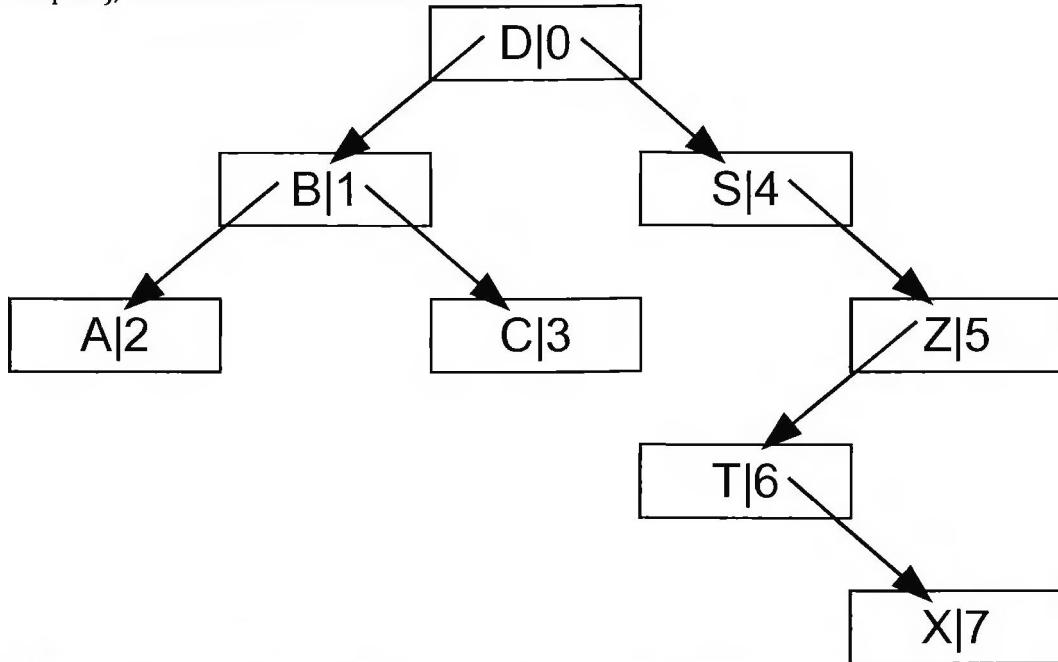


Figure 8.3: Treap items (the letters) and priorities (the numbers)

Nodes maintain parent pointers and subtree node counts as optional augmentations (discussed later in the chapter).

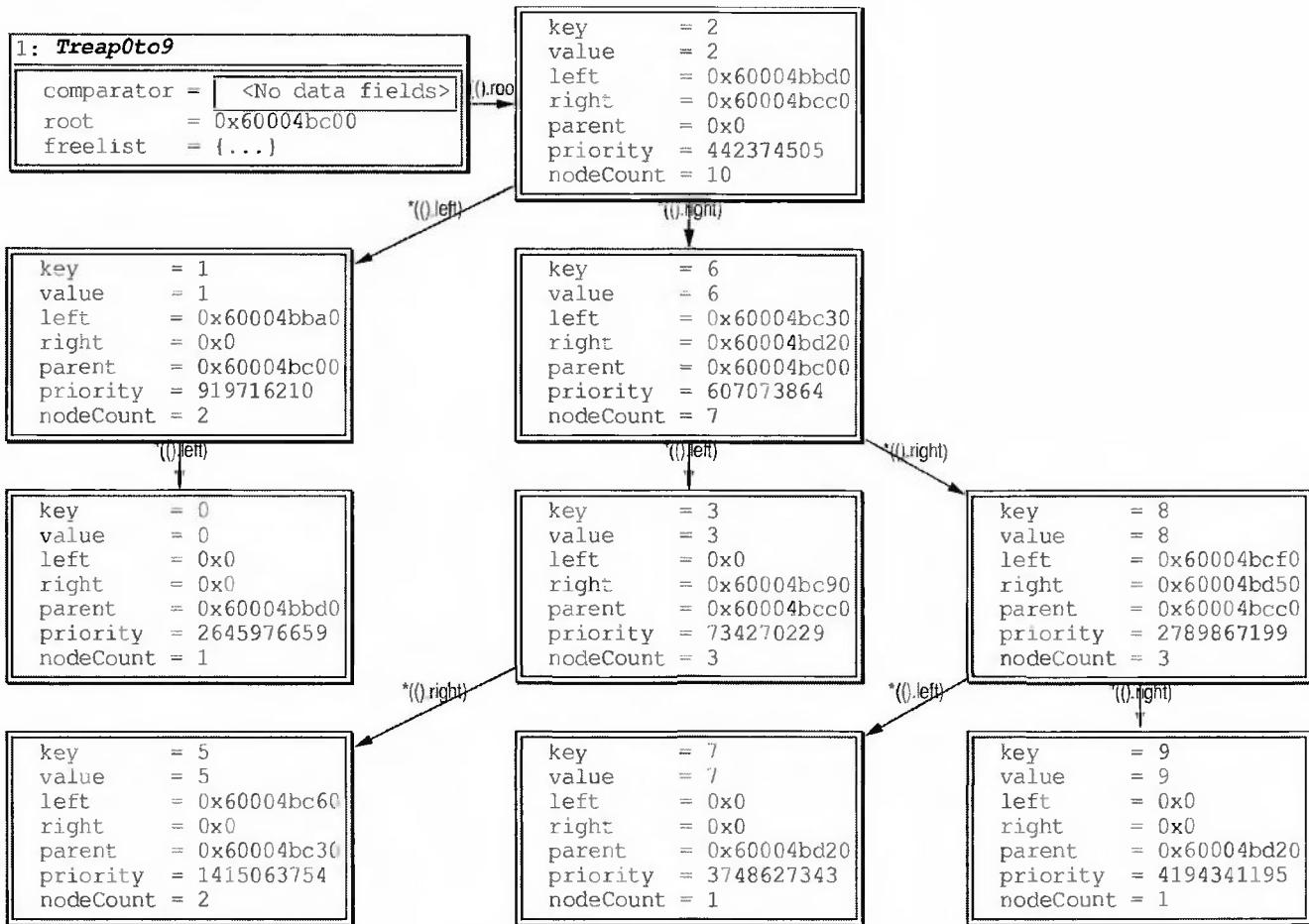


Figure 8.4: Memory layout of a treap with integer items 0–9

The copy constructor is recursive, copying the root and then each child in overall $O(n)$ time. This is a general pattern \forall tree implementation.

```

template<typename KEY, typename VALUE, typename COMPARATOR =
DefaultComparator<KEY>> class Treap
{
    COMPARATOR c;
    struct Node
    {
        KEY key;
        VALUE value;
        Node *left, *right, *parent;
        unsigned int priority, nodeCount;
        Node(KEY const& theKey, VALUE const& theValue): key(theKey),
            value(theValue), left(0), right(0), parent(0),
            priority(GlobalRNG().next()), nodeCount(1)
    }* root;
    Freelist<Node> f;
    Node* constructFrom(Node* node)
    {
        Node* tree = 0;
        if(node)
        {
            tree = new(f.allocate())Node(node->key, node->value);
            tree->priority = node->priority;
            tree->nodeCount = node->nodeCount;
            tree->left = constructFrom(node->left);
            if(tree->left) tree->left->parent = tree;
            tree->right = constructFrom(node->right);
            if(tree->right) tree->right->parent = tree;
        }
        return tree;
    }
}

```

```

public:
    typedef Node NodeType;
    unsigned int getSize() {return root ? root->nodeCount : 0;}
    Treap(COMPARATOR const& theC = COMPARATOR()): root(0), c(theC){}
    Treap(Treap const& other): c(other.c)
    {
        root = constructFrom(other.root);
        if(root) root->parent = 0;
    }
    Treap& operator=(Treap const& rhs) {return genericAssign(*this, rhs);}
}

```

Find:

- 1. Start at the root**
- 2. Until reach the wanted or a null node**
- 3. Go left if key < the current node key, else right**
- 4. Return a pointer to the node containing the item**

find is identical to that of a skip list.

```

Node* findNode (KEY const& key)
{
    Node* node = root;
    while(node && !c.isEqual(key, node->key)) node =
        c(key, node->key) ? node->left : node->right;
    return node;
}

```

Rotations transform the tree to get heap order while preserving the sorted order.

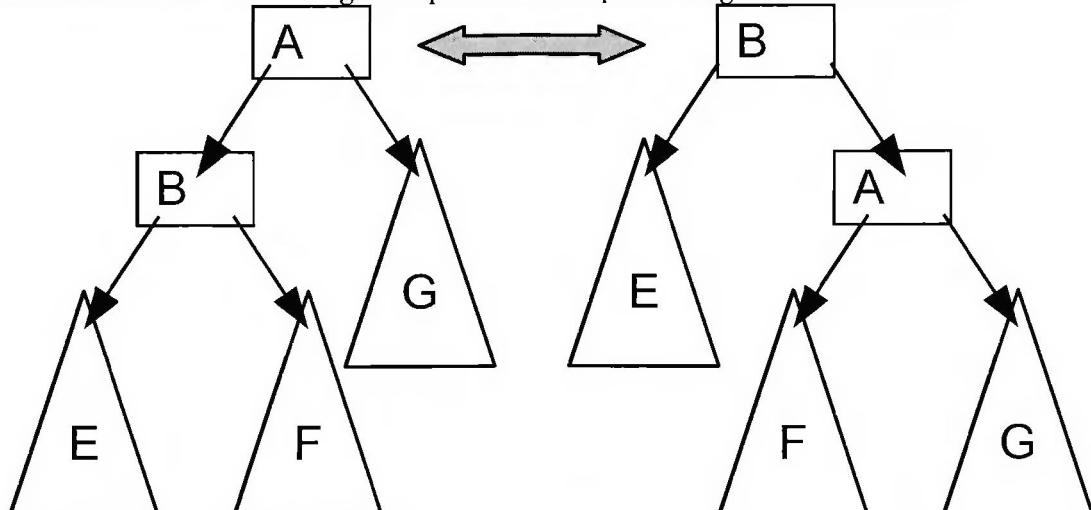


Figure 8.5: Tree rotation pointer rearrangement logic

The right rotation:

- 1. The new root = the subtree's root's left child B**
- 2. The root's left child = the new root's right child F**
- 3. The new root's right child = the root A**
- 4. Reset parent pointers of B and F**
- 5. Node count of B = that of A, and the latter is recalculated**
- 6. Adjust any other augmentations if needed**

The left rotation is symmetric. Both need O(1) time unless the augmentations are complicated. A balanced tree can't have nonunique keys because then rotations break "<". Rotation helpers are implemented generically for reuse in TTT (discussed later in the chapter):

```

template typename NODE> struct TreapGeneric
{
    static void rotateHelper(NODE* node, NODE* goingUp, NODE*& movedChild)
    {
        if(movedChild) movedChild->parent = node;
        movedChild = node;
        goingUp->nodeCount = node->nodeCount;
        goingUp->parent = node->parent;
        node->parent = goingUp;
    }
}

```

```

        node->nodeCount = 1 + (node->left ? node->left->nodeCount: 0) +
            (node->right ? node->right->nodeCount: 0);
    }
};

Rotations are member functions:
```

```

Node* rotateRight(Node* node)
{
    Node *goingUp = node->left, *&movedChild = goingUp->right;
    node->left = child;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}
Node* rotateLeft(Node* node)
{
    Node *goingUp = node->right, *&movedChild = goingUp->left;
    node->right = movedChild;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}
```

A node is first inserted as though the tree is unbalanced, taking left or right branches until hitting a leaf and becoming its child. Then it's rotated up until its priority satisfies heap order—the right rotation lifts the left child and left the right one. Rotations preserve heap order of other items.

```

Node* insertNode(Node* newNode, Node* node)
{
    if(!node) return newNode;
    bool goLeft = c(newNode->key, node->key);
    Node*& chosenChild = goLeft ? node->left : node->right;
    chosenChild = insertNode(newNode, chosenChild);
    chosenChild->parent = node;
    ++node->nodeCount;
    if(chosenChild->priority < node->priority)
        node = goLeft ? rotateRight(node) : rotateLeft(node);
    return node;
}
NodeType* insert(KEY const& key, VALUE const& value)
{
    Node* node = findNode(key);
    if(node) node->value = value;
    else
    {
        node = new(f.allocate()) Node(key, value);
        root = insertNode(node, root);
    }
    return node;
}
```

The resulting tree is unique if the priorities are unique. Using random priorities is equivalent to inserting into an unbalanced tree in random order, so $E[h] = O(\lg(n))$, regardless of the past insert/delete sequence. For a small constant c , $\Pr(h > 1 + 2c \ln(n)) < 2(n/e)^{-cln(c)e}$, effectively ensuring balance for not too small n . The worst-case, highly unlikely $h = O(n)$. $E[\text{the number of rotations for insert}] = O(1)$ (Seidel & Aragon 1996), which matters for augmentations.

Remove uses rotation to leaf, which is the simplest option with augmentations (discussed later in this chapter):

- 1. Find the node corresponding to x**
- 2. Rotate it to a leaf or a single-child node, lifting lower-priority children**
- 3. Remove it**

```

Node* removeFound(Node* node)
{
    Node *left = node->left, *right = node->right;
    if(left && right)
    {
```

```

bool goRight = left->priority < right->priority;
node = goRight ? rotateRight(node) : rotateLeft(node);
Node*& child = goRight ? node->right : node->left;
child = removeFound(child);
if(child) child->parent = node;
--node->nodeCount;
}
else
{
    f.remove(node);
    node = left ? left : right;
}
return node;
}

void remove(KEY const key)
{
    Node* node = findNode(key);
    if(node)
    {
        Node* parent = node->parent;
        bool wasLeft = parent && node == parent->left;
        node = removeFound(node);
        if(node) node->parent = parent;
        (parent ? (wasLeft ? parent->left : parent->right) : root) = node;
        for(; parent; parent = parent->parent) --parent->nodeCount;
    }
}
}

```

The runtime is $O(h)$.

The minimum is the leftmost node, and the maximum the rightmost. Finding both is implemented in `TreapGeneric`:

```

static NODE* findMin(NODE* root)
{
    NODE* node = root;
    if(node) while(node->left) node = node->left;
    return node;
}
static NODE* findMax(NODE* root)
{
    NODE* node = root;
    if(node) while(node->right) node = node->right;
    return node;
}

```

In treap use:

```

NodeType* findMin() {return TreapGeneric<Node>::findMin(root);}
NodeType* findMax() {return TreapGeneric<Node>::findMax(root);}

```

The node count augmentation enables finding n^{th} element, such as the median, and is implemented in `TreapGeneric`. Let n_{lc} = the left child count of the current node. Then iteratively:

1. If $n = n_{lc}$, return the root
2. Else if $n < n_{lc}$, go left
3. Else go right, and reduce n by $n_{lc} + 1$

```

static NODE* nthElement(int n, NODE* root)
{
    assert(n >= 0 && root && n < root->nodeCount);
    NODE* node = root;
    for(;;)
    {
        unsigned int lc = node->left ? node->left->nodeCount : 0;
        if(n == lc) break;
        if(n < lc) node = node->left;
        else
        {

```

```

        n -= lc + 1;
        node = node->right;
    }
}
return node;
}

```

In treap use:

```

NodeType* nthElement(int n)
{
    return TreapGeneric<Node>::nthElement(n, root);
}

```

Successor and predecessor are similar to find and have symmetric logic. For predecessor get as close as possible to x from the left, i.e., maintain a left boundary node as the closest member of the predecessor set:

1. **Pred = 0**
2. **While not at a leaf**
3. **If node < x pred = node, and go right**
4. **Else go left**
5. **Return pred**

Successor is symmetric.

```

NodeType* predecessor(KEY const& key)
{
    Node* pred = 0;
    for(Node* node = root; node;)
        if(c(node->key, key)) //found pred set member
        {
            pred = node;
            node = node->right;
        }
        else node = node->left;
    return pred;
}

NodeType* inclusivePredecessor(KEY const& key)
{
    Node* node = findNode(key);
    return node ? node : predecessor(key);
}

NodeType* successor(KEY const& key)
{
    Node* succ = 0;
    for(Node* node = root; node;)
        if(c(key, node->key)) //found succ set member
        {
            succ = node;
            node = node->left;
        }
        else node = node->right;
    return succ;
}

NodeType* inclusiveSuccessor(KEY const& key)
{
    Node* node = findNode(key);
    return node ? node : successor(key);
}

```

8.5 Tree Iterators

Several ways to iterate a tree in-order:

- Use **parent pointers**, i.e., with children pointing to their parents—makes iterator operations simple and allows bidirectional iteration. But need an extra pointer per node and updating it during insertions and deletions. All operations can return and accept iterators, which is convenient for users but doesn't generalize to structures that don't have iterators, such as a k -d tree (see the "Computation Geometry" chapter). This is the only way to create iterators as lightweight objects and is implemented here and in C++ STL.

- Pass a functor to a range search. E.g., when summing all values the functor has a pointer to the current sum and updates it. This is most efficient and generalizes to a k -d tree, but creating a functor is clumsy for users.
- Do range search as above, with a predefined functor that creates a vector of found items—more convenient for users, but needs memory for the vector.

For iterating with parent pointers, the main idea is that the iterator, when incremented with the right child = null, goes up until returning from a left child. Backward iteration is symmetric. As a convention, iteration from a null node isn't allowed, though it might make sense to remember the previous node if any and allow reverse iteration to it.

```
template<typename NODE> class TreeIterator
{
    NODE* current;
public:
    TreeIterator(NODE* node) {current = node;}
    TreeIterator& operator++()
    {
        assert(current);
        if (current->right)
            //if have right child go there, then maximally left
            current = current->right;
            while (current->left) current = current->left;
        }
        else
            //parent if came from left child, else keep going up
            while (current->parent && current != current->parent->left)
                current = current->parent;
                current = current->parent;
        }
        return *this;
    }
    TreeIterator& operator--()
    {
        assert(current);
        if (current->left)
            //if have left child go there, then maximally right
            current = current->left;
            while (current->right) current = current->right;
        }
        else
            //parent if came from right child, else keep going up
            while (current->parent && current != current->parent->right)
                current = current->parent;
                current = current->parent;
        }
        return *this;
    }
    NODE& operator*() {assert(current); return *current;}
    NODE* operator->() {assert(current); return current;}
    bool operator==(TreeIterator const& rhs) const
        {return current != rhs.current;}
};
```

In treap use:

```
typedef TreeIterator<Node> Iterator;
Iterator begin() {return Iterator(findMin());}
Iterator end() {return Iterator(0);}
Iterator rBegin() {return Iterator(findMax());}
Iterator rEnd() {return Iterator(0);}
```

8.6 Augmentations and API Variants

Common augmentations for trees, as already discussed, include storing the number of nodes in each subtree and parent pointers. But occasionally some others are useful.

A dynamic sorted sequence API can be a:

- Map—implement other variants in terms of it
- **Multimap**, allowing equal items—a map where the item type is a vector of items
- **Set**, with keys but not values—a map where the item type is ignored Boolean or `Empty` for low storage
- **Multiset**, where keys aren't unique—a map where the item is a count of keys

A skip list supports the multi variants directly. STL implements maps in terms of sets using key-item pairs as set items, but items need a default constructor because find constructs a key-item pair. Other than using a vector of keys, can keep a global insertion count, and use it as a secondary priority. This is simpler when maintain node counts.

For only map operations a hash table (see the “Hashing” chapter) are faster. For successor/predecessor or in-order iteration need a dynamic sorted sequence. In general, asking which data structure to use when is one of my favorite interview questions. A treap is a bit more efficient in speed and memory use, so prefer it when the standard augmentations are sufficient. But the simplicity of a skip list might make it better for other augmentations, and it supports faster iteration.

8.7 Vector Keys

When keys are variable-length arrays of up to k objects comparable in $O(1)$ time, key comparisons take $O(k)$ time, giving $O(k \lg(n))$ operations. Also $E[\text{lcp of two random strings over an alphabet of size } a \text{ out of a collection of } n] = \log_a(n)$, so the expected performance is usually better.

The main additional operations are already supported:

- **Prefix search**—given a length, find all items with $\text{lcp}(x, \text{item's key}) \geq \text{length}$. Iterating from the inclusive predecessor of the prefix gives the result.
- **Longest match**—given key x , find the item maximizing $\text{lcp}(x, \text{item's key})$. The predecessor of x or its successor is the result.

In practice the performance is good enough, but can improve efficiency. The idea is finding the lcp and comparing $\text{key}[\text{lcp}]$. Lcp of any key with $-\infty$ or ∞ is 0. \forall vector keys $x \leq y \leq z$:

- $\text{lcp}(x, y) \geq \text{lcp}(x, z)$, with equality only if $y = z$
- $\text{lcp}(x, z) = \min(\text{lcp}(x, y), \text{lcp}(y, z))$

8.8 LCP Augmentation for Trees

Store in each node the lcps between it and its predecessor and successor among the nodes on its search path (Grossi & Italiano 1999; Crescenzi et al. 2003). Use `unsigned short` to save space. The lcps are 0 for the root, relative to the imaginary ∞ keys.

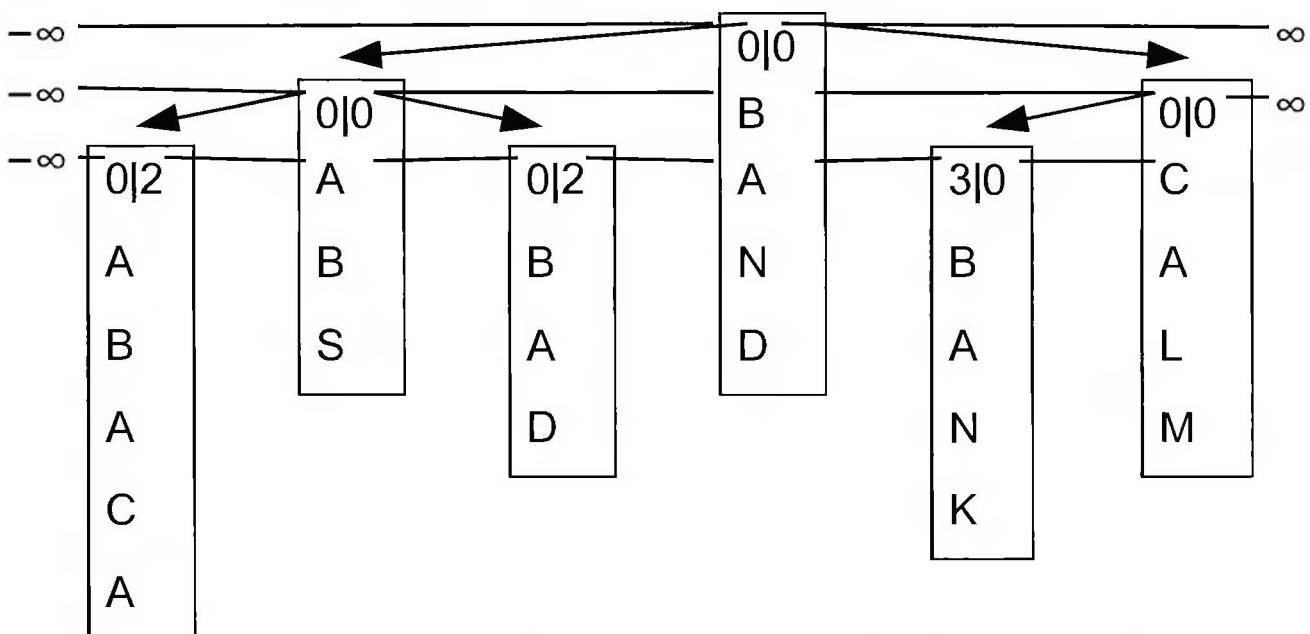


Figure 8.6: Lcps between nodes and their predecessors/successors. E.g., for “bad” only “abs” is in the predecessor set, so pred lcp = 0, and only “band” is in the successor set, so succ lcp = 2.

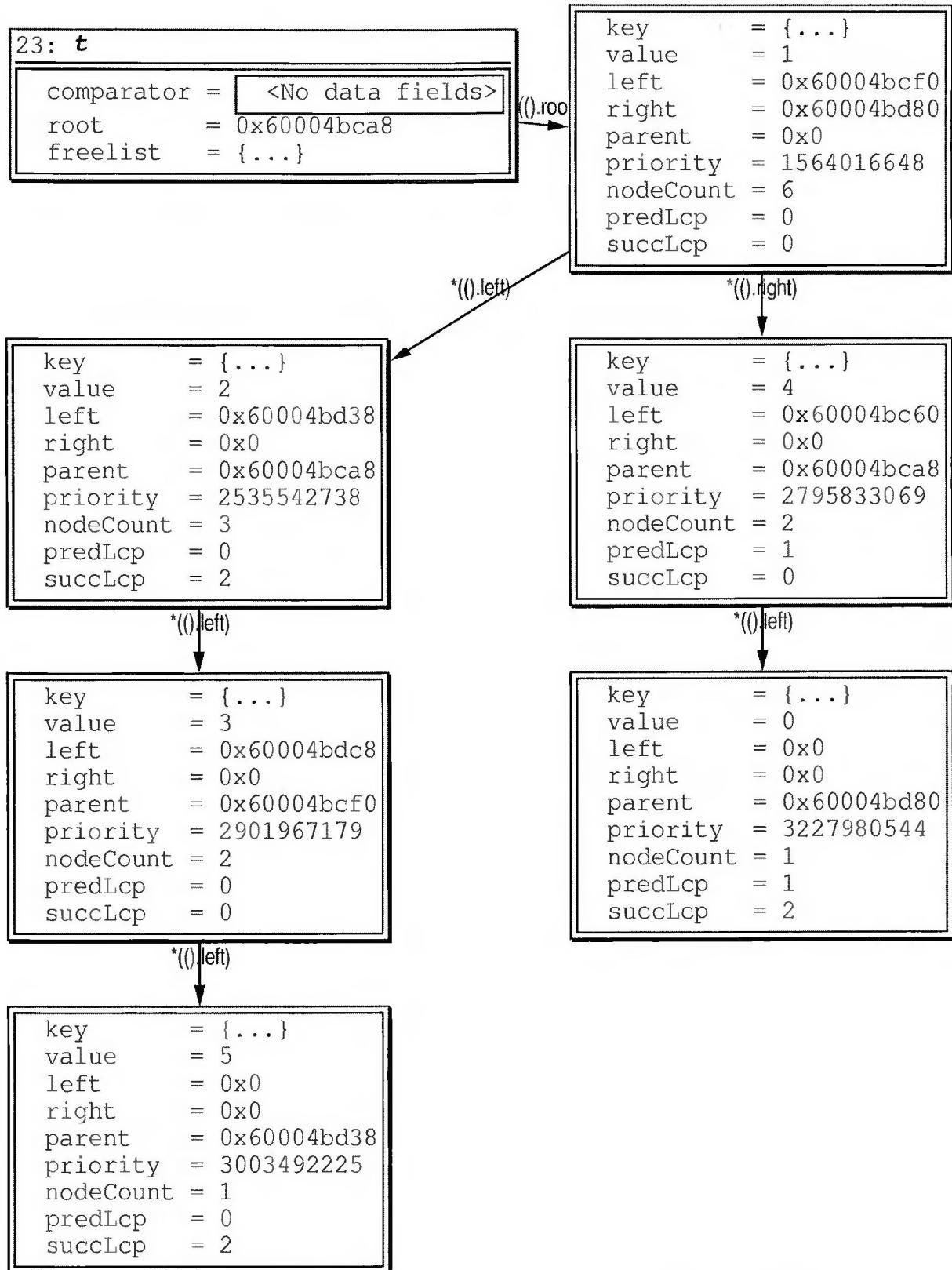


Figure 8.7: Memory layout of a LCPTreap with some words

```
template<typename KEY, typename VALUE, typename INDEXED_COMPARATOR =
    LexicographicComparator<KEY>> class LCPTreap
{
    INDEXED_COMPARATOR c;
    struct Node
    {
        KEY key;
        VALUE value;
        Node *left, *right, *parent;
        unsigned int priority, nodeCount;
    };
}
```

```

unsigned short predLcp, succLcp;
Node(KEY const& theKey, VALUE const& theValue): key(theKey),
    value(theValue), left(0), right(0), priority(GlobalRNG().next()),
    predLcp(0), succLcp(0), parent(0), nodeCount(1) {}

/* root;
Freelist<Node> f;
Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate())Node(node->key, node->value);
        tree->priorit

```

```

public:
typedef Node NodeType;
LCPTreap(INDEXED_COMPARATOR theC = INDEXED_COMPARATOR()):root(0), c(theC) {}
LCPTreap(LCPTreap const& other): c(other.c)
    {root = constructFrom(other.root);}
LCPTreap& operator=(LCPTreap const&rhs) {return genericAssign(*this,rhs);}
unsigned int getSize() {return root ? root->nodeCount : 0;}
};

When compare  $x$  with the current node  $c$  with ancestor predecessor  $p$  and ancestor successor  $s$ :

```

- $p < x < s$, and let $\max(lcp(p, x), lcp(s, x)) = m$
- $p < c < s$, and let $lcp(p, c) = l$ and $lcp(s, c) = r$

If $c \neq x$, $lcp(c, x)$ is the first index at which $c \neq x$, and can find it based on the following logic.

- If $lcp(p, x) = m$
 - If $l < m$, $lcp(c, x) = \min(lcp(c, p), lcp(p, x)) = l$ —can compare $x[l]$ with $c[l]$
 - Else $lcp(c, x) \geq m$ —need to keep comparing
- If $lcp(s, x) = m$
 - If $r < m$, $lcp(c, x) = \min(lcp(s, p), lcp(p, x)) = r$ —can compare $x[r]$ with $c[r]$
 - Else $lcp(c, x) \geq m$ —need to keep comparing

To decide whether m is from p or s , keep $lcp(p, x)$ in $predM$; $findLcp$ calculates $lcp(c, x)$ and updates m to include the current node as its ancestor predecessor.

```

int findLCP(KEY const& key, Node* node, int predM, int& m)
{
    int lcp = predM == m ? node->predLcp : node->succLcp; //get l or r
    if(lcp >= m)
    {
        while(m < c.getSize(key) &&
            c.isEqual(key, node->key, m)) ++m;
        lcp = m;
    }
    return lcp;
}

```

If c and x have the same length = lcp , then $c = x$. $m \leq k$ and is never decreased, so map operations cost $O(k + \lg(n))$. When going right, set $predM = lcp(c, x)$ because c becomes p . $find$ is identical to that of a treap.

```

Node* findNode(KEY const& key)
{
    Node* node = root;
    int m = 0, predM = 0;
}

```

```

while(node)
{
    int lcp = findLCP(key, node, predM, m);
    if(c.getSize(key) == lcp && c.getSize(node->key) == lcp) break;
    if(c(key, node->key, lcp)) node = node->left;
    else
    {
        node = node->right;
        predM = lcp;
    }
}
return node;
}

```

Rotations maintain the lcp information. For the right rotation the node's left child becomes its ancestor predecessor. So set its `predLcp = succLcp` of the left child—the node is its ancestor successor. The left child loses its ancestor successor, so set its `succLcp = lcp`(it, the ancestor successor of the node) = the min of the two `succLcp` values (by the triangle equality). The left rotation is symmetric, and the runtime of both is $O(1)$.

```

Node* rotateRight (Node* node)
{
    Node *goingUp = node->left, *&movedChild = goingUp->right;
    node->predLcp = goingUp->succLcp;
    goingUp->succLcp = min(node->succLcp, goingUp->succLcp);
    node->left = movedChild;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}
Node* rotateLeft (Node* node)
{
    Node *goingUp = node->right, *&movedChild = goingUp->left;
    node->succLcp = goingUp->predLcp;
    goingUp->predLcp = min(node->predLcp, goingUp->predLcp);
    node->right = movedChild;
    TreapGeneric<Node>::rotateHelper(node, goingUp, movedChild);
    return goingUp;
}

```

Insert is a hybrid of find and the treap insert. It creates the node and updates its `predLcp / succLcp` when going respectively left/right. During insertion `predLcp = predM`.

```

Node* insertNode(Node* newNode, Node* node, int m = 0)
{
    if (!node) return newNode;
    int lcp = findLCP(newNode->key, node, newNode->predLcp, m);
    bool goLeft = c(newNode->key, node->key, lcp);
    (goLeft ? newNode->succLcp : newNode->predLcp) = lcp;
    Node*& chosenChild = goLeft ? node->left : node->right;
    chosenChild = insertNode(newNode, chosenChild, m);
    chosenChild->parent = node;
    ++node->nodeCount;
    if(chosenChild->priority < node->priority)
        node = goLeft ? rotateRight(node) : rotateLeft(node);
    return node;
}

```

insert is identical to that of a treap. Same for removeFound, remove, min, max, nthElement, iterator functions, and inclusiveSuccessor.

Successor and predecessor differ only in that they find lcp and compare `key[lcp]`. The runtime is the same as for map operations.

```

NodeType* predecessor(KEY const& key)
{
    int m = 0, predM = 0;
    Node* pred = 0;
    for(Node* node = root; node; )

```

```

    {
        int lcp = findLCP(key, node, predM, m);
        if(c(node->key, key, lcp)) //found pred set member
        {
            pred = node;
            node = node->right;
            predM = lcp;
        }
        else node = node->left;
    }
    return pred;
}
NodeType* successor(KEY const& key)
{
    int m = 0, predM = 0;
    Node* succ = 0;
    for(Node* node = root; node;)
    {
        int lcp = findLCP(key, node, predM, m);
        if(c(key, node->key, lcp)) //found succ set member
        {
            succ = node;
            node = node->left;
            predM = lcp;
        }
        else node = node->right;
    }
    return succ;
}

```

Prefix successor is similar to successor but skips over items > the prefix that have it. The runtime is the same as for map operations.

```

NodeType* prefixSuccessor(KEY const& prefix)
{
    int m = 0, predM = 0;
    for(Node* node = root; node;)
    {
        int lcp = findLCP(prefix, node, predM, m);
        if(c.getSize(prefix) == lcp || !c(prefix, node->key, lcp))
        {
            node = node->right;
            predM = lcp;
        }
        else if(!node->left) return node;
        else node = node->left;
    }
    return 0;
}

```

Though the lcp augmentation is useful, can avoid it in many cases for efficiency:

- Implement a map with integer-pair keys by encoding a pair into `int` or `long long` and using that as a key. Need some care—e.g., given $0 \leq a < m$ and $0 \leq b < n$, with $m > n$, the mapping $x = a + bm$ works, but $x = b + an$ doesn't because $(b + an) \% n \neq b$ if $b > n$. During interviews candidates often prefer to make string keys from pairs because it works well with APIs in various languages (including hash tables), but this is inefficient.

```

template<typename WORD = unsigned long long> class Key2DBuilder
{
    unsigned int n;
    bool firstNotSmaller;
public:
    typedef WORD WORD_TYPE;
    Key2DBuilder(unsigned int theN = numeric_limits<unsigned int>::max(),
                 bool theFirstNotSmaller = true): n(theN),

```

```

        firstNotSmaller(theFirstNotSmaller){}
WORD toID(unsigned int n1, unsigned int n2) const
{
    assert(n1 < n && n2 < n);
    return firstNotSmaller ? n1 * n + n2 : n2 * n + n1;
}
pair<unsigned int, unsigned int> to2D(WORD key) const
{
    pair<unsigned int, unsigned int> n1n2(key/n, key % n);
    assert(n1n2.first < n && n1n2.second < n);
    if(!firstNotSmaller) swap(n1n2.first, n1n2.second);
    return n1n2;
}
};

```

- For longest match on bit sequences \leq word size, e.g., in network routers, use a map with the longest match implemented as predecessor

8.9 Tries

A trie is a map where keys are variable-length arrays of objects comparable in $O(1)$ time. Supposing that vectors are strings, a trie is a tree where each node has a pointer to a string, consisting of the i characters that lead to the node, and a map from the next letter to the corresponding node. The root branches on the letter 0 and points to the item represented by the empty string. A child of a node checking the letter i checks the letter $i + 1$. Find starts at the root and follows pointers until characters or nodes run out, taking $O(hM)$ time, where M is the time to look up the next node.

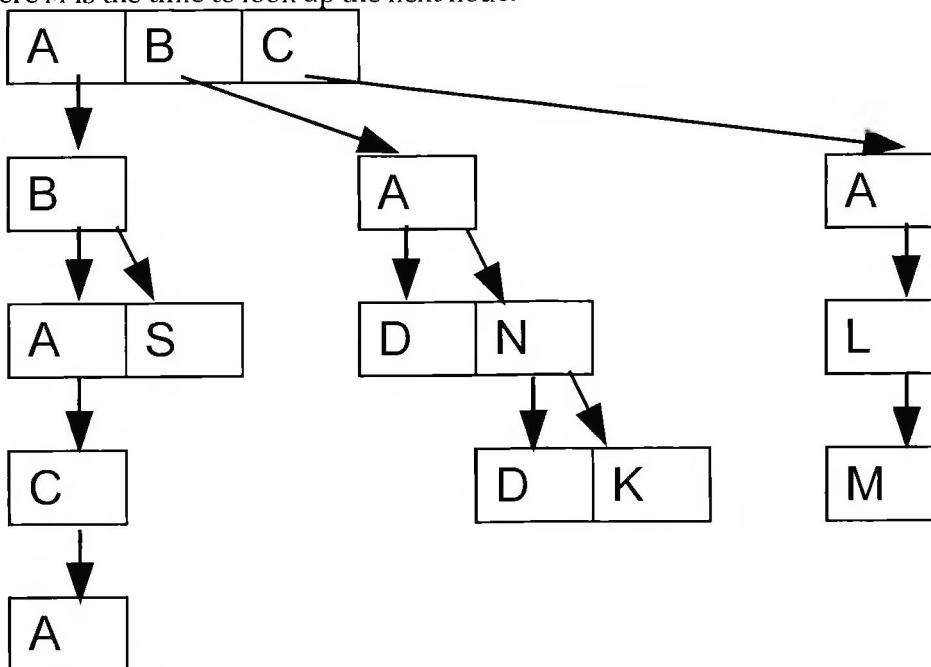


Figure 8.8: Abstract trie pointer structure

Can implement the map in many ways, most importantly as an array or a dynamic sorted sequence. For the former $M = O(1)$, but use too much space, and have inefficient in-order iteration. For the latter $M = O(\lg(\text{the alphabet size}))$. A trie additionally supports:

- **Incremental search**—e.g., searching consecutively for “h”, “he”, “hel”, “hell”, and “hello” takes $O(h)$ time
- Suffix augmentations—keys reachable from a node of height i share a prefix of length i

Needing these operations is the only good reason to use a trie. A trie with any map implementation usually uses more memory than a LCPTreap and doesn't represent keys as single objects, making iteration that needs key values difficult. The map is one reason for high memory use, and **suffix expansion** is another. After some depth a prefix of a key uniquely identifies it, but a trie uses one node per key object instead of a single node with the suffix. So it's memory-hungry for long strings with small average lcp.

8.10 Ternary Treap Trie

A TTT efficiently implements the dynamic sorted sequence node map. It mimics multikey quicksort and uses left, down, and right pointers with a pivot character. Find goes down if the i^{th} character = the node's pivot, left if $<$, and right if $>$. An item is stored at the node where the last key character = the pivot. Treap priorities decide the tree structure of nodes checking the i^{th} character. The operations use the asymptotically optimal space and expected time for comparable key objects. The latter is $O(\lg(n)) + \text{the average key length}$. Don't support the key of length 0, but can handle it externally if needed.

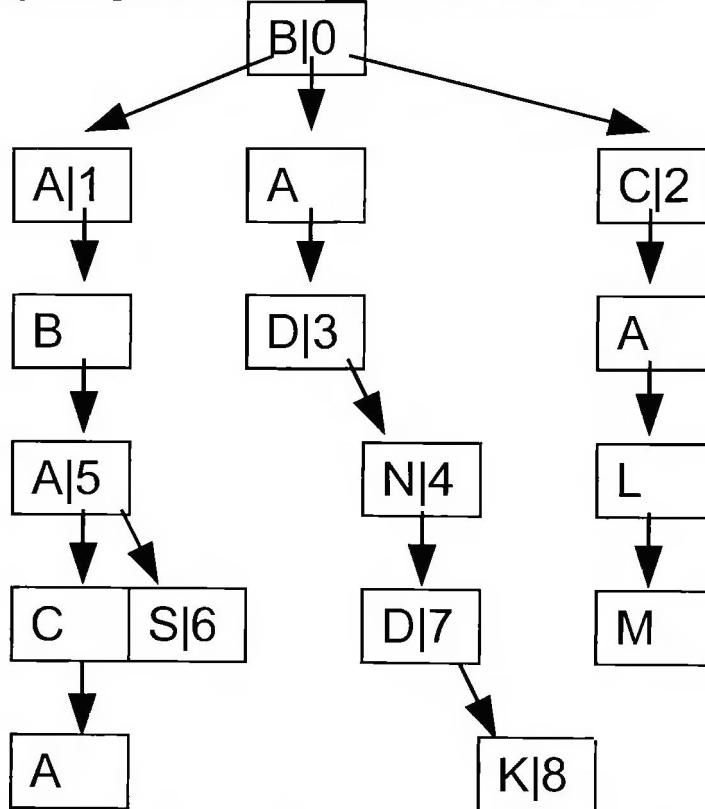


Figure 8.9: TTT pointer structure and priorities. Where sideways links are present, the numbers are treap priorities.

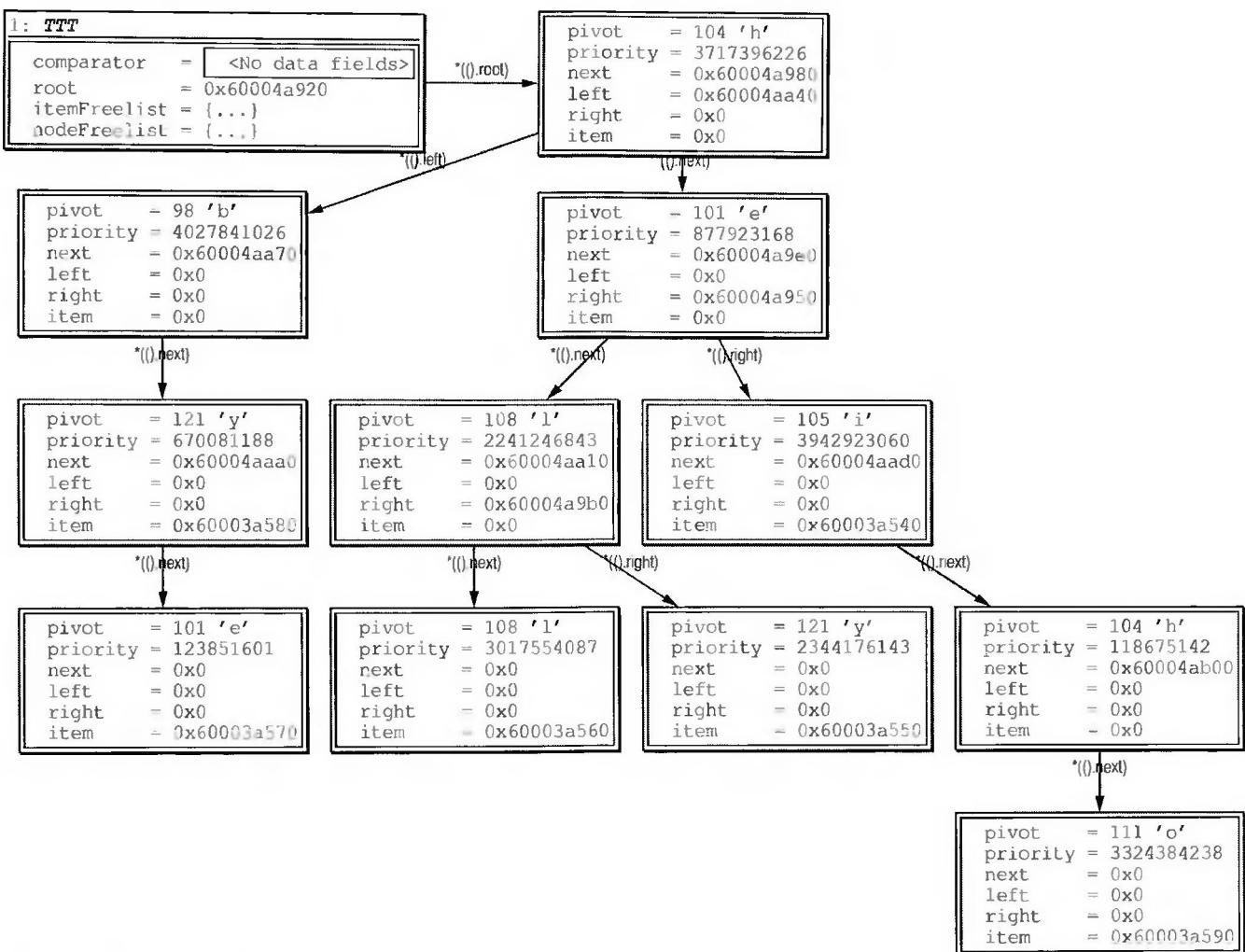


Figure 8.10: Memory layout of a TTT

```

template<typename ITEM, typename KEY_OBJECT = unsigned char, typename
COMPARATOR = DefaultComparator<ITEM>> class TernaryTreapTrie
{
    COMPARATOR c;
    struct Node
    {
        KEY_OBJECT pivot;
        unsigned int priority;
        Node *next, *left, *right;
        ITEM* item;
        Node(KEY_OBJECT const& thePivot): next(0), left(0), right(0),
            item(0), pivot(thePivot), priority(GlobalRNG().next()) {}

        /* root;
        Freelist<ITEM> itemF;
        Freelist<Node> nodeF;
        Node* constructFrom(Node* node)
        {
            Node* result = 0;
            if(node)
            {
                result = new(nodeF.allocate()) Node(node->pivot);
                result->priority = node->priority;
                if(node->item) result->item = new(itemF.allocate()) ITEM(node->item);
                result->left = constructFrom(node->left);
                result->next = constructFrom(node->next);
                result->right = constructFrom(node->right);
            }
            return result;
        }
    }
}

```

```

public:
    TernaryTreapTrie(COMPARETOR const& theC = COMPARETOR()): root(0), c(theC){}
    TernaryTreapTrie(TernaryTreapTrie const& other):
        c(other.c){root = constructFrom(other.root);}
    TernaryTreapTrie& operator=(TernaryTreapTrie const& rhs)
        {return genericAssign(this, rhs);}
};

```

The memory overhead is five words per key object, leading to a costly suffix expansion, but many nodes are shared if the items have a high average lcp. In comparison, LCPTreap has five words of overhead per key (both without augmentations).

Find is implemented in terms of incremental find that manages a handle to the current node. The idea of the latter is to start not from the root but from an existing node that is known to be a prefix of the sought key. A handle object encapsulates this for callers' convenience.

```

struct Handle
{
    Node* node;
    int i;
    Handle(Node* theNode = 0, int theI = 0): node(theNode), i(theI){}
};

Node* findNodeIncremental(KEY_OBJECT* key, int keySize, Handle& h)
{
    while(h.node && h.i < keySize)
    {
        if(c.isEqual(key[h.i], h.node->pivot))
        {
            if(h.i == keySize - 1) return h.node;
            else[h.node = h.node->next; ++h.i];
        }
        else if(c(key[h.i], h.node->pivot)) h.node = h.node->left;
        else h.node = h.node->right;
    }
    h = Handle(); //not found
    return 0;
}
ITEM* findIncremental(KEY_OBJECT* key, int keySize, Handle& h)
{
    if(!h.node) h.node = root;
    Node* result = findNodeIncremental(key, keySize, h);
    return result ? result->item : 0;
}
Node* findNode(KEY_OBJECT* key, int keySize)
{
    Handle h(root, 0);
    return findNodeIncremental(key, keySize, h);
}
ITEM* find(KEY_OBJECT* key, int keySize)
{
    Node* result = findNode(key, keySize);
    return result ? result->item : 0;
}

```

The runtime of incremental find is same as that for regular find, but n is size of the starting node subtree, and key length is reduced by the passed prefix.

Rotations and join are same as for treap. Insert moves as find but adds new nodes when existing ones run out and, as for a treap, rotates the nondown links on the way up to restore heap order.

```

Node* insertNode(Node* node, KEY_OBJECT* key, int keySize,
    ITEM const& item, int i)
{
    if(!node) node = new(nodeF.allocate()) Node(key[i]);
    if(c.isEqual(key[i], node->pivot))//go down
        if(i == keySize - 1)//found the wanted node
        {

```

```

        if(node->item) *node->item = item;
        else node->item = new(itemF.allocate()) ITEM(item);
    }
    else node->next = insertNode(node->next, key, keySize, item,
        i + 1); //keep going
else
//go sideways
    bool goLeft = c(key[i], node->pivot);
    Node*& chosenChild = goLeft ? node->left : node->right;
    chosenChild = insertNode(chosenChild, key, keySize, item, i);
    if(chosenChild->priority < node->priority)
        node = goLeft ? rotateRight(node) : rotateLeft(node);
}
return node;
}
void insert(unsigned char* key, int keySize, ITEM const* item)
{
    assert(keySize > 0);
    root = insertNode(root, key, keySize, item, 0);
}

```

Remove finds the item and on the way back up removes nodes on the path to it that have no item or down pointer. Node remove is done by deleting it directly and joining the children. This is more efficient than rotation to leaf but less friendly to augmentations, which isn't an issue here.

```

Node* join(Node* left, Node* right)
{
    if(!left) return right;
    if(!right) return left;
    if(left->priority < right->priority) //lower priority goes up
    {
        left->right = join(left->right, right);
        return left;
    }
    else
    {
        right->left = join(left, right->left);
        return right;
    }
}
Node* removeR(Node* node, KEY_OBJECT* key, int keySize, int i)
{
    if(node)
    {
        bool isEqual = c.isEqual(key[i], node->pivot);
        if(isEqual && i == keySize - 1)
        {//remove found item if it exists
            if(!node->item) return node;
            itemF.remove(node->item);
            node->item = 0;
        }
        else
        {//go to next node
            Node** child;
            if(isEqual) child = &node->next; ++i;
            else if(c(key[i], node->pivot)) child = &node->left;
            else child = &node->right;
            *child = removeR(*child, key, keySize, i);
        }
        if(!node->item && !node->next)
        {//remove empty node
            Node* left = node->left, *right = node->right;
            nodeF.remove(node);
            node = (left || right) ? join(left, right) : 0;
        }
    }
}

```

```

        }
    }
    return node;
}
void remove(KEY_OBJECT* key, int keySize)
{
    assert(keySize > 0);
    root = removeR(root, key, keySize, 0);
}

```

Longest match finds any item with the greatest lcp with x —select the one with the shortest key. This is particularly fast for tries in general; the runtime is $O(\lg(n) + \text{lcp})$.

```

ITEM* longestMatch(KEY_OBJECT* key, int keySize)
{
    Node* node = root, *result = 0;
    for(int i = 0; node && i < keySize;)
        if(c isEqual(key[i], node->pivot))
        {
            result = node;
            if(i == keySize - 1) break; //reached last key object
            else {node = node->next; ++i;}
        }
        else if(c(key[i], node->pivot)) node = node->left;
        else node = node->right;
    return result ? result->item : 0;
}

```

The usual iteration isn't an option due to breaking up of keys (it can be done clumsily by reconstructing keys while iterating, perhaps with a builder from the caller), so do range search instead. For-each iterates over nodes in-order and executes a user-specified functor on each. E.g., a useful functor copies all items to a vector. Using these can find all items whose keys share a prefix:

```

template<typename ACTION> void forEachNode(Node* node, ACTION& action)
{
    if(node)
    {
        action(node);
        forEachNode(node->left);
        forEachNode(node->next);
        forEachNode(node->right);
    }
}
struct CopyAction
{
    Vector<ITEM*>& result;
    CopyAction(Vector<ITEM*>& theResult): result(theResult) {}
    void operator()(Node* node)
        {if(node->item) result.append(node->item);}
};
Vector<ITEM*> prefixFind(KEY_OBJECT* key, int lcp)
{
    Vector<ITEM*> result;
    CopyAction action(result);
    forEachNode(findNode(key, lcp), action);
    return result;
}

```

The runtime is $O(\text{prefix find} + \text{the number of nodes in the found subtree})$.

8.11 Performance Comparisons

	SkipList		Treap		Chaining		Linear Probing		LCPTreap		TernaryTreapTrie	
Key Type	Time (sec)	Mem (MB)	Time (sec)	Mem (MB)	Time (sec)	Mem (MB)	Time (sec)	Mem (MB)	Time (sec)	Mem (MB)	Time (sec)	Mem (MB)
int	27	42	16	38	8	33	8.3	25				
Struct10	93	95	72	90	49	86	52	117	26	96	123	98
Struct10_2	34	95	21	90	49	86	51	117	26	96	164	1578
Struct10_4									44	96	59	55
Struct10_5									145	96	164	1578

Figure 8.11: Times for 10^6 insertions and 10 times more finds and the same number of deletions. The memory use was recorded after the insertions. The treap and the LCPTreap implementations didn't have the augmentation code here. Struct10 is a structure of 10 integers, designed so that LCP is either high or low, depending on the variant.

A treap wins over a skip list, and a LCPTreap over a TTT. Hash tables win on time as expected, but not by a large factor. Memory use of a linear problem hash table is subject to array doubling, which isn't friendly to large types.

8.12 Implementation Notes

Most of the implementations are original in some way:

- The treap implementation has full iterator support and the common augmentations
- Same for LCPTreap, which is coming directly from the primary literature
- TTT is original based on the suggestion in (Badr & Oommen 2005), though the implementation isn't difficult given that of ternary trie and treap

The decision to rule out other trie implementations is uncommon, but well-justified (see the "Comments" section). I spent a few months experimenting with a variety of tries before deciding to keep only TTT to stay in the key comparison model and because of its good performance.

8.13 Comments

Using a garbage-collecting free list allows to not write destructors. Where need these for a tree, a simple solution is to delete the tree recursively, as for the copy constructor. A cleverer nonrecursive solution is to iteratively remove the top node and implicitly join, using maybe first the left subtree and then the right one. With a balanced tree the general need to remove recursion doesn't hold because the stack is at least probabilistically guaranteed to be small, but it's best to do it anyway if simple.

See Mehlhorn & Näher (1999) for details for other skip list operations, particularly the **back pointer augmentation**. For the lcp augmentation can attach lcp to each tower pointer (Ciriani et al. 2007).

There are many different balanced trees, with some having worst-case $O(\lg(n))$ map operations and/or other nice properties, but none are as simple as a treap and outperform it on average. In particular, many libraries implement a **red-black tree** (Cormen et al. 2009). It also has amortized $O(1)$ rotations per insertion, here amortized and not expected, but its balancing strategy doesn't work for ternary trie. For map operations, experimentally a treap is faster than a red-black tree, which is faster than a skip list (Heger 2004). The **left-leaning red-black tree** (Sedgewick & Wayne 2011; Sedgewick 1999) is simpler but seems to lose the $O(1)$ amortized rotations due to being isomorphic to the older **2-3 tree**, which doesn't have it. A B-tree (see the "External Memory Algorithms" chapter) with $B = 2$ is a **2-3-4 tree** (Mehlhorn et al. 2019), isomorphic to a red-black tree, and enjoys the same properties. The latter was derived from the former to save space on empty pointers.

A **splay tree** (Brass 2008) has impractical constant factors but is theoretically interesting because it doesn't use extra space for balancing, and every sequence of m map operations on k elements takes $O(m\lg(k))$ amortized time. An **AVL tree** (Brass 2008) balances using rotations to maintain bounded height differences. Compared to a red-black tree, it's slightly more balanced, giving slightly faster search, but insertion is slower, and don't have the $O(1)$ amortized rotations. A **weight-balanced tree** (Brass 2008) is similar to an AVL tree but balances on weight, i.e., the number of nodes in each subtree. This has the advantage of getting the node size augmentation at no extra memory cost but is slower for both find and insert. Same for a **randomized tree** (Sedgewick 1999) that adjusts itself to be random after every operation, using node count proportions. See Mehta & Sahni (2018) for a discussion of balancing in general. Brass (2008) mentions several other implementations about which you've likely never heard of (for a good reason).

Using treap priorities seems to be one of the few ways to balance a ternary trie. Don't have a worst-case

balancing strategy because the usual height or weight balancing don't work (Badr & Oommen 2005). Various trie varieties have problems:

- An array-map trie with key object = 4 bits of key—makes map sizes seem reasonable, but still suffers from high memory use, loses generality and simple prefix operations, and becomes a bad hash table.
- A **bucket trie**—put items that share a common prefix into a small, linearly-searchable bucket to reduce the memory due to the suffix expansion. While applicable to any trie, including TTT, this makes operations clumsy and inefficient due to needing to keep track of several node types and occasionally convert between them on insertion and deletion.
- A **digital search tree (DST)** (Sedgewick 1999)—like a tree but uses bits of keys to branch instead of comparisons. I.e., at level i , if the i^{th} bit is 0, go left, else right. So $h \leq \text{key size in bits}$. This works with bit sequences only, is inefficient for sequences with high bit lcp, the performance depends on endianness, and supports other operations such as predecessor only if lexicographic bit order leads to correct comparisons. So DST is also at best a bad hash table.
- A **patricia trie** (Sedgewick 1999; Mehta & Sahni 2018)—improves DST by jumping to bits that differ. This results, e.g., in a recommendation of this data structure for longest bit sequence matching for network routers (Mehta & Sahni 2018). But can do the same with any dynamic sorted sequence using predecessor, as discussed before, and the slowness of testing one bit at a time and applicability to only bit sequences remain.

8.14 Projects

- For a skip list investigate the performance as a function of p . In particular, try $p = 0.25$, as recommended in various sources.
- For a skip list implement the back pointer and the nodes-after-count augmentations to allow bidirectional iteration and finding the i^{th} element.
- For a skip list implement the lcp augmentation.
- Change a treap and a LCPTreap to return iterators instead of node pointers.
- Some library standards might require worst-case logarithmic performance for a dynamic sorted sequence. Here a red-black tree is the best choice. Research it, implement it, and do a performance comparison with a treap, a skip list, and the STL map.
- Automate the performance comparison using a Python (or another language that allows easy OS interaction) script to record memory use of various structures.
- For TTT implement iteration with key reconstruction
- Fix const-correctness of methods such as find. May need to introduce const-iterators like STL does.

8.15 References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Badr, G. H., & Oommen, B. J. (2005). Self-adjusting of ternary search tries using conditional rotations and randomized heuristics. *The Computer Journal*, 48(2), 200–219.
- Brass, P. (2008). *Advanced Data Structures*. Cambridge University Press.
- Ciriani, V., Ferragina, P., Luccio, F., & Muthukrishnan, S. (2007). A data structure for a sequence of string accesses in external memory. *ACM Transactions on Algorithms (TALG)*, 3(1), 6.
- Crescenzi, P., Grossi, R., & Italiano, G. F. (2003). Search data structures for skewed strings. In *Experimental and Efficient Algorithms* (pp. 81–96). Springer.
- Grossi, R., & Italiano, G. F. (1999). Efficient techniques for maintaining multidimensional keys in linked data structures. In *Automata, Languages, and Programming* (pp. 372–381). Springer.
- Heger, D. A. (2004). A disquisition on the performance behavior of binary search tree data structures. *European Journal for the Informatics Professional*, 5(5), 67–75.
- Mehlhorn, K., & Näher, S. (1999). *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press.
- Mehlhorn, K., & Sanders, P., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- Mehta, D. P., & Sahni, S. (Eds.). (2018). *Handbook of Data Structures and Applications*. CRC.
- Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 668–676.
- Sedgewick, R. (1999). *Algorithms in C++*. Parts 1–4. Addison-Wesley.

- Sedgewick, R. (2008). Left-leaning red-black trees. In *Dagstuhl Workshop on Data Structures*.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.
- Seidel, R., & Aragon, C. R. (1996). Randomized search trees. *Algorithmica*, 16(4–5), 464–497.

9 Hashing

9.1 Introduction

From my own education and interviewing experience, hashing algorithms are generally poorly understood by students after having taken an algorithms class. E.g., a request to compare a hash table with a dynamic sorted sequence in detail confuses many. I believe the problem is in randomization and resizing aspects of hash tables, so I give implementations of useful hash tables and hash functions.

9.2 Hash Functions

A **hash function** h maps objects x to integers mod m for some m ; h must be a regular function (see the “Background” chapter) and should have few **collisions**, where for $x \neq y$, $h(x) = h(y)$. Theoretically, need h only for variable-length byte arrays because can interpret any object as such, but for specific data types have more efficient h . Hash other types such as strings by converting them into an efficiently hashable type.

Unfortunately the simple $h(x) = x \% m$ for integers is dangerous. For general objects, such h would concatenate a few nonzero bytes to form an integer x . In the obvious cases the numbers x differ only in the lower bits, which is the ideal use case. Some sources even recommend such h as fast in practice under the justification of **fairness**—i.e., any value is as likely as any other when hashing all possible x . But performance can degrade dramatically, e.g., if switch from a little-endian system to a big-endian one. Or if multiply all x by some factor for some application-specific convenience. In general h must look at all bytes of x and extract unique information content to create a mostly unique id. Speed of h usually isn't the performance bottleneck, but using a bad but computationally fast h is likely to create one.

For a set of objects of size $n > m$, all h hash at least n/m of them to the same value. Every h hashes all x to the same value in the worst case, so evaluate the performance based on typical behavior. To be useful, h must evaluate in $O(1)$ time with respect to n . For x of size s want $O(s)$ time. Also want to avoid collisions on average, so a **random** h is theoretically ideal due to not having any biases. h is:

- **Universal** if it's initialized with random seed a , calculates $h(x) = f(x, a)$, and $\forall x \neq y$, $\Pr(h(x)=h(y)) < \frac{\text{constant}}{m}$. Then for integer k and any x , $\Pr(h(x)=k) < \frac{\text{constant}}{m}$. A random seed doesn't break regularity because it's stored and reused. As discussed later, this is as close as can get in practice to random.
- **Rolling** if used to hash an array of objects, and after changing a single object can update the array hash in $O(s)$ time.
- **Portable** if it produces the same value \forall architecture and word size. This usually isn't a concern unless store h values, but any dependence on the word size or implicit overflow handling leads to unportability.

Most hash functions have the form $h(x) \% m$ for some m , and some applications don't need the “% m ” or can replace it by the slightly faster $\&(m - 1)$ when $m = 2^b$. Here x is typically a number or an array of numbers, but also want to automatically hash other build-in types. So for the implementation:

- A basic h must support integers and arrays of integers of the word type it supports.
- Arrays of `char` are packed to fill the word and hashed as arrays of words. Larger integers are broken up into arrays of words if exact multiples. Others are cast to arrays of `char` and hashed as such. Lose portability for efficiency though.
- Instead of the arrays support more general builder objects that are helpful when need to create h for custom types.
- Separate wrappers handle the “% m ” and the casts above.
- Every h knows its maximum value, and for wrappers $m - 1$ must not exceed it.

```
template <typename HASHER> class MHash
{
    unsigned long long m;
    HASHER h;
public:
    MHash(unsigned long long theM) : m(theM)
        {assert(theM > 0 && theM <= h.max());}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
```

```

unsigned long long max() const{return m - 1;}
unsigned long long operator()(WORD_TYPE const& x) const{return h(x) % m;}
typedef typename HASHER::Builder Builder;
Builder makeBuilder() const{return h.makeBuilder();}
unsigned long long operator()(Builder b) const{return h(b) % m;}
};

template<typename HASHER> class BHash
{
    unsigned long long mask;
    HASHER h;
public:
    BHash(unsigned long long m): mask(m - 1){assert(m > 0 && isPowerOfTwo(m));}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max() const{return mask;}
    unsigned long long operator()(WORD_TYPE const& x) const{return h(x) & mask;}
    typedef typename HASHER::Builder Builder;
    Builder makeBuilder() const{return h.makeBuilder();}
    unsigned long long operator()(Builder b) const{return h(b) & mask;}
};

```

Another wrapper makes basic *h* efficiently applicable to many data types by hashing *x* as number whenever possible and otherwise trying to deduce its smallest-size array representation. But beware that for some types objects with the same value can have different hashes when interpreted as POD:

- For a *double*, $-0 = 0$, and several bit patterns represent NaN
- For a class type a hidden attribute can influence the hash value
- Structure types can have garbage in padding bytes

So no automatic cast-based hashing is safe in general, and for silliness safety only word types are allowed to be hashed blindly. Using a builder, this is the best reasonable strategy for automatic hashing—use custom *h* if want slightly better performance for specialized types.

```

template<typename HASHER> class EHash
{//takes special care to avoid template substitution compile errors
    HASHER h;
    template<typename WORD> unsigned long long hash(WORD x, true_type) const
        {//integral type - hash as word if possible
            if(sizeof(WORD) <= sizeof(WORD_TYPE)) return h(x);
            return operator()(&x, 1); //word too big, will break in chunks
        }
public:
    EHash() {} //for h that uses no m
    EHash(unsigned long long m): h(m) {}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max() const{return h.max();}
    template<typename WORD> unsigned long long operator()(WORD x) const
        {return hash(x, is_integral<WORD>());}//integral words only
    class Builder
    {
        enum{K = sizeof(WORD_TYPE)};
        union
        {
            WORD_TYPE xi;
            unsigned char bytes[K];
        };
        int byteIndex;
        typename HASHER::Builder b;
        friend EHash;
        Builder(EHash const& eh): xi(0), byteIndex(0), b(eh.h.makeBuilder()) {}
        template<typename WORD> void add(WORD const& xi, true_type)
            {//only word type supported for safety
                if(sizeof(WORD) & sizeof(WORD_TYPE) == 0)
                    //exact multiple, add as word type
                    for(int i = 0; i < sizeof(WORD)/sizeof(WORD_TYPE); ++i)
                        b.add(((WORD_TYPE*)&xi)[i]);
            }
    };
}
```

```

        else//as char sequence
            for(int i = 0; i < sizeof(xi); ++i)
                add(((unsigned char*)&xi)[i]);
    }
public:
    void add(unsigned char bi)
    {
        bytes[byteIndex++] = bi;
        if(byteIndex >= K)
        {
            byteIndex = 0;
            b.add(xi);
            xi = 0;
        }
    }
    template<typename WORD> void add(WORD const& xi)
        {add(xi, is_integral<WORD>());}//integral words only
    typename HASHER::Builder operator()()
    {//finalize remaining xi if any
        if(byteIndex > 0) b.add(xi);
        return b;
    }
};

Builder makeBuilder() const{return Builder(*this);}
unsigned long long operator()(Builder b) const{return h(b());}
template<typename WORD>
unsigned long long operator()(WORD* array, int size) const
{
    Builder b(makeBuilder());
    for(int i = 0; i < size; ++i) b.add(array[i]);
    return operator()(b);
}
};

```

For data types that aren't integral words or arrays of words, need to create custom h using the builders provided. Here want to use maximum value of the object. Some less obvious information sources:

- For sequences also hash the size
- For unions also hash the selection of the current member
- For pointers also hash the null status

Any type that is hashed must function as a meaningful id, so anything containing, e.g., a `double` needs a good reason to be hashed. Have the below for a vector of integral types and string (`BUHash` is discussed later):

```

template<typename HASHER = EHash<BUHash>> class DataHash
{
    HASHER h;
public:
    DataHash(unsigned long long m): h(m) {}
    typedef typename HASHER::WORD_TYPE WORD_TYPE;
    unsigned long long max() const{return h.max();}
    unsigned long long operator()(string const& item) const
        {return h(item.c_str(), item.size());}
    unsigned long long operator()(Bitset<> const& item)
        const{return h(item.getStorage().getArray(), item.wordSize());}
    template<typename VECTOR> unsigned long long operator()(VECTOR const& item)
        const{return h(item.getArray(), item.getSize());}
    typedef EMPTY Builder;
};

```

9.3 Universal Hash Functions

Theorem A: If h is universal with constant c , and hash n items, $E[\text{the number of items hashing to the same bin}] \leq cn$

value] < $\frac{cn}{m}$. Proof: Let $X_i = (h(x_i) = k)$. $E[\text{the number of items hashing to } k] = E[\sum X_i] = \sum \Pr(h(x_i)=k) < \sum \frac{c}{m} = \frac{cn}{m}$. \square

Lagrange's theorem (Wikipedia 2018b): A polynomial $\% p$ of degree k , where p is prime, has $\leq k$ solutions mod p if one or more coefficients aren't divisible by p , and $k < p$. It doesn't matter if some coefficients are negative because $\% p$ makes them positive.

Theorem B: For a variable-length array of numbers x , prime p , $m \leq p$, and random seed $a \in [0, p-1]$, $h(x) = (\sum x_i a^i) \% p \% m$ is universal with $c = 2k$, where k is the length of the longest hashed array (here indexing of x is one-based).

Proof: Let $x \neq y$ be variable-length arrays of length $\leq k$. Then $h(x) = h(y) \leftrightarrow (\sum x_i a^i) \% p = (\sum y_i a^i) \% p + qm$, for $qm \in (-p, p) \leftrightarrow (\sum (x_i - y_i) a^i \pm qm) \% p = 0$, with any x_i or y_i that don't exist due to length difference set to 0. This is a polynomial $\% p$ with coefficients $(x_i - y_i)$ and $\pm qm$. By Lagrange's theorem, because $\pm qm$ isn't divisible by p , $\forall q \exists \leq k a$ choices that cause equality for a given $\pm q$. $\exists \leq 2 \frac{p}{m}$ values of $\pm q$, so the fraction of a that cause equality $\leq \frac{2kp/m}{p} = 2 \frac{k}{m}$. \square

- The proof holds only if the calculations don't overflow. Use 64 bits with $x_i, p < 2^{32}$, and compute a^i incrementally mod p .
- If $m = p$, don't need the " $\% m$ ", and h is rolling because can add or subtract the $x_i a^i$ terms in $O(\lg(i))$ time, computing a^i using efficient modular exponentiation (see the "Large Numbers" chapter).
- $k = 1$ leads to a universal h for integers with $c = 2$.
- c is large for long strings.
- Can use h for error detection because only need the seed to rehash. CRC (see the "Miscellaneous Algorithms" chapter) has larger error detection probability, but using several such h makes it arbitrary close to 1.

```
class PrimeHash2
{
    static uint32_t const PRIME = (ull << 32) - 5;
    uint32_t seed;

public:
    PrimeHash2(): seed((GlobalRNG().next() | 1) % PRIME) {} //ensure non-0
    typedef uint32_t WORD_TYPE;
    unsigned long long max() const {return PRIME - 1;}
    unsigned long long operator()(WORD_TYPE x) const
    {return (unsigned long long)seed * x % PRIME;}
    class Builder
    {
        unsigned long long sum;
        WORD_TYPE seed;
        friend PrimeHash2;
        Builder(WORD_TYPE theSeed): sum(0), seed(theSeed) {}
    public://unlikely possible overflow from add & mult but that's ok
        void add(WORD_TYPE xi){sum = seed * (sum + xi) % PRIME;}
    };
    Builder makeBuilder() const {return Builder(seed);}
    unsigned long long operator()(Builder b) const {return b.sum;}
};


```

Theorem C: For variable-length arrays of numbers x , a prime p , $m \leq p$, and k random seeds $a \in [0, p-1]$, where k is the length of the longest hashed vector, $h(x) = (\sum a_i x_i) \% p \% m$ is universal with $c = 2$.

Proof: Let $x \neq y$ be variable-length arrays of length $\leq k$. Then $h(x) = h(y) \leftrightarrow (\sum a_i x_i) \% p = (\sum a_i y_i) \% p + qm$, for $qm \in (-p, p) \leftrightarrow (a_j(y_j - x_j) + \sum_{i \neq j} a_i(x_i - y_i) \pm qm) \% p = 0$, where j is such that $y_j - x_j \neq 0$. Think of a_i for $i \neq j$ as constants, so by Lagrange's theorem the resulting linear polynomial with coefficients $y_j - x_j$ and $\sum_{i \neq j} a_i(x_i - y_i) \pm qm$ has at most one solution for given $\pm q$ and a_j . $\exists \leq 2 \frac{p}{m}$ values of $\pm q$ and p^{k-1} values of a_i with $i \neq j$, so the fraction of $\{a_i\}$ that cause equality $< \frac{2p}{m} \frac{p^{k-1}}{p^k} = \frac{2}{m}$. \square

- The proof holds only if the calculations don't overflow. Use 64 bits with x_i , and need $p < 2^{32}/k$, but in practice can get away with $p < 2^{32}$.
- If $m = p$, don't need the "% m ".
- $k = 1$ leads to a universal h for integers with $c = 2$.

Due to pseudorandom generation of a_i from a single-word seed, the proof doesn't hold for the implementation because it's no longer mathematically possible to isolate arbitrary a_i . But due to the entropy of the pseudorandom generator with respect to the working of h , about the same fraction of seed and q choices should lead to the solution of the equation. Though don't have proof of a constant better than that of theorem B, a mapping from a good pseudorandom generator doesn't seem worse than the powering. So expect a better constant, and only need "% m " once.

```
class PrimeHash
{
    static uint32_t const PRIME = (ull << 32) - 5;
    uint32_t seed;

public:
    PrimeHash(): seed((GlobalRNG().next() + 1) % PRIME) {} //ensure non-0
    typedef uint32_t WORD_TYPE;
    unsigned long long max() const{return PRIME - 1;}
    unsigned long long operator()(WORD_TYPE x) const
        {return (unsigned long long)seed * x % PRIME;}
    class Builder
    {
        unsigned long long sum;
        WORD_TYPE a;
        friend PrimeHash;
        Builder(WORD_TYPE theSeed): sum(0), a(theSeed) {}
    public:
        void add(WORD_TYPE xi)
            {//unlikely possible overflow from adding but that's ok
            sum += (unsigned long long)a * xi;
            a = xorshiftTransform(a);
        }
    };
    Builder makeBuilder() const{return Builder(seed);}
    unsigned long long operator()(Builder b) const{return b.sum % PRIME;}
};
```

Theorem D: For w -bit integers, $m = 2^b$, and odd random $a \in [0, 2^w - 1]$, $h(x) = \frac{xa \% 2^w}{2^{w-b}}$ is universal even if xa overflows (Ditzfelbinger et al. 1997). Combining the h from theorems C and D gives a very efficient combined universal h for $m = 2^b$. Because the former works with 32-bit words, also implement the latter this way even though it supports 64 bits. Though not an issue in practice, beware that small a hash small x to 0.

```
class BUHash
{
    uint32_t a, wLB;
    BHash<PrimeHash> h;
public:
    BUHash(unsigned long long m): a(GlobalRNG().next() + 1), //ensure non-0
        wLB(32 - lgCeiling(m)), h(m) {assert(m > 0 && isPowerOfTwo(m));}
    typedef uint32_t WORD_TYPE;
    unsigned long long max() const{return h.max();}
    uint32_t operator()(WORD_TYPE const& x) const{return (a * x) >> wLB;}
    typedef BHash<PrimeHash>::Builder Builder;
    Builder makeBuilder() const{return h.makeBuilder();}
    unsigned long long operator()(Builder b) const{return h(b);}
};
```

9.4 Nonuniversal Hash Functions

Sometimes h must always give the same result for the same x and so can't be universal. E.g., in Java, every object's `hashCode` gives the same hash (unlike `unordered_map` of C++, which allows to pass a seed

argument to the constructor). High-quality such h do well on (Henke et al. 2008; 2009):

- **Avalanche**—changing any input bit changes each hash bit with 50% probability
- **Bias**—inputs of any length and value hash equally likely into any range
- **Collisions**— $E[\text{the number}] \approx \text{that of a random } h$

For a high-quality **FNV hash function** for variable-length byte arrays, $hC(x, i) = \begin{cases} a, & i=0 \\ (hC(x, i-1)b)^{\wedge}x[i] \end{cases}$.

where $a, b = 2166136261, 16777619$ for 32 bits and $14695981039346656037, 1099511628211$ for 64 (Wikipedia 2018a). The values of b are special primes and of a arbitrary. Only the 32-bit version seems to have been extensively tested by Henke et al. (2008, 2009) and Valloud (2008), but the 64-bit one should be of similar quality.

```
struct FNVHash
{
    typedef unsigned char WORD_TYPE;
    unsigned long long max() const{return numeric_limits<uint32_t>::max(); }
    uint32_t operator()(WORD_TYPE const& x) const
    {
        Builder b(makeBuilder());
        b.add(x);
        return b.sum;
    }
    class Builder
    {
        uint32_t sum;
        friend FNVHash;
        Builder(): sum(216613626lu) {}
    public://unlikely possible overflow from add & mult but that's ok
        void add(WORD_TYPE xi){sum = (sum * 16777619) ^ xi;}
    };
    Builder makeBuilder() const{return Builder();}
    uint32_t operator()(Builder b) const{return b.sum;}
};

struct FNVHash64
{
    typedef unsigned char WORD_TYPE;
    unsigned long long max() const{return numeric_limits<uint64_t>::max(); }
    uint64_t operator()(WORD_TYPE const& x) const
    {
        Builder b(makeBuilder());
        b.add(x);
        return b.sum;
    }
    class Builder
    {
        uint64_t sum;
        friend FNVHash64;
        Builder(): sum(14695981039346656037ull) {}
    public:
        void add(WORD_TYPE xi){sum = (sum * 1099511628211ull) ^ xi;}
    };
    Builder makeBuilder() const{return Builder();}
    uint64_t operator()(Builder b) const{return b.sum;}
};
```

$h_i(x) = \begin{cases} 0, & i=0 \\ f(\text{combine}(h_{i-1}(x), x[i])) \end{cases}$ is the form for a variable-length-array and computable incrementally h . Prefer this over $\text{combine}(f(h_{i-1}(x)), x[i])$ because for simple combiners the last byte doesn't affect all bits of h . Obvious f are random generator transitions and obvious combiners operators "+" and " \wedge ". A combiner should be different from f . For efficiency h should hash arrays of integers and not byte-by-byte, but sometimes can't avoid it, like for FNV.

The **Xorshift hash function** uses the QualityXorshift64 transition and combiner operator "+".

```
struct Xorshift64Hash
```

```

{
    typedef uint64_t WORD_TYPE;
    unsigned long long max() const {return numeric_limits<WORD_TYPE>::max(); }
    uint64_t operator()(WORD_TYPE x) const
        {return QualityXorshift64::transform(x);}
    class Builder
    {
        uint64_t sum;
        friend Xorshift64Hash;
        Builder(): sum(0) {}
    public:
        void add(WORD_TYPE xi) {sum = QualityXorshift64::transform(sum + xi);}
    };
    Builder makeBuilder() const {return Builder();}
    uint64_t operator()(Builder b) const {return b.sum;}
};

```

The following general properties are different and can't be converted to each other:

- Universality—any two items won't collide with high probability
- Avalanche and bias—exploit entropy of the data as much as possible
- Pseudorandom generator lack of correlation—a sequence property similar to avalanche, but nearby numbers can transition to nearby numbers

So until further study, the Xorshift hash function, despite excellent avalanche and fully explained mechanics by design, is only an illustration of the general construction—for the nonuniversal use case always want a well-studied h such as FNV.

9.5 Rolling Hash Functions

The fastest and most convenient rolling h is the **table hash function** (also called **Zobrist hashing**). It interprets x as a variable-length byte array. Then $h(x) = \text{XOR} \sum_{i=0}^n \text{table}[x_i]$, where table has size 256 and holds random numbers. Rolling uses hash $\text{hash} \leftarrow \text{table}[\text{old}] \text{ ^ } \text{table}[\text{new}]$ and needs $O(1)$ time. All byte permutations of x hash to the same number, making h unsuitable for general use.

```

class TableHash
{
    enum {N = 1 << numeric_limits<unsigned char>::digits};
    unsigned int table[N];
public:
    TableHash(){for(int i = 0; i < N; ++i) table[i] = GlobalRNG().next();}
    typedef unsigned char WORD_TYPE;
    unsigned long long max() const {return numeric_limits<unsigned int>::max();}
    unsigned int operator()(WORD_TYPE const& x) const
    {
        Builder b(makeBuilder());
        b.add(x);
        return b.sum();
    }
    unsigned int update(unsigned int currentHash, unsigned char byte)
        const {return currentHash ^ table[byte];}//for both add and remove
    class Builder
    {
        unsigned long long sum;
        TableHash const& h;
        friend TableHash;
        Builder(TableHash const& theH): sum(0), h(theH) {}
    public: //unlikely possible overflow from add & mult but that's ok
        void add(unsigned char xi){sum ^= h.table[xi];}
    };
    Builder makeBuilder() const {return Builder(*this);}
    unsigned long long operator()(Builder b) const {return b.sum;}
};

```

Board games, a major application, hash piece-square pairs. A board generally consists of squares with at most one piece each. E.g., chess has $64 \times (2 \times 6 + 1) = 832$ possible piece-squares. Can represent each with

two bytes, but it's more efficient and higher quality to use *table* of size 832. If \exists many piece-squares, to avoid *table* pick a hash function g , and use $h(x) = \text{XOR} \sum g(x_i)$.

9.6 Collection of Hash Functions

Need k hash functions, e.g., for a Bloom filter (discussed later in this chapter), for $k > 2$. Here using a set generated by $h_i = h_1 + ih_2$ gives sufficiently random results and is more efficient than hashing k times (Kirsch & Mitzenmacher 2008).

9.7 Hash Tables

A hash table efficiently supports map operations by placing objects x into an array of size m at location $h(x)$. Duplicates aren't allowed and must be handled by external logic. With n items, m/n is the **load factor**. Hash tables differ by when to resize and how to resolve **collisions**—i.e., when several x land in the same location. For all reasonable implementations $E[\text{the runtime of a map operation}] = O(1)$. It's impossible to have a map of size $O(n)$ with both amortized $O(1)$ find and insert (Ditzelbinger et al. 1994). So the worst case is usually $O(n)$. Beware that if using h with a random seed, \forall seed the iteration order of items is different, which can cause problems when computing order-sensitive functions such as max.

Due to more efficient h table sizes are powers of two.

9.8 Chaining Hash Table

Each array cell has a linked list with items that hashed to the corresponding index. Unlike in some other presentations, the array only has pointers to the list nodes and not the items. Need this for address persistence under resizing.

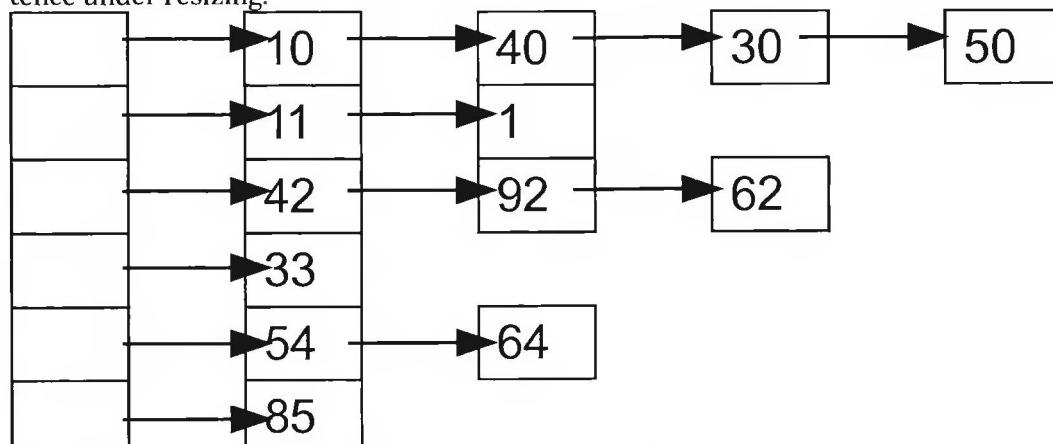


Figure 9.1: Chaining hash table structure with $h(x) = x \% 10$

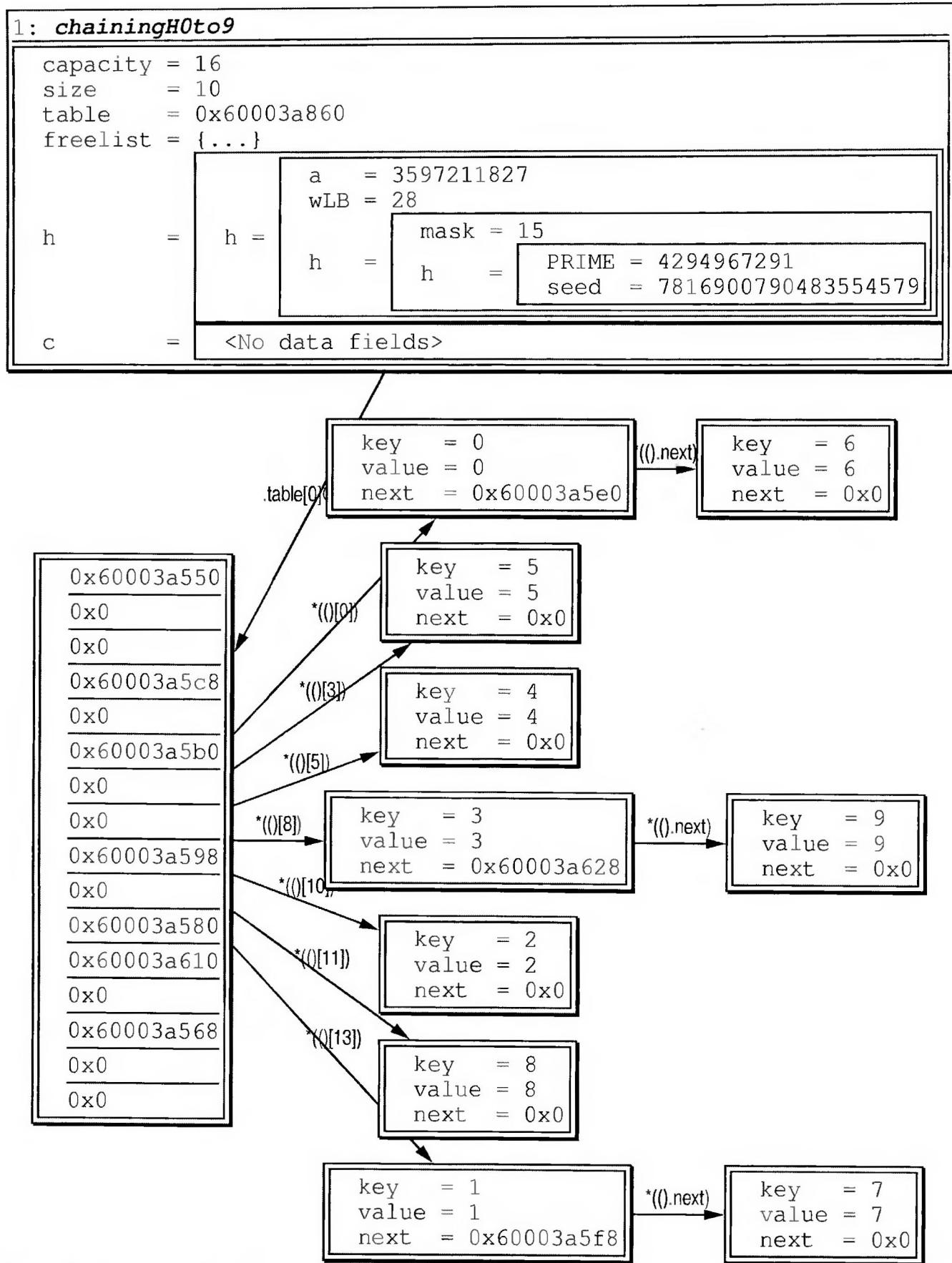


Figure 9.2: Memory layout of a chaining hash table with integer items 0-9

```

template<typename KEY, typename VALUE, typename HASHER = EHash<BUHash>,
        typename COMPARATOR = DefaultComparator<KEY>> class ChainingHashTable
{
    int capacity, size; //capacity must be before size
    struct Node

```

```

{
    KEY key;
    VALUE value;
    Node* next;
    Node(KEY const& theKey, VALUE const& theValue): key(theKey),
        value(theValue), next(0) {}

    ** table;
    Freelist<Node> f;
    Hasher h;
    Comparator c;
    enum{MIN_CAPACITY = 8}; //for efficiency require at least size 8
    void allocateTable()
    {
        h = Hasher(capacity);
        table = new Node*[capacity];
        for(int i = 0; i < capacity; ++i) table[i] = 0;
    }
public:
    typedef Node NodeType;
    int getSize() {return size;}
    ChainingHashTable(int initialCapacity = 8, Comparator const&
        theC = Comparator()): capacity(nextPowerOfTwo(max<int>(initialCapacity,
            MIN_CAPACITY))), c(theC), h(capacity), size(0) {allocateTable();}
    ChainingHashTable(ChainingHashTable const& rhs): capacity(rhs.capacity),
        size(rhs.size), h(rhs.h), table(new Node*[capacity]), c(rhs.c)
    {//copy just mirrors the source, without trying to compact
        for(int i = 0; i < capacity; ++i)
        {
            table[i] = 0;
            Node** target = &table[i];
            for(Node* j = rhs.table[i]; j; j = j->next)
            {
                *target = new(f.allocate()) Node(*j);
                target = &(*target)->next;
            }
        }
    }
    ChainingHashTable& operator=(ChainingHashTable const& rhs)
    {
        return genericAssign(*this, rhs);
    }
    ~ChainingHashTable() {delete[] table;}
};

Map operations hash the key, linearly search the list for the item with the key, and update the list if inserting/removing.

```

```

Node** findPointer(KEY const& key)
{//for code reuse get pointer to node pointer
    Node** pointer = &table[h(key)];
    for(;*pointer && !c isEqual((*pointer)->key, key);
        pointer = &(*pointer)->next);
    return pointer; //if not found return pointer to next of last node
}
//chaining has node persistence, so alloc pointer return
Node* findNode(KEY const& key) {return *findPointer(key);}
VALUE* find(KEY const& key)
{
    Node* next = findNode(key);
    return next ? &next->value : 0;
}

```

Insert updates the value if found and appends to the list otherwise, doubling if the load factor ≥ 1 . If h is universal, by theorem A with $O(1)$ load factor $E[\text{list size}] = O(1)$, so $E[\text{the runtime of a map operation}] = O(1)$.

```

Node* insert(KEY const& key, VALUE const& value)
{

```

```

Node** pointer = findPointer(key);
if(*pointer)
    //already exists, just update value
    (*pointer)->value = value;
    return *pointer;
}
else
{
    Node* node = *pointer = new(f.allocate())Node(key, value);
    if(++size >= capacity) resize(); //not > where will have x4 size
    return node;
}
}

```

Resize iterates over the old table and copies all nodes into the new one of $2 \times$ the size. $E[\text{the runtime}] = O(n)$, and this happens at most every $O(n)$ insertions, giving amortized expected $O(1)$ insertion. After a resize, $a = 0.5$, and node pointers don't change.

```

void resize()
{
    int oldCapacity = capacity;
    Node** oldTable = table;
    capacity = nextPowerOfTwo(size * 2);
    allocateTable();
    for(int i = 0; i < oldCapacity; ++i)
        for(Node* j = oldTable[i], *tail; j; j = tail)
        {
            tail = j->next;
            j->next = 0;
            *findPointer(j->key) = j; //insert node
        }
    delete[] oldTable;
}

```

Remove cuts out the found node and resizes if $a < 0.1$, which seems to be a reasonable balance between the space use and the frequency of resizing:

```

void remove(KEY const& key)
{
    Node** pointer = findPointer(key);
    Node* i = *pointer;
    if(i)
        //found
        *pointer = i->next;
        f.remove(i);
        if(--size < capacity * 0.1 && size * 2 >= MIN_CAPACITY) resize();
    }
}

```

The runtime is also amortized expected $O(1)$. Iteration loops over the array, skipping empty cells and going into and looping over each nonempty cell's list. Going through all items takes $O(n)$ time. Only forward iteration is supported because don't have order.

```

class Iterator
{
    int i; //current cell index
    Node* node; //node in cell
    ChainingHashTable& t;
    friend ChainingHashTable;
    void advanceCell() //if at null node and not at end, try next cell
    {
        if(!node) while(i + 1 < t.capacity && !(node = t.table[+i]));
        Iterator(ChainingHashTable& theHashTable, int theI = -1): i(theI),
            node(0), t(theHashTable) {advanceCell();}
    }
    public:
        Iterator& operator++()
        {
            assert(node);
        }
}

```

```

        node = node->next;
        advanceCell();
        return *this;
    }
    NodeType& operator*() {assert (node); return *node; }
    NodeType* operator->() {assert (node); return node; }
    bool operator==(Iterator const& rhs) const {return node == rhs.node; }
};

Iterator begin() {return Iterator(*this); }
Iterator end()
{
    Iterator result(*this, capacity);
    return result;
}

```

With random h , $\Pr(\text{an item lands in a given cell}) = 1/m$. Let $k = \text{the list size in a given cell after inserting } n \text{ items}$, which $\sim \text{binomial}(k, 1/m, n) \approx \text{Poisson}(k, a)$. The latter has PDF = $\frac{a^k}{k! e^a}$ (Wikipedia 2018c). So:

- $E[\text{the list size}] = a$
- $E[\text{the \% of cells with } k \text{ items}] = \text{PoissonPDF}(k, a)$, which for $a = 1$ is ≈ 0.37 for $k = 0$ and 1 , 0.18 for 2 , 0.06 for 3 , etc.
- $\Pr(\text{the list size} = l \geq k) = \sum_{k \leq l \leq \infty} \text{PoissonPDF}(l, a) = \sum_{k \leq l \leq \infty} \frac{a^l}{l! e^a} = \frac{a^k}{k!} \sum_{0 \leq l \leq \infty} \frac{a^l}{l! e^a} = \frac{a^k}{k!} < 10^{-12}$ for $k = 15$ and $a = 1$

At the expense of higher constant factors in both memory use and expected runtime the chaining collision resolution (discussed later) allows using dynamic sorted sequences instead of linked lists to guarantee $O(\lg(n))$ runtimes. This doesn't seem worth it because superconstant runtimes are exponentially unlikely with good h .

9.9 Linear Probing Hash Table

Disadvantages of chaining are space for links, use of a free list, and clumsy iteration. Linear probing avoids these by using an array of items and an array of Booleans to mark the occupied cells. Collided items go to the next free cell, like books in library shelves. The probing sequence is cache-oblivious (see the "External Memory Algorithms" chapter) and allows deletions.

1	10	
1	11	
1	40	
0		
1	54	
1	64	

Figure 9.3: Linear probing hash table structure—the left column is the Booleans, and the right the items

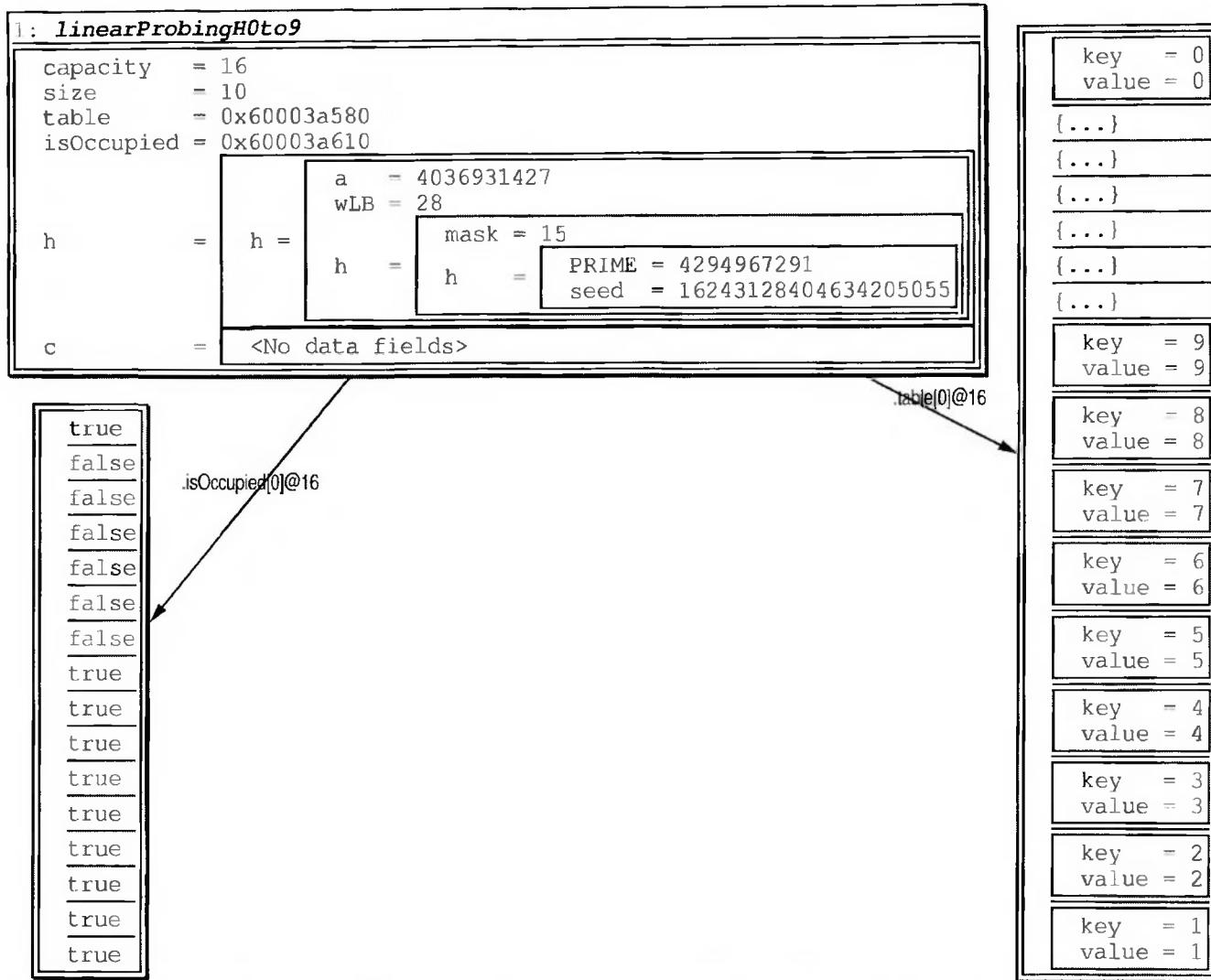


Figure 9.4: Memory layout of a linear probing hash table with integer items 0–9

```

template< typename KEY, typename VALUE, typename HASHER = EHash<BUHash>,
typename COMPARATOR = DefaultComparator<KEY> > class LinearProbingHashTable
{
    int capacity, size; //capacity must be before h
    typedef KVPair<KEY, VALUE> Node;
    Node* table;
    bool* isOccupied;
    HASHER h;
    COMPARATOR c;
    enum{MIN_CAPACITY = 8}; //for efficiency require at least size 8
    void allocateTable()
    { //create an unoccupied table of size capacity
        h = HASHER(capacity);
        size = 0;
        table = rawMemory<Node>(capacity);
        isOccupied = new bool[capacity];
        for(int i = 0; i < capacity; ++i) isOccupied[i] = false;
    } //helper to remove an unused table
    static void cleanUp(Node* theTable, int theCapacity, bool* isOccupied)
    { //destruct the occupied nodes, and deallocate the arrays
        for(int i = 0; i < theCapacity; ++i)
            if(isOccupied[i]) theTable[i].~Node();
        rawDelete(theTable);
        delete[] isOccupied;
    }
public:
    typedef Node NodeType;
    int getSize() {return size;}
}

```

```

LinearProbingHashTable(int initialCapacity = 8, COMPARATOR const& theC =
    COMPARATOR()): capacity(nextPowerOfTwo(max<int>(initialCapacity,
    MIN_CAPACITY))), c(theC), h(capacity) {allocateTable();}
LinearProbingHashTable(LinearProbingHashTable const& rhs):
    capacity(rhs.capacity), h(rhs.h), size(rhs.size), c(rhs.c),
    isOccupied(new bool[capacity]), table(rawMemory<Node>(capacity))
//copy just mirrors the source, without trying to compact
    for(int i = 0; i < capacity; ++i)
        if(isOccupied[i] = rhs.isOccupied[i]) table[i] = rhs.table[i];
}
LinearProbingHashTable& operator=(LinearProbingHashTable const& rhs)
{
    return genericAssign(*this, rhs);
~LinearProbingHashTable(){cleanUp(table, capacity, isOccupied);}
};

```

Map operations find index i of the cell where the item should be. If a different item is there, go to cell $(i+1)\%m$ until find an empty one.

```

int findNode(KEY const& key)
{//find the cell where the key would belong if inserted
    int cell = h(key);
    for(;isOccupied[cell] && !c isEqual(key, table[cell].key);
        cell = (cell + 1) % capacity);
    return cell;
}
VALUE* find(KEY const& key)
{
    int cell = findNode(key);
    return isOccupied[cell] ? &table[cell].value : 0;
}

```

Insert puts the item in the next available cell or updates the value of the found item. With random h , $E[\text{the number of probes for a successful find}] = \frac{1}{2} \left(1 - \frac{1}{1-a}\right)$. For an unsuccessful find or insert it's $\frac{1}{2} \left(1 - \frac{1}{(1-a)^2}\right)$ (Cormen et al. 2009). E.g., these are respectively 3 and 13 for $a = 0.8$. Don't return a value handle because a future resizing will invalidate it.

```

void insert(KEY const& key, VALUE const& value)
{
    int cell = findNode(key);
    if(isOccupied[cell]) table[cell].value = value; //update
    else
        {//insert
            new(&table[cell]) Node(key, value);
            isOccupied[cell] = true;
            if(++size > capacity * 0.8) resize(); //resize if reach a
        }
}

```

Due to resizing insert and delete are amortized expected $O(1)$, as for chaining. Resize changes a to 0.5, and reinserts all occupied items.

```

void resize()
{
    int oldCapacity = capacity;
    Node* oldTable = table;
    bool* oldIsOccupied = isOccupied;
    capacity = nextPowerOfTwo(size * 2);
    allocateTable();
    for(int i = 0; i < oldCapacity; ++i) //reinsert
        if(oldIsOccupied[i]) insert(oldTable[i].key, oldTable[i].value);
    cleanUp(oldTable, oldCapacity, oldIsOccupied); //remove old table
}

```

Remove removes the found item and, because this can break the probing for the next items until an unoccupied cell, deletes and reinserts them, resizing if $a < 0.1$. The runtime is expected $O(1)$ because

$E[\text{chain length}]$ is and worst-case quadratic in the chain's length.

```

void destroy(int cell)
{
    table[cell].~Node();
    isOccupied[cell] = false;
    --size;
}
void remove(KEY const& key)
{
    int cell = findNode(key);
    if(isOccupied[cell])
        //reinsert subsequent nodes in the found value's chain
        destroy(cell); //remove item
        if(size < capacity * 0.1 && size * 2 >= MIN_CAPACITY) resize();
        else //reinsert chain
            while(isOccupied[cell = (cell + 1) % capacity])
            {
                Node temp = table[cell];
                destroy(cell); //destroy item
                insert(temp.key, temp.value); //reinsert it
            }
    }
}

```

Going through all items takes $O(n)$ time and is cache-efficient.

```

class Iterator
{
    int i; //current cell index
    LinearProbingHashTable& t;
    friend LinearProbingHashTable;
    void advance() while(i < t.capacity && !t.isOccupied[i]) ++i;
    Iterator(LinearProbingHashTable& theHashTable, int theI = 0): i(theI),
        t(theHashTable) {advance();}
public:
    Iterator& operator++()
    {
        ++i;
        advance();
        return *this;
    }
    NodeType& operator*() const{assert(i < t.capacity); return t.table[i];}
    NodeType* operator->() const{assert(i < t.capacity); return &t.table[i];}
    bool operator==(Iterator const& rhs) const{return i == rhs.i;}
};
Iterator begin(){return Iterator(*this);}
Iterator end()
{
    Iterator result(*this, capacity);
    return result;
}

```

A universal h doesn't guarantee $E[\text{the runtime of a map operation}] = O(1)$ because the implicit lists of items hashing to the same cell aren't independent. But joint randomness of a universal h and the data usually give performance of a random h (Mitzenmacher & Vadhan 2008). Can replace the Boolean array by a slower bit vector—from my rough timings, if the key and the item take 8 bytes, this saves $\approx 10\%$ memory for $\approx 50\%$ slowdown.

9.10 Timings

A 32-bit machine was used here, and a 64-bit one would favor 64-bit h .

Hasher	Int	Str10	Ch	+-	Min	Max	LP	+-	Min	Max	Ch	+-	Min	Max	LP	+-	Min	Max
E-BU	0.63	13.11	0.36	0.04	0.25	1.56	0.21	0.03	0.13	1.218	0.18	0	0.17	0.25	0.218	0.01	0.187	0.469
E-B-Prime	6	12.92	0.45	0.02	0.36	1.19	0.33	0.03	0.22	1.485	0.18	0	0.17	0.2	0.219	0.01	0.187	0.515
E-B-Prime2	6	21.6	0.44	0.01	0.31	1	0.36	0.06	0.22	3.14	0.3	0	0.28	0.34	0.334	0.03	0.296	1.985
E-B-FNV	2.12	4.015	0.35	0	0.33	0.44	0.15	0	0.14	0.172	0.09	0	0.08	0.11	0.1	0	0.093	0.11
E-M-FNV	8.84	4.953	0.45	0	0.44	0.47	0.29	0	0.27	0.312	0.1	0	0.09	0.11	0.111	0	0.094	0.125
E-B-FNV64	2.11	4.031	0.34	0	0.33	0.36	0.15	0	0.14	0.157	0.09	0	0.08	0.11	0.1	0	0.093	0.11
E-B-X64	3.06	11.33	0.29	0	0.27	0.34	0.17	0	0.16	0.172	0.16	0	0.16	0.17	0.183	0	0.171	0.219
E-B-Table	1.89	4.094	0.48	0	0.47	0.5	0.45	0	0.44	0.469	0.08	0	0.08	0.09	0.113	0	0.109	0.125

Figure 9.5: Test results—the times are in seconds for 10^6 and 10^5 insertions for respectively `int`s and `Struct10` objects of 10 `int`s, and 10 times more finds. For only hashing performance `int` and `Struct10` objects were hashed respectively 10^9 and 10^8 times. The best and near-best results are highlighted. The hash table tests ran 100 times, and the results are the means. The “+‐” is the 95% confidence standard error on the means, and min/max are reported as well.

Based on the timings, use E-BU as default. The m-hashes are slower, particularly for integers. Chaining vs linear probing:

- Chaining has item persistence—the C++ standard requires this for `unordered_map`
- Linear probing uses less memory for small items
- Chaining has better collision resolution and expected $O(1)$ guarantee with a universal h
- Memory allocator quality and use of a free list affect chaining speed
- Linear probing has faster iteration

So use chaining by default and linear probing for small items to cut memory use.

9.11 Bloom Filter

A Bloom filter supports only insert and correct-with-high-probability `isInserted`. It has a fixed size but uses much less space than a hash table. This is useful, e.g., to reduce the memory of word dictionaries for spell checking. The use case is rare, but see Wikipedia (2019) for some applications.

It consists of a bit set of size m and k hash functions. To insert x , set each $h_i(x)$ bit. If all the $h_i(x)$ bits are set, x was inserted with high probability, and not inserted otherwise. Rebuilding isn't supported. Hashing uses the collection-of-hash-functions logic, as described before.

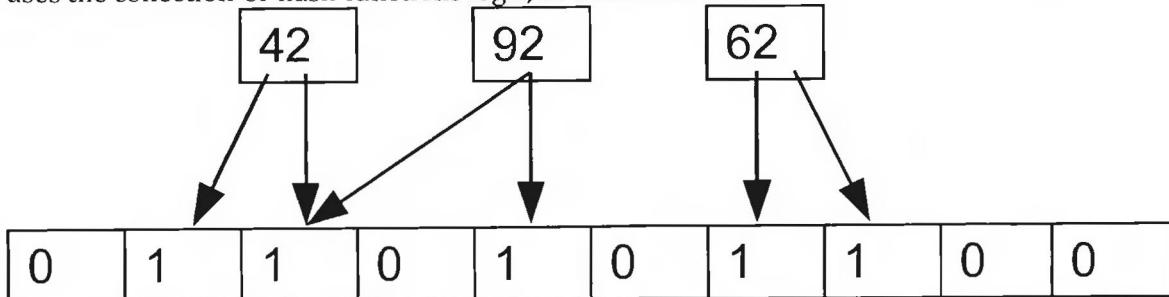


Figure 9.6: Bloom filter structure

```

template<typename KEY, typename HASHER = EHash<BUHash>>
class BloomFilter
{
    Bitset<unsigned char> items; //must be before h1, h2
    HASHER h1, h2;
    int nHashes;
    int hash(int hash1, int hash2, int i)
    {
        if(i == 0) return hash1;
        if(i == 1) return hash2;
        return (hash1 + i * hash2) % items.getSize();
    }
public:
    BloomFilter(int m, int theNHashes = 7): nHashes(theNHashes), items(
        nextPowerOfTwo(m)), h1(items.getSize()), h2(items.getSize())
        {assert(m > 0 && theNHashes > 0);}
    void insert(KEY const& key)
    {
        for(int i = 0; i < nHashes; i++)
            items.set(hash(h1(key), h2(key), i));
    }
    bool isInserted(KEY const& key) const
    {
        for(int i = 0; i < nHashes; i++)
            if(!items.test(hash(h1(key), h2(key), i)))
                return false;
        return true;
    }
};
  
```

```

        int hash1 = h1(key), hash2 = h2(key);
        for(int i = 0; i < nHashes; ++i) items.set(hash(hash1, hash2, i));
    }
    bool isInserted(KEY const& key)
    {
        int hash1 = h1.hash(key), hash2 = h2.hash(key);
        for(int i = 0; i < nHashes; ++i)
            if(!items[hash(hash1, hash2, i)]) return false;
        return true;
    }
};

```

As with chaining, $\Pr[x \text{ items are in a particular cell}] = \text{PoissonPDF}(x, ka)$. For $x = 0$, this $= e^{-ka} = \Pr[\text{a given bit remains } 0]$. So $\Pr[\text{isInserted is wrong}] = p = \Pr[k \text{ given bits are } 1] = (1 - e^{-ka})^k$; $k = \frac{\ln(2)}{a}$ minimizes p , so, neglecting that k is integer, $p = \frac{0.5 \ln(2)}{a} = e^{-0.48/a}$, and $m = \frac{-n \ln(p)}{0.48}$. Given an application-specific p and an estimate of maximum n , solve for m , a , and k , in this order. Linear probing uses $8(\text{sizeof(Node)} + 1)n/a$ bits, so only if it's infeasible should consider whether a Bloom filter allows a good enough p with a feasible m .

9.12 Implementation Notes

The most difficult part is actually the API for a hash function. I wanted code reuse, and C++11 was needed to make that happen. Otherwise would need to follow the STL and define a hash function override \forall word type in the language, which is clumsy and error-prone.

Picking specific h to support is also hard. The universal ones are few and so easy to pick, but nonuniversal ones aren't. FNV was chosen mostly based on its simplicity, popularity, and the performance studies, though I think xorshift-based h are potentially better, but not yet well-studied.

The Bloom filter implementation isn't very useful as is because the number of hashes should be calculated automatically from the success probability, which is the logical parameter. This isn't a particularly useful data structure for general use, but specific use cases for very large data sets have been reported in the literature.

9.13 Comments

Can turn any pseudorandom generator into h by interpreting a key as the seed (though need their sizes to match). Many h are criticized by Valloud (2008) and Henke et al. (2008, 2009). http://www.strchr.com/hash_functions has source code and a large number of tests of various h , but only focuses on runtime and collisions. Among all these, Bob Jenkins's h , which usually excel all tests, are complicated and have many unexplained decisions (google if curious). Cryptographic h which excel all tests shouldn't be used because they are very slow and even more complicated. With speed and simplicity taken into account, FNV appears to be the best nonuniversal choice.

As a C++ peculiarity, specifying h changes the type of the container, which logically should only be dependent on the data. Can fix this using a single class for all h and a choice selector, but shouldn't need this.

For linear probing deletion, one of the reinserted items will end up in the deleted item's place unless all hashed to a later location. So to improve efficiency can:

1. Try to move the last item in the chain to the deleted item's location
2. If that fails, move the next-to-last
3. Once move successfully, recursively process the deletion of the moved item
4. Stop when move the last item or no items need to be moved

Linear probing is a special case of more general **open addressing**, of which a textbook-popular variant is **double hashing**. The idea is to improve collision resolution of linear probing by jumping to a random cell using another h , instead of to the next cell. This reduces the expected insertion time but doesn't improve performance in practice (Heileman & Luo 2005) and is a bad choice because:

- Lose cache-efficiency
- Need to evaluate another h and the caller to specify one
- Lose deletions, and can only do weak ones (see the "Miscellaneous Algorithms and Techniques" chapter)

Quadratic probing (Cormen et al. 2009) uses deterministic step sizes to remove the cost and inconvenience of using another h . But this doesn't help with the other issues and needs $a < 0.5$.

Cuckoo hashing allows $O(1)$ find and remove by hashing an item in one of only two possible locations, chosen by two independent h . But insertion, despite being expected $O(1)$ for h with certain theoretical guarantees (stronger than universal; Mitzenmacher 2009; none implemented here), has no worst-case bound and may need to rebuild the table. This makes cuckoo dangerous and clumsy in practice because the caller might not give h with the wanted properties, and bugs such as a flawed random number generator are catastrophic. In my experiments only the seeded version of the Xorshift hash function (i.e., with the initial sum a remembered random number) did well, and this isn't a guarantee by any means.

Perfect hashing (Cormen et al. 2009) makes collision-free two-level hash tables for fixed n . But its advantage over chaining and linear probing in that case is unclear, and the bad-case memory use and construction time are problematic.

Can try to improve Bloom filter in several ways, but at best get slightly better efficiency with a much more complicated implementation (Putze et al. 2009). See also Mehta & Sahni (2018).

9.14 Projects

- Implement the more efficient (and also more complicated) deletion for linear probing. Compare performance to see whether the gain is worth the extra complexity.
- Compare performance with C++11 `unordered_map`.

9.15 References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., & Tarjan, R. E. (1994). Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23(4), 738–761.
- Dietzfelbinger, M., Hagerup, T., Katajainen, J., & Penttonen, M. (1997). A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1), 19–51.
- Heileman, G. L., & Luo, W. (2005). How caching affects hashing. In *ALENEX/ANALCO* (pp. 141–154). SIAM.
- Henke, C., Schmoll, C., & Zseby, T. (2008). Empirical evaluation of hash functions for multipoint measurements. *ACM SIGCOMM Computer Communication Review*, 38(3), 39–50.
- Henke, C., Schmoll, C., & Zseby, T. (2009). Empirical evaluation of hash functions for packetid generation in sampled multipoint measurements. In *Passive and Active Network Measurement* (pp. 197–206). Springer.
- Kirsch, A., & Mitzenmacher, M. (2008). Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms*, 33(2), 187–218.
- Mehlhorn, K., & Sanders, P., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- Mehta, D. P., & Sahni, S. (Eds.). (2018). *Handbook of Data Structures and Applications*. CRC.
- Mitzenmacher, M. (2009). Some open questions related to cuckoo hashing. In *Algorithms-ESA 2009* (pp. 1–10). Springer.
- Mitzenmacher, M., & Vadhan, S. (2008). Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 746–755). SIAM.
- Putze, F., Sanders, P., & Singler, J. (2009). Cache-, hash-, and space-efficient Bloom filters. *Journal of Experimental Algorithmics (JEA)*, 14, 4.
- Tarkoma, S., Rothenberg, C. E., & Lagerspetz, E. (2012). Theory and practice of Bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1), 131–155.
- Valloud, A. (2008). *Hashing in Smalltalk: Theory and Practice*.
- Wikipedia (2018a). Fowler–Noll–Vo hash function. https://en.wikipedia.org/wiki/Fowler%20%93Noll%20%93Vo_hash_function. Accessed July 22, 2018.
- Wikipedia (2018b). Lagrange's theorem (number theory). [https://en.wikipedia.org/wiki/Lagrange%27s_theorem_\(number_theory\)](https://en.wikipedia.org/wiki/Lagrange%27s_theorem_(number_theory)). Accessed July 22, 2018.
- Wikipedia (2018c). Poisson distribution. https://en.wikipedia.org/wiki/Poisson_distribution. Accessed July 29, 2018.
- Wikipedia (2018d). Universal hashing. https://en.wikipedia.org/wiki/Universal_hashing. Accessed July 22, 2018.
- Wikipedia (2019). Bloom filter. https://en.wikipedia.org/wiki/Bloom_filter. Accessed December 21, 2019.

10 Priority Queues

10.1 Introduction

A binary heap is commonly discussed in an algorithms class. My focus is on the very useful change key operation which unfortunately isn't. Another goal is to convince you that more complicated, O-fast priority queues, such as Fibonacci or pairing heaps that the class might have mentioned, are useless in practice.

10.2 The API

A **priority queue** can:

- Insert an item with a given priority
- Find the smallest-priority item
- Remove the smallest-priority item

This is the standard interface. Some additional operations:

- Change an item's priority—traditionally called “change key”, essential for many algorithms
- Remove an item—essentially change key to the smallest value, and remove the smallest-priority item
- Merge two priority queues—very fast for some implementations, but rarely used

The functionality limitation allows a more efficient implementation than by a dynamic sorted sequence that supports nonunique keys. Because a priority queue can sort, delete min or insert needs \geq amortized $O(\lg(n))$ time.

10.3 Binary Heap

Use a vector to represent a binary tree where for the node at index i . The:

- Parent = $(i - 1)/2$
- Left child = $2i + 1$
- Right child = $2i + 2$

The item in any node \leq the items in its children in terms of priority.

Vector: abicdjkefghlmno

Tree:

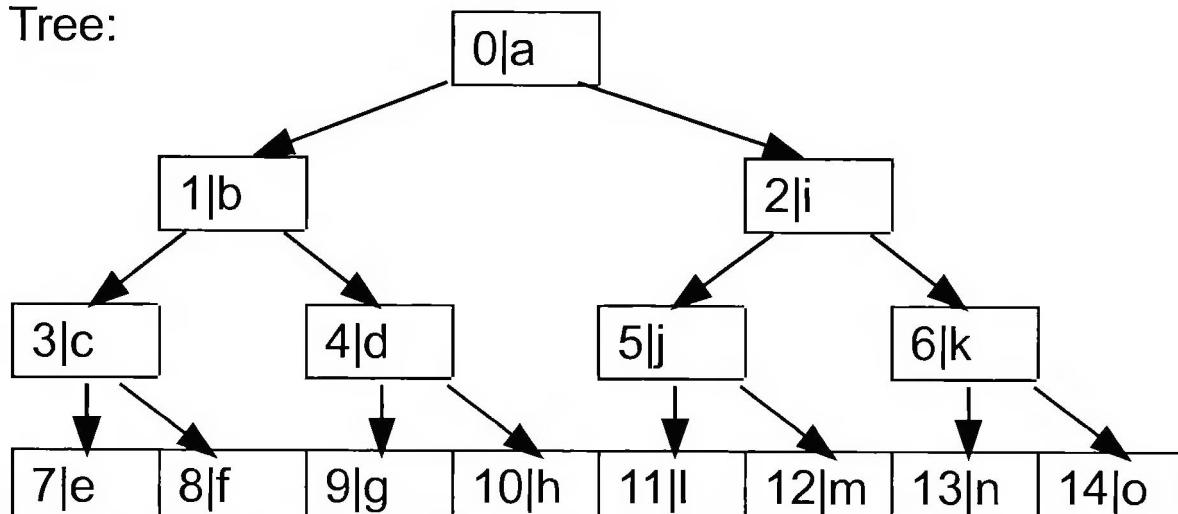


Figure 10.1: Tree interpretation of the heap vector. The numbers are array indices.

Item movement is optionally reported to the caller for indexing, as discussed later in the chapter. I haven't seen this augmentation presented or implemented elsewhere, but due to lack of item persistence, without it the change key operation is useless.

```
template <typename ITEM>
struct ReportDefault<void operator()(ITEM& item, int i){};
```

```
template <typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>,
          typename REPORTER = ReportDefault<ITEM>> class Heap
```

```

{
    REPORTER r;
    int getParent(int i) const { return (i - 1)/2; }
    int getLeftChild(int i) const { return 2 * i + 1; }
    Vector<ITEM> items;
public:
    COMPARATOR c;
    Heap(COMPARATOR const& theC = COMPARATOR(), REPORTER const&
        theReporter = REPORTER()): r(theReporter), c(theC) {}
    bool isEmpty() const { return items.getSize() == 0; }
    int getSize() const { return items.getSize(); }
    ITEM const& getMin() const
    {
        assert(!isEmpty());
        return items[0];
    }
    ITEM const& operator[](int i) const
    { // random access is useful with item handles
        assert(i >= 0 && i < items.getSize());
        return items[i];
    }
}

```

Move up exchanges the item with its parent while parent < item, reporting all movements:

```

void moveUp(int i)
{
    ITEM temp = items[i];
    for(int parent; i > 0 && c(temp, items[parent = getParent(i)]);
        i = parent) r(items[i] = items[parent], i);
    r(items[i] = temp, i);
}

```

Move down exchanges the item with its smallest child while it has one, reporting all movements:

```

void moveDown(int i)
{
    ITEM temp = items[i];
    for(int child; (child = getLeftChild(i)) < items.getSize(); i = child)
    { // find smaller child
        int rightChild = child + 1;
        if(rightChild < items.getSize() && c(items
            [rightChild], items[child])) child = rightChild;
        // replace with the smaller child if any
        if(!c(items[child], temp)) break;
        r(items[i] = items[child], i);
    }
    r(items[i] = temp, i);
}

```

Insert appends the item and moves it up:

```

void insert(ITEM const& item)
{
    items.append(item);
    moveUp(items.getSize() - 1);
}

```

To remove item *i*:

1. Replace the item with a copy of the last item
2. Move down the copy (it won't touch the last item)
3. Remove the last item

Reporting location -1 signals that the item was deleted. For the reporting of the replacement item note that it might be the removed one if the last.

```

ITEM deleteMin() { return remove(0); }
ITEM remove(int i)
{
    assert(i >= 0 && i < items.getSize());
}

```

```

ITEM result = items[i];
r(result, -1);
if(items.getSize() > i)
{ //not last item
    items[i] = items.lastItem();
    r(items[i], i); //report move
    moveDown(i); //won't touch the last item
}
items.removeLast();
return result;
}

```

Change key of item i makes the change and moves it up if the new key $<$ the old key and down otherwise. To replace the minimum, use `changeKey(0, item)`.

```

void changeKey(int i, ITEM const& item)
{
    assert(i >= 0 && i < items.getSize());
    bool decrease = c(item, items[i]);
    items[i] = item;
    decrease ? moveUp(i) : moveDown(i);
}

```

All operations take $O(\text{the height}) = O(\lg(n))$ time, but `insert` and `remove` are amortized due to vector doubling. To merge several heaps, insert into one all items from the rest (not implemented here). If all items will eventually be removed, the cost is amortized $O(\lg(n))$.

Heapsort uses a heap to sort by inserting items into it and getting them out in sorted order using `delete min`. Can run a heap directly on the unsorted array of items and use faster bulk insertion, called **heapify** (not implemented here). This takes $O(n\lg(n))$ time but with a larger constant than other O -optimal sorts.

10.4 Indexed Heaps

An **indexed heap** uses, e.g., an address-persistent chaining hash table to associate some items with caller-provided handles from inserting into a binary heap. This is useful, e.g., for Dijkstra's shortest paths algorithm (see the "Graph Algorithms" chapter). With some creativity can make an indexed heap out of standard components:

- Use a set with tuple items consisting of a priority and a handle, sorted on both in this order
- Map from a handle to a priority using a hash table
- To update given a handle, use the map to get the priority, update it, then in the set delete the old tuple and insert the new one
- To pop remove the set's smallest element, and update the map

But, while this is considered a fair game for interview questions (someone once asked me to implement a modified version of Dijkstra's algorithm), the constant factors are better if use a priority queue instead of a set. Though a set typically doesn't allow duplicate keys unlike a priority queue, here the secondary key guarantees overall key uniqueness. This is where reporting comes—the heap tells the map to where it moves the items.

Another approach, applicable only in special cases, is to insert the new-priority item without removing the old one. E.g., for Dijkstra's algorithm (see the "Graph Algorithms" chapter), this works because the old item will never be retrieved. But this is unnecessary cleverness and memory waste.

For an indexed heap the main implementation trick is managing the reporting correctly. The heap item consists of the real item and the pointer to the hash table node containing the handle as key and heap location as value. The reporter uses the pointers to update item indices. The heap's amortized $O(\lg(n))$ and the hash table's expected $O(1)$ lead to expected amortized $O(\lg(n))$ operations.

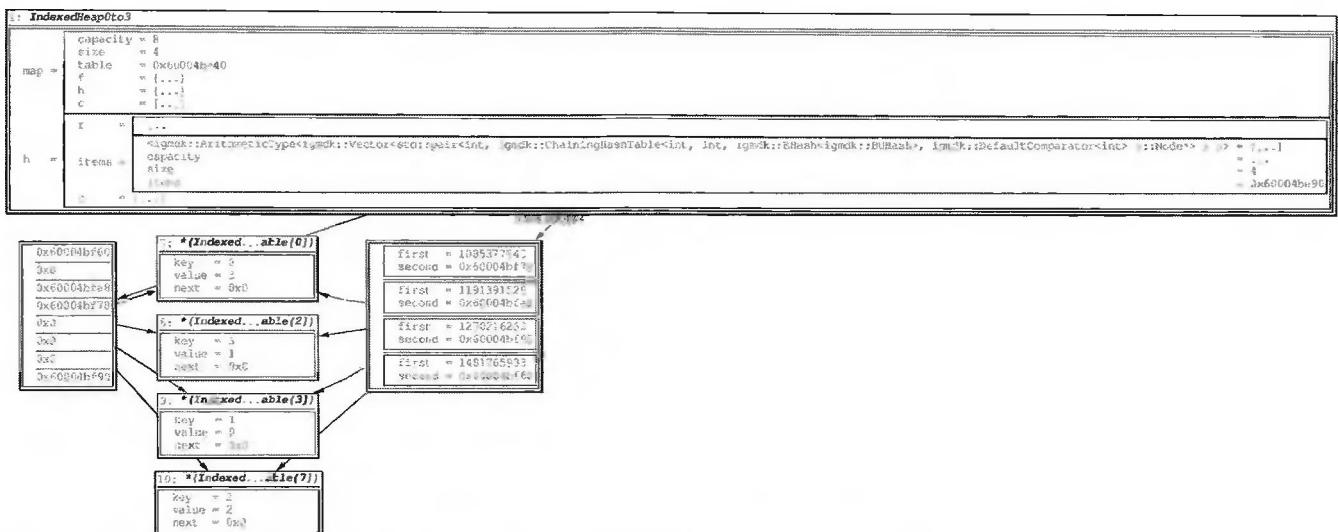


Figure 10.2: Memory layout of an indexed heap with integer items 0–3

The operations reduce to calling the corresponding heap operations and updating the map when the reporter can't.

```
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>,
         typename HANDLE = int, typename HASHER = EHash<BUHash>> class IndexedHeap
{
    typedef ChainingHashTable<HANDLE, int, HASHER> MAP;
    MAP map;
    typedef typename MAP::NodeType* POINTER;
    typedef pair<ITEM, POINTER> Item;
    typedef PairFirstComparator<ITEM, POINTER, COMPARATOR> Comparator;
    struct Reporter
    {
        void operator()(Item& item, int i){item.second->value = i;};
    };
    Heap<Item, Comparator, Reporter> h;
public:
    IndexedHeap(COMPARATOR const& theC = COMPARATOR()): h(Comparator(theC)) {}
    int getSize() const{return h.getSize();}
    ITEM const* find(HANDLE handle)
    {
        int* index = map.find(handle);
        return index ? &h[*index].first : 0;
    }
    bool isEmpty() const{return h.isEmpty();}
    void insert(ITEM const& item, HANDLE handle)
    {
        assert(!find(handle)); //else map will fail with duplicate
        h.insert(Item(item, map.insert(handle, h.getSize())));
    }
    pair<ITEM, HANDLE> getMin() const
    {
        Item temp = h.getMin();
        return make_pair(temp.first, temp.second->key);
    }
    pair<ITEM, HANDLE> deleteMin()
    {
        Item temp = h.deleteMin();
        pair<ITEM, HANDLE> result = make_pair(temp.first, temp.second->key);
        map.remove(temp.second->key);
        return result;
    }
    void changeKey(ITEM const& item, HANDLE handle)
    {
        POINTER p = map.findNode(handle);
        if(p) h.changeKey(p->value, Item(item, p));
        else insert(item, handle);
    }
}
```

```

void deleteKey(HANDLE handle)
{
    int* index = map.find(handle);
    assert(index);
    h.remove(*index);
    map.remove(handle);
}
};

```

Using a vector as the map is more efficient for small-integer handles. Its size is dynamically increased to the maximum handle value. The indices are handles and the items are heap locations. Value -1 means no location at the index.

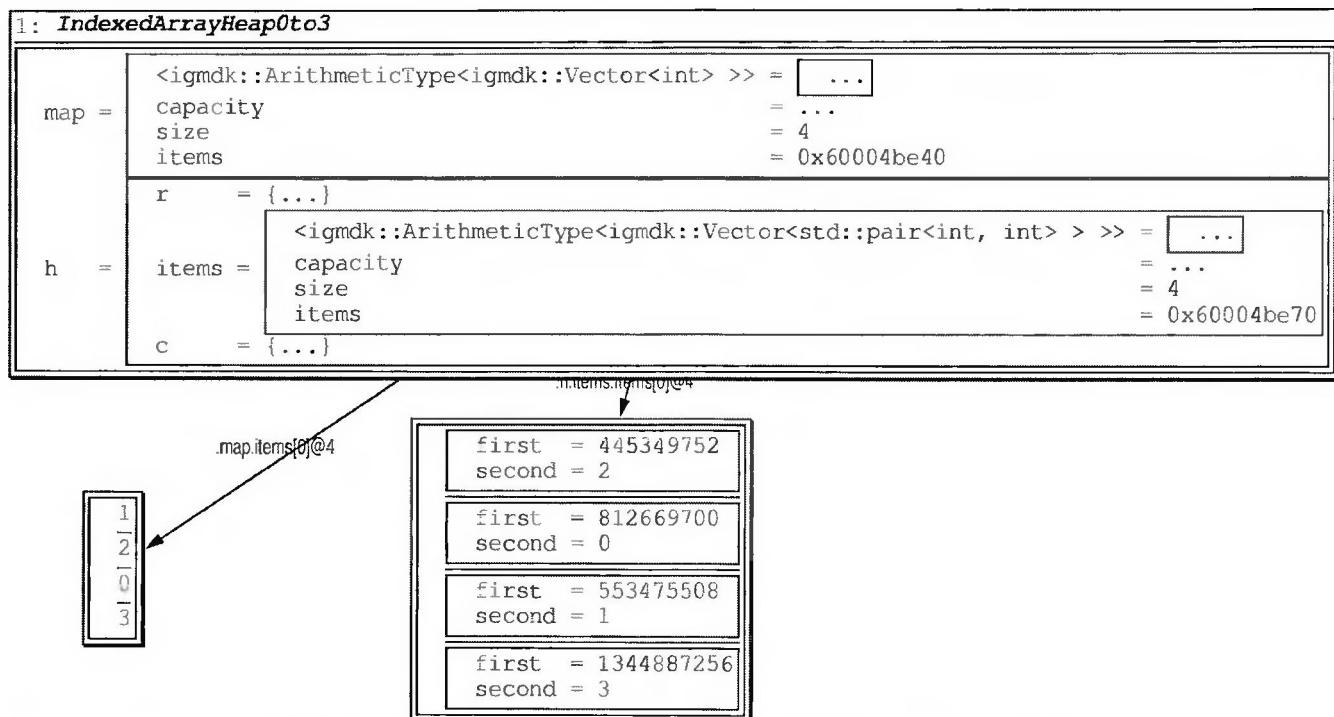


Figure 10.3: Memory layout of vector-based indexed heap with integer items 0–3

```

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
class IndexedArrayHeap
{
    Vector<int> map;
    typedef pair<ITEM, int> Item;
    typedef PairFirstComparator<ITEM, int, COMPARATOR> Comparator;
    struct Reporter
    {
        Vector<int>& pmap;
        Reporter(Vector<int>& theMap) : pmap(theMap) {}
        void operator() (Item& item, int i) {pmap[item.second] = i;}
    };
    Heap<Item, Comparator, Reporter> h;
public:
    typedef Item ITEM_TYPE;
    IndexedArrayHeap(COMPARATOR const& theC = COMPARATOR()):
        h(Comparator(theC), Reporter(map)) {}
    int getSize() const {return h.getSize();}
    ITEM const* find(int handle)
    {
        assert(handle >= 0);
        return handle >= map.getSize() || map[handle] == -1 ? 0 :
            &h[map[handle]].first;
    }
    bool isEmpty() const {return h.isEmpty();}
    void insert(ITEM const& item, int handle)
    {

```

```

assert(handle >= 0);
if(handle >= map.getSize())
    for(int i = map.getSize(); i <= handle; ++i) map.append(-1);
    h.insert(Item(item, handle));
}
pair<ITEM, int> const& getMin() const return h.getMin();
pair<ITEM, int> deleteMin()
{
    Item result = h.deleteMin();
    map[result.second] = -1;
    return result;
}
void changeKey(ITEM const& item, int handle)
{
    assert(handle >= 0);
    if(handle >= map.getSize() || map[handle] == -1) insert(item, handle);
    else h.changeKey(map[handle], Item(item, handle));
}
void deleteKey(int handle)
{
    assert(handle >= 0 && handle < map.getSize());
    int pointer = map[handle];
    assert(pointer != -1);
    h.remove(pointer);
    map[handle] = -1;
}
};

```

An indexed heap (either implementation) also gives a clean solution to the following interview problem. Have a marathon track with m sensors and n runners who start at the same time (unrealistically the track is wide enough). A sensor sends an event consisting of its and the runner's ids to some system when a runner passes by. The system maintains a list of current k leading runners as the race progresses, printing it on demand.

A simple solution is to keep an ordered list of runners that passed a particular sensor (assume that several runners can't come to the same sensor at exactly the same time). When asked for a ranking, start from the last sensor on the track, collect the runners from its list if any, and move on to the next sensor, until have k . When runners are deleted from all but their most recent sensor list, printing current ranks takes $O(n + m)$ time and memory.

A better solution is to recognize that at any time only the current top k runners (take any k at the start) are relevant. A priority of a runner is a composition of the first sensor and the arrival time. Use a global counter for arrival times. Put the current top k runners in an indexed heap. When a sensor sends an event, the runner is either top or not. Then:

- If a runner is currently in the top, update its priority
- If it gets to the top, put it in the heap, and remove the current k^{th} runner

Printing takes $O(k \lg(k))$ by making a copy of the heap and repeatedly removing the top. Processing an event takes $O(\lg(k))$ time, and use $O(n)$ memory. Can drop the $\lg(k)$ factor using a data structure consisting of a linked list and two hash tables, but this is too complicated for interviews and impractical.

10.5 Implementation Notes

The idea of an indexed heap appears in Sedgewick & Wayne (2011), who give a limited implementation of an indexed array heap and don't mention using a hash table instead. The reporting functionality is original and allows clean decoupling. Given that using change key functionality requires some mapping, it's an obvious but a useful concept, though other implementations apply. I tested a number of different ones before deciding to pick the reported-based solution over some level of indirection.

10.6 Comments

☰ many other priority queues with various properties. None are useful in practice because a binary heap is simpler, performs competitively in all cases, and uses less memory. Major alternatives include a:

- **Bucket queue**—for items with priorities $\in [0, N - 1]$, it's an array of N linked lists, with list i containing the items with priority i , and an integer containing the smallest index of a nonempty list.

E.g., can schedule tasks by giving them priorities $\in [1, 10]$. For small N it's faster than a binary heap but uses much more memory. Many real-life queues run similarly.

- **Pairing heap**—uses a collection of heap-ordered pointer-based trees. Decrease key and merge take $O(1)$ time (see Mehlhorn et al. 2019 for implementation details). The problems are clumsy implementation, much higher memory use, and larger constant factors (Bruun et al. 2010). Some experimental conclusions about binary heaps are that the expected cost of inserting is $O(1)$, the expected cost of decreasing a key for random data is $O(1)$, cache and using-less-memory effects are significant, and the difference between $\lg(n)$ and a constant is small, particularly for the move up due to the short loop.
- Even more so for a better-known **Fibonacci heap** and other complicated pointer-based heaps. These were designed as indexed heaps due to easy change key. A **Brodal queue** (Brass 2008) seems to be the winner theoretically among these “weak heaps” that attempt to cut down the amortized number of comparisons.

For a **double-ended heap** supporting insertion and deletion of both max and min, use a skip list with nonunique items. A **min-max heap** (Mehta & Sahni 2018) is specially designed for this and may be a little more efficient, but need to implement and maintain it. A further generalization is a **multidimensional heap**, with items ranked by each dimension (Brass 2008). Can implement it using an indexed heap \forall coordinate, putting every item in every heap and using the indices to delete from other heaps after a delete in a particular one.

10.7 References

- Brass, P. (2008). *Advanced Data Structures*. Cambridge University Press.
- Bruun, A., Edelkamp, S., Katajainen, J., & Rasmussen, J. (2010). Policy-based benchmarking of weak heaps and their relatives. In *International Symposium on Experimental Algorithms* (pp. 424-435). Springer, Berlin, Heidelberg.
- Mehta, D. P. & Sahni, S. (Eds.). (2018). *Handbook of Data Structures and Applications*. CRC.
- Mehlhorn, K., & Sanders, P., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.

11 Graph Algorithms

11.1 Introduction

Graph algorithms are very useful for interview questions—particularly depth-first and breadth-first search. The standard topics are discussed. I also describe efficient implementations of several algorithms using indexed heaps. More advanced algorithms based on networks flows are introduced briefly.

11.2 Basics

A graph consists of V vertices connected by E edges such that \exists at most one edge between any two vertices, and no vertex has an edge to itself. A sequence of edges is a **path**. A graph is:

- **Sparse** if it has $< V^2/2$ edges, and **dense** otherwise. For most useful graphs, $E = O(V)$.
- **Undirected** if edges are bidirectional, and **directed** otherwise.
- **Strongly connected** if directed and \forall vertex \exists path to any other vertex.
- **Connected** if undirected and strongly connected. A disconnected graph consists of several connected components.
- **Acyclic** if directed, and \forall vertex \exists a path to itself—very common, called **DAG**.
- **A tree** if acyclic and each vertex has at most one incoming edge.
- **Bipartite** if vertices form two groups, and \exists edges between same-group vertices.
- **Implicit** if not represented by a data structure, and some function determines vertices, edges and edge data.

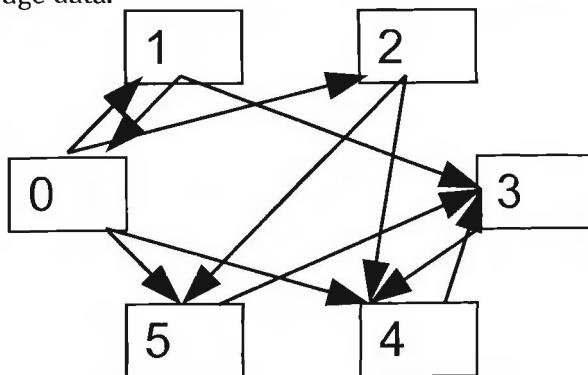


Figure 11.1: A directed, cyclic, not strongly connected, and sparse graph

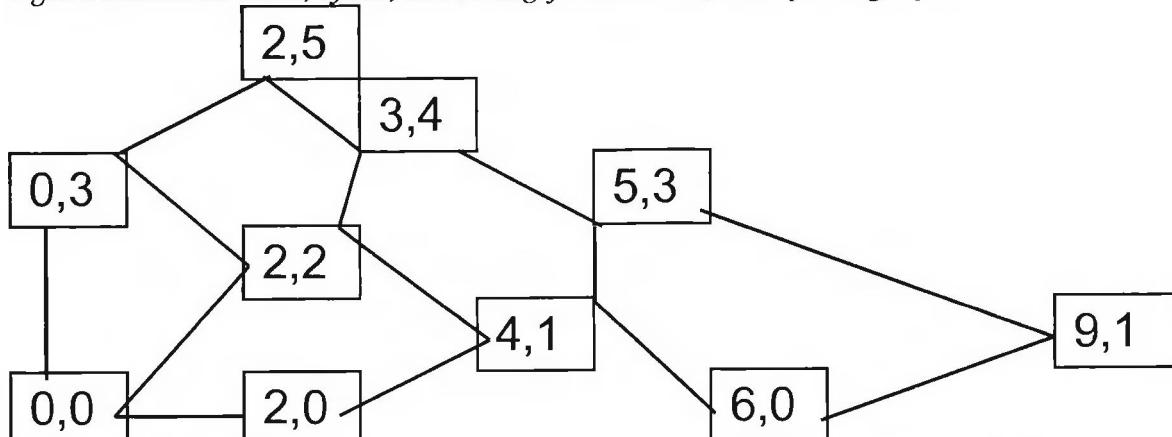


Figure 11.2: An undirected, sparse, and connected graph, with edge data = implicit Euclidean distance between vertex coordinates

11.3 Graph Representation

A graph supports:

- Construction
- Iteration over vertices
- Iteration over edges of a vertex

Some algorithms need map operations with edges or vertices, particularly finding outgoing edges of a vertex. Any representation takes $O(V + E)$ space unless the graph is at least partially implicit. A dense graph needs $O(V^2)$ bits. Undirected graphs are represented as directed, but with both edges present. Many graphs don't have vertex data, so it's more efficiently represented by the caller, and edge data by the graph.

Consider a dynamic sorted sequence of vertices, each of which has its number, and a dynamic sorted sequence of outgoing vertex numbers with the corresponding edge data. This data structure efficiently supports all operations, but a vector of vectors (also called an **adjacency array**) has less overhead, supports incremental construction and iteration, and most algorithms use it.

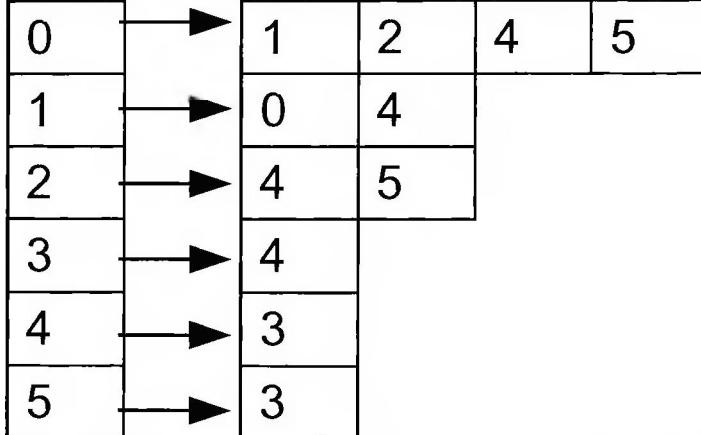


Figure 11.3: Structure of the adjacency array for the graph in Figure 4.1

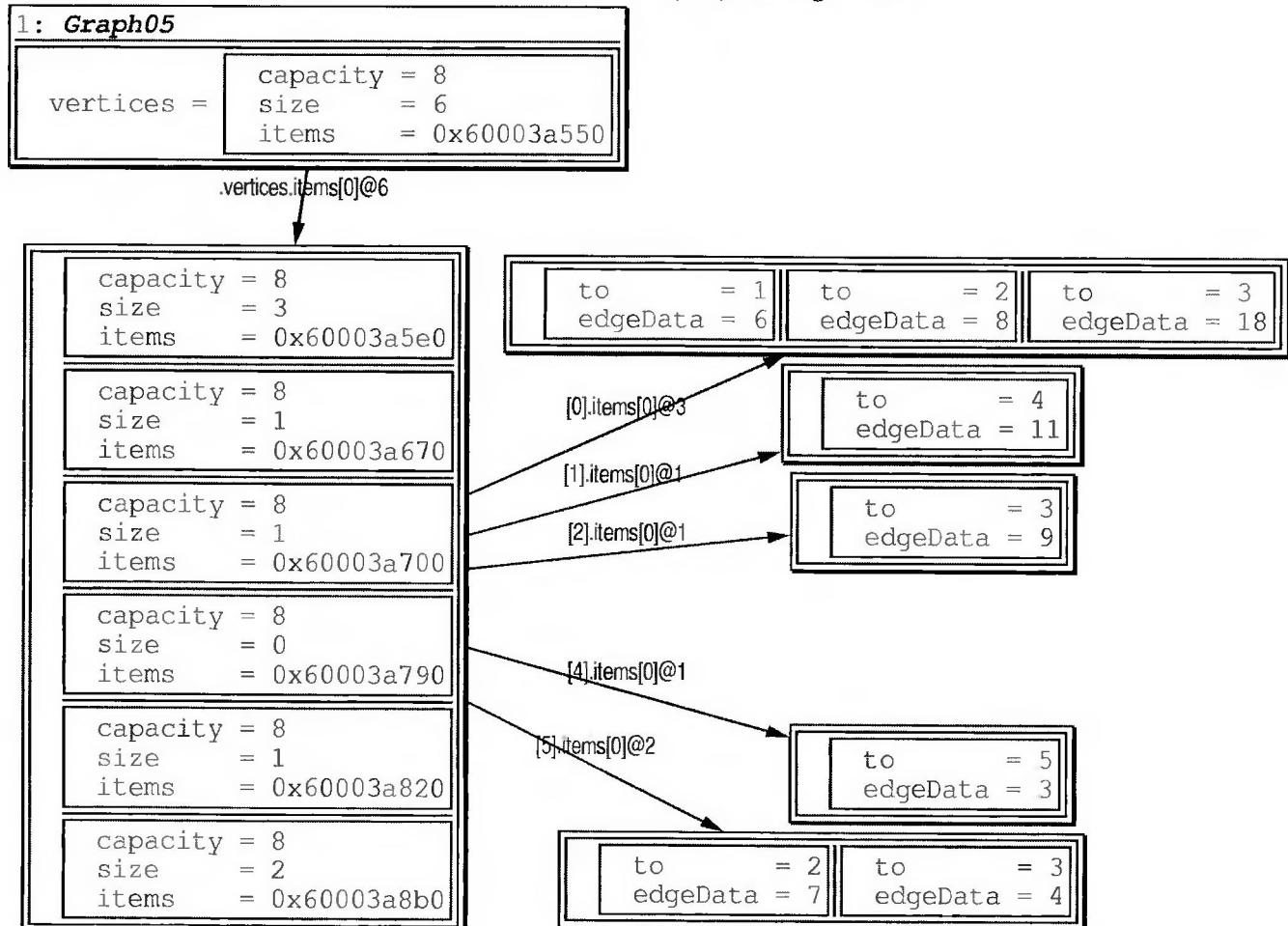


Figure 11.4: Memory layout of the adjacency array for the graph in Figure 4.1 with some edge data

Adjacency iteration is going through all outgoing edges of a vertex.

```

template< typename EDGE_DATA > class GraphAA
{
    struct Edge
    {
        int to;
        EDGE_DATA edgeData;
    };
}
```

```

    Edge(int theTo, EDGE_DATA const& theEdgeData): to(theTo),
        edgeData(theEdgeData) {}
};

Vector<Vector<Edge>> vertices;
public:
    GraphAA(int initialSize = 0): vertices(initialSize) {}
    int nVertices() const{return vertices.getSize();}
    int nEdges(int v) const{return vertices[v].getSize();}
    void addVertex(){vertices.append(Vector<Edge>());}
    void addEdge(int from, int to, EDGE_DATA const& edgeData = EDGE_DATA())
    {
        assert(to >= 0 && to < vertices.getSize());
        vertices[from].append(Edge(to, edgeData));
    }
    void addUndirectedEdge(int from, int to,
        EDGE_DATA const& edgeData = EDGE_DATA())
    {
        addEdge(from, to, edgeData);
        addEdge(to, from, edgeData);
    }
    class AdjacencyIterator
    {
        Vector<Edge> const* edges;
        int j; //current edge
    public:
        AdjacencyIterator(GraphAA const& g, int v, int theJ):
            edges(&g.vertices[v]), j(theJ){}
        AdjacencyIterator& operator++()
        {
            assert(j < edges->getSize());
            ++j;
            return *this;
        }
        int to(){return (*edges)[j].to;}
        EDGE_DATA const& data(){return (*edges)[j].edgeData;}
        bool operator==(AdjacencyIterator const& rhs){return j == rhs.j;}
    };
    AdjacencyIterator begin(int v) const
    {
        return AdjacencyIterator(*this, v, 0);
    }
    AdjacencyIterator end(int v) const
    {
        return AdjacencyIterator(*this, v, nEdges(v));
    }
};

```

For a directed graph it's easy to reverse the edges:

```

template<typename GRAPH> GRAPH reverse(GRAPH const& g)
{
    GRAPH result(g.nVertices());
    for(int i = 0; i < g.nVertices(); ++i)
        for(typename GRAPH::AdjacencyIterator j = g.begin(i);
            j != g.end(i); ++j) result.addEdge(j.to(), i, j.data());
    return result;
}

```

For a static graph a more compact representation is to merge all edge arrays. It has all edges from vertex 0, then 1, etc., and the vertex array indexes into the edge array. Byte code or some other mechanism (see the "Compression" chapter) can compress this further.

11.4 Search

Vertex and edge iteration allows graph traversal, but special orders such as the one generated by **depth-first search (DFS)** have useful properties. Called on a source vertex, DFS iterates over its edges, recursively calling itself on every unvisited destination vertex. The traversal forms a tree with edges classified as:

- **Tree**—when visiting an unvisited vertex
- **Backward**—when backtracking from a tree edge

- **Forward**—when jumping to a visited descendant of the current vertex
- **Cross**—when jumping to a visited nondescendant of the current vertex

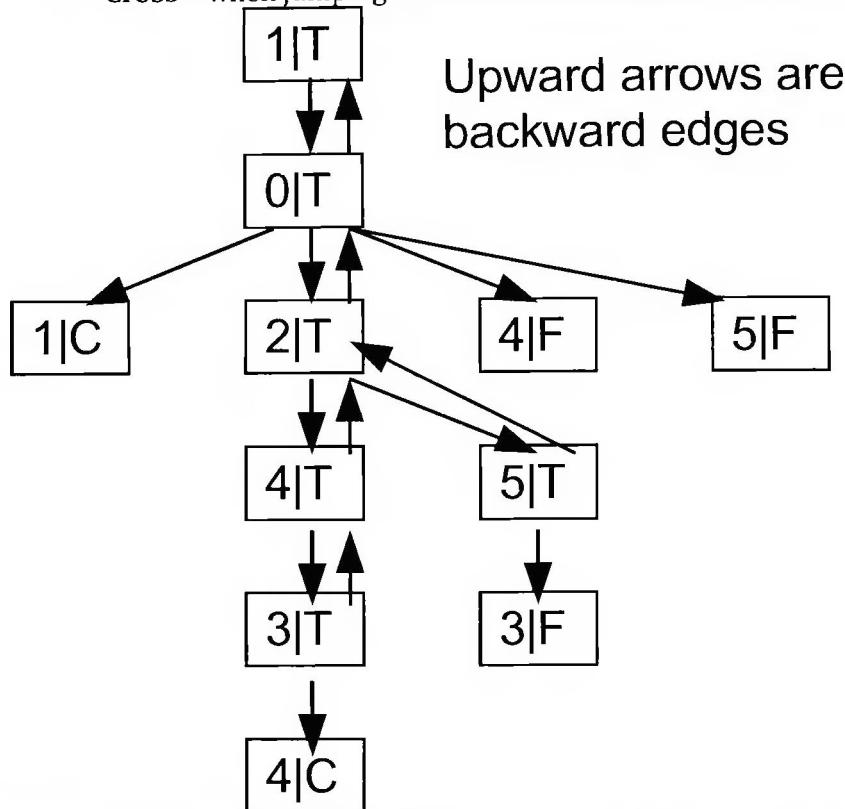


Figure 11.5: DFS transition logic of the graph in Figure 4.1; the numbers are vertex indices and the letter represent edge types

Use a stack to not run out of memory for large graphs. Push the current vertex and the next child to be visited, and pop when taking a backward edge, calling an action functor when entering source or taking a tree, forward, cross, or backward edge. A default functor is provided for code reuse. Forward and cross edges are grouped into a nontree edge because can't distinguish them efficiently. The below helper explores a single connected component.

```
struct DefaultDFSAction
{
    void source(int v){}
    void treeEdge(int v){}
    void nonTreeEdge(int v){}
    void backwardEdge(int v){}
};

template<typename GRAPH, typename ACTION> void DFSCOMPONENT(GRAPH const& g,
    int source, Vector<bool>& visited, ACTION& a = ACTION())
{
    typedef typename GRAPH::AdjacencyIterator ITER;
    Stack<pair<ITER, int>> s; //current vertex and next child
    s.push(make_pair(g.begin(source), source));
    while(!s.isEmpty())
    {
        ITER& j = s.getTop().first;
        if(j != g.end(s.getTop().second))
        {
            int to = j.to();
            if(visited[to]) a.nonTreeEdge(to);
            else
            {
                a.treeEdge(to);
                visited[to] = true;
                s.push(make_pair(g.begin(to), to));
            }
            ++j;
        }
    }
}
```

```
        }
    else
    {
        s.pop();
        if(!s.isEmpty()) a.backwardEdge(s.getTop().second);
    }
}
```

Because a graph may be disconnected, call DFS on each vertex. This doesn't affect the correctness and the asymptotic runtime because mark vertices visited, and don't call a nontree action when enter a source.

```

template<typename GRAPH, typename ACTION> void DFS(GRAPH const* g,
ACTION& a = ACTION())
{
    Vector<bool> visited(g.nVertices(), false);
    for(int i = 0; i < g.nVertices(); ++i) if(!visited[i])
    {
        a.source(i);
        visited[i] = true;
        DFSComponent(g, i, visited, a);
    }
}

```

Breadth-first search (BFS) is a different, also useful way to explore a graph:

1. Enqueue the source vertex
 2. Until the queue is empty
 3. Dequeue a vertex
 4. Iterate over its edges, enqueueing each

Don't have action points because essentially the only result of BFS is the number of edges taken to reach each vertex from the source. Visit vertices in order of this distance, and set distances when enqueueing vertices that aren't enqueued, which ensures correctness.

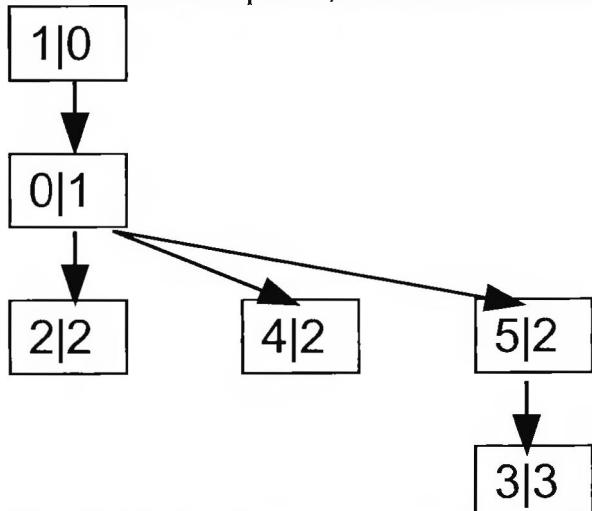


Figure 11.6: BFS transition logic of the graph in Figure 4.1: The first number in a node is the vertex, and the second its number-of-edges distance from the source.

```
template<typename GRAPH> Vector<int> BFS(GRAPH& g, int source)
{
    Vector<int> distances(g.nVertices(), -1);
    Queue<int> q(g.nVertices());
    distances[source] = 0;
    q.push(source);
    while(!q.isEmpty())
    {
        int i = q.pop();
        for(typename GRAPH::AdjacencyIterator j = g.begin(i); j != g.end(i);
            ++j) if(distances[j.to()] == -1)
        {
            distances[i.to()] = distances[i] + 1;
            q.push(j.to());
        }
    }
}
```

```

        q.push(j.to());
    }
}

return distances;
}

```

BFS is useful only for exploring a single component. DFS walks a continuous path, and BFS jumps across vertices. Both take $O(V + E)$ time, assuming actions take $O(1)$ time.

11.5 Some Applications of Search

DFS finds **connected components** of a graph when called with a functor which adds a new component when entering a source and adds any newly visited vertex to the current component.

```

struct ConnectedComponentAction: public DefaultDFSAction
{
    Vector<Vector<int>> components;
    void source(int v)
    {
        components.append(Vector<int>());
        treeEdge(v);
    }
    void treeEdge(int v) {components.lastItem().append(v);}
};

template<typename GRAPH>
Vector<Vector<int>> connectedComponents(GRAPH const& g)
{
    ConnectedComponentAction a;
    DFS(g, a);
    return a.components;
}

```

Topological sort of a DAG is an order of vertices such that $\forall \text{vertex } v \exists \text{ a path to any vertex earlier in the ordering}$. E.g., an order in which can take classes to satisfy prerequisites. When DFS returns to a vertex and goes into its next outgoing edge, can visit the vertices on the returned path last in topological order. So assigning ranks, starting from $V - 1$, to vertices when taking backward edges finds a topological sort. If take a nontree edge to a vertex without a rank, this must be a cross edge because if it were a forward edge the vertex would have been ranked.

A graph with a cross edge has a **cycle**, so can't topologically sort it. E.g., in Figure 5.9, the cross edges (0,1) and (3,4) correspond to cycles. If they are removed, when DFS returns to 2 from 3 and goes into 5, it assigns rank 5 to 3 and 4 to 4. Then when returning from 5 to 1, it assigns 3 to 5, 2 to 2, 1 to 0, and 0 to 1.

```

struct TopologicalSortAction
{
    int currentRank, leaf; //current DFS tree leaf
    Vector<int> ranks;
    bool hasCycle;
    TopologicalSortAction(int nVertices): currentRank(nVertices), leaf(-1),
        ranks(nVertices, -1), hasCycle(false) {}
    void source(int v){treeEdge(v);}
    void treeEdge(int v){leaf = v;} //potential leaf
    //unranked v = cross edge
    void nonTreeEdge(int v){if(ranks[v] == -1) hasCycle = true;}
    void backwardEdge(int v)
    {
        if(leaf != -1)
            //assign rank to DFS tree leaf if any
            ranks[leaf] = --currentRank;
        leaf = -1;
    }
    ranks[v] = --currentRank;
}
;

template<typename GRAPH> Vector<int> topologicalSort(GRAPH const& g)
{

```

```

TopologicalSortAction a(g.nVertices());
DFS(g, a);
if(a.hasCycle) a.ranks = Vector<int>(); //empty ranks signals cycle
return a.ranks;
}

```

DFS can generate a random maze:

1. Start with a rectangular grid of unit-size cells with walls between any two neighboring cells and a bounding box around all cells
2. Represent the grid as a graph, with vertices corresponding to cells and edges to walls
3. Randomly permute each edge array
4. Starting from any vertex, run DFS that \forall tree edge erases the corresponding wall
5. Pick any starting and ending square, and erase their bounding box walls

11.6 Minimum Spanning Tree

When edges hold distances between the vertices they connect, a **MST** connects all vertices with $V - 1$ edges such that \sum edge distance is minimal.

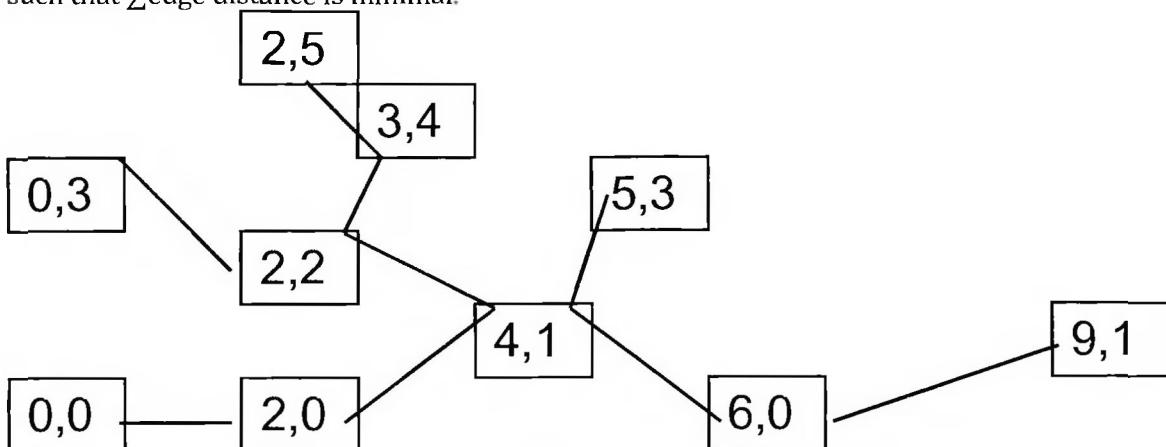


Figure 11.7: MST of a Euclidean-data graph

Prim's algorithm iteratively adds the vertex whose edge cost from any vertex already in the tree is minimal. Represent the resulting tree as a sequence of vertices.

1. Create an indexed priority queue that orders by distance from any vertex in the tree
2. Enqueue all vertices with priority ∞
3. Until the queue is empty
4. Dequeue a vertex
5. Decrease known distances to its children
6. \forall child vertex update the parent if decreased distance

```

template<typename GRAPH> Vector<int> MST(GRAPH& g)
{//represent MST as edges to parent vertices (first node won't have any)
Vector<int> parents(g.nVertices(), -1);
typedef pair<double, int> QNode;
IndexedArrayHeap<QNode, PairFirstComparator<double, int>> pQ;
for(int i = 0; i < g.nVertices(); ++i)
    pQ.insert(QNode(numeric_limits<double>::max(), i), i);
while(!pQ.isEmpty())
{
    int i = pQ.deleteMin().first.second;
    for(typename GRAPH::AdjacencyIterator j = g.begin(i); j != g.end(i); ++j)
        //adjust best known distances to child vertices not yet in the tree
        QNode const* child = pQ.find(j.to()); //child may no longer be in q
        if(child && j.data() < child->first)
        {
            pQ.changeKey(QNode(j.data(), j.to()), j.to());
            parents[j.to()] = i; //update to closer parent
        }
}
}

```

```

    return parents;
}

```

The runtime is $O(E\ln(V))$ due to $O(E)$ change key operations. Use $O(V)$ memory.

11.7 Shortest Paths

When edges have associated distances between the vertices they connect, \exists a shortest path from any vertex to any other. Distance(any vertex, itself) = 0.

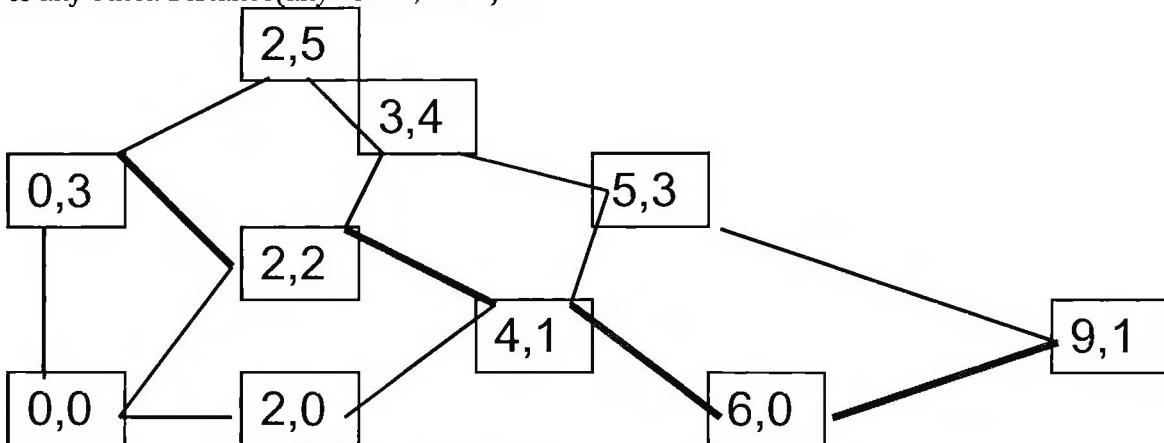


Figure 11.8: The shortest path from (0, 3) to (9, 1)

When all the distances ≥ 0 , **Dijkstra's algorithm** finds the distances from a source vertex to all others and stops when the distance to a destination vertex is found:

1. Create an indexed priority queue that orders by the distance from the source
2. Put all vertices with distance ∞ and the source with distance 0
3. Until the queue is empty, or reach a goal state if any
4. Dequeue the next vertex
5. \forall child vertex
6. The distance = the distance to the vertex + distance(the vertex, destination) if shorter

The result gives \forall vertex the next vertex on the path back to the source.

```

template<typename GRAPH>
Vector<int> ShortestPath(GRAPH& g, int from, int dest = -1)
{//no goal state by default
    assert(from >= 0 && from < g.nVertices());
    Vector<int> pred(g.nVertices(), -1);
    typedef pair<double, int> QNode;
    IndexedArrayHeap<QNode, PairFirstComparator<double, int> > pQ;
    for(int i = 0; i < g.nVertices(); ++i) pQ.insert(
        QNode(i == from ? 0 : numeric_limits<double>::infinity(), i), i);
    while(!pQ.isEmpty() && pQ.getMin().second != dest)
    {
        int i = pQ.getMin().first.second;
        double dj = pQ.deleteMin().first.first; //distance to the current node
        for(typename GRAPH::AdjacencyIterator j = g.begin(i); j != g.end(i);
            ++j)
        { //child may no longer be in q
            double newChildDistance = dj + j.data();
            QNode const* child = pQ.find(j.to());
            if(child && newChildDistance < child->first)
            {
                pQ.changeKey(QNode(newChildDistance, j.to()), j.to());
                pred[j.to()] = i; //new best parent
            }
        }
    }
    return pred;
}

```

The runtime is $O(E\ln(V))$ due to $O(E)$ change key operations. Use $O(V)$ memory.

When some distances < 0 , can have **negative cycles** of distance < 0 , traversing which makes the total distance arbitrarily small. This happens, e.g., if \exists an arbitrage opportunity in a currency market, with possible exchanges modeled as edges. If \exists a negative cycle, shortest paths are undefined.

With some distances < 0 , Dijkstra's algorithm is wrong and unable to detect negative cycles. **Bellman-Ford algorithm** doesn't assume that the best distance to a vertex is known after it's processed:

1. Set the distances to all vertices to ∞
2. Enqueue the source with distance 0
3. Until the queue is empty or a negative cycle is found
4. Dequeue a vertex
5. \forall child vertex
6. Enqueue it if not already on the queue
7. The distance = the distance to the vertex + distance(the vertex, destination) if shorter

The result gives \forall vertex the next vertex on the path back to the source.

```
template<typename GRAPH> struct BellmanFord
{
    int v; //must be first
    Vector<double> distances;
    Vector<int> pred;
    bool hasNegativeCycle;
    BellmanFord(GRAPH& g, int from): v(g.nVertices()), pred(v, -1),
        distances(v, numeric_limits<double>::infinity()),
        hasNegativeCycle(false)
    {
        assert(from >= 0 && from < v);
        Queue<int> queue;
        Vector<bool> onQ(v, false);
        distances[from] = 0;
        queue.push(from);
        onQ[from] = true;
        for(int nIterations = 0; !queue.isEmpty() && !hasNegativeCycle;)
        {
            int i = queue.pop();
            onQ[i] = false;
            for(typename GRAPH::AdjacencyIterator j = g.begin(i);
                j != g.end(i); ++j)
            {
                double newChildDistance = distances[i] + j.data();
                if(newChildDistance < distances[j.to()])
                {
                    distances[j.to()] = newChildDistance;
                    pred[j.to()] = i; //new best parent
                    if(!onQ[j.to()])
                    {
                        queue.push(j.to());
                        onQ[j.to()] = true;
                    }
                }
            } //check for negative cycles every V inner iterations
            if(++nIterations % v == 0)
                hasNegativeCycle = findNegativeCycle();
        }
    }
};
```

The negative cycle check makes sure that no vertex is its own ancestor and runs every V distance updates to amortize its cost. Using union-find, it joins each vertex with its parent. \exists a cycle if no vertex is in the same subset as its parent before they are joined because otherwise the join completes the cycle.

```
bool findNegativeCycle()
{
    UnionFind uf(v);
    for(int i = 0; i < v; ++i)
```

```

    {
        int parent = pred[i];
        if(parent != -1)
            //can't be in same subset as parent before join
            if(uf.areEquivalent(i, parent)) return true;
            uf.join(i, parent);
    }
}
return false;
}

```

The runtime is $O(VE)$ because each edge is enqueued $\leq V$ times if \exists a negative cycle, otherwise it's detected in this time (Ahuja et al. 1993). In practice, the algorithm is much faster due to less enqueueing.

11.8 Flow Algorithms

Flow is a useful general model of distribution and delivery of goods. Given a connected undirected graph with **source** and **sink** vertices, want an assignment of flows to edges such that:

- Except for the source and the sink, \sum flow into a vertex = \sum flow out of it
- \forall vertex, flow out > 0

The **maximum flow** problem gives each edge a capacity \geq the flow assignment and asks to find a flow assignment maximizing \sum flow out of the source.

Let edge capacities be 2 for (5,3) to (9,1) and 1 for others

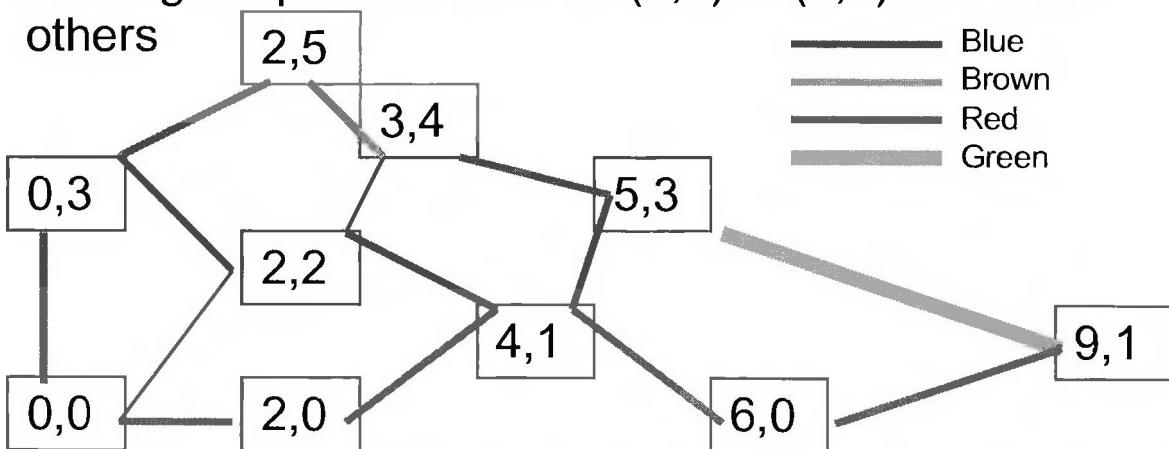


Figure 11.9: Flow comes through the blue, the yellow, and the red paths. The blue and the brown merge into the green.

The **minimum-cost flow** problem gives each edge a capacity ≥ 0 and a cost and asks to find flow \geq the needed amount, such that \sum flow costs is minimal. If the needed amount is too large, don't have a feasible solution.

Let edge capacities be 1 and costs distances for all edges

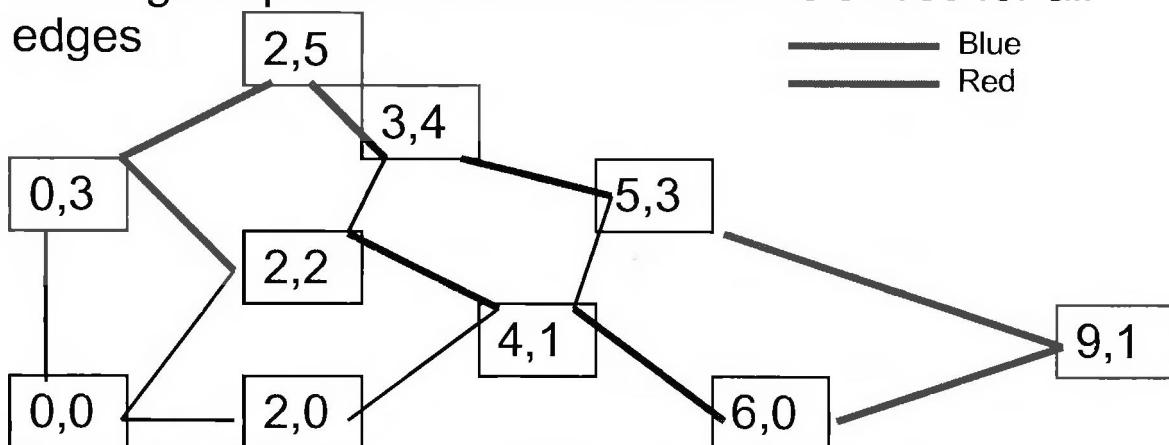


Figure 11.10: Total flow = 2 comes through the blue and the red paths

The flow edge data representation includes a flow direction to allow determining the remaining capacity. For forward flow can send another $capacity - flow$, and for backward can send back flow. Each edge is asso-

ciated with its from-vertex. Can increase or decrease flow along an edge.

```
struct FlowData
{
    int from;
    double flow, capacity, cost; //cost used only for min flow
    FlowData(int theFrom, double theCapacity, double theCost = 0):
        from(theFrom), capacity(theCapacity), flow(0), cost(theCost) {}
    double capacityTo(int v) const {return v == from ? flow : capacity - flow;}
    //flow can step out of (0, capacity) numerically but OK
    void addFlowTo(int v, double change)
        {flow += change * (v == from ? -1 : 1);}
};
```

The augmenting path algorithm for maximum flow:

1. While can find a path from the source to the sink, using which can send flow > 0
2. Send through the path max amount

This works for both maximum and minimum-cost flow, depending on how the paths are found. From the caller expect neededFlow = 0 for maximum flow and -the wanted amount if can have negative costs. To save memory due to undirected edges, the flow data is given in a separate array, and the graph edge data are the indices. The array will also contain the final flow assignments. Data structures pred and path store respectively the vertex sequence and the indices of the corresponding flow edges.

```
template<typename GRAPH> class ShortestAugmentingPath
{
    int v;
    Vector<int> path, pred;
    double totalFlow;
public:
    double getTotalFlow() const {return totalFlow;}
    ShortestAugmentingPath(GRAPH const& g, Vector<FlowData>& fedges, int from,
                           int to, double neededFlow = 0): v(g.nVertices()), totalFlow(0),
                           path(v, -1), pred(v, -1)
    {//iteratively, first find a path
        assert(from >= 0 && from < v && to >= 0 && to < v);
        while(neededFlow == 0 ? hasAugmentingPath(g, fedges, from, to) :
               hasMinCostAugmentingPath(g, fedges, from, to, neededFlow))
        {//then from it the amount of flow to add
            double increment = numeric_limits<double>::max();
            for(int j = to; j != from; j = pred[j])
                increment = min(increment, fedges[path[j]].capacityTo(j));
            if(neededFlow != 0) //only relevant to min cost flow
                increment = min(increment, abs(neededFlow) - totalFlow);
            //then add to all edges
            for(int j = to; j != from; j = pred[j])
                fedges[path[j]].addFlowTo(j, increment);
            totalFlow += increment;
        }
    }
};
```

Finding a path for max flow uses BFS augmented to ignore edges filled to capacity or already part of an existing path:

```
bool hasAugmentingPath(GRAPH const& g, Vector<FlowData>& fedges, int from,
                        int to)
{
    for(int i = 0; i < v; ++i) pred[i] = -1;
    Queue<int> queue;
    queue.push(pred[from] = from);
    while(!queue.isEmpty())
    {
        int i = queue.pop();
        for(typename GRAPH::AdjacencyIterator j = g.begin(i);
              j != g.end(i); ++j)
            if(pred[j.to()] == -1 && //unvisited with capacity
```

```

        fedges[j.data()].capacityTo(j.to()) > 0)
    {
        pred[j.to()] = i;
        path[j.to()] = j.data();
        queue.push(j.to());
    }
}
return pred[to] != -1;
}
}

```

The runtime is $O(VE^2)$ because can have $O(VE)$ augmentations with BFS (Ahuja et al. 1993). In practice the algorithm is much faster.

For minimum cost flow, the algorithm finds the shortest path with respect to cost and ignores full edges. The implementation builds another graph to reuse shortest path algorithms. Then find the corresponding edges and store the flow data indices.

```

bool hasMinCostAugmentingPath(GRAPH const& g, Vector<FlowData>& fedges,
    int from, int to, double neededFlow)
//if need more flow, make a graph from edges with available capacity
if(totalFlow >= abs(neededFlow)) return false;
GraphAA<double> costGraph(v);
for(int i = 0; i < v; ++i)
    for(typename GRAPH::AdjacencyIterator j = g.begin(i);
        j != g.end(i); ++j)
        if(fedges[j.data()].capacityTo(j.to()) > 0)
            costGraph.addEdge(i, j.to(), fedges[j.data()].cost);
if(neededFlow > 0) pred = ShortestPath(costGraph, from, to);
else
//negative costs
    BellmanFord<GraphAA<double>> bf(costGraph, from);
    if(bf.hasNegativeCycle)
    {
        totalFlow = numeric_limits<double>::infinity();
        return false;
    }
    pred = bf.pred;
};

//extract edges for the path
for(int i = to; pred[i] != -1; i = pred[i])
    for(typename GRAPH::AdjacencyIterator j = g.begin(pred[i]);
        j != g.end(pred[i]); ++j)
        if(j.to() == i)
        {
            path[i] = j.data();
            break;
        }
return pred[to] != -1;
}
}

```

The algorithm is fast in practice but finds shortest paths $O(VU)$ time, where $U \geq$ any vertex outflow (Ahuja et al. 1993).

11.9 Bipartite Matching

Given two groups of vertices and a set of allowed edges, want a subset of the latter such that as many vertices as possible have only one edge. The matching is **perfect** if all vertices get an edge.

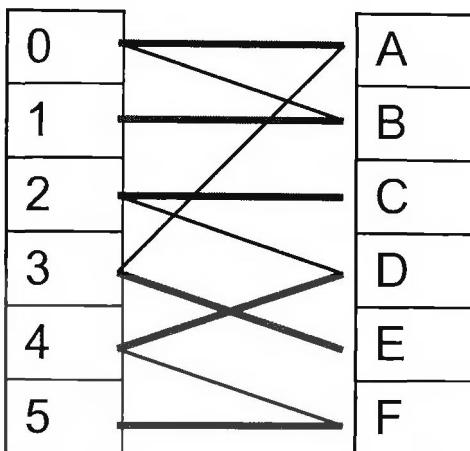


Figure 11.11: The wide edges form a perfect matching

The simplest solution is reducing to maximum flow. Attach a source to all vertices in group one, a sink to all in group two, and give all edges capacity 1. The runtime is $O(VE)$ (Ahuja et al. 1993).

```
Vector<pair<int, int>> bipartiteMatching(int n, int m,
    Vector<pair<int, int>> const allowedMatches)
{ //v = n + m + 2, s = n + m + allowedMatches.getSize(), time is O(ve)
    GraphAA<int> sp(n + m + 2); //setup graph and flow edges
    Vector<FlowData> data;
    for(int i = 0; i < allowedMatches.getSize(); ++i)
    {
        data.append(FlowData(allowedMatches[i].first, 1));
        sp.addUndirectedEdge(allowedMatches[i].first,
            allowedMatches[i].second, i);
    }
    int source = n + m, sink = source + 1; //setup source and sink groups
    for(int i = 0; i < source; ++i)
    {
        int from = i, to = sink;
        if(i < n)
        {
            from = source;
            to = i;
        }
        data.append(FlowData(from, 1));
        sp.addUndirectedEdge(from, to, i + allowedMatches.getSize());
    } //calculate the matching
    ShortestAugmentingPath<GraphAA<int>> dk(sp, data, source, sink);
    //return edges with positive flow
    Vector<pair<int, int>> result;
    for(int i = 0; i < allowedMatches.getSize(); ++i)
        if(data[i].flow > 0) result.append(allowedMatches[i]);
    return result;
}
```

11.10 Stable Matching

When each of m “men” and n “women” has preferences, a matching is **stable** if no two pairs can break up because a man in one pair and a woman in the other prefer each other to their current partners. If $n \leq m$, men are choosing women, otherwise women are choosing men. Without loss of generality assume $n \leq m$. For convenience, the preferences for a man are an ordered list of women, and for a women the equivalent ranks of men.

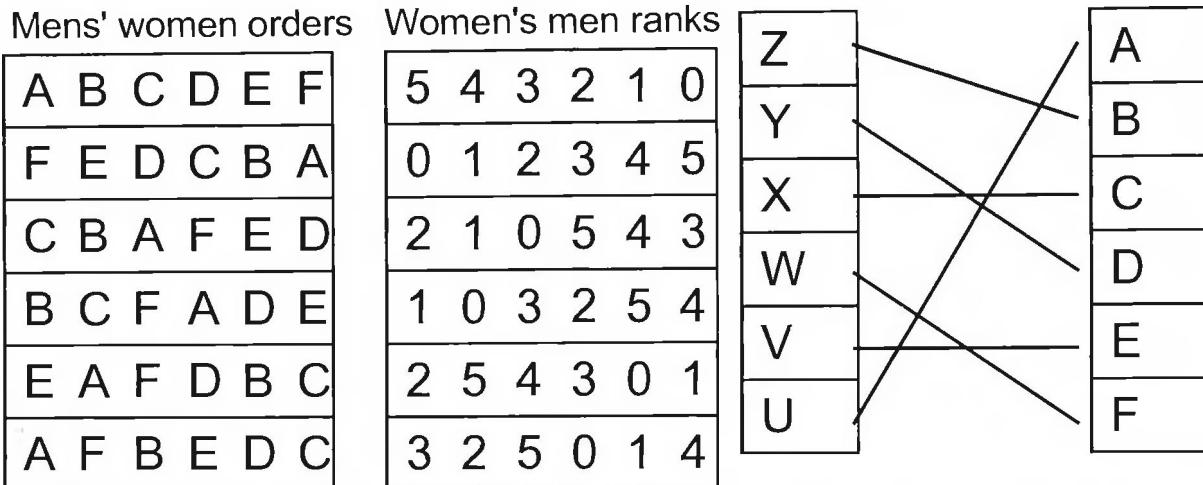


Figure 11.12: Preference ranks and the resulting matches (the men start with "Z" and the women with "A")

Gale-Shapley algorithm computes a stable matching iteratively:

1. Initially all are unassigned
2. Until all men are assigned
3. Any unassigned man proposes to the best woman to whom he hasn't proposed
4. She accepts him if he > her current partner and rejects otherwise

```
Vector<int> stableMatching(Vector<Vector<int>> const& womenOrders,
                           Vector<Vector<int>> const& menScores)
{
    int n = womenOrders.getSize(), m = menScores.getSize();
    assert(n <= m);
    Stack<int> unassignedMen; //any list type will do
    for(int i = 0; i < n; ++i) unassignedMen.push(i);
    Vector<int> currentMan(m, -1), nextWoman(n, 0);
    while(!unassignedMen.isEmpty())
    {
        int man = unassignedMen.pop(), woman, currentM;
        do
            { //won't run out of bounds due to n <= m
                woman = nextWoman[man]++;
                currentM = currentMan[woman];
            } while(currentM != -1 && //man finds best woman that prefers him
                    menScores[woman][man] <= menScores[woman][currentM]);
        currentMan[woman] = man; //found match
        //divorcee, if any, to search more
        if(currentM != -1) unassignedMen.push(currentM);
    }
    return currentMan;
}
```

If preferences of each woman are the same, the algorithm runs fastest when men are in order of descending rank, in which case no woman chooses a different partner. Because each woman and man are paired at most once, the runtime is $O(nm)$.

11.11 Assignment Problem

Similar to bipartite matching, but with edges having weights, and want a matching that minimizes \sum edge costs.

Let edges cost 1 if straight and 0 if diagonal

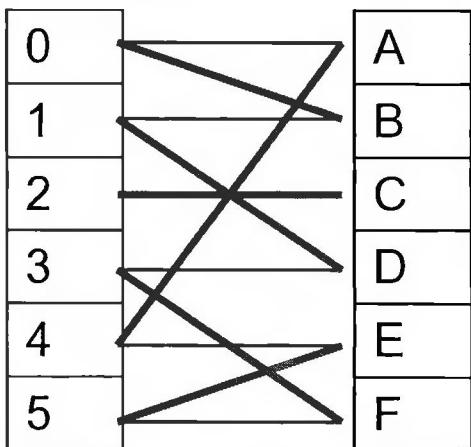


Figure 11.13: Change in matchings from Figure 5.10 when diagonals are free

A simple solution is reducing to minimum-cost flow. It's the same as for bipartite matching, except using weights $\neq 1$.

```
Vector<pair<int, int>> assignmentProblem(int n, int m,
    Vector<pair<pair<int, int>, double>> const allowedMatches)
{ //v = n + m + 2, e = n + m + allowedMatches.getSize()
    GraphAA<int> sp(n + m + 2); //setup graph and flow edges
    Vector<FlowData> data;
    for(int i = 0; i < allowedMatches.getSize(); ++i)
    {
        data.append(FlowData(allowedMatches[i].first.first, 1,
            allowedMatches[i].second));
        sp.addUndirectedEdge(allowedMatches[i].first.first,
            allowedMatches[i].first.second, i);
    }
    int source = n + m, sink = source + 1; //setup source and sink groups
    for(int i = 0; i < source; ++i)
    {
        int from = i, to = sink;
        if(i < n)
        {
            from = source;
            to = i;
        }
        data.append(FlowData(from, 1, 0));
        sp.addUndirectedEdge(from, to, i + allowedMatches.getSize());
    } //calculate the matching
    ShortestAugmentingPath<GraphAA<int>> dummy(sp, data, source, sink,
        min(n, m));
    //return edges with positive flow
    Vector<pair<int, int>> result;
    for(int i = 0; i < allowedMatches.getSize(); ++i)
        if(data[i].flow > 0) result.append(allowedMatches[i].first);
    return result;
}
```

When using the shortest augmenting path method, the runtime is $O(V^2E)$ with Bellman-Ford (Ahuja et al. 1993).

11.12 Generating Random Graphs

Proper testing of graph algorithms needs large graphs. A generated graph should be sparse and may need to satisfy other algorithm-specific requirements.

A simple method is to create an edge between two vertices with probability p . But $E[\text{the space use of the result}] = O(pV^2)$. A more useful model is to have a directed graph with k outgoing edges per vertex:

```
template<typename GRAPH>
```

```

GRAPH randomDirectedGraph(int vertices, int edgesPerVertex)
{
    assert(edgesPerVertex <= vertices);
    GRAPH g(vertices);
    for(int i = 0; i < vertices; ++i)
    {
        Vector<int> edges = GlobalRNG().sortedSample(edgesPerVertex, vertices);
        for(int j = 0; j < edgesPerVertex; ++j) g.addEdge(i, edges[i]);
    }
    return g;
}

```

Another useful model is generating n points in the unit square and creating a graph that connects each vertex to its k nearest neighbors or to all points within some distance, but these are less efficient. Batagelj & Brandes (2005) give efficient generators for various models.

11.13 Implementations Notes

The only novelty is the use of an indexed heap in shortest path and MST calculations. All other algorithms are standard.

The network flow algorithms are perhaps the hardest to find because few sources discuss them. My simple implementations, particularly for the minimum cost flow, don't show up anywhere else, but they aren't using the best algorithms (see the "Comments" section).

11.14 Comments

For finding an MST, **Kruskal's algorithm** sorts edges by distance and iteratively adds the next shortest edge that connects two not-yet-connected components, found using union-find. But it needs $O(E)$ memory and doesn't work directly with the adjacency array representation.

Dijkstra's algorithm can also find a shortest path with the shortest longest edge by using $\max(\text{distance to the vertex}, \text{distance}(\text{the vertex, child}))$ instead of $\text{distance to the vertex} + \text{distance}(\text{the vertex, child})$, but this is rarely useful.

For finding shortest paths in very large graphs, preprocessing gives huge speedups for queries. Google **contraction hierarchies** if curious. A different model is finding several shortest paths of good quality, so that users can pick the best according to their needs. Finding several different shortest paths isn't useful because they tend to be very similar. A better model sets required edge alternatives or defines several distance functions using, e.g., travel time, fuel costs, number of transfers, etc., solving for each.

The fastest known flow algorithms use more complicated **push-relabel** techniques (Ahuja et al. 1993; Mehlhorn & Näher 1999). Also see Goldberg & Tarjan (2014) and Thulasiraman et al. (2015) for a brief review and some recent references. For minimum-cost flow some recent experimental work is Kiraly & Kovacs (2012) and Kovács (2015). Most other graph algorithms and their practical implementations have been resolved by about 1990. One topic not discussed here is matchings. It and other less popular, complicated algorithmic topics such as planarity testing, with much supporting theory, are discussed in Kocay & Kreher (2016) and Thulasiraman et al. (2015).

11.15 Projects

- Investigate and implement the main push-relabel algorithms for maximum and minimum-cost flows. Are they faster on large test problems than the corresponding augmenting path algorithms?
- Add 0 checks for random graph generation parameters

11.16 References

- Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. MIT Press.
- Batagelj, V., & Brandes, U. (2005). Efficient generation of large random networks. *Physical Review E*, 71(3), 036113.
- Goldberg, A. V., & Tarjan, R. E. (2014). Efficient maximum flow algorithms. *Communications of the ACM*, 57(8), 82-89.
- Heineman, G. T., Pollice, G., & Selkow, S. (2016). *Algorithms in a Nutshell*. O'Reilly.
- Kiraly, Z., & Kovacs, P. (2012). Efficient implementations of minimum-cost flow algorithms. *Acta Univ. Sapientiae Mathematica*, 4(1), 1-18.

- entiae*, 4(1), 67-118.
- Kocay, W., & Kreher, D. L. (2016). *Graphs, Algorithms, and Optimization*. CRC.
- Kovács, P. (2015). Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software*, 30(1), 94-127.
- Mehlhorn, K., & Näher, S. (1999). *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press.
- Mehlhorn, K., & Sanders, P., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.
- Thulasiraman, K., Arumugam, S., Nishizeki, T., & Brandstädt, A. (2015). *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*. CRC.

12 Miscellaneous Algorithms and Techniques

12.1 Introduction

This chapter is the first with material outside the standard undergraduate curriculum. It contains topics not discussed enough to make independent chapters, mostly general data structure techniques, cache algorithms, and combinatorial generation.

12.2 Making Static Data Structures Dynamic

A data structure is **dynamic** if it supports updates, **semi-dynamic** if it doesn't support deletions, and **static** if it doesn't support updates. If parameters determined at construction prevent growing, **rebuilding** applies if have enough information. E.g., it applies to a vector but not to a Bloom filter.

Partial rebuilding rebuilds part of a data structure after a sequence of updates or an event. E.g., to balance a tree without rotating, randomized rebuilding keeps the tree random by giving every node in every subtree an equal chance of being its root. Keep node counts, and insertion into a subtree with n nodes rebuilds it with probability $1/n$, using incremental construction that processes the new node and then the nodes in the existing subtree, rooting the subtree with the new node. With probability $\frac{n-1}{n}$ insertion proceeds as unbalanced tree insertion. Because for a random tree $\Pr(\text{height} > \text{clg}(n))$ for $c > 2$) is exponentially small, any node has \leq factor $1/b$ more descendants relative to its sibling, for $0.5 < b < 1$. If rebuilding costs $n \lg(n)$, $E[\text{the cost of insertion}] = C(n) \leq \frac{1}{n} n \lg(n) + \frac{n-1}{n} C(nb) \approx \lg(n) + C(nb)$. Using the master theorem, $C(n) = O(\lg(n)^2)$. This holds if b is $O(1)$ -bounded away from 0, and the expected behavior is better than the worst case.

Another way to maintain $0.5 < b < 1$ is creating a perfectly balanced subtree out of the highest subtree made unbalanced by insertion. The amortized cost of insertion is $O(\lg(n)^2)$ if rebuilding takes $O(n \lg(n))$ time, though on average it's smaller. Weight balance guarantees $O(\lg(n))$ height (Overmars 1983).

Total rebuilding rebuilds the whole data structure. **Array doubling** does this for vector. Updates are **weak** if after $O(n)$ updates the resource use of all operations increases by $O(1)$. In this case, rebuilding after every $O(n)$ updates cuts the amortized cost of rebuilding by a factor of $O(n)$.

The ability to rebuild and weakly delete enables making any semi-dynamic data structure dynamic. When deleting an item, mark it with a Boolean **tombstone**. After deleting enough items, rebuild to remove them. The cost of a weak deletion is amortized $O(\text{rebuild}/\text{the number of deleted items})$. Rebuilding for deletions must be rarer than for insertions, so that a sequence of insertions and deletions doesn't cause frequent rebuilds.

A problem is **order-decomposable** if can derive the result of a query on a data set by combining results of queries on any of the set's partitions. Order-decomposability allows representing a data structure by a collection of blocks. In particular, for a data structure with n items, blocks have sizes 2^i for $0 \leq i \leq \lg(n)$, $\sum \text{sizes} \geq n$, and each i is unique; have $\leq \lg(n)$ blocks. Insertion builds a structure of size 1 and, if have one already, rebuilds both into a structure of size 2, etc. This is like maintaining a binary counter and enables amortized $O(\lg(n))$ insertion when rebuilding is $O(n)$. Deletions are weak.

If blocks support insertion before they are full, inserting into the largest block until it's full and allocating another block of twice the size is more efficient.

12.3 Making Data Structures Persistent

After every update, a data structure as a whole is different, but may want to record its past versions. E.g., a version-controlled code commit creates a new version of the repository. A data structure is **partially persistent** when each version is accessible, and updates affect only the latest version. Cloning the data structure on every update does this.

The **fat node method** is more efficient. When the data structure is a collection of linked nodes, replace each node pointer with a vector of (version number, pointer) pairs, and keep a data structure version number.

- During an update, copy and update every affected node, append (the new version number, the new pointer) to every affected vector, and increment the version number. The extra cost of an update is the time to copy the affected nodes.

- To access the latest version, use the last pair in each vector.
- To access a version k versions ago in extra $O(\lg(k))$ time, use binary search.

A persistent node can be the whole data structure, a single bit, or something in-between, such as a memory location, and unrelated to the specifics of the data structure. Choose nodes so that an update touches few of them. E.g., for a tree make each node persistent, and copy if item or child pointers change:

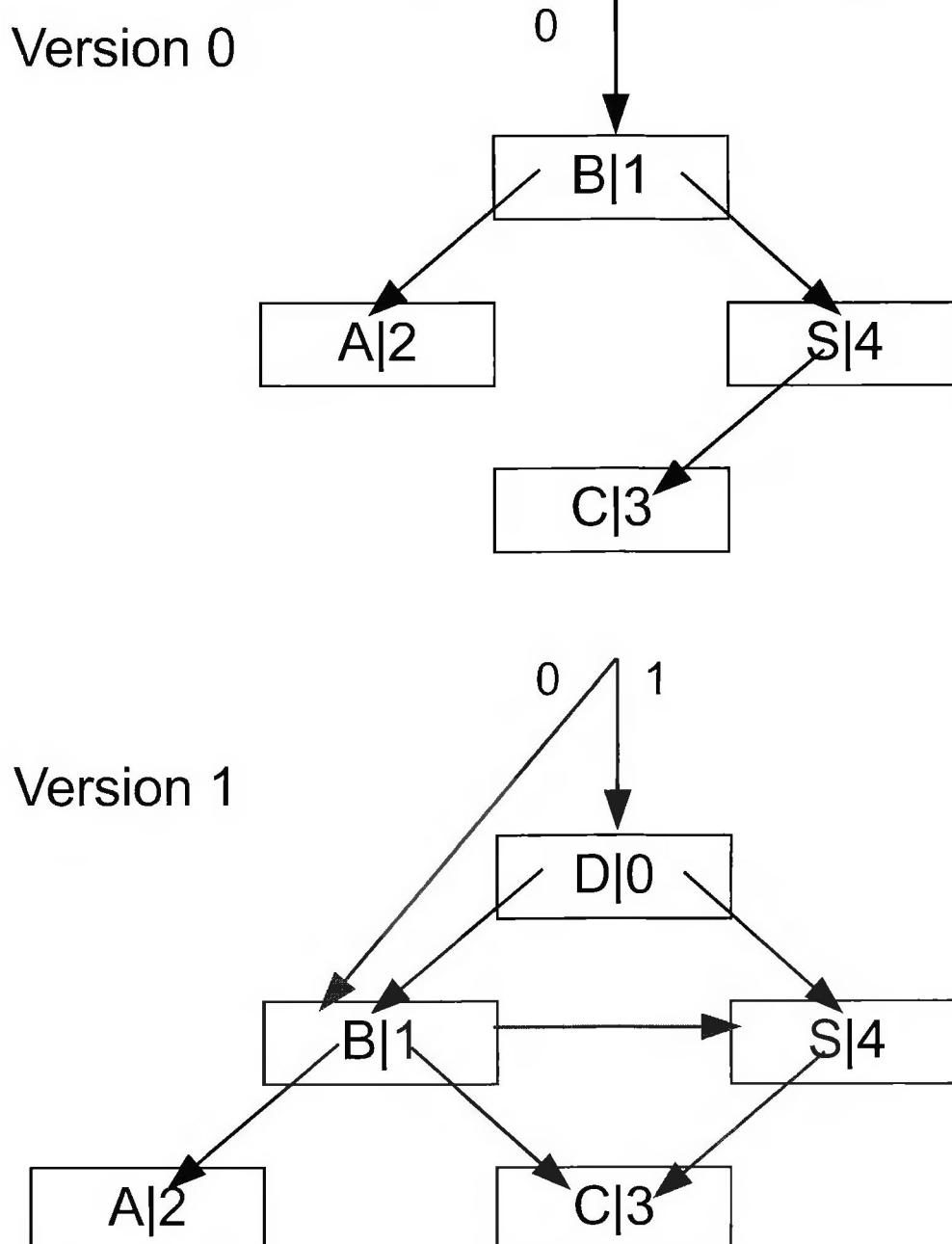


Figure 12.1: Persistent treap after insertion of 'D' with priority 0

12.4 Maintaining a Cache

Size and replacement policy define cache behavior. For a fixed size, an ideal policy caches resources to optimize their future access. The **least recently used** (LRU) policy has competitive ratio 2 (Wikipedia 2013). When the cache is full, and a resource \notin cache is accessed, it evicts the oldest accessed resource. The optimal implementation is using a linked list, indexed by a hash table:

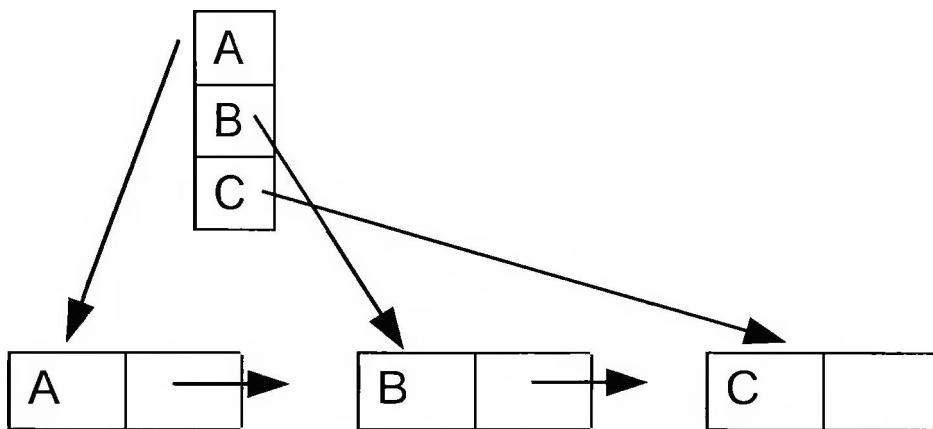


Figure 12.2: LRU cache structure, using a hash table and a linked list

The list orders items by access time, moving accessed items to the front, and the hash table allows efficient search, so that the operations take $O(\text{hash table})$ time.

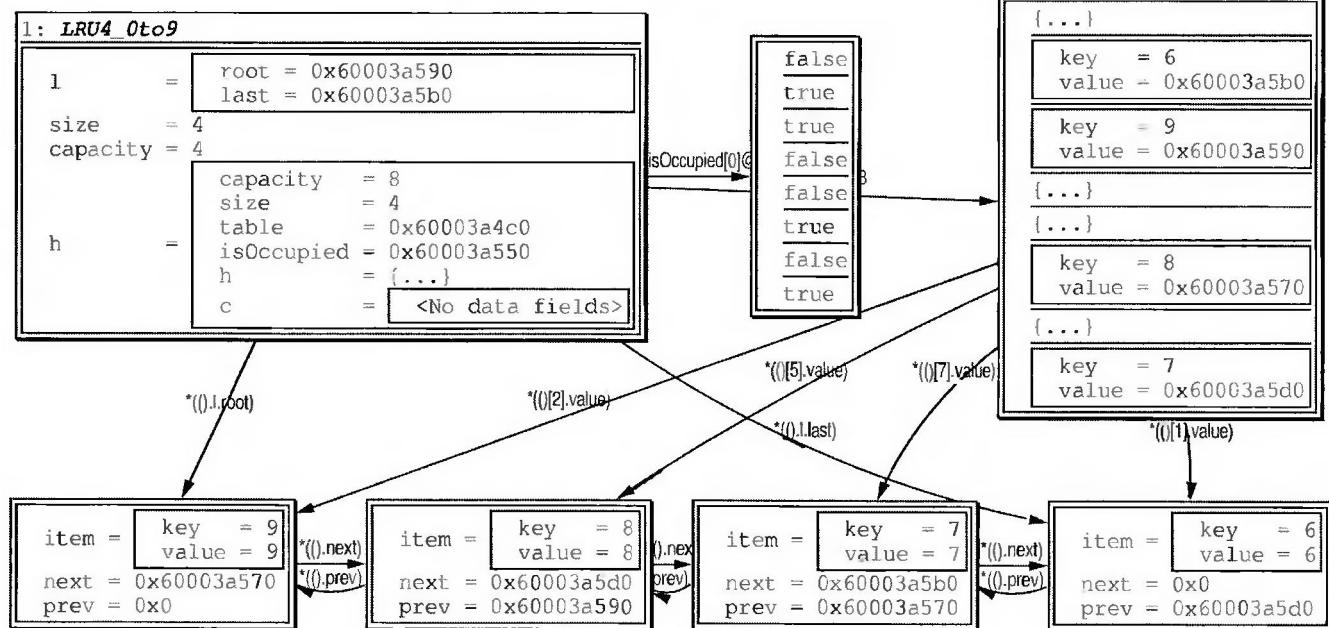


Figure 12.3: Memory layout of a LRU cache with integer items 0–9 inserted and 0–5 removed

```
template<typename KEY, typename VALUE,
        typename HASHER = EHash<BUHash>> class LRUCache
{
    typedef KVPair<KEY, VALUE> ITEM;
    typedef SimpleDoublyLinkedList<ITEM> LIST;
    typedef typename LIST::Iterator I;
    LIST l;
    int size, capacity;
    LinearProbingHashTable<KEY, I, HASHER> h;
public:
    LRUCache(int theCapacity): size(0), capacity(theCapacity)
    {
        assert(capacity > 0);
    }
    VALUE* read(KEY const& k)
    {
        I* np = h.find(k);
        if(np)
        {
            //put in front on access
            l.moveBefore(*np, l.begin());
            return &(*np)->value;
        }
        return 0;
    }
    typedef I Iterator;
    Iterator begin(){return l.begin();}
    Iterator end(){return l.end();}
}
```

```

Iterator evicteeOnWrite(KEY const& k) //none if not full or item in cache
{
    if (size < capacity || h.find(k) == end()) : l.rBegin();
    void write(KEY const& k, VALUE const& v)
    {
        VALUE* oldV = read(k); //first check if already inserted
        if (oldV != oldV) *oldV = v; //found, update
        else
        {
            Iterator evictee = evicteeOnWrite(k);
            if (evictee != end())
            {
                h.remove(evictee->key);
                l.moveBefore(evictee, l.begin()); //recycle evictee
                evictee->key = k;
                evictee->value = v;
            }
            else
            {
                ++size;
                l.prepend(ITEM(k, v));
            }
            h.insert(k, l.begin());
        }
    }
}

```

Can use this to implement a facade over various resources, so that the API users are unaware that a cache is used, e.g., as is usually the case for disk access. The main difference between the various facades is whether need to write, and if yes, whether commit immediately or as late as possible, with the latter being the most useful option. Here need to check if any evicted block has changes and flush it.

```

template <typename KEY, typename VALUE, typename RESOURCE,
         typename HASHER = EHash<BUHash> > class DelayedCommitLRUCache
{
    RESOURCE& r;
    typedef LRUCache<KEY, pair<VALUE, bool>, HASHER> LRU;
    typedef typename LRU::Iterator I;
    LRU c;
    void commit(I i)
    {
        if (i->value.second) r.write(i->key, i->value.first);
        i->value.second = false;
    }
    void writeHelper(KEY const& k, VALUE const& v, bool fromWrite)
    { //first commit evictee if any
        I i = c.evicteeOnWrite(k);
        if (i != c.end()) commit(i);
        c.write(k, pair<VALUE, bool>(v, fromWrite));
    }
    DelayedCommitLRUCache(DelayedCommitLRUCache const&); //no copying allowed
    DelayedCommitLRUCache& operator=(DelayedCommitLRUCache const&);

public:
    DelayedCommitLRUCache(RESOURCE& theR, int capacity): r(theR), c(capacity)
    {
        assert(capacity > 0);
    }
    VALUE const& read(KEY const& k)
    { //first check if in cache
        pair<VALUE, bool>* mv = c.read(k);
        if (!mv)
        { //if not then read from resource and put in cache
            writeHelper(k, r.read(k), false);
            mv = c.read(k);
        }
        return mv->first;
    }
}

```

```

void write(KEY const& k, VALUE const& v){writeHelper(k, v, true);}
void flush(){for(I i = c.begin(); i != c.end(); ++i) commit(i);}
~DelayedCommitLRUCache(){flush();}
};

```

12.5 k-bit-word Vector

To save space for representing sequences of enumerations such as DNA, can use a vector of k -bit integers representing values $\in [0, 2^k - 1]$. It's a facade implemented on top of bitset. Random access is $O(1)$, and append amortized $O(1)$. But it's not particularly useful in general, so the implementation is minimal.

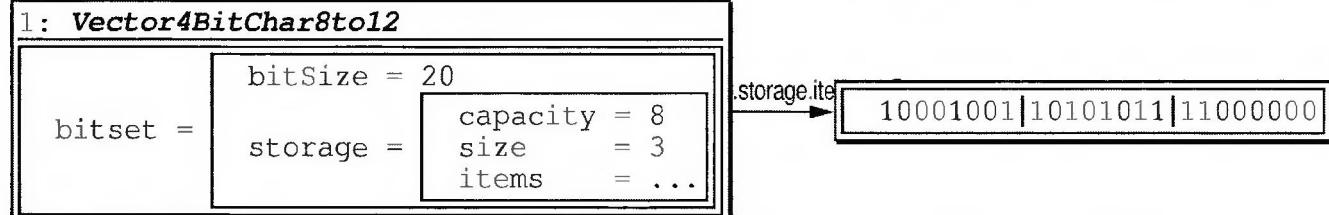


Figure 12.4: Memory layout of a 4-bit-word vector with byte storage containing integers 8–12

```

template<int N, typename WORD = unsigned long long> class KBitWordVector
{
    Bitset<WORD> b;
public:
    unsigned long long getSize() const{return b.getSize()/N;}
    KBitWordVector(): b();
    KBitWordVector(int n, WORD item = 0): b(n * N)
    {
        if(Bits::getValue(item, 0, N) == 0) b.setAll(0);
        else for(unsigned long long i = 0; i < getSize(); ++i) set(item, i);
    }
    WORD operator[](unsigned long long i) const
    {
        assert(i < getSize()); return b.getValue(i * N, N);
    }
    void set(WORD value, unsigned long long i)
    {
        assert(i < getSize()); b.setValue(value, i * N, N);
    }
    void append(WORD value){b.appendValue(value, N);}
};

```

12.6 Set Union on Intervals

Maintain partitions of the interval $[0, n]$, and can find a subinterval containing $0 \leq i \leq n$, merge two adjacent subintervals, and split a subinterval. A simple solution is to store the split points in a dynamic sorted sequence where a split point is the rightmost point in the interval and represents the interval. Then find, merge, and split correspond to successor, delete, and insert respectively:

```

class IntervalSetUnion
{
    Treap<int, bool> treap;
public:
    int find(int i){return treap.getSize() == 0 ? -1: treap.successor(i)->key;}
    void merge(int i){return treap.remove(i);}
    void split(int i){treap.insert(i, 0);}
};

```

The use case is rare though.

12.7 Generating the First N Primes

Sieve of Eratosthenes keeps a list of first n numbers and \forall prime $\leq \sqrt{n}$ removes its multiples (Wikipedia 2016). The list correctly tells which numbers are prime when going through it and removing in increasing order. For memory efficiency the list is a bit set representing odd numbers ≥ 3 .

```

class PrimeTable
{
    long long maxN;
    Bitset<> table; //marks odd numbers starting from 3
};

```

```

long long nToI(long long n) const {return (n - 3)/2; }

public:
    PrimeTable(long long primesUpto) : maxN(primesUpto - 1),
        table(nToI(maxN) + 1)
    {
        assert(primesUpto > 1);
        table.setAll(true);
        for (long long i = 3; i <= sqrt(maxN); i += 2)
            if (isPrime(i)) // set every odd multiple i <= k <= maxN/i to false
                for (long long k = i; i * k <= maxN; k += 2)
                    table.set(nToI(i * k), false);
    }
    bool isPrime(long long n) const
    {
        assert(n > 0 && n <= maxN);
        return n == 2 || (n > 2 && n % 2 && table[nToI(n)]);
    }
};

```

The algorithm needs $O(n)$ time and $\approx n/16$ bytes of space.

12.8 Generating All Permutations

To compute the lexicographic successor of a permutation:

1. Find the last element < the next element
2. Swap it with its successor from the elements to its right
3. Reverse the latter

E.g., for 045837621, swap 3 with 6, and reverse to produce 045861237. This works because the elements to the right are in decreasing order after the swap.

To skip a class of permutations, sort the remaining elements in decreasing order, and go to the next permutation. E.g., if don't need permutations beginning with 04586, skip them by producing 045867321 and going to 045871236.

```

struct Permutator
{
    Vector<int> p;
    Permutator(int size) {for (int i = 0; i < size; ++i) p.append(i);}

    bool next()
    { // find largest i such that p[i] < p[i + 1]
        int j = p.getSize() - 1, i = j - 1; // start with one-before-last
        while (i >= 0 && p[i] >= p[i + 1]) --i;
        bool backToIdentity = i == -1;
        if (!backToIdentity)
            { // find j such that p[j] is next largest element after p[i]
                while (i < j && p[i] >= p[j]) --j;
                swap(p[i], p[j]);
            }
        p.reverse(i + 1, p.getSize() - 1);
        return backToIdentity; // true if returned to smallest permutation
    }

    bool advance(int i)
    {
        assert(i >= 0 && i < p.getSize());
        quickSort(p.getArray(), i + 1, p.getSize() - 1,
                  ReverseComparator<int>());
        return next();
    }
};

```

Going to the next permutation takes worst-case $O(n)$ and amortized $O(1)$ time for a permutation of n elements because $(n - m)!$ out of $n!$ permutations need reversing m elements, and the total average work = $\sum_{0 \leq m < n} \frac{m(n-m)!}{n!}$ = $O(1)$. Visiting all permutations takes $O(nn!)$ time and is the bottleneck. The algorithm also works with repeated elements and can be templated to work on any copyable and comparable ele-

ments.

12.9 Generating All Combinations

Because order doesn't matter for combinations, work with sorted order for algorithm simplicity. To compute the lexicographic successor of a combination c of m out of n numbers $\in [0, n - 1]$:

1. Find the last index i at which $c[i] < n - m + i$
2. Increment $c[i]$
3. $\forall j > i$ reset $c[j]$ to $c[j - 1] + 1$

E.g., for a (4, 6) combination 0145, $i = 1$, and the next combination is 0234. Skipping at i resets at i to the largest possible values. The algorithm takes $O(m)$ time per combination.

```
struct Combinator
{
    int n;
    Vector<int> c;
    Combinator(int m, int theN): n(theN), c(m, -1)
    {
        assert(m <= n && m > 0);
        skipAfter(0);
    }
    void skipAfter(int i)
    { //increment c[i], and reset all c[j] for j > i
        assert(i >= 0 && i < c.getSize());
        ++c[i];
        for (int j = i + 1; j < c.getSize(); ++j) c[j] = c[j - 1] + 1;
    }
    bool next()
    { //find rightmost c[i] which can be increased
        int i = c.getSize() - 1;
        while(i >= 0 && c[i] == n - c.getSize() + i) --i;
        bool finished = i == -1;
        if (!finished) skipAfter(i);
        return finished;
    }
};
```

12.10 Generating All Subsets

A subset of a set of size m is most economically represented as a bit string of size m with present items marked with 1 bits. To generate all subsets in reverse lexicographic order, set the bit string to $2^n - 1$, and decrement until 0. To skip, zero out the wanted lower bits. The operations take $O(1)$ time if use a single word as the bit string.

This method is an application of the general **ranking/unranking** technique. The idea is to define a mapping from the objects of interest to integers. Then can generate all objects by counting and unranking the current integer.

12.11 Generating All Partitions

A **partition** is an equivalence relation, where each item belongs to a group. For n items the most economical representation is an array p of group numbers starting from 0. Because putting every item in group 0 = putting every item in any other group, the maximum allowable value of $p[i]$ is $m_i = \begin{cases} \max(m_{i-1}, p[i-1]+1), & i > 0 \\ 0, & i = 0 \end{cases}$. To generate the lexicographic successor, increase the rightmost increaseable element, and set the rest to 0. E.g., for 010123, the second 0 is the rightmost increaseable element, and the next partition is 011000. To skip at i , make the values to its right the largest possible, and generate the successor. The algorithm takes $O(m)$ time per partition.

```
struct Partitioner
{
    Vector<int> p;
    Partitioner(int n): p(n, 0) {assert(n > 0);}
```

```

bool skipAfter(int k)
{ //set trailing elements to maximum values and call next
    assert(k >= 0 && k < p.getSize());
    for(int i = k; i < p.getSize(); ++i) p[i] = i;
    return next();
}
bool next()
{ //find rightmost p[j] which can be increased
    int m = 0, j = -1;
    for(int i = 0; i < p.getSize(); ++i)
    {
        if(p[i] < m) j = i;
        m = max(m, p[i] + 1);
    }
    bool finished = j == -1;
    if(!finished)
    { //increase it and reset the tail
        ++p[j];
        for(int i = j + 1; i < p.getSize(); ++i) p[i] = 0;
    }
    return finished;
}
};

```

12.12 Generating All Constrained Objects

E.g., want to generate all syntax trees for a function of five variables using only three levels of operators "+" and "*". For generating all values of any type:

- Check if the type is isomorphic to an easily generatable type
- Find the most economical representation
- Use lexicographic order
- Avoid storing state

If all else fails, can generate a less constrained class of objects, and accept/reject each, ultimately generating all binary sequences of some large length and outputting those that represent valid objects. But at some point this becomes very inefficient.

12.13 Implementation Notes

Using a LRU cache to create a delayed-commit cache is an original API choice that proves convenient for disk algorithms (see the "External Memory" algorithms chapter). Despite the easy algorithmic idea and available components the implementation is tricky to get right, and I found several bugs over the years.

The other implementations are fairly simple, needing only one source for all the details.

12.14 Comments

Can make array doubling worst-case-efficient, but doing so is clumsy, memory-inefficient, and not useful. When the array is $\frac{1}{2}$ full, create another of twice its capacity. On append, append the item to both, and copy a different item from the old array to the new one. So when the old array is full, the new one has all items and is $\frac{1}{2}$ full. Delete the old array, and repeat the process.

The idea behind k -bit-word vector is original, though somewhat obvious.

12.15 Projects

- Implement automated tests for generation algorithms. One idea is to use small n , and make sure that all possibilities are reached in the correct order. Same for jumps.
- Research and implement algorithms for computing approximate percentiles in large data sets and other big data algorithms such as hyperloglog.

12.16 References

Brass, P. (2008). *Advanced Data Structures*. Cambridge University Press.

- Overmars, M. H. (1983). *The Design of Dynamic Data Structures*. Springer.
- Nayak, A., & Stojmenovic, I. (Eds.). (2007). *Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems*. Wiley-IEEE Press.
- Wikipedia (2013). Cache algorithms. http://en.wikipedia.org/wiki/Cache_algorithms. Accessed May 18, 2013.
- Wikipedia (2016). Sieve of Eratosthenes. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes. Accessed October 30, 2016.

13 External Memory Algorithms

13.1 Introduction

It's important to understand how to work with files on disk, particularly sequential and map layouts. So these are the ones I emphasize with implementation details.

13.2 Disks and Files

A **disk drive** is a slow-to-access persistent memory. Technologies differ, but in all cases:

- Random access called **seeking** is supported
- Writing is negligibly or substantially slower than reading
- Reading and writing contiguous-byte blocks is orders of magnitude faster than accessing one byte at a time
- Both seeking and **data transfer** are costly, and data transfer is costlier
- Need to read the data into internal memory to work on it

Logically, a **file** is an array of bytes. In C++ a file is a dynamic array that remembers the last accessed index and increments it after an I/O. The OS handles the dynamic aspect. The implementation here is designed for byte-block access and uses a subset of the C++ file API that implements such an abstraction. See the code comments for some design decisions. Beware:

- The OS limits the number of files open by a single executable, so close unneeded files.
- C++ `fstream` fails to open a file for writing if the permissions don't allow writes.
- The random access pointer of the read stream is a **get pointer**, and of the write stream a **put pointer**. In `fstream` they are the same because a single buffer is used, and can use either set of accessors. Flush after every write if both reading and writing because otherwise the behavior of a read after a write is undefined.
- `fstream` is likely (i.e., not standard-guaranteed but implementations do it) to use a buffer of size `BUFSIZ` (macro from `<cstdio>`). So reading several smaller-size contiguous blocks is still fast, but writing is slow due to needing to flush per above. It's also likely that for blocks exceeding the buffer size the I/O is done directly instead of being broken up in `BUFSIZ` chunks.

```
class File
{
    fstream f; //C++ IO object
    long long size; //cached here for efficiency
    void create(const char* filename) {ofstream dummy(filename, ios::trunc); }
    File(File const&); //no copying
    File& operator=(File const&);
    void goToEnd()
    {
        f.seekg(0, ios::end);
        assert(f); //watch out for external issues
    }
public:
    static bool exists(const char* filename) {return bool(ifstream(filename));}
    static void remove(const char* filename)
    {
        if(exists(filename))
        {
            int returnCode = std::remove(filename);
            assert(returnCode == 0); //watch out for external issues
        }
    }
    File(const char* filename, bool truncate)
    {
        if(truncate || !exists(filename)) create(filename);
        f.open(filename, ios::binary | ios::in | ios::out);
        assert(f); //make sure f not locked, etc.
        //calculate size
    }
};
```

```

        goToEnd();
        size = getPosition();
        setPosition(0); //come back to the beginning

    }

long long getPosition(){return f.tellg();}
long long getSize() const{return size;}
long long bytesToEnd(){return getSize() - getPosition();}
void setPosition(long long position)
{
    assert(0 <= position && position <= getSize());
    f.seekg(position);
    assert(f); //watch out for external issues
}
void read(unsigned char* buffer, long long n)
{
    assert(n > 0 && n <= bytesToEnd());
    f.read((char*)buffer, n);
    assert(f); //watch out for external issues
}
void write(unsigned char* buffer, long long n)
{
    assert(n > 0 && n <= bytesToEnd()); //to prevent errors, not needed
    f.write((char*)buffer, n);
    f.flush();
    assert(f); //watch out for external issues
}
void append(unsigned char* buffer, long long n)
{
    goToEnd();
    size += n; //do this first to prevent write assertion fail
    write(buffer, n); //write from one-past-last
}
};

```

On my 2011 computer, file system block size = 4096, and the logical one = 512 = BUFSIZ. Looks like the compiler is trying to set BUFSIZ to the best possible value—on various forums other developers report $\text{BUFSIZ} \in [512, 4096]$. Seems that due to larger disk sizes modern disks and file systems currently prefer 4096 over the older 512 (which resulted in smaller file sizes to save space; see, e.g., <https://support.microsoft.com/en-us/help/140365/default-cluster-size-for-ntfs-fat-and-exfat> and <https://unix.stackexchange.com/questions/178899/optimizing-logical-sector-size-for-physical-sector-size-4096-hdd>).

B	Seconds	Slowdown to Previous
4	0.66	
8	0.65	0.98
16	0.64	0.98
32	0.65	1.01
64	0.66	1.03
128	0.68	1.02
256	0.71	1.05
512	0.74	1.04
1024	0.83	1.12
2048	1.06	1.28
4096	1.47	1.39
8192	2.28	1.55
16384	3.88	1.70
32768	7.15	1.84
65536	13.78	1.93

Figure 13.1: Times for 10^5 reads of random blocks of size B from a file of size 10^7

So experimentally:

- The optimal block size B to work with may exceed BUFSIZ

- The marginal cost of doubling B gradually $\rightarrow 2$. While B doesn't seem sensitive if not too much beyond the file system block size, with high fragmentation the numbers could be much worse. But for newer SSD disks fragmentation isn't an issue due to not needing to seek (see, e.g., <https://www.pcgamer.com/should-i-defrag-my-ssd/>).

On modern computers, $B = \max(\text{BUFSIZ}, 4096)$ is a safe heuristic choice—not too large and not too small and used here as the default for all disk-based algorithms and data structures. Alternatively, trusting `BUFSIZ` is also not wrong. If willing to sacrifice portability, check an OS-specific API.

Using external memory gives:

- Persistence**—any external memory data structure is stored on disk after the computation
- Increased memory—for applications working with massive data !Enough internal memory

External memory algorithms are all about efficiency because not working with contiguous data gives orders-of-magnitude slowdowns. The most common use case of reading a file, doing the computation in memory, and writing the result to another file is the most efficient way to process a file. But in many cases, e.g., for databases, need to work on only a small part of a large file.

13.3 File Layout

Serialization is converting a data structure into a byte sequence. Assuming a single file holds the entire data structure, the following layout patterns can represent every data structure:

- Integers representing offsets from the beginning of a file act as pointers.
- Items and arrays have known fixed or stored variable length.
- Split a file into a variable-length part and a fixed-length item array part to ensure fast contiguous operations. For efficiency the former needs to be padded to ensure that the array items don't land on file system block boundaries.
- Use a fixed-length header to specify file format, check sums, permissions, and other useful data.
- Use **comma-separated value (CSV)** files for simple matrix-like data.
- Use JSON- or XML-like encoding to make complex data structure representation human-readable. Binary and text files are equivalent when using only printable characters.
- To avoid issues with portability due to existence of big- and little-endian systems, each of which views bytes of an integer in different order, represent integers using byte or reinterpret code (see the "Compression" chapter). When designing a file format, specify each byte logically. The algorithms in this chapter ignore this for simplicity.

E.g., might have `/HEADER/LENGTH/{JSON_VAR: VAL1, JSON_ARR: [VAL2, VAL3]}/`.

13.4 Working with CSV Files

CSV files are easy to create from a string matrix:

```
void createCSV(Vector<Vector<string> > const& matrix, const char* filename)
{
    ofstream file(filename);
    assert(file);
    for(int i = 0; i < matrix.getSize(); ++i)
    {
        for(int j = 0; j < matrix[i].getSize(); ++j)
        {
            if(j > 0) file << ",";
            file << matrix[i][j];
        }
        file << endl;
    }
}
```

They are useful in particular for recording and analyzing any experimental measurements. A very typical use case is when run several algorithms against several problems, and collect comparable metrics such as solution quality score, runtime, etc. For reading and writing such files, it's useful to be able to convert a floating-point number to a string with predictable formatting (`to_string` doesn't allow to specify precision):

```
string toStringDouble(double x)
{
    stringstream s;
```

```

s << setprecision(17) << x;
string result;
s >> result;
return result;
}

```

Want to be able to split the combined in-memory data matrix into one per metric for easy follow-up analysis. For this assume the layout of the matrix to be:

Problem 1	Algorithm 1	Metric value	1	Metric value	2	Algorithm 2	Metric value	1	Metric value	2
Problem 2	Algorithm 1	Metric value	1	Metric value	2	Algorithm 2	Metric value	1	Metric value	2

This is convenient to write row-by-row when run all algorithms on one problem at a time and usually more efficient than running one algorithm at a time on all problems due to repeated input loading. The caller needs to know the names of the metrics. An individual metric layout becomes:

	Algorithm 1	Algorithm 2
Problem 1	Metric value	Metric value
Problem 2	Metric value	Metric value

```

Vector<Vector<Vector<string> > > splitRegularMatrix(
    Vector<Vector<string> > const& matrix, int nMetrics)
{//must have proper number of columns in every row
    assert(nMetrics > 0); //calculate the number of rows, columns,
    //and metrics
    for(int i = 0; i < matrix.getSize(); ++i)
        assert((matrix[i].getSize() - 1) % (nMetrics + 1) == 0);
    int nNewRows = matrix.getSize() + 1,
        nNewColumns = 1 + (matrix[0].getSize() - 1)/(nMetrics + 1);
    //do the splitting
    Vector<Vector<Vector<string> > > result(nMetrics,
        Vector<Vector<string> >(nNewRows, Vector<string>(nNewColumns)));
    for(int i = 0; i < nMetrics; ++i)
        //copy over algorithms names from first row
        for(int c = 1; c < nNewColumns; ++c) result[i][0][c] =
            matrix[1][1 + (c - 1) * (nMetrics + 1)];
        for(int r = 1; r < nNewRows; ++r)
            //copy over problem metricNames
            result[i][r][0] = matrix[r - 1][0];
            //copy over all relevant columns
            for(int c = 1; c < nNewColumns; ++c) result[i][r][c] =
                matrix[r - 1][2 + i + (c - 1) * (nMetrics + 1)];
    }
}
return result;
}

```

For analysis, to a single-metric file add (OpenOffice Calc) spreadsheet formulas that compare by ranks (see the "Computational Statistics" chapter):

1. Convert the scores into ranks (assuming smaller is better)
2. Average the ranks
3. Rank algorithms by the average ranks

Need several helper functions, in particular to convert an index column number into a cell name. For understanding these see the Calc documentation. Beware that with ties spreadsheet formulas give the lower and not the average rank, but this makes little difference to statistical analysis.

```

string cellValue(int r, int c)
{//convert cell and reference
    return "INDIRECT(ADDRESS(" + to_string(r + 1) + ";" +
        to_string(c + 1) + ";4))";
}
string fixNumber(int r, int c)
{

```

```

//convert cell, reference, and convert to number if scientific notation
    return "VALUE(TRIM(" + cellValue(r, c) + "));";
}
string rankFormula(int r, int c, int c0, int cLast)
{//rank column c in range [c0, cLast] in given row
    return "RANK(" + cellValue(r, c) + ";" + cellValue(r, c0) +
           ":" + cellValue(r, cLast) + ";1)";
}
string minFormula(int r, int c0, int cLast)
{
    return "MIN(" + cellValue(r, c0) + ":" + cellValue(r, cLast) + ")";
}
string aveFormula(int r0, int c0, int rLast, int cLast)
{//average over a row to a column
    assert(r0 == rLast || c0 == cLast);
    return "AVERAGE(" + cellValue(r0, c0) + ":" +
           cellValue(rLast, cLast) + ")";
}
}

```

Allow several runs of an algorithm by adjusting the sample size so that this won't produce extra statistical confidence.

```

void augmentComparableMatrix(Vector<Vector<string>>& matrix, int nRepeats = 1)
{//assume first row is algorithm names + first column problem names
    int nDataRows = matrix.getSize() - 1, nColumns = matrix[0].getSize();
    //append empty row as separator
    matrix.append(Vector<string>(nColumns, ""));
    //extract numerical values in all columns
    int fixedStart = matrix.getSize();
    for(int r = 0; r < nDataRows; ++r)
    {
        Vector<string> newRow(nColumns, "");
        for(int c = 1; c < nColumns; c++)
            newRow[c] = string("=") + fixNumber(1 + r, c);
        matrix.append(newRow);
    }
    //append empty row as separator
    matrix.append(Vector<string>(nColumns, ""));
    //make rank formulas for all data points
    for(int r = 0; r < nDataRows; ++r)
    {
        Vector<string> newRow(nColumns, "");
        for(int c = 1; c < nColumns; c++) newRow[c] = string("=") +
            rankFormula(fixedStart + r, c, 1, nColumns - 1);
        matrix.append(newRow);
    }
    //average the ranks in each column
    Vector<string> newRow2(nColumns, "Ave Ranks");
    for(int c = 1; c < nColumns; c++) newRow2[c] = string("=") + aveFormula(
        matrix.getSize() - nDataRows, c, matrix.getSize() - 1, c);
    matrix.append(newRow2);
    //rank the averages
    Vector<string> newRow(nColumns, "Total Rank");
    for(int c = 1; c < nColumns; c++) newRow[c] = string("=") +
        rankFormula(matrix.getSize() - 1, c, 1, nColumns - 1);
    matrix.append(newRow);
    //find significant rankDifference
    double maxDiff = findNemenyiSignificantAveRankDiff(nColumns - 1,
        nDataRows/nRepeats);
    Vector<string> newRow3(nColumns, toStringDouble(maxDiff));
    newRow3[0] = "Significant Diff";
    matrix.append(newRow3);
    Vector<string> newRow4(nColumns, "Cutoff Rank");
    for(int c = 1; c < nColumns; c++) newRow4[c] = string("=") +
        minFormula(matrix.getSize() - 3, 1, nColumns - 1) + "+" +
        cellValue(matrix.getSize() - 1, c);
}

```

```

matrix.append(newRow4);
Vector<string> newRow5(nColumns, "Same as Best");
for(int c = 1; c < nColumns; c++) newRow5[c] = string("=") + "IF(" +
    cellValue(matrix.getSize() - 1, c) + ">" +
    cellValue(matrix.getSize() - 4, c) + ";1;0)";
matrix.append(newRow5);
}

```

	Child	Computer
Animal Recognition	1	5
Simple Multiplication	2	1
	1	5
	2	1
	1	2
	2	1
Ave Ranks	1.5	1.5
Total Rank	1	1
Significant Diff	1.6928008593	1.6928008593
Cutoff Rank	3.1928008593	3.1928008593
Same as Best	1	1

Figure 13.2: A sample output file; note the number formatting

Put this together into a single function. In the resulting files the metric names are the prefixes and the file name the suffix.

```

void createAugmentedCSVFiles(Vector<Vector<string>> const& matrix,
    Vector<string> const& metricNames, string filename, int nRepeats = 1)
{
    Vector<Vector<string>> pieces(
        splitRegularMatrix(matrix, metricNames.getSize()));
    for(int i = 0; i < pieces.getSize(); ++i)
    {
        augmentComparableMatrix(pieces[i], nRepeats);
        createCSV(pieces[i], (metricNames[i] + "_" + filename).c_str());
    }
}

```

13.5 The I/O Model

It's fastest to access contiguous data. This is similar to transporting a shopping cart of items from a supermarket to your fridge instead of one item at a time. Details:

- Seek times cost 0 because they are done only before the slower data transfer.
- The read cost = the write cost. This is true for older magnetic disks but not for newer flash or SSD disks.
- Data is transferred in blocks of size $\leq B$, with B picked so that increasing it gives no further speedup. Each transfer is an I/O.
- The disk size is unlimited, and the memory size is M , which is much larger than B .
- Internal memory operations cost 0 because I/Os are orders-of-magnitude slower.

The model also holds when disk is memory and memory cache, except for the constant factors. The assumptions lead to lower bounds on the number of I/Os. In particular for (Dementiev 2008):

- Scanning— $O\left(\frac{n}{B}\right)$
- Map operations— $O(\log_B(n))$
- Sorting— $O\left(\frac{n}{B} \log_{M/B}\left(\frac{n}{B}\right)\right)$
- Priority queue—amortized $O\left(\frac{1}{B} \log_{M/B}\left(\frac{n}{B}\right)\right)$ per insert/delete

Though generally accurate, the I/O model has some flaws:

- Disks have large fast caches.
- Seek can dominate the runtime if doing many more seeks than I/Os.
- The OS can allocate large files in blocks of size $< B$. The size can be as small as 512 bytes, and adjacent blocks needn't be contiguous, which effectively reduces B to the file system block size. The latter is also the size of a single-character text file. Increasing it speeds up access but wastes space because most files are tiny.
- Internal memory computation can be the bottleneck.

Many algorithms and data structures such as a B-tree (discussed later in this chapter) need $O(B)$ internal memory work, so B can't be too large. A file layout that supports the I/O model is a variable-length array of blocks of a large fixed size. Data structures using it can encode any configuration data into a fixed-size header or use a separate file. So the ability to use a header is provided by the implementation here. To avoid misaligning the blocks, it's encoded into the first few, with padding if needed.

A general requirement is that want everything to be stored that is needed to recreate the observable state of the data structure, including any parameters. The block file itself encodes the block size in case a file is read by a data structure that want to use another size. The assumption is that both the wanted size and the current one will do—e.g., for a vector both should be multiples of the item's size. This gives file portability for when, e.g., the file system block size changes.

Any operation that must read data makes ≥ 1 I/O, so it's important to use internal memory effectively without using too much of it. So use a cache of frequently used blocks to reduce disk access. An asymptotically optimal strategy keeps a buffer of several least recently used blocks using a LRU cache (see the "Miscellaneous Algorithms" chapter).

```

class BlockFile
{
    File f;
    long long size; //the number of blocks, excluding header ones
    enum{SELF_HEADER_SIZE = 4};
    int headerSize, blockSize;
    int getNHeaderBlocks() const
    {
        return ceiling(SELF_HEADER_SIZE + headerSize, blockSize);
    }
    void setBlock(long long blockId) {f.setPosition(blockId * blockSize);}
    void write(long long blockId, Vector<unsigned char> const& block)
    {
        assert(block.getSize() == blockSize);
        setBlock(blockId);
        f.write(block.getArray(), blockSize);
    }
    Vector<unsigned char> read(long long blockId)
    {
        Vector<unsigned char> block(blockSize);
        setBlock(blockId);
        f.read(block.getArray(), blockSize);
        return block;
    }
    typedef DelayedCommitLRUCache<long long, Vector<unsigned char>, BlockFile>
        CACHE;
    friend CACHE; //to allow access to read and write
    CACHE cache; //declared last to be destructed first
    Vector<unsigned char> getHelper(long long blockId, int start, int n)
    {
        Vector<unsigned char> data(n);
        Vector<unsigned char> const& block = cache.read(blockId);
        for(int i = 0; i < n; ++i) data[i] = block[start + i];
        return data;
    }
    void setHelper(Vector<unsigned char> const& data, long long blockId,
        int start)
    {
        Vector<unsigned char> block = cache.read(blockId);
        for(int i = 0; i < data.getSize(); ++i) block[start + i] = data[i];
        cache.write(blockId, block);
    }
}

```

```

public:
    constexpr static int targetBlockSize() { return max<int>(BUFSIZ, 4096); }
    int getBlockSize() const { return blockSize; }
    //header blocks not included in size
    long long getSize() const { return size - getNHeaderBlocks(); }
    BlockFile(string const& filename, int theBlockSize, int cacheSize,
              int theHeaderSize = 0): f(filename.c_str(), false), size(0),
              headerSize(theHeaderSize), blockSize(theBlockSize),
              cache(*this, cacheSize)
    {
        assert(blockSize > 0);
        long long fileSize = f.getSize();
        if(fileSize > 0)
            //already exists, importing settings, can't use getHelper because
            //don't know blockSize yet
            Vector<unsigned char> selfHeader(SELF_HEADER_SIZE);
            f.setPosition(0);
            f.read(selfHeader.getArray(), SELF_HEADER_SIZE);
            blockSize = ReinterpretDecode(selfHeader);
            assert(blockSize > 0 && fileSize % blockSize == 0); //basic check
            size = fileSize/blockSize;
    }
    else
        //append header blocks and write blockSize to own header
        for(int i = 0; i < getNHeaderBlocks(); ++i) appendEmptyBlock();
        setHelper(ReinterpretEncode(blockSize, SELF_HEADER_SIZE), 0, 0);
    }
}
void appendEmptyBlock()
{
    ++size;
    Vector<unsigned char> block(blockSize, 0);
    f.append(block.getArray(), blockSize);
}
Vector<unsigned char> get(long long blockId, int start, int n)
{//for non-header blocks
    assert(0 <= blockId && blockId < getSize() && n > 0 && start >= 0 &&
           start + n <= blockSize);
    return getHelper(blockId + getNHeaderBlocks(), start, n);
}
void set(Vector<unsigned char> const& data, long long blockId, int start)
{//for non-header blocks
    assert(0 <= blockId && blockId < getSize() && start >= 0 &&
           start + data.getSize() <= blockSize);
    setHelper(data, blockId + getNHeaderBlocks(), start);
}
void writeHeader(Vector<unsigned char> const& header)
{//caller header may span several blocks
    assert(header.getSize() == headerSize);
    for(int i = 0, toWrite = headerSize; i < getNHeaderBlocks(); ++i)
    {
        int start = i == 0 ? SELF_HEADER_SIZE : 0,
            n = min(toWrite, blockSize - start);
        Vector<unsigned char> headerBlockData(n);
        for(int j = 0; j < n; ++j) headerBlockData[j] =
            header[(headerSize - toWrite) + j];
        setHelper(headerBlockData, i, start);
        toWrite -= n;
    }
}
Vector<unsigned char> readHeader()
{//caller header may span several blocks
    assert(headerSize > 0);
}

```

```

Vector<unsigned char> header;
for(int i = 0, toRead = headerSize; i < getNHeaderBlocks(); ++i)
{
    int start = i == 0 ? SELF_HEADER_SIZE : 0,
        n = min(toRead, blockSize - start);
    header.appendVector(getHelper(i, start, n));
    toRead -= n;
}
return header;
};

```

Read and write use ≤ 1 I/O per operation for a block (x the number of used blocks for the header). Beware:

- External memory data structures can contain only POD items, or more generally items for which the user provides a serializer that converts to a known-fixed-size byte sequence.
- Cache item references invalidate when the corresponding blocks are replaced, which can happen in the same line of code. E.g., for an external memory vector v with a cache size 1, $v[i] = v[j]$ assigns to a dead reference if $v[i]$ gets the current block reference, and $v[j]$ causes block replacement. So accessors can't safely return items by reference. This can also happen in recursion, with child calls invalidating references of the parent calls.

To use D concurrently accessible disks, make a superblock of size DB , and work with D subblocks in parallel (not implemented here). Usually some controller or indirection layer handles this **RAID0** configuration and appears as a single disk to the application. Particular algorithms may be more efficient accessing all disks directly, but the speedup is usually minor and not worth the complexity increase.

When B is unknown, the I/O model is the **cache-oblivious model**. An algorithm is cache-oblivious if it has an $O(1)$ factor slowdown relative to an optimal algorithm that knows B . For this to work, the model assumes that the OS transfers data in optimal-size blocks. Major building blocks:

- Scanning an array sequentially takes the optimal $\frac{n}{B}$ I/Os.
- Divide and conquer splitting eventually gets down to a block size $< B$. E.g., mergesort and quicksort are cache-oblivious, taking $O\left(\frac{n}{B} \lg\left(\frac{n}{B}\right)\right)$ I/Os.

Though cache-friendliness is important, complicated cache-oblivious algorithms aren't worth implementing:

- Constant factors lost due to not knowing B are large
- When disk is memory, the gains are minor because caches are large, constant factor differences small, and the OS task scheduler evicts cached data

13.6 External Memory Vector

Want to implement vector operations, but with the items stored on disk. For efficiency partition items in blocks, and load an entire block to memory when needed, using a user-specified buffer strategy. To handle blocks at the end of the file that aren't full, store in the header the number of extra items that can fit without increasing the file length.

Some differences from an in-memory vector:

- Append doesn't double but allocates another block when needed.
- Have a set method because can't return by reference.
- Remove only decreases internal size without reducing memory use due to lack of standard C++ API support for that. Users can do so on their own by rebuilding.
- Need a file name to work with; this allows persistence but disallows copy construction.

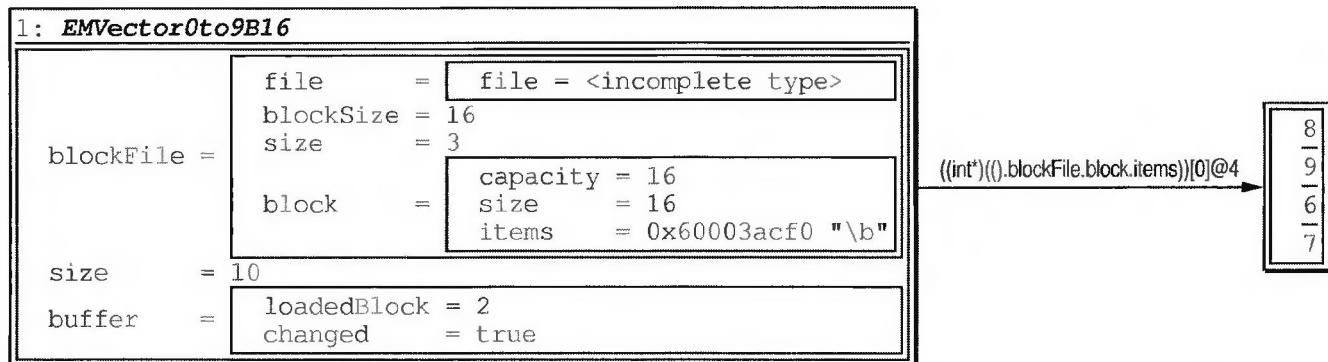


Figure 13.3: Memory layout of EMVector with $B = 16$ bytes (4 ints), after inserting integers 0 to 9. The last two (8 and 9) are part of the last block, and its other items are garbage.

The items are converted to bytes using a user-provided serializer. The default convenience serializer is a cast that requires a POD item type. On a single architecture it works fine, but isn't portable with change of endianness. For portability allow user-provided serializers, such as the one explicitly converting an item into a sequence of bytes using the reinterpret code (see the "Compression" chapter).

```

template<typename POD> struct CastSerializer//unportable - only for convenience
{//uncomment when this is supported in a few years
    //CastSerializer(){assert(is_trivially_copyable<POD>::value);}
    constexpr static int byteSize(){return sizeof(POD);}
    POD operator()(Vector<unsigned char> const& bytes)
    {
        assert(bytes.getSize() == byteSize());
        POD item;
        for(int i = 0; i < byteSize(); ++i)
            ((unsigned char*)&item)[i] = bytes[i];
        return item;
    }
    Vector<unsigned char> operator()(POD const& item)
    {
        Vector<unsigned char> bytes(byteSize());
        for(int i = 0; i < byteSize(); ++i)
            bytes[i] = ((unsigned char*)&item)[i];
        return bytes;
    }
};

template<typename POD> struct IntegralSerializer
{//common use case
    IntegralSerializer(){assert(is_integral<POD>::value);}
    constexpr static int byteSize(){return sizeof(POD);}
    POD operator()(Vector<unsigned char> const& bytes)
    {
        assert(bytes.getSize() == byteSize());
        return ReinterpretDecode(bytes);
    }
    Vector<unsigned char> operator()(POD const& item)
        {return ReinterpretEncode(item, byteSize());}
};

template<typename POD, typename SERIALIZER = CastSerializer<POD>>
class EMVector
{
    BlockFile blockFile;//must be first
    long long size;
    int itemsPerBlock() const{return blockFile.getBlockSize()/sizeof(POD);}
    long long block(long long i){return i/itemsPerBlock();}
    long long index(long long i){return i % itemsPerBlock();}
    SERIALIZER s;
    enum{HEADER_SIZE = 4};
    int extraItems() const{return blockFile.getSize() * itemsPerBlock() - size;}
    static int calculateBlockSize()
};

```

```

//if not exact try to go under, if not go over
    int result = BlockFile::targetBlockSize() / SERIALIZER::byteSize();
    if(result == 0) ++result; //can't go under, must go over
    return result * SERIALIZER::byteSize();
}
EMVector(EMVector const&); //no copying allowed
EMVector& operator=(EMVector const&);

public:
    long long getSize() {return size;}
    EMVector(string const& filename, int cacheSize = 2): size(0),
        blockFile(filename, calculateBlockSize(), cacheSize, HEADER_SIZE)
    {
        assert(blockFile.getBlockSize() % SERIALIZER::byteSize() == 0);
        //check if file already exists - header is number of extra items
        if(blockFile.getSize() > 0) size = blockFile.getSize() *
            itemsPerBlock() - ReinterpretDecode(blockFile.readHeader());
    }
    ~EMVector()
    { //write number of extra items to header
        Vector<unsigned char> header =
            ReinterpretEncode(extraItems(), HEADER_SIZE);
        blockFile.writeHeader(header);
    }
    void append(POD const& item)
    {
        ++size;
        if(extraItems() < 0) blockFile.appendEmptyBlock();
        set(item, size - 1);
    }
    void set(POD const& item, long long i)
    {
        assert(i >= 0 && i < size);
        blockFile.set(s(item), block(i), index(i) * SERIALIZER::byteSize());
    }
    POD operator[](long long i)
    {
        assert(i >= 0 && i < size);
        return s(blockFile.get(block(i), index(i) * SERIALIZER::byteSize(),
            SERIALIZER::byteSize()));
    }
    void removeLast()
    {
        assert(size > 0);
        --size;
    }
};

```

Due to caching random access takes $1/B$ and 1 I/Os for respectively contiguous and random locations.

13.7 Sorting

A simple way to sort efficiently:

1. Divide the vector into $Q + 1$ chunks of C items each (the last one gets the remainder)
2. Sort each in internal memory, and write the result to a temporary file
3. Create $Q + 1$ buffers of size B and a priority queue containing the first item from each chunk and the chunk's number
4. While the queue isn't empty
5. Write the dequeued item to the vector
6. Put the next item from the written item's buffer into the queue, refilling the buffer if it's empty, and the chunk has more items

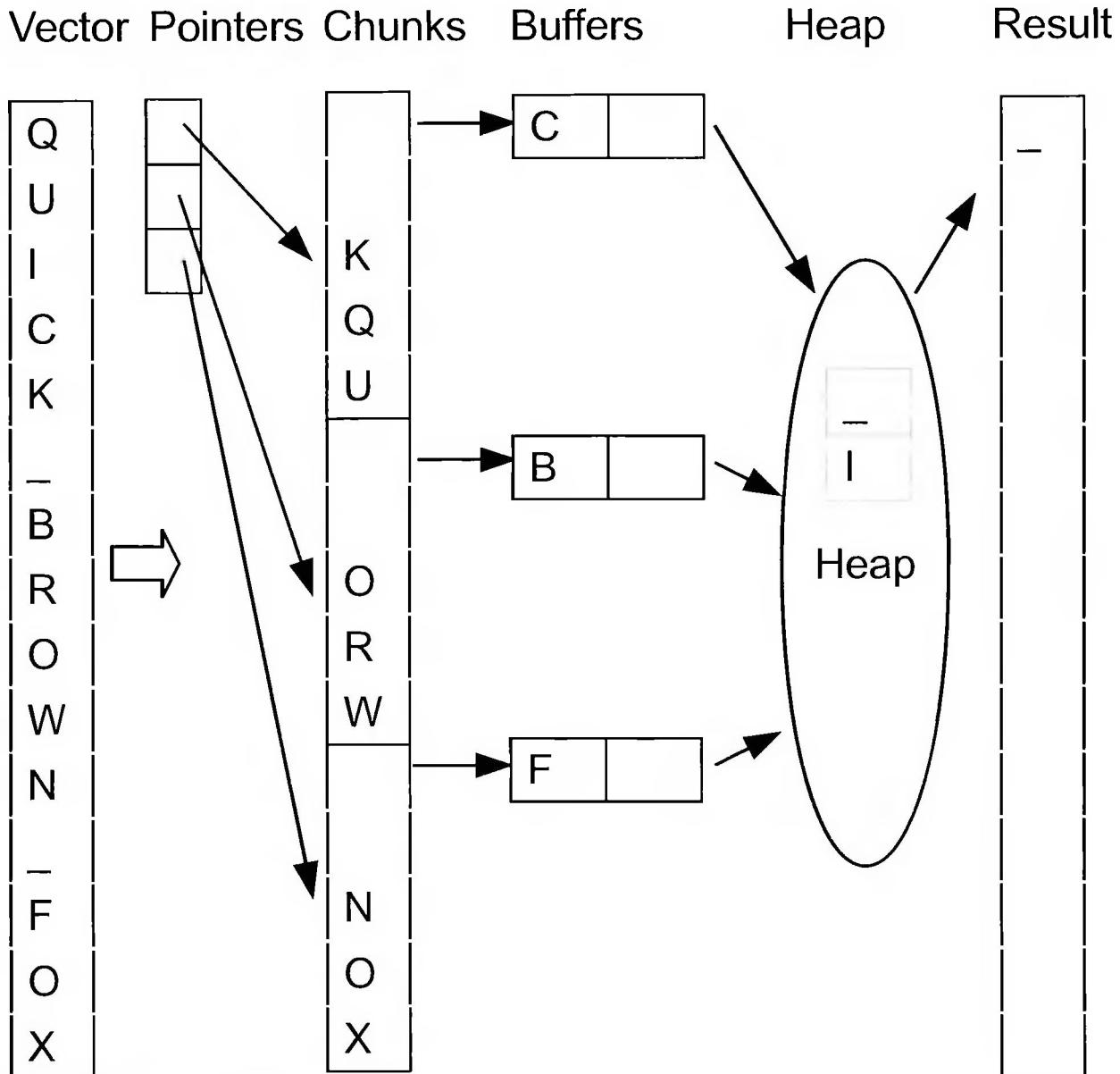


Figure 13.4: Logic of EM sorting

With m items per block must have enough internal memory for C items during sorting and mQ items during merging. So $C = \sqrt{nm}$, and $Q = n/C$ minimizes internal memory use.

```
friend void IOSort(EMVector& vector)
{
    //scope to remove temp vector before file deletion
    long long n = vector.getSize(), C = sqrt(n *
        vector.itemsPerBlock()), Q = n/C, lastQSize = n % C;
    File::remove("IOSortTempFile.igmdk"); //in case exists already
    EMVector temp("IOSortTempFile.igmdk"); //potentially different
    //block size if old file and BUFSIZ changed, but ok
    typedef pair<POD, long long> HeapItem;
    Heap<HeapItem, PairFirstComparator<POD, long long>> merger;
    for(long long q = 0, i = 0; q < Q + 1; ++q)
    {
        long long m = q == Q ? lastQSize : C;
        if(m > 0)
            //sort each block
            Vector<POD> buffer;
            for(long long j = 0; j < m; ++j) buffer.append(vector[i++]);
            quickSort(buffer.getArray(), buffer.getSize());
            //put smallest item of each block on the heap
            merger.insert(HeapItem(buffer[0], q));
            //and the rest to the temp vector
    }
}
```

```

        for(long long j = 1; j < m; ++j) temp.append(buffer[j]);
    }
}

Vector<Queue<POD>> buffers(Q + 1);
Vector<long long> pointers(Q + 1, 0);
for(long long i = 0; i < n; ++i)
//merge, remember that temp blocks are 1 less
    long long q = merger.getMin().second;
    vector.set(merger.deleteMin().first, i);
    if(buffers[q].isEmpty())//refill if needed
        while(pointers[q] < (q == Q ? lastQSize : C) - 1)
            buffers[q].push(temp[q * (C - 1) + pointers[q]++]);
    if(!buffers[q].isEmpty())//check if done with block
        merger.insert(HeapItem(buffers[q].pop(), q));
}
}

File::remove("IOSortTempFile.igmdk");
}

```

The algorithm needs $O(n/B)$ I/Os if have enough internal memory. If don't, merge the sorted chunks with each other in a mergesort-like process until Q is small enough to begin the heap merge.

13.8 Vector-based Data Structures

Many data structures can be implemented I/O-optimally using vector:

- Stack—beware that with cache of size 1 a push/pop sequence on a block boundary causes 1 I/O per operation. Size 2 is enough for the optimal amortized $1/B$ I/Os per operation.
- Free list—one vector contains the nodes and another indices of returned nodes.
- Queue—use doubling on a circular queue, avoiding temporaries after doubling by copying the first half into the second.

A free list is a useful component for many other data structures and is implemented here:

```

template<typename POD, typename SERIALIZER = CastSerializer<POD>>
class EMFreelist
{
    EMVector<POD, SERIALIZER> nodes;
    EMVector<long long, IntegralSerializer<long long>> returned;
    //disallow copying
    EMFreelist(EMFreelist const&);
    EMFreelist& operator=(EMFreelist const&);

public:
    EMFreelist(string const& filenameSuffix, int cacheSize = 2):
        nodes("Nodes" + filenameSuffix, cacheSize),
        returned("Returned" + filenameSuffix){}
    long long allocate(POD const& item = POD())
    {
        if(returned.getSize() > 0)
        //reuse the last deallocated node
            long long result = returned[returned.getSize() - 1];
            returned.removeLast();
            nodes.set(item, result);
            return result;
    }
    else
    {
        nodes.append(item);
        return nodes.getSize() - 1;
    }
}

void deallocate(long long i)
//no efficient way to check if already deallocated
    assert(i >= 0 && i < nodes.getSize());
    returned.append(i);
}

```

```

POD operator[](long long i)
{ //no efficient way to check if already deallocated
    assert(i >= 0 && i < nodes.getsize());
    return nodes[i];
}
void set(POD const item, long long i)
{ //no efficient way to check if already deallocated
    assert(i >= 0 && i < nodes.getsize());
    nodes.set(item, i);
}
};

```

13.9 B+ Tree

A B+ tree is a dynamic sorted sequence where the leaves hold values and the internal nodes keys. For many database use cases keys are small, and values huge, so the separation improves efficiency and allows linking the leaves for iteration. Keys for some nodes appear in the tree several times, but in total the internal nodes have fewer keys than the leaves.

An internal node has a size and M (key, I/O pointer) pairs. M is even, ≥ 4 , and such that its byte size $\approx B$. A node of size k generalizes binary tree node and has k pointers and $k - 1$ keys, where pointer _{i} points to the node with keys $<$ key _{i} , and pointer _{$i + 1$} to the node with keys \geq key _{i} . The pair $k - 1$ only has a pointer. Search inside an internal node finds the pointer to the next node according to this definition.

A leaf has a sorted array of items of size L . L is such that the node's byte size $\approx B$, ≥ 2 , and even.

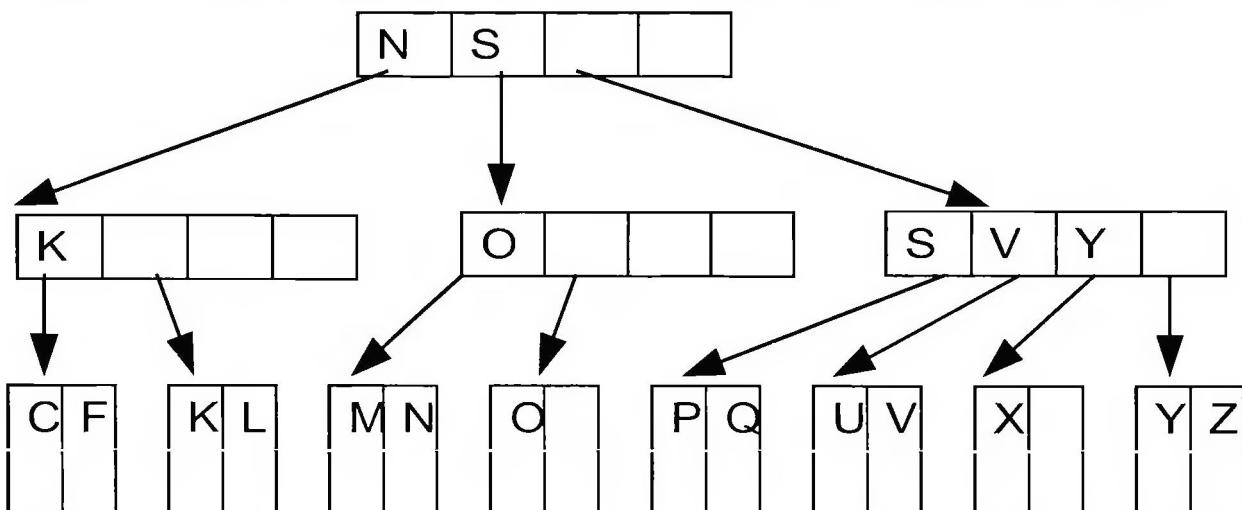


Figure 13.5: A possible B+ tree structure for letter keys and unspecified values

To save space, use a convention that -1 is a null pointer, numbers ≥ 0 are internal node pointers, and numbers < -1 leaf pointers. The internal nodes support finding the child corresponding to a key, and the leaves inclusive successor of a key. Use linear search for both. Leaves are allocated using a free list and internal nodes a vector because deletion doesn't touch the latter. Both use custom serializers based on the user-provided ones for keys and values.

```

template<typename KEY, typename VALUE, typename KEY_SERIALIZER =
CastSerializer<KEY>, typename VALUE_SERIALIZER = CastSerializer<VALUE> >
class EMBPlusTree
{
    typedef KVPair<KEY, long long> Key;
    typedef KVPair<KEY, VALUE> Record;
    //satisfy the constraints on M and L, but first from internal node make
    //space for size and from leaf also the next pointer
    enum{NULL_IO_POINTER = -1, NODE_SIZE_BYTES = 4, POINTER_SIZE = 8,
        KEY_SIZE = KEY_SERIALIZER::byteSize() + POINTER_SIZE, RECORD_SIZE =
        KEY_SERIALIZER::byteSize() + VALUE_SERIALIZER::byteSize(), B =
        BlockFile::targetBlockSize() - NODE_SIZE_BYTES, M = 2 * min<int>(2,
        B/KEY_SIZE), L = 2 * min<int>(1, (B - POINTER_SIZE)/2/RECORD_SIZE)
    };
    struct Node

```

```

    {
        int size;
        Key next[M];
        Node(): size(1) {next[0].value = NULL_IO_POINTER; }
        int findChild(KEY const& key)
        {//last child not larger than the key
            int i = 0;
            while(i < size - 1 && key >= next[i].key) ++i;
            return i;
        }
        struct Serializer
        {
            KEY_SERIALIZER ks;
            constexpr static int byteSize()
            {
                return NODE_SIZE_BYTES + int(M) * int(KEY_SIZE);
            }
            Node operator()(Vector<unsigned char> const& bytes)
            {
                assert(bytes.getSize() == byteSize());//basic file check
                Node node;
                BitStream bs(bytes);//first decode size
                node.size = ReinterpretDecode(bs.readBytes(NODE_SIZE_BYTES));
                for(int i = 0; i < M; ++i)//then the next pointers
                {
                    node.next[i].key =
                        ks(bs.readBytes(KEY_SERIALIZER::byteSize()));
                    node.next[i].value =
                        ReinterpretDecode(bs.readBytes(PTR_SIZE));
                }
                return node;
            }
            Vector<unsigned char> operator()(Node const& node)
            {
                BitStream bs;//first encode size
                bs.writeBytes(ReinterpretEncode(node.size, NODE_SIZE_BYTES));
                for(int i = 0; i < M; ++i)//then the next pointers
                {
                    bs.writeBytes(ks(node.next[i].key));
                    bs.writeBytes(ReinterpretEncode(node.next[i].value,
                        PTR_SIZE));
                }
                return bs.bitset.getStorage();
            }
        };
    };
    struct Leaf
    {
        int size;
        long long next;
        Record records[L];
        Leaf(): size(0), next(NULL_IO_POINTER) {}
        int inclusiveSuccessorRecord(KEY const& key)
        {//last record smaller than the key
            int i = 0;
            while(i < size && key > records[i].key) ++i;
            return i;
        }
        struct Serializer
        {
            KEY_SERIALIZER ks;
            VALUE_SERIALIZER vs;
            constexpr static int byteSize()
            {
                return int(NODE_SIZE_BYTES) + int(PTR_SIZE) +

```

```

        int(L) * int(RECORD_SIZE);
    }
Leaf operator()(Vector<unsigned char> const& bytes)
{
    assert(bytes.getSize() == byteSize()); //basic file check
    Leaf leaf;
    BitStream bs(bytes); //first decode size
    leaf.size = ReinterpretDecode(bs.readBytes(NODE_SIZE_BYTES));
    //then next pointer
    leaf.next = ReinterpretDecode(bs.readBytes(PTR_SIZE));
    for(int i = 0; i < L; ++i) //then the records
    {
        leaf.records[i].key =
            ks(bs.readBytes(KEY_SERIALIZER::byteSize()));
        leaf.records[i].value =
            vs(bs.readBytes(VALUE_SERIALIZER::byteSize()));
    }
    return leaf;
}
Vector<unsigned char> operator()(Leaf const& leaf)
{
    BitStream bs; //first encode size
    bs.writeBytes(ReinterpretEncode(leaf.size, NODE_SIZE_BYTES));
    //the next pointer
    bs.writeBytes(ReinterpretEncode(leaf.next, PTR_SIZE));
    for(int i = 0; i < L; ++i) //then the records
    {
        bs.writeBytes(ks(leaf.records[i].key));
        bs.writeBytes(vs(leaf.records[i].value));
    }
    return bs.bitset.getStorage();
}
};

//leaf indices start at -2
long long leafIndex(long long index){return -(index + 2);}
long long inverseLeafIndex(long long lIndex){return -lIndex - 2;}
long long root;
File header; //for the root pointer, uncached access
EMVector<Node, typename Node::Serializer> nodes;
EMFreelist<Leaf, typename Leaf::Serializer> leaves;
EMBPlusTree(EMBPlusTree const&); //no copying allowed
EMBPlusTree& operator=(EMBPlusTree const&);

public:
EMBPlusTree(string const& filenameSuffix): root(NULL_PTR),
    header("Header" + filenameSuffix).c_str(), false),
    nodes("Keys" + filenameSuffix, 8), leaves("Records" + filenameSuffix)
{
    Vector<unsigned char> temp(PTR_SIZE);
    if(header.getSize() > 0)
    {
        header.read(temp.getArray(), PTR_SIZE);
        root = ReinterpretDecode(temp);
    }
    else header.append(temp.getArray(), PTR_SIZE); //make space for root
}
~EMBPlusTree()
{//write root to header
    header.setPosition(0);
    header.write(ReinterpretEncode(root, PTR_SIZE).getArray(),
        PTR_SIZE);
}
};

```

Find uses the internal node invariant to find the leaf containing the value and searches it:

```

pair<long long, long long> findLeaf(KEY const& key)
{
    long long current = root, parent = NULL_IO_POINTER;
    while(current >= 0) //stop at leaf or null
    {
        parent = current;
        Node node = nodes[current];
        current = node.next[node.findChild(key)].value;
    }
    return make_pair(current, parent);
}
ITEM find(KEY const& key, bool& status)
{
    status = true;
    long long current = findLeaf(key).first;
    if(current != NULL_IO_POINTER)
    {
        Leaf leaf = leaves[leafIndex(current)];
        int i = leaf.inclusiveSuccessorRecord(key);
        if(i < leaf.size && key == leaf.records[i].key)
            return leaf.records[i].value;
    }
    status = false;
    return VALUE();
}

```

Insert balances using splits. When a node is full, and its parent isn't:

1. Put the right half of the keys into a new node
2. Insert a pointer to the new node into the parent after the pointer to the node
3. Copy the middle key to the parent, and move it to the new node

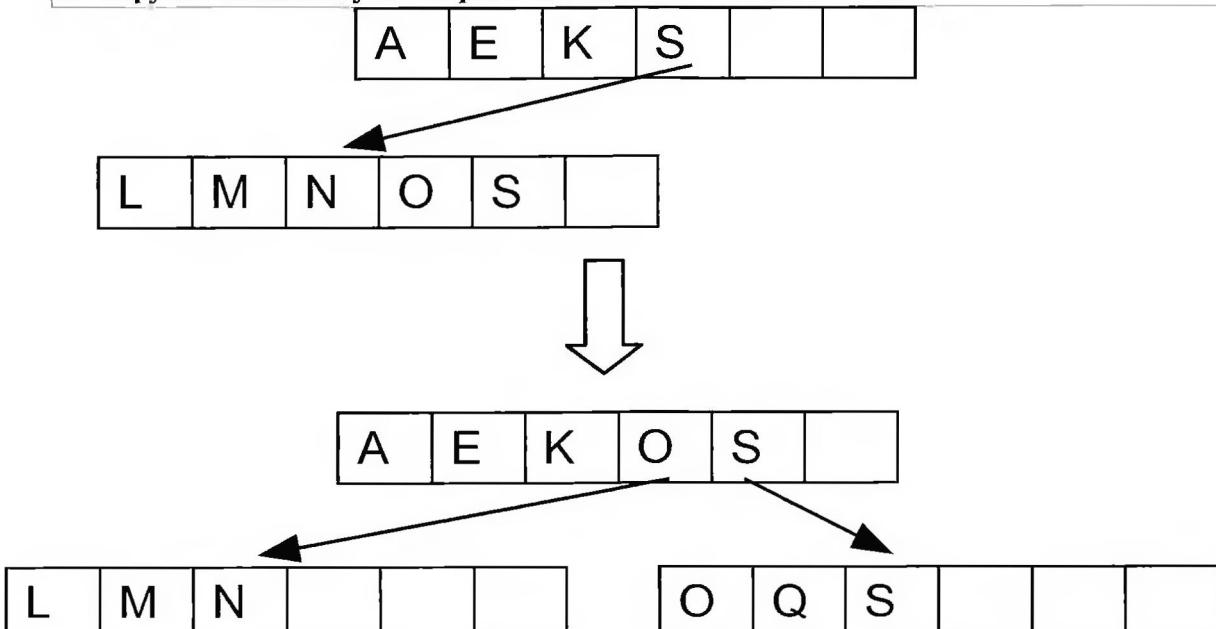


Figure 13.6: B+ tree node splitting (recall that the last key cell is unused)

```

void splitInternal(long long index, int child)
{
    Node parent = nodes[index];
    long long childIndex = parent.next[child].value;
    Node left = nodes[childIndex], right;
    //copy middle item key into the parent, shifting the latter's other
    //keys
    for(int i = parent.size++; i > child; --i)
        parent.next[i] = parent.next[i - 1];
    parent.next[child].key = left.next[M/2 - 1].key;
    parent.next[child + 1].value = nodes.getSize();
    //move items starting from middle into right
}

```

```

right.size = M/2 + 1;
for(int i = 0; i < right.size; ++i)
    right.next[i] = left.next[i + M/2 - 1];
left.size = M/2;
nodes.append(right); //write the nodes
nodes.set(left, childIndex);
nodes.set(parent, index);
}

```

Splitting a leaf follows the same logic. But it's slightly different because a leaf contains no dummy keys, maintains a pointer to the next leaf, and the items are stored differently:

```

void splitLeaf(long long index, int child)
{
    Node parent = nodes[index];
    long long childIndex = parent.next[child].value,
           newChildIndex = inverseLeafIndex(leaves.allocate());
    Leaf left = leaves[leafIndex(childIndex)], right;
    //copy middle item key into the parent internal node, shifting the
    //latter's other keys
    for(int i = parent.size++; i > child; --i)
        parent.next[i] = parent.next[i - 1];
    parent.next[child].key = left.records[L/2].key;
    parent.next[child + 1].value = newChildIndex;
    //move items starting from middle into right
    left.size = right.size = L/2;
    for(int i = 0; i < right.size; ++i)
        right.records[i] = left.records[i + L/2];
    right.next = left.next;
    left.next = newChildIndex;
    leaves.set(right, leafIndex(newChildIndex)); //write the nodes
    leaves.set(left, leafIndex(childIndex));
    nodes.set(parent, index);
}

```

To insert:

1. Create the root if it's missing, and split if full
2. Find the wanted leaf in the same way as find, inserting the key into each node, and splitting full nodes on the way down
3. Insert the item into the appropriate leaf

```

bool shouldSplit(long long node)
{
    return node < NULL_IO_POINTER ?
           leaves[leafIndex(node)].size == L : nodes[node].size == M;
}

void insert(KEY const& key, ITEM const& value)
//the first node is root as leaf
{
    if(root == NULL_IO_POINTER) root = inverseLeafIndex(leaves.allocate());
    else if(shouldSplit(root))
        //split the root if needed
        Node newRoot;
        newRoot.next[0].value = root;
        bool wasLeaf = root < NULL_IO_POINTER;
        root = nodes.getSize();
        nodes.append(newRoot);
        wasLeaf ? splitLeaf(root, 0) : splitInternal(root, 0);
    }

    long long index = root;
    while(index > NULL_IO_POINTER) //internal node work
        //go down, inserting and splitting
        Node node = nodes[index];
        int childI = node.findChild(key), child = node.next[childI].value;
        if(shouldSplit(child))
            //split child if needed

```

```

        child < NULL_IO_POINTER ? splitLeaf(index, childI) :
            splitInternal(index, childI);
        if(key > nodes[index].next[childI].key) //go to next child
            child = nodes[index].next[childI + 1].value;
    }
    index = child;
}
//insert the item into the leaf
Leaf leaf = leaves[leafIndex(index)];
int i = leaf.inclusiveSuccessorRecord(key);
if(i < leaf.size && key == leaf.records[i].key)
    leaf.records[i].value = value;
else
{
    for(int j = leaf.size++; j > i; --j)
        leaf.records[j] = leaf.records[j - 1];
    leaf.records[i] = Record(key, value);
}
leaves.set(leaf, leafIndex(index));
}

```

A concurrent implementation would lock a node when an operation looks at it and unlock when done. The top-down nature of all operations makes this correct. Remove removes the node from the leaf without merging any nodes. Databases use this simple strategy because it's efficient for concurrency. Also, real-world databases are rebuilt occasionally, particularly when adding new data fields, which takes care of the merging. With substantial extra logic, can merge nodes $< \frac{1}{2}$ full in a process opposite of splitting. Empty leaves are also removed.

```

void remove(KEY const& key)
{
    pair<long long, long long> pointerAndParent = findLeaf(key);
    long long pointer = pointerAndParent.first;
    if(pointer != NULL_IO_POINTER)
    {
        Leaf leaf = leaves[leafIndex(pointer)];
        int i = leaf.inclusiveSuccessorRecord(key);
        if(i < leaf.size && key == leaf.records[i].key)
        {
            --leaf.size;
            for(int j = i; j < leaf.size; ++j)
                leaf.records[j] = leaf.records[j + 1];
        }
        if(leaf.size > 0) leaves.set(leaf, leafIndex(pointer));
        else
        {//remove leaf
            leaves.deallocate(leafIndex(pointer));
            Node parent = nodes[pointerAndParent.second];
            parent.next[parent.findChild(key)].value = NULL_IO_POINTER;
            nodes.set(parent, pointerAndParent.second);
        }
    }
}

```

B+ tree is I/O-optimal for dynamic sorted sequence operations. Without deletions, each node $\geq \frac{1}{2}$ full, so with n items the height is $O(\log_B(n))$, which < 4 in practice. An LRU cache will always contain the root. Retrieving a large record takes the optimal $O(\text{the height} + \text{reading the record})$ time.

For use cases such as adding a field to a database, need to rebuild the tree. To construct a B+ tree bottom-up from a sorted file:

1. Sort
2. Copy every B^{th} node to the next level
3. Recurse (2) on the B -separators until all fit in one block, which becomes the root

This costs $O(\text{sort}) < O(n \text{ insertions})$.

13.10 Comments

When `BUFSIZ < B`, it seems best to adjust the buffer size of `fstream`—can use `f.rdbuf() ->pubsetbuf(array, size)` before opening the file. But this isn't done here and won't be considered further because:

- The C++ standard doesn't guarantee that this will even take effect (unless both arguments are 0, which is useless).
- Based on my tests setting the buffer size to 4096 or even standard-regulated 0 doesn't change the block read times at all. For reading several small- B contiguous blocks, the times are close to best-buffer-size ones already at 128 and don't improve after 512.
- A too large buffer (≥ 8192 for me) slows down block reads—looks like the implementation tries to fill it up even for smaller B .

Can implement a linked list efficiently, though the idea is perhaps more useful than the implementation. If a IO-size block can fit k nodes, any two consecutive blocks must have $k/2$ elements. This slows down scanning only by a factor of 2. When inserting put the new item into its block. If there is no space, kick out one of the block's border items into the neighboring block. If it has no space, split the block in two before proceeding with insertion, which is the worst case for the capacity restriction. Deletions are simple unless capacity limit is reached, in which case the offending blocks are merged.

For B-tree, keys and items aren't separated, leading to worse runtime. For simplicity, it's common to call B+ tree "B-tree".

For map operations can use slightly faster hash tables such as linear probing (cache-oblivious) or **linear hashing** (see Folk et al. 1998 for details). But hash tables don't support ordered operations, can't be efficiently rebuilt from sorted data, and have problematic worst cases (Dementiev 2007).

Many algorithms are difficult or impossible to make I/O-efficient. E.g., for graph algorithms DFS's jumping causes 1 I/O per vertex.

13.11 Projects

- Implement swap-shrinking a vector given a temporary file name. Extend this to other data structures.
- Implement a stack and a queue.
- Extend vector sorting to handle the not-enough-memory case. Run a benchmark study for file. size of at least 16GB (or more if have disk space). Determine effect of B on the runtime. Analyze runtime assuming k merge stages are needed.
- Implement forward iteration for a B-tree.
- Implement the ability to create a B-tree efficiently from a vector of sorted records.
- Research and implement the node count augmentation for a B-tree.
- Implement persistence tests for all discussed data structures.

13.12 References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Folk, M. Zoellick, B. and Riccardi G. (1998). *File Structures: An Object-Oriented Approach with C++*. Addison-Wesley.
- Dementiev, R. (2007). *Algorithm Engineering for Large Data Sets*. Springer.
- Mehlhorn, K., & Sanders, P., Dietzfelbinger, M., & Dementiev, R. (2019). *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer.
- Meyer, U., Sanders, P., & Sibeyn, J. F. (2003). *Algorithms for Memory Hierarchies*. Springer.

14 String Algorithms

14.1 Introduction

If you ever wondered how your text editor looks for patterns in a text file, including using patterns made of various expressions, this is what string algorithms do. Many useful algorithms are discussed in detail, but this is a very wide field, so see the references for further study.

A **string** is a vector of characters from some alphabet. Character comparisons are assumed to take $O(1)$ time. An **alphabet** is:

- **Indexed** if every character corresponds to a number—e.g., ASCII and Unicode, but not UTF8
- **Bounded** if the number of possible characters is bounded—e.g., also ASCII and Unicode, but not UTF8
- **Ordered** if it supports meaningful comparisons—e.g., indexed alphabets are comparable but not meaningfully (except for subsets like ASCII letter or numbers)
- **General** if it supports equality comparison

Some important substrings (also called **factors**):

- **q -gram**—has size q
- **Prefix**—starts at the beginning
- **Suffix**—ends at the end

14.2 Single-pattern Search

Want to find all occurrences of a pattern string of length m in a larger text string of length n . Need at least a general alphabet. The slow brute-force algorithm checks for matches at every position. The runtime = $O(mn)$, and $E[\text{the runtime}] = O(n)$ (Crochemore et al. 2007). A useful part of this algorithm is checking occurrence at a given position of the text. It assumes that the text has enough characters and compares them one-by-one with the pattern ones:

```
template<typename VECTOR, typename VECTOR2> bool matchesAt(int position,
    VECTOR2 text, VECTOR pattern, int patternSize)
//allows different text and pattern types
int i = 0;
while(i < patternSize && pattern[i] == text[i + position]) ++i;
return i == patternSize;
```

Lower bounds for the number of character comparisons:

- Worst case $O(n)$
- Expected $O\left(\frac{n \log_a(m)}{m}\right)$ for random text and pattern over a bounded alphabet of size a

Horspool algorithm shifts the pattern from left to right. E.g., given pattern *apple* and text *there_is_a_particularly_healthy_fruit_called_apple*, the brute force algorithm compares *apple* with *there*, *here*, *ere_i*, etc. Instead of shifting by 1, Horspool preprocesses the pattern to compute \forall character except the last the max possible shift = $m - 1$ – the last position of the character, with positions of characters \notin pattern = -1 . E.g., for *apple* the precomputed shifts are:

a	p	l	e	Any other
4	2	1	5	5

Using this information, the last character of the current text factor determines the shift. E.g., after comparing *apple* with *there*, shift by 5 to align with *_is_a* because *apple* has no other *e*'s. Then by 4 to align with *a_par* because *apple* starts with *a*. Then by 5 until aligned with *_appl*, and by 1 to find a match. For random text and pattern, $E[\text{the runtime}] = O\left(\frac{n}{\min(m, a)} + n_{\text{matched}}\right)$ (Navarro & Raffinot 2002).

Algorithm **HashQ** has the optimal expected runtime (Lecroq 2007), is practically faster for small alphabets or very long patterns, and works for general alphabets (with a proper user-supplied hash function). Both are worst-case $O(mn)$. HashQ calculates the shifts using q -grams for $q \geq 1$, which requires $m \geq q$; $q = 2 \log_a(m)$ is optimal (Baeza-Yates & Ribeiro-Neto 2011). It uses a hash table of size m/q and resolves collisions using linear probing.

sions by giving preference to smaller shifts. To find the shift amount during matching, lookup the last q -gram of the current text factor. The algorithm assumes that h comes with a builder.

For small alphabets, specialized h are more efficient. E.g., for DNA use $q = 4$ with $h = \text{concatenation of all bits}$. For $q = 1$, identity h , and table size a , the algorithm is equivalent to Horspool. For $q = 3$ Lecroq (2007) proposed a simple h based on adding and shifting—this is implemented here without loop unrolling for general q .

```

struct LecroqHash//identity for q = 1
//ignore the size parameter - the matcher will have enough table size
LecroqHash(int dummy) {}
struct Builder
{
    unsigned char result;
    Builder(): result(0){}
    void add(unsigned char c){result = result << 1 + c;}
};
Builder makeBuilder(){return Builder();}
unsigned char operator()(Builder b){return b.result;}
};

template<typename VECTOR, typename HASHER = LecroqHash>
class HashQ
{
    enum{CHAR_ALPHABET_SIZE = 1 << numeric_limits<unsigned char>::digits};
    int patternSize, q;
    Vector<int> shifts;/size is a power of 2 for fast hashing
    VECTOR const &pattern;
    HASHER h;
    typedef typename HASHER::Builder B;
public:
    HashQ(VECTOR const &thePattern, int thePatternSize, int theQ = 1): q(theQ),
        pattern(thePattern), patternSize(thePatternSize), shifts(max<int>(
            CHAR_ALPHABET_SIZE, nextPowerOfTwo(ceiling(patternSize, q)))),
        h(shifts.getSize())
    {//precompute shifts
        assert(patternSize >= q);
        int temp = patternSize - q;
        for(int i = 0; i < shifts.getSize(); ++i) shifts[i] = temp + 1;
        for(int i = 0; i < temp; ++i)
        {
            B b(h.makeBuilder());
            for(int j = 0; j < q; ++j) b.add(pattern[i + j]);
            shifts[h(b)] = temp - i;
        }
    }//return match position and next start position of (-1, -1)
    template<typename VECTOR2> pair<int, int> findNext(VECTOR2 const &text,
        int textSize, int start = 0)//allow different text and pattern types
    {
        while(start + patternSize <= textSize)
        {
            int result = start, hStart = start + patternSize - q;
            B b(h.makeBuilder());
            for(int j = 0; j < q; ++j) b.add(text[hStart + j]);
            start += shifts[h(b)];
            if(matchesAt(result, text, pattern, patternSize))
                return make_pair(result, start);
        }
        return make_pair(-1, -1);
    }
};

```

14.3 Multiple Patterns

Can match k patterns of different lengths one-by-one, but **Wu-Manber algorithm** is more efficient. It's a

generalization of HashQ with $E[\text{the runtime}] = O\left(\frac{n \log_a(mk)}{m}\right)$, $q = \log_a(mk)$, and table size $\frac{mk}{q}$, with m

the average length (Navarro & Raffinot 2002; historically Wu-Manber was discovered first). Match any patterns of length $< q$ separately, by Wu-Manber with $q = 1$ or one-by-one. The algorithm hashes each pattern's suffix q -gram and \forall hash value creates a list of patterns that hash to it. The hash of the last q -gram of the current text factor determines the shift = the min possible shift \forall pattern and possible match, which are checked by brute force.

```
template<typename VECTOR, typename HASHER = LecroqHash> class WuManber
{
    enum{CHAR_ALPHABET_SIZE = 1 << numeric_limits<unsigned char>::digits};
    int q, minPatternSize;
    Vector<pair<VECTOR, int>> const& patterns;
    Vector<int> shifts;//size is a power of 2 for fast hashing
    Vector<Vector<int>> candidates;
    HASHER h;
    typedef typename HASHER::Builder B;
public:
    WuManber(Vector<pair<VECTOR, int>> const& thePatterns, int theQ = 1,
              double avePatternSize = 1): q(theQ), patterns(thePatterns), shifts(max<int>(CHAR_ALPHABET_SIZE, nextPowerOfTwo(avePatternSize *
                patterns.getSize()/q))), candidates(shifts.getSize()), h(shifts.getSize()), minPatternSize(numeric_limits<int>::max())
    { //precompute shifts
        for(int i = 0; i < patterns.getSize(); ++i)
            minPatternSize = min(patterns[i].second, minPatternSize);
        assert(patterns.getSize() > 0 && minPatternSize >= q);
        int temp = minPatternSize - q;
        for(int i = 0; i < shifts.getSize(); ++i) shifts[i] = temp + 1;
        for(int j = 0; j < patterns.getSize(); ++j)
            for(int i = 0; i < temp + 1; ++i)
            {
                B b(h.makeBuilder());
                for(int k = 0; k < q; ++k) b.add(patterns[j].first[i + k]);
                int hValue = h(b);
                if(i == temp) candidates[hValue].append(j);
                else shifts[hValue] = min(temp - i, shifts[hValue]);
            }
        } //return match position and next start position of (-1, -1) and indices
        //of patterns that match; allow different text and pattern types
    template<typename VECTOR2> pair<Vector<int>, int> findNext(
        VECTOR2 const& text, int textSize, int start = 0)
    {
        Vector<int> matches(patterns.getSize(), -1);
        while(start + minPatternSize <= textSize)
        {
            B b(h.makeBuilder());
            for(int j = 0; j < q; ++j)
                b.add(text[start + minPatternSize - q + j]);
            int hValue = h(b);
            bool foundAMatch = false;
            for(int i = 0; i < candidates[hValue].getSize(); ++i)
            {
                int j = candidates[hValue][i];
                if(start + patterns[j].second <= textSize && matchesAt(
                    start, text, patterns[j].first, patterns[j].second))
                {
                    foundAMatch = true;
                    matches[j] = start;
                }
            }
            start += shifts[hValue];
            if(foundAMatch) return make_pair(matches, start);
        }
    }
};
```

```

    }
    return make_pair(matches, -1);
}
};

```

14.4 Regular Expressions

A **regular expression** matches text specified by rules. It's formed by applying operators "*" (the left operand can occur zero or more times), "|" (left or right), and "&" (left concatenated with right) to clauses, in this precedence order; "&" isn't explicitly written. A **clause** is any parenthesized subexpression, including a single character with implicit parenthesis. Use escape character "/" if operators occur as characters. E.g., / (xxx/)xxxx-xxx for $0 \leq x \leq 9$ matches phone numbers. Can represent other operators such as jokers and character classes in terms of these. To find if the text contains a pattern matched by *expression*, use **expression**.

Glushkov algorithm (Sedgewick & Wayne 2011; Navarro & Raffinot 2002) converts the expression to a nondeterministic finite automaton, where **epsilon transitions** don't consume characters. Parsing an expression of m characters creates a graph with a vertex for the final state, a vertex \forall state before reading a character, and an edge \forall epsilon transition, which is added:

- To the next state for "*", "(", and ")"
- Between a clause start and the next state if it's "*"
- For "|" from before the left start to the right start and from before-the-right-start to the clause end

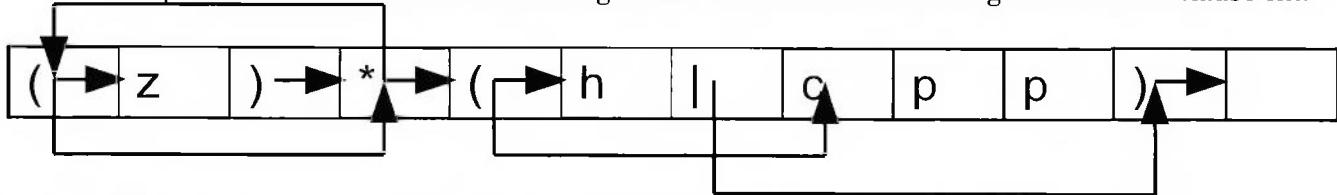


Figure 14.1: Regular expression for a "boring" C++ file zzzzzz.cpp and its variants

A stack holds open clauses so that "(" opens a clause, "|" is inside a clause, and ")" closes it. For simplicity, assume "|" only inside parenthesis and no escape characters.

```

class RegularExpressionMatcher
{
    string re;
    int m;
    GraphAA<bool> g; //dummy edge data
public:
    RegularExpressionMatcher(string const& theRe): re(theRe), m(re.length()), g(m + 1)
    {
        Stack<int> clauses;
        for(int i = 0; i < m; ++i)
        {
            int clauseStart = i;
            if(re[i] == '(' || re[i] == '|') clauses.push(i);
            else if(re[i] == ')')
            {
                int clauseOp = clauses.pop();
                if(re[clauseOp] == '|')
                {
                    clauseStart = clauses.pop();
                    g.addEdge(clauseStart, clauseOp + 1);
                    g.addEdge(clauseOp, i);
                }
                else clauseStart = clauseOp;
            }
            if(i < m - 1 && re[i + 1]=='*') //to next start from clause start
                g.addUndirectedEdge(clauseStart, i + 1);
            if(re[i] == '(' || re[i] == '*' || re[i] == ')')
                g.addEdge(i, i + 1); //to next state from current
        }
    }
};

```

Active states initially consist of state 0. DFS computes all states reachable from them by taking epsilon transitions. Taking read-character transitions from the current active states gives the next set of active states. The text matches if the final state m is active after reading all characters.

```

Vector<int> findActiveStates(Vector<int> const& sources)
{
    Vector<bool> visited(g.nVertices(), false);
    DefaultDfsAction a;
    for(int i = 0; i < sources.getSize(); ++i) if(!visited[sources[i]])
    {
        visited[sources[i]] = true;
        DFSComponent(g, sources[i], visited, a);
    }
    Vector<int> activeStates;
    for(int i = 0; i < visited.getSize(); ++i)
        if(visited[i]) activeStates.append(i);
    return activeStates;
}
bool matches(string const& text)
{
    Vector<int> activeStates = findActiveStates(Vector<int>(1, 0));
    for(int i = 0; i < text.length() && activeStates.getSize() > 0; ++i)
        //must be in >= 1 active state to keep going
        Vector<int> stillActive;
        for(int j = 0; j < activeStates.getSize(); ++j)
            if(activeStates[j] < m && re[activeStates[j]] == text[i])
                stillActive.append(activeStates[j] + 1);
        activeStates = findActiveStates(stillActive);
    }
    for(int j = 0; j < activeStates.getSize(); ++j)
        if(activeStates[j] == m) return true;
    return false;
}

```

The runtime is $O(nm)$ due to needing to update active states after every character of the text.

14.5 Extended Patterns

An **extended pattern** is a regular expression that uses “*” and “|” on only single characters, allowing faster search. The simplest solution is to extend single pattern search algorithm **shift-and** (Navarro & Raffinot 2002).

For a substring pattern of length m , it maintains an active state bit string s of m bits, such that bit i is on if the last $i + 1$ characters of the text matched the pattern prefix of length $i + 1$ by anding with read-character-specific bit strings. During matching a set bit remains set in the shifted position if the corresponding character matched. Assume $m <$ word size w . If not, use a bit set—reporting all matches takes $O(nm/w)$ time.

1. \forall possible character
2. Set bit string c of size m , to 0
3. \forall pattern character at position i
4. Set bit i of corresponding c
5. Initially $s = 0$
6. While $j < n$
7. Read text character $t[j]$
8. Set s to $(s \ll 1 \mid 1) \& c[t[j]]$
9. Report match at position $j - (m - 1)$ if bit $m - 1$ is set

E.g. for pattern “apple” and character “p” $c = 01100$. To extend for:

- **Joker characters** matching any character and “|” of several characters, set the corresponding c bits of every matching character.
- Repeatable characters corresponding to regular expression xx^* , remember their matching preshift positions in s after shifting. If bit string R marks the positions, use $((s \ll 1 \mid 1) \mid (s \& R)) \& c$ (see the online extras website for Navarro & Raffinot 2002 for an explanation).
- Possibly consecutive optional characters that may be omitted, let bit string O indicate their posi-

tions. After reading a character, \forall set bit in s , if the next k positions are optional, set the next k bits. The preprocessing calculates L and P —respectively the last and the preceding positions of blocks of consecutive set bits in O . E.g., if $O = 01010110$, and $s = 00100001$, s becomes 01100111 , F 01010100 , and P 00101001 . The calculation of P assumes no optional characters in bit 0, corresponding to pattern start, which isn't a problem because can omit optional characters at the beginning and the end of the pattern. After a character update, set s to $s \mid O \& ((s \mid L) ^ \sim((s \mid L) - P))$ (see Navarro & Raffinot 2002 for an explanation).

$"x^*$ " is equivalent to making x optional and repeatable. The implementation assumes the caller sets up `charPos` mask to allow "/" of characters. The interface is somewhat different from that of exact matching algorithms because only work with a bounded alphabet, and keep state other than the current position.

```

class ShiftAndExtended
{  

    //Joker handling omitted for simplicity  

    enum{ALPHABET_SIZE = 1 << numeric_limits<unsigned char>::digits};  

    unsigned char*pattern;  

    int patternSize, position;//patternSize must be before masks  

    unsigned long long charPos[ALPHABET_SIZE], O, P, L, R, state;  

    unsigned long long makeMask(Vector<int> const& positions)const  

    {  

        unsigned long long mask = 0;  

        for(int i = 0; i < positions.getSize(); ++i)  

        {  

            assert(positions[i] >= 0 && positions[i] < patternSize);  

            Bits::set(mask, positions[i], true);  

        }  

        return mask;  

    }  

public:  

    ShiftAndExtended(unsigned char* thePattern, int thePatternSize,  

        Vector<int> const& repeatedPositions = Vector<int>(),  

        Vector<int> const& optionalPositions = Vector<int>(): position(0),  

        state(0), patternSize(thePatternSize), pattern(thePattern),  

        R(makeMask(repeatedPositions)), O(makeMask(optionalPositions))  

    {//first precompute character bit strings  

        assert(patternSize <= numeric_limits<unsigned long long>::digits &&  

            !Bits::get(O, 0));//position 0 can't be optional  

        for(int i = 0; i < ALPHABET_SIZE; ++i) charPos[i] = 0;  

        for(int i = 0; i < patternSize; ++i)  

            Bits::set(charPos[pattern[i]], i, true);  

//then masks for optional characters  

        unsigned long long sides = O ^ (O >> 1);  

        P = (O >> 1) & sides;  

        L = O & sides;  

    }  

    int findNext(unsigned char* text, int textSize)  

    {  

        while(position < textSize)  

        {//first regular and repeatable update  

            state = (((state << 1) | 1) | (state & R)) &  

                charPos[text[position++]];  

//then optional character update  

            unsigned long long sL = state | L;  

            state |= O & (sL ^ ~(sL - P));  

            if(Bits::get(state, patternSize - 1))return position - patternSize;  

        }  

        return -1;  

    }  

};
}

```

The runtime is the same as for regular shift-and.

14.6 String Distance Algorithms

The main distances between strings, based on the number of operations to edit one string into another;

allow characters to be:

- **Hamming**—replaced
- **Indel**—inserted/deleted
- **Lewenstein**—inserted/deleted/replaced
- **Transpose**—inserted/deleted/replaced/transposed

The first three are metric with unit costs \forall operation. Commonly compute a **difference between two strings**, which uses the indel distance to create an edit script that changes string a into string b by a sequence of commands:

- Insert i —insert $b[i]$ into position i of a
- Delete i —delete $a[i]$

To apply the script to a without knowing b , store the inserted $b[i]$ in a vector c so that the j^{th} insertion to position i inserts $c[j]$ into position i of a . This is useful, e.g., for version control software, where text files are strings of line “characters” from a general alphabet, and all past versions of a file are stored as differences between consecutive versions.

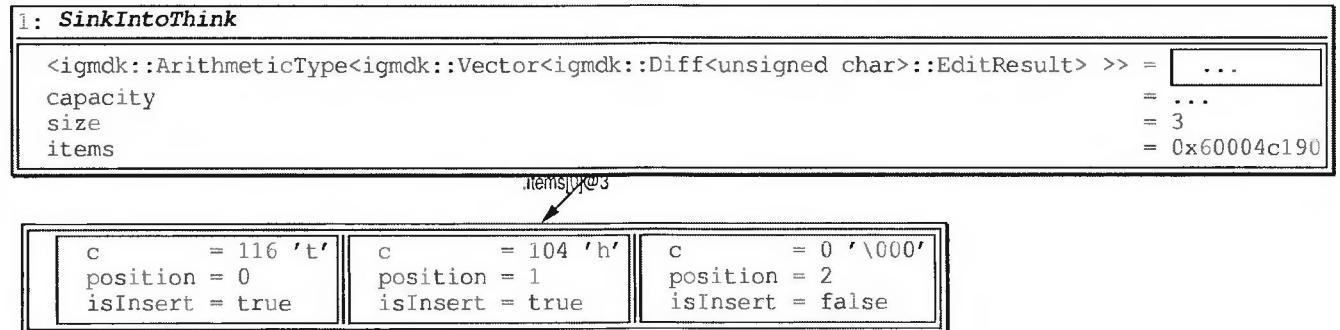


Figure 14.2: Memory layout of an edit sequence, stored in a vector

```

template<typename CHAR> class Diff
{
public:
    struct EditResult
    {
        CHAR c;
        int position;
        bool isInsert;
    };
};

```

Can define distances recursively \forall metric d in a computationally friendly form. Let $p(s, j)$ be the first j letters of s . Then: $d(a, b) = \begin{cases} n, & \text{if } a \text{ empty} \\ d(p(a, m-1), p(b, n-1)), & \text{if } a[m-1] = b[n-1] \\ \min(d(a, p(b, n-1)), d(p(a, m-1), b)) + 1 & \end{cases}$. The corresponding dynamic programming uses $O(nm)$ time and space but is extendable to other distances and nonunit costs.

WMMM algorithm is more efficient. Let x correspond to $p(b, x+1)$, and y to $p(a, y+1)$. Then x and y define a set of diagonals with constant $x - y$. Diagonal $m+1$ has $(-1, -1)$, and diagonal $n+1$ has $(n-1, m-1)$. If need p deletions:

- The dynamic programming relation can reach the base case $(-1, -1)$ from $(n-1, m-1)$ by computing within diagonals $m+1-p$ to $n+1-p$
- The script consists of p deletions and $n-m$ insertions
- $d = n-m+p$

The solution is a shortest path from $(-1, -1)$ to $(n-1, m-1)$, where can move from (x, y) to:

- $(x+1, y+1)$ at no cost if $a[y] = b[x]$
- $(x+1, y)$ at the cost of an insertion
- $(x, y+1)$ at the cost of a deletion

5	6	7	8	9	10	11				
4			-1	0	1	2	3	4		
3				t	h	i	n	k		
2	-1					3	4	5		
1	0	s	1	2		4	5	6		
0	1	i	2	3	4		4	5		
	2	n	3	4	5	4		4		
	3	k	4	5	6	5	4			

Figure 14.3: "sink" into "think": work with diagonal indices 0 to 11, and the shortest edit path increases in cost to 3

The shortest path logic maintains \forall diagonal its current x and the sequence of edits that lead to (x, y) , calculating $y = x - (d - 1 - m)$. The x are stored as a **frontier** vector, initialized to -2 as a convenient base case. Because don't know p , it's initially 0 and iteratively incremented, extending all diagonals until reaching $(n - 1, m - 1)$. Need $\leq n + m + 3$ diagonals. Extending border diagonal first leads to $O(np)$ and expected $O(n + dp)$ time and space use, where d is the edit distance (Wu et al. 1989). The implementation assumes $m \leq n$.

```

struct Edit
{
    Edit* prev; //used only for intermediate work and not the final result
    int position;
    bool isInsert;
};

static Vector<EditResult> DiffInternal(Vector<CHAR> const& a,
                                         Vector<CHAR> const& b, CHAR const& nullC)
{
    int M = a.getSize(), N = b.getSize(), size = M + N + 3,
        mainDiagonal = N + 1;
    assert(M <= N); //s must be shorter than b for this helper
    Vector<int> frontierX(size, -2);
    Vector<Edit*> edits(size, 0);
    Freelist<Edit> f;
    for(int p = 0; frontierX[mainDiagonal] < N - 1; ++p)
        //from lower left to main
        for(int d = M + 1 - p; d < mainDiagonal; ++d)
            extendDiagonal(d, frontierX, edits, a, b, f);
        //from upper right to main
        for(int d = mainDiagonal + p; d >= mainDiagonal; --d)
            extendDiagonal(d, frontierX, edits, a, b, f);
    //retrieve the computed path in reverse order
    Vector<EditResult> result;
    for(Edit* link = edits[mainDiagonal]; link; link = link->prev)
    {
        EditResult er = {nullC, link->position, link->isInsert};
        result.append(er);
    } //fix the order
}

```

```

        result.reverse();
        return result;
    }
}

```

The algorithm extends a diagonal by allowing one more edit operation. Because a deletion from the *frontier* point on diagonal $i + 1$ or an insertion from the one on diagonal $i - 1$ lead to the furthest point on diagonal i , can compute $x = \max(\text{frontier}[i-1] + 1, \text{frontier}[i+1])$. This costs an edit, except as the base case when jumping into $(-1, -1)$. Then can extend the computed furthest point at no edit cost by incrementing x while $a[y+1] == b[x+1]$.

```

static void extendDiagonal(int d, Vector<int>& frontierX, Vector<Edit*>&
                           edits, Vector<CHAR> const& a, Vector<CHAR> const& b, Freelist<Edit*>& f)
{
    // pick next best edit
    int x = max(frontierX[d - 1] + 1, frontierX[d + 1]),
        y = x - (d - 1 - a.getSize());
    if(x != -1 || y != -1)
    {
        // apply it if not base case
        bool isInsert = x != frontierX[d + 1];
        edits[d] = new(f.allocate())Edit();
        edits[d]->isInsert = isInsert;
        edits[d]->prev = edits[d + (isInsert ? -1 : 1)];
        edits[d]->position = isInsert ? x : y;
        // move diagonally as much as possible
        while(y + 1 < a.getSize() && x + 1 < b.getSize() &&
              a[y + 1] == b[x + 1])
        {
            ++y;
            ++x;
        }
        frontierX[d] = x;
    }
}

```

Editing a into b is the same as b into a with *insert* and *delete* swapped, so $m \leq n$ isn't a problem. Need to add to the deletion positions (the number of insertions – the number of deletions) because each edit shifts the remaining character positions. Insertions aren't affected because each character of b inserted into a is in the same position in both.

```

static Vector<EditResult> diff(Vector<CHAR> const& a, Vector<CHAR> const& b,
                                 CHAR const& nullC = CHAR()) // null char used for delete action as dummy
{
    // edits needed to get a into b - positions are with respect to b
    bool exchange = a.getSize() > b.getSize();
    Vector<EditResult> result = exchange ? DiffInternal(b, a, nullC) :
        DiffInternal(a, b, nullC);
    for(int i = 0, netInserted = 0; i < result.getSize(); ++i)
    {
        // exchange if needed, set characters, and adjust deletion positions
        if(exchange) result[i].isInsert = !result[i].isInsert;
        if(result[i].isInsert)
        {
            ++netInserted;
            result[i].c = b[result[i].position];
        }
        else result[i].position += netInserted--;
    }
    return result;
}

```

To apply an edit script from a to b to a , construct b character-by-character. So iteratively until the next edit position take characters from a .

```

static Vector<CHAR> applyDiff(Vector<CHAR> const& a,
                               Vector<EditResult> const& script)
{
    Vector<CHAR> b;
    int nextA = 0;
    for(int i = 0; i < script.getSize(); ++i)
    {
        // take chars from a until next edit position
        while(b.getSize() < script[i].position)

```

```

    //basic input check - must not run out of a before next position
    assert(nextA < a.getSize());
    b.append(a[nextA++]);
}
if(script[i].isInsert) b.append(script[i].c);
else ++nextA; //skip one a char on delete
}//done with script, append the rest from a
while(nextA < a.getSize()) b.append(a[nextA++]);
return b;
}

```

14.7 Inverted Index

An inverted index maps a term to the list of all documents that contain it and supports the Boolean query *contains(term)*. E.g., a book index is an inverted index, where the map is a sorted vector of words. To create an index, run a document-specific parser on each document and insert its id into the list of every term found by the parser. This works when words have separators such as spaces and punctuation marks.

Can define many types of queries, in particular Boolean formulas such as *contains("inverted") & contains("index")*. Here this would find the corresponding id lists and compute their intersection. E.g., the intersection of two sorted sequences of sizes n and m can be optimally done by a relatively simple algorithm in $O(n \lg(m/n))$ time, where n is the size of the shorter sequence (Baeza-Yates & Salinger 2010). With huge data sets might have many servers, with each dedicated to a particular term range. Then to intersect two sequences pick one server randomly, and send its data to the other server, which computes and returns the intersection with its own data. Other operations like union and excluded words can also be supported.

For very large indices stored on disk, such as the ones managed by search engines, the map is a B-tree. To compress the lists sort them, and store the first number in byte code (see the "Compression" chapter) and any other number as a byte-encoded difference to the previous number. To make the index parallel can use a hash table, and dedicate each server to a particular range of the hash function's output.

14.8 Suffix Index

A **suffix array** of a string of size n is an array of positions of the string's suffixes in lexicographic order. Multilevel quicksort computes it in $O(n^2 \lg(n))$ and expected $O(n \lg(n))$ time.

mississippi

10	i
7	ippi
4	issippi
1	ississippi
0	mississippi
9	pi
8	ppi
6	sippi
3	sissippi
5	ssippi
2	ssissippi

Figure 14.4: Suffix array for "mississippi", with the corresponding suffixes

An asymptotically optimal algorithm for a general alphabet uses the **doubling lemma** (Crochemore et al. 2007): Let $r(i, k)$ be 0-based the rank of the suffix at position i in the list of suffixes sorted on the first k

letters. E.g., for aba , $r(0, 1) = 0$, $r(1, 1) = 1$, and $r(2, 1) = 0$. Then $r(i, 2k)$ is the rank of $(r(i, k), r(i + k, k))$ in a lexicographically sorted list of all such pairs. This holds even if the ranks are shifted by a constant.

The rank for positions $\geq n$ is -1 for a suffix array and position $\% n$ for Burrows-Wheeler transform (see the "Compression" chapter). For the former map ranks from $[-1, n - 1]$ to $[0, n]$ to allow sorting with KSort.

```
struct SARank
{
    int* ranks;
    int n, k;
    int operator()(int i) const { i += k; return i < n ? ranks[i] + 1 : 0; }
};

struct BWTRank
{
    int* ranks;
    int n, k;
    int operator()(int i) const { return ranks[(i + k) % n]; }
};
```

To compute the suffix array sort all characters to get ranks with $k = 1$ and double k until $k \geq n$, or all ranks are unique. The algorithm sorts rank pairs with KSort in $O(n)$ time, giving $O(n \lg(n))$ runtime and $3n$ -word working space. It updates and returns the current permutation of suffixes.

```
template<typename RANKER, typename ITEM>
Vector<int> suffixArray(ITEM* const vector, int n)
{
    Vector<int> ranks(n, 0), sa(n, 0);
    for(int i = 0; i < n; ++i) sa[i] = i;
    quickSort(sa.getArray(), 0, n - 1, IndexComparator<ITEM>(vector));
    ranks[sa[0]] = 0; //set ranks based on first char
    for(int i = 1, r = 0; i < n; ++i)
    {
        if(vector[sa[i]] != vector[sa[i - 1]]) ++r;
        ranks[sa[i]] = r;
    }
    for(int k = 1; k < n; k *= 2)
    { //double
        RANKER r1 = {ranks.getArray(), n, k}, r2 = {ranks.getArray(), n, 0};
        KSort(sa.getArray(), n, n + 1, r1);
        KSort(sa.getArray(), n, n + 1, r2);
        if(k * 2 < n) //else ranks already unique after sort
        { //set doubled ranks based on the tuples
            Vector<int> ranks2(n, 0);
            ranks2[sa[0]] = 0;
            for(int i = 1, r = 0; i < n; ++i)
            { //advance rank if either tuple different
                if(r1(sa[i]) != r1(sa[i - 1]) || r2(sa[i]) != r2(sa[i - 1]))
                    ++r;
                ranks2[sa[i]] = r;
                if(r == n - 1) return sa; //ranks already unique
            }
            ranks.swapWith(ranks2); //more efficient than assignment
        }
    }
    return sa;
}
```

An **lcp array** stores lcps between adjacent suffixes in the suffix array; $lcp[i] = lcp(sa[i - 1], sa[i])$, with $i = 0$ value undefined. The computation uses the **permuted lcp array**, defined by $PLCP[sa[i]] = lcp[i]$, and temporary array $pred$, defined by $pred[i] = \begin{cases} sa[size - 1], & \text{if } i = 0 \\ sa[i - 1] & \end{cases}$; $pred[0]$ is just a convenience value.

$PLCP[i] = lcp(i, pred[i])$. For $i > 0$, $PLCP[i] \geq PLCP[i - 1] - 1$ (Karkkainen et al. 2009; in their notation $pred[0] = \Phi[0]$ is undefined, which is less convenient). So compute $PLCP$ by a linear scan of $pred$ and, for $i > 0$, not looking at the first $PLCP[i - 1] - 1$ characters. Because \sum differences between consecutive $PLCP$ values = $O(n)$, the runtime is $O(n)$.

```
template<typename ITEM> Vector<int> LCPArray(ITEM* text, int size, int* sa)
```

```

{
    Vector<int> pred(size, 0), PLCP(size, 0);
    for(int i = 0; i < size; ++i) pred[sa[i]] = sa[(i ? i : size) - 1];
    for(int i = 0, p = 0; i < size; ++i)
    {
        while(text[i + p] == text[pred[i] + p]) ++p;
        PLCP[i] = p;
        p = max(p - 1, 0);
    } //pred becomes the LCP array now, 0 has lcp(sa[0], sa[n - 1])
    for(int i = 0; i < size; ++i) pred[i] = PLCP[sa[i]];
    return pred;
}

```

Suffix and lcp arrays form a suffix index, which is efficient for many pattern matching tasks. E.g., finding all pattern occurrences takes $O(m\lg(n))$ time with two binary searches, by computing an interval where all suffixes are prefixes of the pattern. The number of matches = right - left + 1.

1: Mississipi

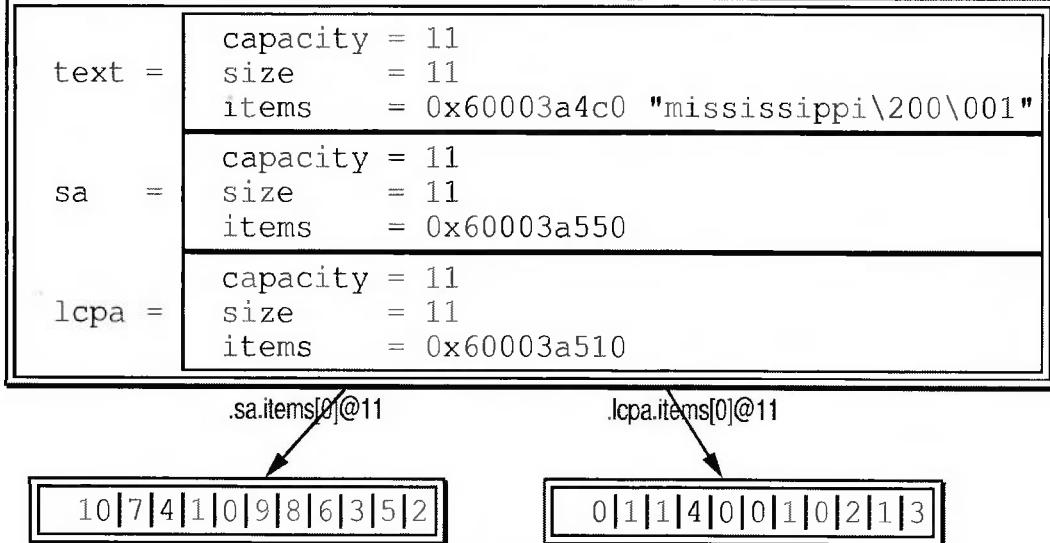


Figure 14.5: Memory layout of a suffix index for "mississippi"

```

template<typename ITEM> struct SuffixIndex
{
    Vector<ITEM> const& text;
    Vector<int> sa, lcpa;
    SuffixIndex(Vector<ITEM> const& theText): text(theText),
        sa(suffixArray<SRank>(text.getArray(), text.getSize())),
        lcpa(LCPArray(text.getArray(), text.getSize(), sa.getArray())) {}
    bool isKLess(ITEM const* a, int aSize, ITEM const* b, int bSize, int k)
    const
    {
        for(int i = 0; i < min(aSize, bSize); ++i)
            if(a[i] > b[i]) return false;
        return max(aSize, bSize) < k;
    }
    pair<int, int> interval(ITEM* pattern, int size)
    {
        int left = 0, right = sa.getSize();
        while(left < right)
        {
            int i = (left + right)/2;
            if(isKLess(&text[sa[i]], sa.getSize() - sa[i], pattern, size,
                      size)) left = i + 1;
            else right = i - 1;
        }
        int left2 = left - 1, right2 = sa.getSize() - 1;
        while(left2 < right2)
        {

```

```

        int i = (left2 + right2)/2;
        if(isKLess(pattern, size, &text[sa[i]], sa.getSize() - sa[i],
                    size)) right2 = i - 1;
        else left2 = i + 1;
    }
    return make_pair(left, right2);
}
}

```

Can find the longest repeated substring in $O(n)$ time by checking which two suffixes have the largest lcp.

14.9 Syntax Tree

In a **syntax tree** each node has a value that is a constant or a function of its children. The root's value = the value of its functional expression. A function can represent any relationship of its node's children.

A syntax tree is usually the result of converting text commands. E.g., $5 + 4$ corresponds to the tree represented by the array $\{(+, 1, 2), (5, -1, -1), (4, -1, -1)\}$ (the last two numbers in a tuple are child pointers). **Lexical analysis** partitions character sequences into symbols and uses specific grammar rules, such as precedence order, to create a tree.

For a readable and editable specification of something, formats such as XML and JSON usually give more than enough functionality and have much API support in many languages.

14.10 Introduction to Succinct Data Structures

A data structure is **succinct** if it uses space $O(\text{the information-theoretical minimum number of bits needed to distinguish the object it represents})$. The latter is usually $\lg(|\text{the set of all possible objects}|)$. E.g., a permutation takes $\lg(n!) \approx n\lg n$ bits to represent, though representing it in such way using a bit set gains little and slows down operations. External fragmentation waste due to memory management isn't considered in this model even though it is important.

A bit set is a natural core data structure, and to be most useful it must be augmented with a **rank-select data structure** that supports:

- Rank(i)—the number of 1's before position i
- Select(i)—the position of the i^{th} 1
- Rank₀(i) and select₀(i)—as above for 0

Rank and select are inverses of each other with the properties (rank₀ and select₀ are symmetric):

- Rank₀(i) = $i - \text{rank}(i)$
- Rank(select(i)) = i
- Select(rank(i)) = (the i^{th} bit is set ? $i : i - 1$)
- If $j = \text{select}(i)$, then rank(j) = i , and rank₀(j) = $j - i$

Beware that in the literature these are sometimes defined for 1-based array indexing, with rank(i) also counting the current bit. Many complex implementations have been proposed for a bit set of size n , getting $O(n)$ extra space and $O(1)$ time. Here prefer the simplest reasonably efficient implementation: \forall word of the bit set store the cumulative bit count. To support sizes larger than 2^{32} , both the bit set and the counters use 64 bit words giving $O(n)$ extra space:

- $O(1)$ rank(i)—use counts[$i/64 - 1$] and the bits of bit set word $i/64$
- $O(\lg(n))$ select(i)—use binary search on the bit counts and the bits of the found bit set word

The bits of bit set words are looked at byte-by-byte using a precomputed table of counts \forall byte. The implementation of select₀(i) is like that of select(i), just for 0's.

```

int rank64(unsigned long long x, int i)
{//ith bit not included
    assert(0 <= i && i < numeric_limits<unsigned long long>::digits);
    //set bits >= i to 0 and popCount
    return popCountWord(x & ((ull)1ll << i) - 1);
    //to include ith bit use i+1 instead of i
    //overflow is fine because result is -1
}
int select64(unsigned long long x, int i, bool is0 = false)
{
    assert(0 <= i && i < numeric_limits<unsigned long long>::digits);
    static PopCount8 p8;

```

```

int result = 0, byteBits = numeric_limits<unsigned char>::digits;
//use popCount to get to the byte with the desired bit position
while(x)
{
    int temp = p8(x & 0xff);
    if(is0) temp = byteBits - temp;
    if(i - temp < 0) break;
    result += byteBits;
    x >>= byteBits;
    i -= temp;
}//scan the bits in the found byte for the desired bit position
for(int j = 0; j <= byteBits; ++j)
{
    bool temp = x & 1 << j;
    if(is0) temp = !temp;
    if(temp && i-- == 0) return result;
    ++result;
}
return -1;
}

class RankSelect
{
    enum{B = numeric_limits<unsigned long long>::digits};
    Bitset<unsigned long long> bitset;
    //cumulative counts of every last bit in a word
    Vector<unsigned long long> counts;
    long long counts0(long long i){return (i + 1) * B - counts[i];}
    long long getCount(long long i, bool is0)
        {return is0 ? (i + 1) * B - counts[i] : counts[i];}
public:
    RankSelect(unsigned long long initialSize = 0): bitset(initialSize){}
    Bitset<unsigned long long>& getBitset(){return bitset;}
    void finalize()
    {
        counts = bitset.getStorage();
        for(long long i = 0; i < bitset.wordSize(); ++i) counts[i] =
            popCountWord(bitset.getStorage()[i]) + (i == 0 ? 0 : counts[i-1]);
    }
    long long rank(long long i)
    {
        assert(0 <= i && i < bitset.getSize());
        long long index = i / B;
        return (index > 0 ? counts[index - 1] : 0) +
            rank64(bitset.getStorage()[index], i % B);
    }
    long long rank0(long long i){return i - rank(i);}
    long long select(long long i, bool is0 = false)
    {
        assert(0 <= i && i < bitset.getSize());
        long long left = 0, right = bitset.wordSize() - 1;
        while(left < right)
        {
            long long middle = (left + right)/2;
            if(getCount(middle, is0) <= i) left = middle + 1;
            else right = middle - 1;
        }
        long long result = select64(bitset.getStorage()[left],
            i - (left == 0 ? 0 : getCount(left - 1, is0)), is0);
        if(result != -1) result += left * B;
        return result;
    }
    long long select0(long long i){return select(i, true);}
    long long prev(long long i){return select(rank(i) - 1);}
}

```

```

long long prev0(long long i){return select0(rank0(i) - 1);}
long long next(long long i){return select(rank(i));}
long long next0(long long i){return select0(rank0(i));}
};

```

Because rank-select is static, structures relying on it can't be made dynamic. Can cut the extra space by two-level structure, but this isn't recommended:

- Level 1 contains 64-bit cumulative sums for 256-bit blocks
- Level 2 contains 8-bit rank counts for 64- or 32-bit blocks

This results in $0.5n$ or $0.33n$ instead of n bits over the bit set.

A simple application is representing a tree. The space redundancy is in the left, right, and parent pointers. For a complete binary tree, these aren't needed because the tree can be represented as a heap, which has no redundancy at all. For a general binary tree, want to have an array of items like for the heap and a small additional structure for navigation. There are $\binom{2n+1}{n}/(2n+1)$ trees with n nodes, $2n$ bits is more than enough to represent such structure instead of the $96n$ bits (32 bits per pointer for the three pointers). The space for the data isn't changed.

Jacobson (1989) proposed the below simple implementations. The general tree one is called **LOUDS**, which is simple and efficient but must be build by level-order traversal, which inconveniently needs a queue or iterative deepening. It might be useful, e.g., to represent certain large XML files.

```

class BinaryTree
{
    //this does not extend to d-ary trees unlike the heaps
    //which need at least log(d)n bits to be distinguished
    RankSelect rs;
    int convert(int i){return rs.getBitset()[i] ? rs.rank(i) : -1;}
public:
    void addNodeInLevelOrder(bool isNotExternalDummyLeaf)
        {rs.getBitset().append(isNotExternalDummyLeaf);}
    void finalize(){rs.finalize();}
    int parent(int i){return (rs.select(i)+1)/2-1;}
    int leftChild(int i){return convert(2 * rs.rank(i) + 1);}
    int rightChild(int i){return convert(2 * rs.rank(i) + 2);}
};

class OrdinalTree
{
    RankSelect rs;
    int convert(int i){return rs.getBitset()[i] ? rs.rank(i) : -1;}
public:
    OrdinalTree(){rs.getBitset().append(1);rs.getBitset().append(0);}
    void addNodeInLevelOrder(int nChildren)
    {
        for(int i = 0; i < nChildren; ++i) rs.getBitset().append(1);
        rs.getBitset().append(0);
    }
    void finalize(){rs.finalize();}
    int parent(int i){return rs.select(i) - i - 1;}
    int firstChild(int i){return convert(rs.select0(i) + 1);}
    int nextChild(int i) {return convert(rs.select(i) + 1);}
};

```

Another simple application is compressed pattern matching in text. Assume the text is compressed and the pattern isn't. Can uncompress the text, and do matching and other operations with it, but sometimes can work on the compressed text directly, which has advantages for potentially very large texts. If the text is compressed using Huffman code or another symbol-to-symbol method, can compress the pattern and match using any method that doesn't skip characters (because don't have random access).

A fairly simple scheme allowing random access is as follows (Fredriksson & Nikitin 2009):

1. Rank all symbols by frequency of occurrence, and let the codewords \forall symbol be rank codewords $0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 00, 3 \rightarrow 01$, etc., i.e., all binary codes of particular length.
2. This code isn't prefix-free, so create a bit set of the same size as the encoded text, with 1 set for each start-of-symbol bit.
3. Put a rank-select data structure on the bit set to allow random access into the structure.

This supports all the operations supported by the original string, but with $O(\lg(n))$ slowdown (due to the

implementation of select here). To get the best code, should preprocess the text using move-to-front transform and store its symbol mapping. This is actually a general method to provide random access to any symbol code and is efficient because the code needn't be prefix free. Specialized matching algorithms allow more effective compression methods.

A major area of research in string algorithms is compressed indices such as the **FM index** (Adjeroh et al. 2008). A suffix array takes a lot of space to compute and represent, so want to compress it as much as possible using bit algorithms, though at the cost of a slowdown. The idea is that for applications such as DNA sequencing, this constant-factor memory reduction can make a difference between solvable and unsolvable. But with 64-bit architectures and growing memory sizes, this is questionable. See Navarro (2016) for much more on these and other succinct data structures, including a supposedly better balanced-parenthesis representation of a tree.

14.11 Implementation Notes

Despite several dedicated textbooks, all the implementations were difficult and needed substantial primary literature research. Selection of algorithms was also problematic.

- Basic search—merging Horspool and HashQ is original, and not many sources choose Horspool as the primary search algorithm
- Suffix array construction—I had difficulty even running implementations of several algorithms that various researchers wrote and eventually settled on the presented one which has a simple idea and works for general alphabets
- LCP array construction needed primary literature
- Computing a string difference surprisingly needed primary literature because the presented algorithm is old but none of the textbooks, all of which came out later, cover it
- I chose to not implement any compressed matching algorithms because they are highly specialized and still an active research field

Most of the tests were added years after the first implementations, and several bugs were discovered, some of which are still not fixed.

14.12 Comments

Many algorithms for string search have been proposed and compared (Faro & Lecroq 2010). Horspool and HashQ are among the simplest ones and perform competitively in almost all cases.

Many algorithms have been proposed for finding a string difference, but in the typical case of relatively few edits, WMMM is the most efficient. The Unix diff for some reason uses its older, less efficient version (Hunt et al. 1998). A related task is that of longest common subsequence, algorithms for which are reviewed in Bergroth et al. (2000).

For approximate pattern matching, done, e.g., by spell-checking software, one simple approach to matching a word against a dictionary of known words is by first trying to find it as is first. If don't, generate all words with distance 1, and try to find them also. Repeating with distance 2, etc., until the search becomes too expensive. The generation itself is simple for a word of k characters:

1. Create k words by deleting each character
2. Create $(k + 1)|A|$ words by inserting each alphabet character into every possible position
3. Create $(k - 1)(|A| - 1)$ words by substituting each character with a different one
4. Create $(k - 1)$ words by swapping each adjacent character pair

The i^{th} iteration needs $O((k|A|)^i)$ time, even if remove duplicates, so the method is practical maybe for distance ≤ 3 .

A potentially better method is putting the dictionary into a VP tree (see the “Computational Geometry” chapter) with edit distance as distance, and using nearest-neighbor search to find nearest words.

14.13 Projects

- Rename “VECTOR” and “VECTOR2” types to the more specific “TEXT” and “PATTERN”.
- For HashQ implement logic that picks q automatically and uses Lecroq's h . Compare it with some other h , such as those of higher quality from the “Hashing” chapter.
- Customize HashQ to compare DNA sequences, store in a k -bit-word vector for $k = 2$ (see the “Miscellaneous Algorithms” chapter).
- Implement joker handling for the extended shift-and. Hint—compute separate masks \forall joker, and use these instead of character-specific ones.

- Implement substring search based on a suffix array of the text. Compare the performance against that of Horspool.
- Research and implement other functionality for the suffix index. Much of it comes from more time-but less memory-efficient **suffix tree**.

14.14 References

- Adjerooh, D., Bell, T. C., & Mukherjee, A. (2008). *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer.
- Baeza-Yates, R., & Salinger, A. (2010). Fast intersection algorithms for sorted sequences. In *Algorithms and Applications* (pp. 45-61). Springer, Berlin, Heidelberg.
- Bergrøth, L., Hakonen, H., & Raita, T. (2000). A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000* (pp. 39-48). IEEE.
- Crochemore, M., Hancart, C., & Lecroq, T. (2007). *Algorithms on Strings*. Cambridge University Press.
- Faro, S., & Lecroq, T. (2010). The exact string matching problem: a comprehensive experimental evaluation. *arXiv preprint arXiv:1012.2547*.
- Fredriksson, K., & Nikitin, F. (2009). Simple random access compression. *Fundamenta Informaticae*, 92(1-2), 63-81.
- Hunt, J. J., Vo, K. P., & Tichy, W. F. (1998). Delta algorithms: an empirical analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2), 192–214.
- Jacobson, G. (1989). Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science* (pp. 549-554). IEEE Computer Society.
- Kärkkäinen, J., Manzini, G., & Puglisi, S. J. (2009). Permuted longest-common-prefix array. In *Combinatorial Pattern Matching* (pp. 181–192). Springer.
- Lecroq, T. (2007). Fast exact string matching algorithms. *Information Processing Letters*, 102(6), 229–235.
- Navarro, G. (2016). *Compact Data Structures: A Practical Approach*. Cambridge University Press.
- Navarro, G., & Raffinot, M. (2002). *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press. Online extras at <https://www.dcc.uchile.cl/~gnavarro/FPMbook/extras.html>. Accessed August 19, 2018.
- Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern Information Retrieval*. Addison-Wesley.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.
- Wu, S., Manber, U., Myers, G., & Miller, W. (1990). An O(NP) sequence comparison algorithm. *Information Processing Letters*, 35(6), 317–323.

15 Compression

15.1 Introduction

Data compression is ubiquitous in modern computing, and all the main algorithms except for LZ77 (the one behind GZIP) are discussed in this chapter.

15.2 Fundamental Limits

Compression saves space by converting a bit string, representing maybe a large image file, into a shorter bit string, which can be converted back to the original. No algorithm can shrink every bit string, otherwise applying it repeatedly shrinks the size to 0. Compression and decompression are one-to-one functions, and \exists more sequences of length $n + 1$ than n , so every algorithm that shortens some bit strings lengthens others. Lengthened bit strings could have only 1 extra bit though—e.g., output shorter(the original, the compressed) and a bit to show which. But in practice compression does save substantial space.

Can highly compress certain bit strings. E.g., “million 0's” compresses a bit string consisting of a million 0's. The shortest description of a bit string is its **Kolmogorov complexity**, computing which is undecidable because a seemingly incompressible, long bit string could be generated by a clever pseudorandom generator and be representable as the seed and generator's code. Technically Kolmogorov complexity is defined with respect to a **universal machine** (such as valid C++ programs with no library help), but this is irrelevant because the idea only has conceptual value.

15.3 Entropy

For a sequence of symbols from an alphabet of size k , occurring with probabilities given by distribution X , a practical measure of compressibility is **first-order entropy** $H(X) = -\sum \Pr(\text{symbol}) \lg(\Pr(\text{symbol}))$. When each symbol is a **supersymbol** of k symbols (e.g., a `short` is usually two `chars`), **entropy** is first-order entropy for $k \rightarrow \infty$. The length of an optimal codeword for a symbol $\approx -\lg(\Pr(\text{symbol}))$.

Creating a superalphabet doesn't change entropy if symbols are independent. E.g., using byte and bit symbols gets the same result if the bits are independent. Otherwise, creating supersymbols reduces entropy—always $H(X) \geq H(X^*)/k$. E.g., entropy of an ASCII text is lower using letters than bits because most 8-bit values never occur, and e is more likely than z.

Entropy is the minimum $E[\text{the number of bits per symbol}]$ to communicate a supersymbol sequence, meaning it's the limit of compressibility without other information. A sequence's representation's size – its entropy \geq the maximum compression gain. E.g., entropy of English > 1 bit per letter, and some algorithms compress a variety of ASCII texts to < 2 bits per letter (Sayood 2002).

15.4 Bit Stream

A bit stream can read and write bits and is a wrapper around a bit set. Its interface is an abstraction and can be implemented as an external memory or a byte stream. For efficiency also allow reading and writing words. When work with files, expect certain portability:

- As compiled, the code will work on any machine
- A file compressed on one architecture will be decompressible on another

These are actually in conflict because not every machine has an 8-bit word type to represent a byte, though almost all current ones do. Also a character, which is assumed to be a byte in C++, needn't have 8 bits (still `sizeof(char) == 1`), though on almost all current machines it does. Finally C++ streams are specialized only for characters and not `uint_8`. So a practical solution here is to use `unsigned char` as byte type and not assume it has 8 bits. This breaks file portability between architectures of different `char` bit size, but using `uint_8` would give compiler error due to absence of the type.

In general, binary files aren't portable, only text files are. When need portability of binary files, e.g., for network protocols, specify the logical bit layout that is friendly to common architectures. Any uncommon ones must read such files in their own way using the specific APIs provided. These would, e.g., use a portable ASCII character stream but only pack the first 8 bits in a character, no matter how many it can hold. This isn't considered further here because the provided vector of characters can be converted to a special representation by further processing. Same for the related functionality of error-correction coding the cryptography (discussed in later chapters).

```

struct Stream
{
    unsigned long long position;
    Stream(): position(0) {}
};

struct BitStream : public Stream
{
    Bitset<unsigned char> bitset; //unsigned char for portability
    enum{B = numeric_limits<unsigned char>::digits};
    BitStream() {}
    BitStream(Bitset<unsigned char> const& aBitset): bitset(aBitset) {}
    BitStream(Vector<unsigned char> const& vector): bitset(vector) {}
    void writeBit(bool value){bitset.append(value);}
    bool readBit()
    {
        assert(bitsLeft());
        return bitset[position++];
    }
    void writeByte(unsigned char byte){writeValue(byte, B);}
    void writeBytes(Vector<unsigned char> const& bytes)
        {for(int i = 0; i < bytes.getSize(); ++i) writeByte(bytes[i]);}
    unsigned char readByte(){return readValue(B);}
    Vector<unsigned char> readBytes(int n)
    {
        assert(n <= bytesLeft());
        Vector<unsigned char> result(n);
        for(int i = 0; i < n; ++i) result[i] = readByte();
        return result;
    }
    void writeValue(unsigned long long value, int bits)
        {bitset.appendValue(value, bits);}
    unsigned long long readValue(int bits)
    {
        assert(bits <= bitsLeft());
        position += bits;
        return bitset.getValue(position - bits, bits);
    }
    unsigned long long bitsLeft()const{return bitset.getSize() - position;}
    unsigned long long bytesLeft()const{return bitsLeft()/B;}
};

```

The stream acts as a builder for a bit set. To convert a bitset to a byte vector, encode the number of bits in the last byte of the bit set storage into the last byte of the vector:

```

Vector<unsigned char> ExtraBitsCompress(Bitset<unsigned char> const& bitset)
{
    assert(bitset.getSize() > 0); //makes no sense otherwise
    Vector<unsigned char> result = bitset.getStorage();
    result.append(bitset.garbageBits());
    return result;
}

Bitset<unsigned char> ExtraBitsUncompress(Vector<unsigned char> byteArray)
{
    assert(byteArray.getSize() > 1 && byteArray.lastItem() < BitStream::B);
    int garbageBits = byteArray.lastItem();
    byteArray.removeLast();
    Bitset<unsigned char> result(byteArray);
    while(garbageBits--) result.removeLast();
    return result;
}

```

Both take $O(n)$ time.

15.5 Codes

A code assigns a bit sequence $\forall \text{symbol} \in \text{alphabet}$. Assume that the alphabet is indexed. In general, compression methods **model**, transforming the input sequence into a compressed or easier-to-compress sequence, and **code**, encoding the input into a bit string.

Prefix-free codes, where no codeword is a prefix of another, are uniquely decodable. They satisfy **Kraft's inequality**: $\sum 2^{-\text{length}(\text{codeword}_i)} \leq 1$, which is stronger than the entropy bound because can't use fractional bits. A codeword consists of a value and a **length indication**. The latter is one of:

- A **terminating character** such as a particular bit sequence
- An encoded length preceding the value
- A fixed-length convention—e.g., always using 32 bits for a certain value

For bit-error robustness, preceding length < terminating character < length convention. The input alphabet is usually a code with fixed-length convention, such as ASCII.

15.6 Static Codes

A static code assigns the same bit sequence to the same symbol, independent of context. E.g., binary code usually uses 8 bits for `char` and 32 for `int`. It's almost perfect when know how many bits to use. E.g., for DNA $(A, C, G, T) \rightarrow (00, 01, 10, 11)$.

Unary code outputs value-many 1's and a 0 for termination—e.g., $5 \rightarrow 11110$. A b -bit number needs $O(2^b)$ bits, so the code is useful only as a building block. Other efficient codes are between binary and unary and need $O(b)$ time and codeword space.

```
void UnaryEncode(int n, BitStream& result)
{
    while (n--) result.writeBit(true);
    result.writeBit(false);
}
int UnaryDecode(BitStream& code)
{
    int n = 0;
    while (code.readBit()) ++n;
    return n;
}
```

Gamma code expresses the number as $2^x + y$ for the largest possible x and writes x in unary and y in binary using x bits—e.g., $5 \rightarrow 11001$ because $5 = 2^2 + 1$. $\approx 2\lg(n)$ bits represent n , which makes the code asymptotically optimal because $\lg(n)$ is the minimum. ! \exists code for 0, but $1 \rightarrow 1$.

```
void GammaEncode(unsigned long long n, BitStream& result)
{
    assert(n > 0);
    int N = lgFloor(n);
    UnaryEncode(N, result);
    if (N > 0) result.writeValue(n - twoPower(N), N);
}
unsigned long long GammaDecode(BitStream& code)
{
    int N = UnaryDecode(code);
    return twoPower(N) + (N > 0 ? code.readValue(N) : 0);
}
```

Static codes correspond to strategies for finding an unknown positive number using comparisons. E.g., unary code is similar to linear search, and gamma to exponential.

Any n is uniquely represented as $\sum \text{some of the Fibonacci numbers} \leq n$. So no two consecutive Fibonacci numbers appear in the sum (otherwise they would be replaced by their sum). **Fibonacci code**:

1. **Find the numbers representing the sum**
2. $\forall \text{number, from smallest to largest, put 1 if included and 0 if not}$
3. **Put terminator 1**

Two consecutive 1's signal termination. E.g., $7 \rightarrow 01011$ because $7 = 0 \times 1 + 1 \times 2 + 0 \times 3 + 1 \times 5$. Because the i^{th} Fibonacci number $\approx G^i$, where G is the golden ratio, and $i+1$ bits represent n , $G^i \leq n < G^{i+1}$, and $\approx \lg(n)/\lg(G) + 1 \approx 1.44\lg(n) + 1$ bits represent n .

```
void advanceFib(unsigned long long& f1, unsigned long long& f2)
{
```

```

unsigned long long temp = f2;
f2 += f1;
f1 = temp;
}

void FibonacciEncode(unsigned long long n, BitStream& result)
{
    assert(n > 0);
    //find largest fib number f1 <= n
    unsigned long long f1 = 1, f2 = 2;
    while(f2 <= n) advanceFib(f1, f2);
    //mark the numbers from highest to lowest
    Bitset<unsigned char> reverse;
    while(f2 > 1)
    {
        reverse.append(n >= f1);
        if(n >= f1) n -= f1;
        unsigned long long temp = f1;
        f1 = f2 - f1;
        f2 = temp;
    } //change order to lowest to highest and add terminator
    reverse.reverse();
    result.bitset.appendBitset(reverse);
    result.writeBit(true);
}

unsigned long long FibonacciDecode(BitStream& code)
{
    unsigned long long n = 0, f1 = 1, f2 = 2;
    for(bool prevBit = false;; advanceFib(f1, f2))
    { //add on the next Fibonacci number until see 11
        bool bit = code.readBit();
        if(bit)
        {
            if(prevBit) break;
            n += f1;
        }
        prevBit = bit;
    }
    return n;
}

```

Byte code represents numbers in base 128 (assuming a `char` has 8 bits) by a little-endian sequence of bytes (i.e., the smallest byte first), where the highest bit signals the last character, with 0 meaning last. E.g., $128^2 \rightarrow 10000000|10000000|00000001$. Byte code is fast due to not needing bit manipulations, efficient, using $\lceil \log_{128}(n) \rceil$ bytes = $\Theta(1.14\lg(n))$ bits, and usually the method of choice for data structure compression. UTF-8 uses byte code for Unicode characters.

```

void byteEncode(unsigned long long n, BitStream& result)
{
    enum{M05 = 1 << (numeric_limits<unsigned char>::digits - 1)};
    do
    {
        unsigned char r = n % M05;
        n /= M05;
        if(n) r += M05;
        result.writeByte(r);
    }while(n);
}

unsigned long long byteDecode(BitStream& stream)
{
    unsigned long long n = 0, base = 1;
    enum{M05 = 1 << (numeric_limits<unsigned char>::digits - 1)};
    for(; base *= M05)
    {
        unsigned char code = stream.readByte(), value = code % M05;

```

```

        n += base * value;
        if(value == code) break;
    }
    return n;
}

```

Can portably transform a number into a sequence of bytes without compression:

```

Vector<unsigned char> ReinterpretEncode(unsigned long long n, int size)
{
    assert(size > 0);
    enum{M = 1 << numeric_limits<unsigned char>::digits};
    Vector<unsigned char> result;
    while(size-- > 0)
    {
        result.append(n % M);
        n /= M;
    }
    return result;
}

unsigned long long ReinterpretDecode(Vector<unsigned char> const& code)
{
    assert(code.getSize() > 0);
    unsigned long long n = 0, base = 1;
    enum{M = 1 << numeric_limits<unsigned char>::digits};
    for(int i = 0; i < code.getSize(); ++i)
    {
        n += base * code[i];
        base *= M;
    }
    return n;
}

```

The best code for values $\in [1, 7]$ is gamma, $\in [4, 33] \cup [128, 1596]$ Fibonacci, and $\in [21, 127] \cup [987, \infty)$ byte. So in practice use byte code unless know that deal with very small numbers.

Value	Unary	Gamma	Fibonacci	Byte
0	0	N/A	N/A	'00000000
1	10	1	'11	'00000001
2	110	100	'011	'00000010
3	1110	101	'0011	'00000011
4	11110	11000	'1011	'00000100
5	111110	11001	'00011	'00000101
6	1111110	11010	'10011	'0'0000110
7	11111110	11011	'01011	'00000111
8	111111110	1110000	'000011	'00001000
16	Too long	111100000	'0010011	'00010000
32		111110000000	'00101011	'00100000
64		11111100000000	'1000100011	'01000000
128		1111111000000000	'100010001011	'10000000000000001
256		'11111110000000000	'0100001000011	'1000000000000010
512		'111111110000000000	'10101001010011	'1000000000000100
1024		'11111111100000000000	'0010000100000011	'1000000000000100

15.7 Huffman Codes

Huffman codes are optimal \forall particular empirical distribution of observed symbols. The algorithm gathers probabilities in the first pass over the data and uses them to calculate the codes in the second. Binary trees with a leaf \forall observed symbol represent all prefix-free codes. Walking to a symbol's leaf and outputting 0 for going left and 1 for right gives its codeword.

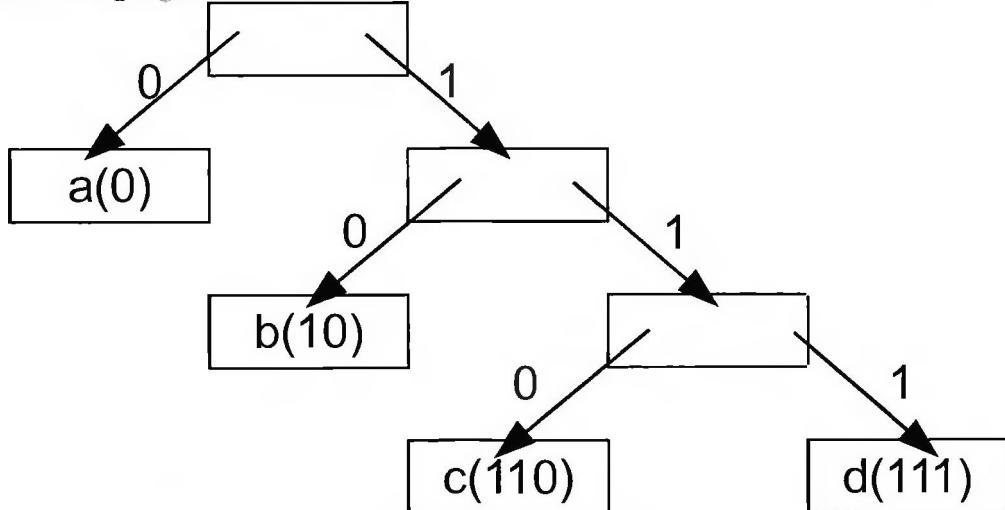


Figure 15.1: A possible Huffman encoding for alphabet $\{a, b, c, d\}$

E.g., given a, b, c, d , possible codeword sets include $(00, 01, 10, 11)$ and $(0, 10, 110, 111)$. If the letters occur with probability $\frac{1}{4}$, $E[\text{the codeword length}] = 2$ for the first set and 2.25 for the second, but for probabilities $\frac{1}{2}, \frac{1}{4}, \frac{1}{8},$ and $\frac{1}{8}$, for the second set $E[\text{the length}] = 1.75$. Want to find for a given distribution a tree that minimizes $E[\text{the length}]$.

Huffman coding provably computes an optimal tree, building it bottom-up:

1. Create a forest with a symbol in each leaf
2. Until have a single tree
3. Merge two trees with the smallest symbol occurrence counts

So every node has 0 or 2 children, and to decide if it's a leaf only check if the left child is null.

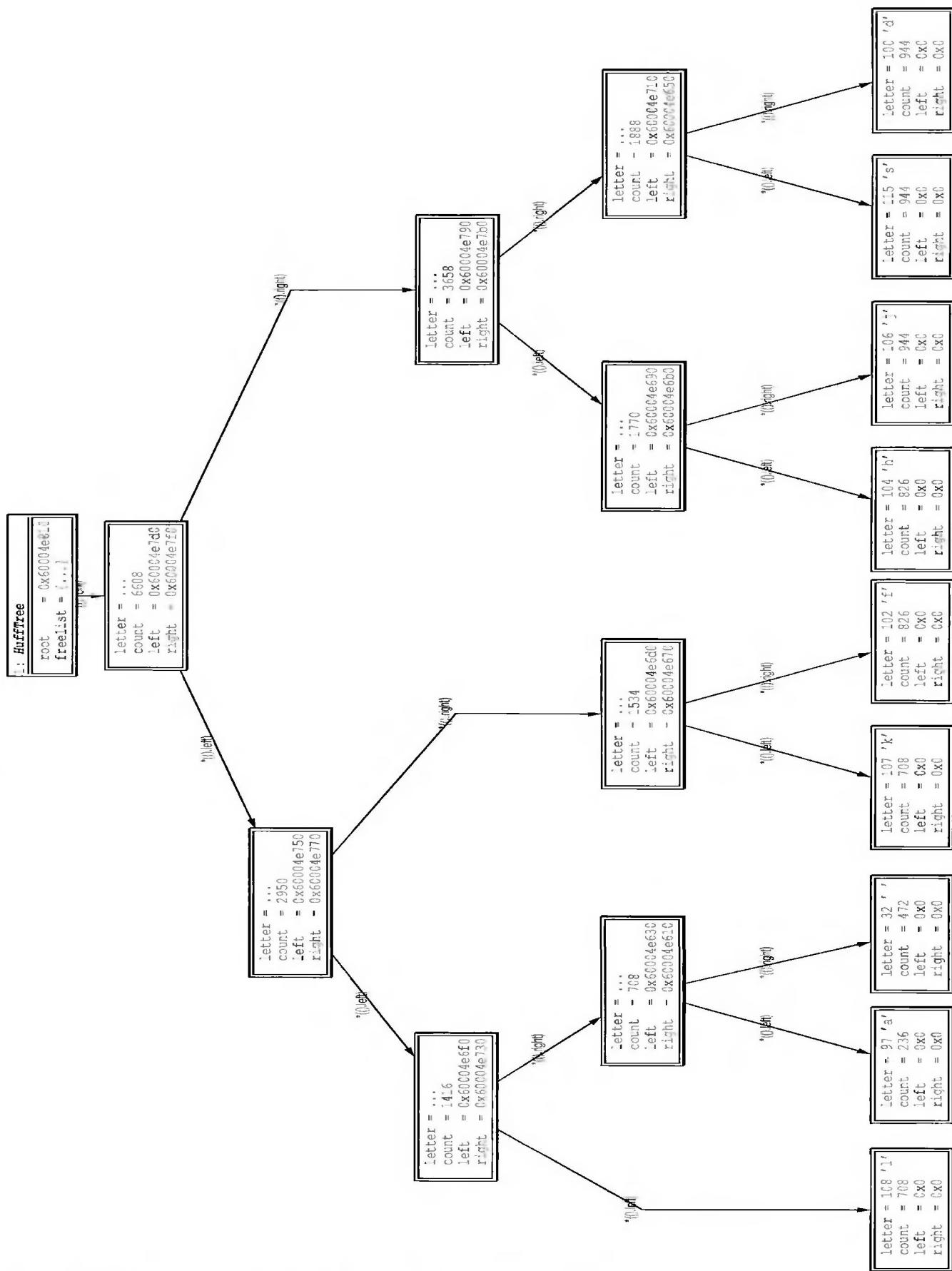


Figure 15.2: Memory layout of a Huffman tree for a large ASCII string

```

struct HuffmanTree
{
    enum{W = numeric_limits<unsigned char>::digits, N = 1 << W};

    struct Node
    {
        unsigned char letter;
    };
}
```

```

int count;
Node *left, *right;
Node(int theCount, Node* theLeft, Node* theRight,
     unsigned char theLetter): left(theLeft), right(theRight),
     count(theCount), letter(theLetter) {}
bool operator<(Node const& rhs) const{return count < rhs.count;}
} * root;
Freelist<Node> f;
HuffmanTree(Vector<unsigned char> const& byteArray)
{//calculate frequencies
    int counts[N];
    for(int i = 0; i < N; ++i) counts[i] = 0;
    for(int i = 0; i < byteArray.getSize(); ++i) ++counts[byteArray[i]];
    //create leaf nodes
    Heap<Node*, PointerComparator<Node>> queue;
    for(int i = 0; i < N; ++i) if(counts[i] > 0) queue.insert(
        new(f.allocate())Node(counts[i], 0, 0, i));
    //merge leaf nodes to create the tree
    while(queue.getSize() > 1)//until forest merged
    {
        Node *first = queue.deleteMin(), *second = queue.deleteMin();
        queue.insert(new(f.allocate())
            Node(first->count + second->count, first, second, 0));
    }
    root = queue.getMin();
}
};

```

Traversing the tree, implemented as a member function of `Node`, creates a **codebook** that maps each symbol to a code.

```

void populateCodebook(Bitset<unsigned char>* codebook)
{
    Bitset<unsigned char> temp;
    root->traverse(codebook, temp);
}
void traverse(Bitset<unsigned char>* codebook,
             Bitset<unsigned char>& currentCode)
{
    if(left)//internal node
    {
        currentCode.append(false); //went left
        left->traverse(codebook, currentCode);
        currentCode.removeLast();
        currentCode.append(true); //went right
        right->traverse(codebook, currentCode);
        currentCode.removeLast();
    }
    else codebook[letter] = currentCode; //leaf
}

```

Encoding writes the codebook and the codeword \forall symbol to a byte vector with the extra bits encoding.

```

Vector<unsigned char> HuffmanCompress(Vector<unsigned char> const& byteArray)
{
    HuffmanTree tree(byteArray);
    Bitset<unsigned char> codebook[HuffmanTree::N], result;
    tree.populateCodebook(codebook);
    tree.writeTree(result);
    for(int i = 0; i < byteArray.getSize(); ++i)
        result.appendBitset(codebook[byteArray[i]]);
    return ExtraBitsCompress(result);
}

```

Write the codebook by preorder traversal of the tree—for a nonleaf write 0, and recurse on both children, and for a leaf write 1 and the 8-bit symbol. E.g., for the tree in Figure 15.1 have 01a01b01c1d, with the characters replaced by the corresponding bit sequences. `append` is a member function of `Node`.

```

void append(Bitset<unsigned char>& result)
{
    result.append(!left); // 0 for nonleaf, 1 for leaf
    if(left)
    {
        left->append(result);
        right->append(result);
    }
    else result.appendValue(letter, W);
}
void writeTree(Bitset<unsigned char>& result){root->append(result);}

```

Decoding reads the tree and uses it to decode each symbol by using the code to walk to leaves:

```

Vector<unsigned char> HuffmanUncompress(Vector<unsigned char> const& byteArray)
{
    BitStream text(ExtraBitsUncompress(byteArray));
    HuffmanTree tree(text);
    return tree.decode(text);
}

Node* readHuffmanTree(BitStream& text)
{
    Node *left = 0, *right = 0;
    unsigned char letter;
    if(text.readBit()) letter = text.readValue(W); // got to a leaf
    else
        // process internal nodes recursively
        left = readHuffmanTree(text);
        right = readHuffmanTree(text);
    }
    return new(f.allocate())Node(0, left, right, letter);
}

HuffmanTree(BitStream& text){root = readHuffmanTree(text);}

Vector<unsigned char> decode(BitStream& text)
// wrong bits will give wrong result, but not a crash
{
    Vector<unsigned char> result;
    for(Node* current = root;;
        current = text.readBit() ? current->right : current->left)
    {
        if (!current->left)
        {
            result.append(current->letter);
            current = root;
        }
        if (!text.bitsLeft()) break;
    }
    return result;
}

```

For a text of length n and alphabet A with $n > |A|$, need $O(n \lg(|A|))$ time to encode and $O(n)$ to decode. $E[\text{the code length}] - H(X) < 1$ (Salomon & Motta 2010). So for a binary alphabet have no compression, and, for a model with k joined symbols, $E[\text{the code length}] - H(X) < 1/k$. E.g., for 8-bit symbols, bit redundancy $\leq 12.5\%$. For small alphabets use supersymbols for efficiency. For a very large file, can divide it into blocks for convenience, and compress each independently with minimal efficiency loss. Huffman works with a general alphabet when using a hash table to hold the codebook.

15.8 Dictionary Compression

The idea is to encode a word by its position in the list of all seen words. **LZW** is the simplest dictionary method. Encoding and decoding start with a dictionary containing all single bytes and maintain the same dictionary of size n . The dictionary of the encoder before/after writing word j matches the dictionary of the decoder respectively before/after reading word $j + 1$.

Encoding:

- 1. Initialize the dictionary with all single-symbol words**

2. The current word $w = \text{blank}$
3. Until EOF
4. Append the read byte to w
5. If $w \notin \text{dictionary}$
 - 6. Output the index of w without the last byte
 - 7. Add w to the dictionary if have space
 - 8. Set w to the read byte
9. Output the index of w

The word indices are encoded in binary using $\lceil \lg(n) \rceil$ bits, which is the number of bits needed to read any previous word. E.g., to encode *abhababa*:

Read Letter	Current Word	In the Dictionary?	Next Index	Output
<i>a</i>	<i>a</i>	yes	256	none
<i>b</i>	<i>ab</i>	no	256	97(<i>a</i>), 8 bits
<i>h</i>	<i>bh</i>	no	257	98(<i>b</i>), 9 bits
<i>a</i>	<i>ha</i>	no	258	104(<i>h</i>), 9 bits
<i>b</i>	<i>ab</i>	yes	258	none
<i>a</i>	<i>aba</i>	no	259	256(<i>ab</i>), 9 bits
<i>b</i>	<i>ab</i>	yes	259	none
<i>a</i>	<i>aba</i>	yes	259	none
				259(<i>aba</i>), 9 bits

The dictionary has size $\leq 2^{\max\text{Bits}}$ and is a ternary treap trie for incremental search. A good default value for $\max\text{Bits}$ is 16—don't want a large value because will get poor compression with large number of bits per index. Though implemented here with a bit stream, LZW is online and can be changed to work with disk streams.

```
void LZWCompress(BitStream& in, BitStream& out, int maxBits = 16)
{
    assert(in.bytesLeft());
    byteEncode(maxBits, out); // store as config
    TernaryTreapTrie<int> dictionary;
    TernaryTreapTrie<int>::Handle h;
    int n = 0;
    while (n < (1 << numeric_limits<unsigned char>::digits))
        // initialize with all bytes
        unsigned char letter = n;
        dictionary.insert(&letter, 1, n++);
    }
    Vector<unsigned char> word;
    while (in.bytesLeft())
    {
        unsigned char c = in.readByte();
        word.append(c);
        // if found keep appending
        if (!dictionary.findIncremental(word.getArray(), word.getSize(), h))
            // word without the last byte guaranteed to be in the dictionary
            out.writeValue(*dictionary.find(word.getArray(),
                word.getSize() - 1), lgCeiling(n));
        if (n < twoPower(maxBits)) // add new word if have space
            dictionary.insert(word.getArray(), word.getSize(), n++);
        word = Vector<unsigned char>(1, c); // set to read byte
    }
    out.writeValue(*dictionary.find(word.getArray()), word.getSize(),
        lgCeiling(n));
}
```

$E[\text{the runtime}] = O(n \times \max\text{Bits})$ due to incremental search but can be smaller, depending on the text.

Decoding builds an array dictionary from indices to words. When reading an index not-for-the-first time, it inserts the new word = the word corresponding to the last index + the first character of the word corresponding to the index. The latter \in dictionary unless it = the word added to the dictionary after outputting the last word. This can only happen if it = the last word + its first character. E.g., if *ababa* is decoded when *ab* \in dictionary, *aba* is added to the dictionary and immediately used to decode the *aba* suffix. The first word is read using 8 bits because it's not added to the dictionary, and \forall other word the number of bits of the next possible index = $\min(\maxBits, \lceil \lg(n+1) \rceil)$. For a given word, the number of index bits is the same after/before respectively the encoder/decode adds it. To decode the encoding example output:

Next Index	Last Word	Read Index	Is in the dictionary	Added word
256	none	97(<i>a</i>), 8 bits	yes	none
256	97(<i>a</i>)	98(<i>b</i>), 9 bits	yes	<i>ab</i>
257	98(<i>b</i>)	104(<i>h</i>), 9 bits	yes	<i>bh</i>
258	104(<i>h</i>)	257(<i>ab</i>), 9 bits	yes	<i>ha</i>
259	257(<i>ab</i>)	259(<i>aba</i>), 9 bits	no	<i>aba</i>

```
void LZWUncompress(BitStream& in, BitStream& out)
{
    int maxBits = byteDecode(in), size = twoPower(maxBits), n = 0,
        lastIndex = -1;
    assert(maxBits >= numeric_limits<unsigned char>::digits);
    Vector<Vector<unsigned char>> dictionary(size);
    for(; n < (1 << numeric_limits<unsigned char>::digits); ++n)
        dictionary[n].append(n);
    while(in.bitsLeft())
    {
        int index = in.readValue(lastIndex == -1 ? 8 :
            min(maxBits, lgCeiling(n + 1)));
        if(lastIndex != -1 && n < size)
        {
            Vector<unsigned char> word = dictionary[lastIndex];
            word.append((index == n ? word : dictionary[index])[0]);
            dictionary[n++] = word;
        }
        for(int i = 0; i < dictionary[index].getSize(); ++i)
            out.writeByte(dictionary[index][i]);
        lastIndex = index;
    }
}
```

The runtime is $O(n + 2^{\maxBits})$.

15.9 Run-length Encoding

Count repeating bytes, and output bytes and counts, which compresses when many bytes are repeated and is efficient because of working with bytes. One way to set this up is to reserve **escape symbols** $a = 255$ and $b = 254$, and \forall character c output:

- cak if the count of the remaining same symbols $k > 1$, or $c = a$ and $k = 1$
- ab if $c = a$, and $k = 0$
- c otherwise

Counts ≤ 253 fit into a byte and don't collide with the escape symbols. E.g., the byte sequence 0,0,0,0,127,127,255,254,255,255 \rightarrow 0,255,4,127,127,255,254,254,255,1.

```
enum {RLE_E1 = (1 << numeric_limits<unsigned char>::digits) - 1,
      RLE_E2 = RLE_E1 - 1};
Vector<unsigned char> RLECompress(Vector<unsigned char> const& byteArray)
{
    Vector<unsigned char> result;
    for(int i = 0; i < byteArray.getSize(); )
    {
        unsigned char byte = byteArray[i++];
        if(byte == RLE_E1)
        {
            result.push_back(RLE_E1);
            result.push_back(0);
        }
        else if(byte == RLE_E2)
        {
            result.push_back(RLE_E2);
            result.push_back(1);
        }
        else
        {
            result.push_back(byte);
            int count = 1;
            while(i < byteArray.getSize() && byteArray[i] == byte)
            {
                i++;
                count++;
            }
            if(count > 1)
            {
                result.push_back(count);
                result.push_back(byte);
            }
        }
    }
}
```

```

        result.append(byte);
        int count = 0;
        while(count < RLE_E2 - 1 && i + count < byteArray.getSize() &&
              byteArray[i + count] == byte) ++count;
        if(count > 1 || (byte == RLE_E1 && count == 1))
        {
            result.append(RLE_E1);
            result.append(count);
            i += count;
        }
        else if(byte == RLE_E1) result.append(RLE_E2);
    }
    return result;
}

```

Only compression is possible if *a* is never present. After reading *e*, the next byte is a count unless it's *a* or *b*. If *a* the next byte is a count, and, if *b*, decode a single *a*.

```

Vector<unsigned char> RLEUncompress(Vector<unsigned char> const& byteArray)
{
    Vector<unsigned char> result;
    for(int i = 0; i < byteArray.getSize();)
    {
        unsigned char byte = byteArray[i++];
        if(byte == RLE_E1 && byteArray[i] != RLE_E1)
        {
            unsigned char count = byteArray[i++];
            if(count == RLE_E2) count = 1;
            else byte = result.lastItem(); //need temp if vector reallocates
            while(count--) result.append(byte);
        }
        else result.append(byte);
    }
    return result;
}

```

The runtime for both is $O(n)$.

15.10 Move-to-front Transform

MTFT transforms frequent numbers into small ranks, leading to compression during coding. E.g., repeating symbols lead to sequences of 0's. To encode:

1. Put all symbols \in alphabet into a list in known order, e.g., by increasing numeric values.
2. \forall input symbol
3. Output its rank
4. Move it to the front of the list

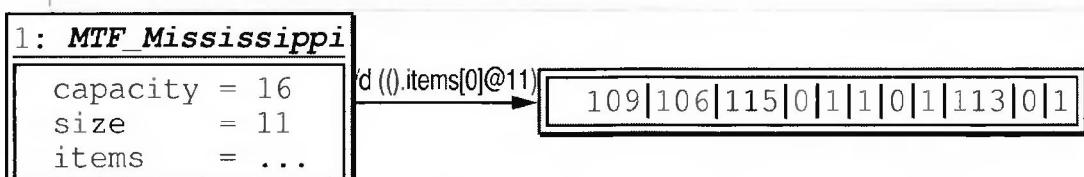


Figure 15.3: MTFT of "mississippi"

To decode:

1. Initialize a list of ranks with identity ranks
2. \forall rank
3. Output the symbol in that position
4. Move the rank to the front

```

Vector<unsigned char> MoveToFrontTransform(bool compress,
                                             Vector<unsigned char> const& byteArray)
{
    unsigned char list[1 << numeric_limits<unsigned char>::digits], j, letter;
    for(int i = 0; i < sizeof(list); ++i) list[i] = i;
    Vector<unsigned char> resultArray;
}

```

```

for(int i = 0; i < byteArray.getSize(); ++i)
{
    if(compress)
        //find and output rank
        j = 0;
        letter = byteArray[i];
        while(list[j] != letter) ++j;
        resultArray.append(j);
    }
    else
        //rank to byte
        j = byteArray[i];
        letter = list[j];
        resultArray.append(letter);
    //move list back to make space for front item
    for(; j > 0; --j) list[j] = list[j - 1];
    list[0] = letter;
}
return resultArray;
}

```

The runtime is $O(n|A|)$ for an alphabet A for encoding and decoding, but with small constant factors and cache-friendliness.

15.11 Burrows-Wheeler Transform

BWT of a string consists of the last column of the character matrix, formed by the sorted list of the string's

rotations, and the index of the original rotation. E.g., *click* has rotations: $\begin{bmatrix} ckcli \\ click \\ ickcl \\ kclic \\ lickc \end{bmatrix}$; the original rotation is in row 1, and the last column is *iklcc*. BWT creates easier-to-compress output because every section tends to use few distinct characters, which, e.g., works well with MTFT.

The suffix array algorithm computes BWT using the BWT rank functor to sort the rotations. E.g., for *click* get BWT array [3, 0, 2, 4, 1]. Let t be the last column, corresponding to the transformed string. The index of the original rotation is that of the rotation with the suffix at index 0. Because each row is a rotation, its rightmost character precedes the leftmost, so $t[i] = \text{string}[(\text{BWT}[i] - 1) \% n]$. The runtime is $O(n \lg(n))$.

```

Vector<unsigned char> BurrowsWheelerTransform(
    Vector<unsigned char> const& byteArray)
{
    int original = 0, size = byteArray.getSize();
    Vector<int> BTWArray = suffixArray<BTWRank>(byteArray.getArray(), size);
    Vector<unsigned char> result;
    for(int i = 0; i < size; ++i)
    {
        int suffixIndex = BTWArray[i];
        if(suffixIndex == 0)
            //found the original string
            original = i;
            suffixIndex = size; //avoid the + size in next step
        }
        result.append(byteArray[suffixIndex - 1]);
    } //assume that 4 bytes is enough
    Vector<unsigned char> code = ReinterpretEncode(original, 4);
    for(int i = 0; i < code.getSize(); ++i) result.append(code[i]);
    return result;
}

```

The reverse transformation is $O(n)$. It uses character counts to create arrays:

- **Ranks**—how many same characters in t precede the character at a given position. The left column ranks = the right column ranks because \forall rows x_c and y_c , ending with same character c , x_c

and cy are their right rotations, and if $xc < yc$ then $cx < cy$. E.g., for $iklcc$ with original position 1, the ranks are 00001.

- **First positions**—the first positions of characters in the left column, computable because these are sorted and $\in [0, 255]$. First positions for $cikl$ are 0234. If extending the algorithm to work with a different range, extract the sorted list of unique characters from t .

Intuitively, *firstPositions* finds all rotations that begin with c , and *ranks* tells which one corresponds to the right column character because ranks in both columns are the same. Traverse the original rotation in reverse order. $t[\text{the original index}]$ is the last character of the string. \forall other character $c = t[j]$ the previous text character = $t[\text{firstPositions}[c] + \text{ranks}[j]]$ because it's the position of the rotation with c in the first column and the wanted next character in the last. For $iklcc$ the rotation sequence is $1 \rightarrow k, 3 + 0 = 3 \rightarrow c, 0 + 0 = 0 \rightarrow i, 2 + 0 = 2 \rightarrow l, 4 + 0 = 4 \rightarrow c$, the reverse of which is *click*.

```
Vector<unsigned char> BurrowsWheelerReverseTransform(
    Vector<unsigned char> const& byteArray)
{
    enum{M = 1 << numeric_limits<unsigned char>::digits};
    int counts[M], firstPositions[M], textSize = byteArray.getSize() - 4;
    for(int i = 0; i < M; ++i) counts[i] = 0;
    Vector<int> ranks(textSize); //compute ranks
    for(int i = 0; i < textSize; ++i) ranks[i] = counts[byteArray[i]]++;
    firstPositions[0] = 0; //compute first positions
    for(int i = 0; i < M - 1; ++i)
        firstPositions[i + 1] = firstPositions[i] + counts[i];
    Vector<unsigned char> index, result(textSize); //extract original rotation
    for(int i = 0; i < 4; ++i) index.append(byteArray[i + textSize]);
    //construct in reverse order
    for(int i = textSize - 1, ix = ReinterpretDecode(index); i >= 0; --i)
        ix = ranks[ix] + firstPositions[result[i] = byteArray[ix]];
    return result;
}
```

The compression system $\text{output} = \text{Huffman}(\text{RLE}(\text{MTFT}(\text{BWT}(\text{input}))))$, implemented by **BZIP2**, is fast and effective. MTFT exploits symbol grouping of BWT, producing small numbers. RLE compresses runs of 0's, 1's, and 2's, and its $a = 255$ is unlikely to appear in MTFT output.

```
Vector<unsigned char> BWTCompress(Vector<unsigned char> const& byteArray)
{
    return HuffmanCompress(RLECompress(MoveToFrontTransform(true,
        BurrowsWheelerTransform(byteArray))));
}
Vector<unsigned char> BWTUncompress(Vector<unsigned char> const& byteArray)
{
    return BurrowsWheelerReverseTransform(MoveToFrontTransform(false,
        RLEUncompress(HuffmanUncompress(byteArray))));
}
```

File	Size	Huffman	BWT	LZW	7-Zip BZIP2
a.txt	1	3	9	2	37
bible.txt	4047392	2218529	934693	1417732	846185
dickens.txt	31457485	17786401	9046529	13252448	8899726
ecoli.txt	4638690	1159679	1319213	1213574	1250991
moby dick.txt	1191463	667648	410470	495146	371466
pi10mm.txt	10000000	4249754	4381817	4524962	4308898
world192.txt	2473400	1558714	489749	925798	489543

Figure 15.4: Some tests of the presented compressors

BWT is best overall in most cases, particularly text, as predicted. The BZIP2 does better than my implementation, but it's highly optimized.

15.12 Implementation Notes

Only RLE has some creativity in parameter settings. Other implementations are in direct correspondence to textbook ones. The stream functionality is a natural on top of bit set, though the API might benefit from some other functionality such as appending a bit set at a time.

15.13 Comments

Information theory is full of appealing definitions, but even its inventor acknowledged that using them doesn't make the problems any easier.

\exists many other static codes. Some of the most important ones:

- **Turnstall code** uses binary code and, when alphabet size \neq a power of two, assigns the remaining values to pairs of symbols. E.g., with alphabet $a, b, c, ab \rightarrow 3$. But it's hard to decide which pairs are most likely.
- **Fibonacci code** (Sayood 2002) is slightly more efficient than gamma for values ≥ 8 but is only 1 bit better than byte code for values ≥ 16 up to some value where the latter takes over. It's more complicated and doesn't seem to have a use case.
- Some codes are optimal for specific data distributions, e.g., **Golomb** for geometric, and **Vigna** for power law. But there are more complex and do poorly for different distributions.

Arithmetic codes are an alternative to Huffman coding. They are complicated to implement and much slower but tiny bit better compressors, particularly for smaller alphabets. They also used to be patented (now expired). Huffman codes are a more practical option. See Moffat & Turpin (2002) for details if curious.

The run-length encoding can use a different number of escape symbols or different values of them. Use two escape symbols with largest possible values seems most efficient based on my experiments with BTW-based compression.

Some data structures such as **splay tree** can make MTFT optimally asymptotically efficient, but are complicated and have high constant factors. MTFT is usually far from being the compression pipeline's performance bottleneck, so no point using those.

For BWT, \exists complicated $O(n)$ algorithms, and new ones are still being proposed. BZIP2 is the best general compressor. If efficiency is critical, prefer faster dictionary methods such as **GZIP**. Slightly more effective but much slower are variants of **PPM** (Salomon & Motta 2010).

Decompression speed is usually more important than compression speed because a file is often compressed once and decompressed many times, e.g., when it's uploaded to a server and downloaded many times. For applications such as file backups, compression speed is more important because most files are never accessed.

Compression is **lossy** when decompressed data closely resembles the original. This is useful for media and implemented by standards such as MP3 and MPEG. The idea is to reduce quality, but so that humans don't notice any difference, and apply lossless methods. Quality reduction is media-dependent, but often use a Fourier transform, and discard higher frequencies. See Salomon & Motta (2010) for details.

15.14 Projects

- Consider a simpler RLE encoding that uses only a : output cak if $k > 1$, or $c = a$, otherwise c . Study the effect on performance of BWT compression.
- Implement **LZ77** compression used by GZIP. Compare its performance and efficiency to LZW. Given that most current compression software uses LZ77 variants, is LZW useful at all despite its implementation simplicity?
- Research and implement a PPM compressor, and compare it to BZIP2.

15.15 References

- Adjerooh, D., Bell, T. C., & Mukherjee, A. (2008). *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer.
- Moffat, A., & Turpin, A. (2002). *Compression and Coding Algorithms*. Springer.
- Salomon, D. & Motta, G. (2010). *Handbook of Data Compression*. Springer.
- Sayood, K. (2002). *Lossless Compression Handbook*. Academic Press.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-Wesley.

16 Combinatorial Optimization

"A donkey unable to chose between two similar piles of hay stays hungry" – Russian folk saying

"Calculate all possibilities only once" – Alexander Kotov, regarding chess moves

16.1 Introduction

This chapter is mostly about general global combinatorial optimization techniques. Numerical optimization is discussed in its own chapter later in the book. The emphasis is on the main metaheuristic methods. Constraint processing is also briefly introduced.

Want to find the best choice among possibly uncountably many, using some quality score. Solution algorithms look for the minimum, because the maximum is the minimum over the negated values. Conceptually, the value of a choice is its **economic utility**, which is happiness derived from the results of making the choice. E.g., empirical utility for human happiness as a function of wealth is believed to be $O(\lg(\text{wealth}))$. But usually utilities are mathematically inconvenient or unknown, so use the identity function utility.

Checking each choice is too inefficient, but for some problems smart brute force is feasible for moderately large instances. Combinatorial generation algorithms (see the "Miscellaneous Algorithms" chapter) are useful here.

16.2 Complexity Theory

This section gives only a brief overview of the most important ideas. See Sipser (2013) for more details.

All problems have intrinsic difficulty. The most important difficulty classes:

- **Easy/tractable**—can solve in polynomial time $O(n^k)$, for some constant k
- **Hard/intractable**—need more than polynomial resources
- **Undecidable**—can't solve by any algorithm

Can't solve tasks for which \exists information such as predicting the future, but \exists unsolvable tasks with complete information. E.g., the **halting problem**: \forall program X , does X go into an infinite loop for some input? If \exists program A (source of X , input) that computes *terminates/loops*, make program B (source of X) that calls A (source of X , source of X), and if the answer is *terminates*, enters an infinite loop. Call B (source of B). If A (source of B , source of B) called by B returned *terminates*, B (source of B) goes into a loop, and if *loops*, B (source of B) terminates, contradicting A 's answer in both cases. The main characteristic of an undecidable problem is going into a loop and not knowing when to exit. Technically \exists undecidability on finite-memory computers because any program whose memory state repeats is in an infinite loop, but this makes no practical difference.

A problem $\in \text{NP}$ if it has a *yes/no* answer and \exists a polynomial-time algorithm that can verify its correctness. A computation is **easy** with respect to class C if performing it takes less resources than solving the most difficult problem $\in C$. E.g., a polynomial-time algorithm is easy with respect to NP. A problem X is **C -hard** if can use it as a black-box solver \forall problem $A \in C$, after reducing an instance of A to an instance of X by an easy-with-respect-to- C algorithm. A problem is **C -complete** if it's C -hard and $\in C$.

Many NP-complete problems ask for minimum-cost solutions. They are equivalent to **decision problems** that ask if \exists a solution of a specific cost because using exponential search (see the "Numerical Algorithms—Working with Functions" chapter) on cost with a black-box decision solver solves optimization. Most useful optimization problems are NP-hard.

Problems $\in \text{PSPACE}$ need polynomial space and are harder than NP-complete, e.g., multiplayer games. Think about how to prove checkmate in n moves in chess, no matter what the opponent does.

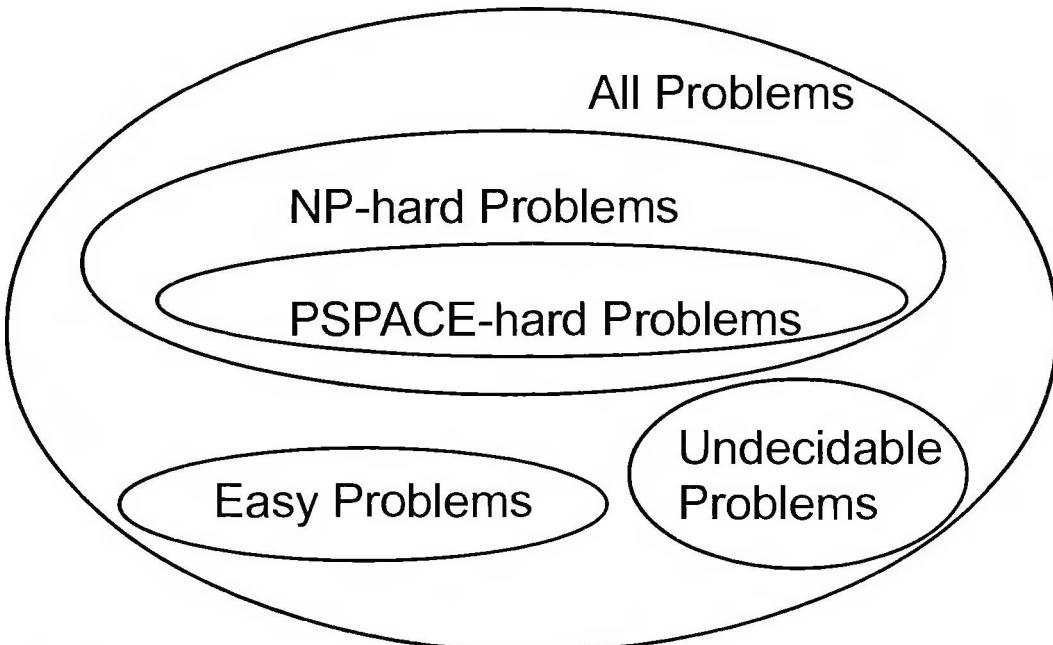


Figure 16.1: Relations among various complexity classes

An algorithm trying to solve a hard problem must give up one of:

- Worst case polynomial time—try to be fast on average
- Finding the exact answer—get a close-enough approximation
- Working \forall case—only solve problems with a specific structure that allows efficient solution

16.3 Typical Hard Problems

The following problems are simple to describe and common in the optimization literature. They cover most natural type of solution representation including permutation, subset, and partition. Combination problems are omitted though. In all cases use a simple random generator to create problem instances of specific size. The focus is on testing and comparing the solution algorithms, so pay no attention to the difficulty of the resulting problems or picking the best possible generators. Every presented problem has extensive published problem-set collections, often with at least some optimal solutions. Also every problem has one or several best-known algorithms. For these consult the literature—here only show how general algorithms can be applied to common problems with minor effort.

The **traveling salesman problem (TSP)** gives a collection of points and distances between every pair. Want a permutation such that visiting the points in its order minimizes \sum traveled distance. Can include or exclude coming back from the last point to the first. The distances are usually Euclidean.

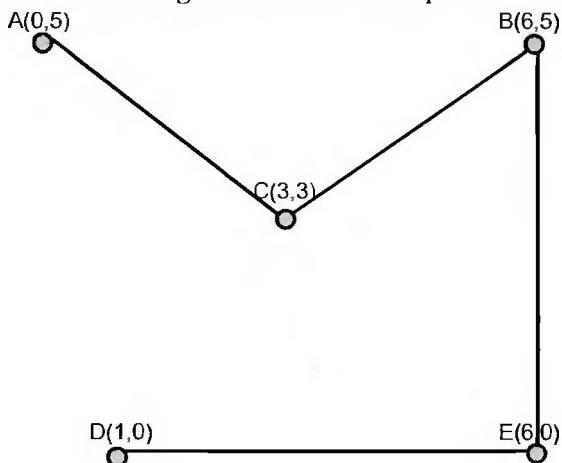


Figure 16.2: Solution to a 5-point Euclidean TSP without return

TSP is the benchmark for new optimization algorithms, and much is known about it (Applegate et al. 2007). E.g.:

- For random points $\in [0, 1]^2$, $E[\text{the best tour cost}] = \sqrt{\frac{n}{2}}$
- For Euclidean distance, the best solution visits convex hull vertices in convex hull order (see the

“Computation Geometry” chapter)

To generate a random instance for testing, can use the below code. The solvers expect any problem data structure to be able to evaluate its solution representation. For permutation problems also expect the ability to evaluate a single step, with indication whether it's a return to the starting position. `EuclideanDistance` is discussed in the “Computational Geometry” chapter.

```
struct TSPRandomInstance
{
    Vector<Vector<double>> points;
    TSPRandomInstance(int n)
    {
        for(int i = 0; i < n; ++i)
        {
            Vector<double> point;
            for(int j = 0; j < 2; ++j) point.append(GlobalRNG().uniform01());
            points.append(point);
        }
    }
    double evalStep(int from, int to, bool isReturn = false) const
    {//evaluate single step
        if(isReturn) return 0;//no cost to return
        assert(from >= 0 && from < points.getSize() && to >= 0 &&
               to < points.getSize());
        EuclideanDistance<Vector<double>>::Distance d;
        return d(points[from], points[to]);
    }
    double operator()(Vector<int> const& permutation) const
    {//evaluate complete solution
        assert(permutation.getSize() == points.getSize());
        double result = 0;
        for(int i = 1; i < points.getSize(); ++i)
            result += evalStep(permutation[i - 1], permutation[i]);
        return result;
    }
};
```

The **knapsack problem** gives a maximum weight and a set of items with profits and weights. Want a subset of items that maximizes $\sum \text{profits}$ such that $\sum \text{weights} \leq \text{the capacity}$.

A simple generator takes both profits and weights as `uniform01` and sets capacity to $\sum \text{weights}/2$.

```
class KnapsackRandomInstance
{
    Vector<double> profits, weights;
    double capacity;
public:
    double getProfit(int i) const
    {
        assert(i >= 0 && i < profits.getSize());
        return profits[i];
    }
    double getWeight(int i) const
    {
        assert(i >= 0 && i < weights.getSize());
        return weights[i];
    }
    int getSize() const {return profits.getSize();}
    double getCapacity() const {return capacity;}
    KnapsackRandomInstance(int n): capacity(0)
    {//uniformly random weights and profits and capacity half of weights
        for(int i = 0; i < n; ++i)
        {
            profits.append(GlobalRNG().uniform01());
            weights.append(GlobalRNG().uniform01());
            capacity += weights[i];
        }
    }
```

```

        capacity /= 2;
    }
double operator() (Vector<bool> const& subset) const
{
    assert(subset.getSize() == getSize());
    double totalProfit = 0, totalWeight = 0;
    for(int i = 0; i < subset.getSize(); ++i) if(subset[i])
    {
        totalProfit += getProfit(i);
        totalWeight += getWeight(i);
    }//infeasible marked by infinite score
    return totalWeight <= capacity ? -totalProfit :
        numeric_limits<double>::infinity();
}
};

```

The **satisfiability problem (SAT)** gives n Boolean variables and a formula that is an *and* of many clauses, each of which is an *or* of several variables. Want a subset of true variables, such that when the rest are false the formula is true, or to declare the problem unsatisfiable. The optimization problem is called **maximum satisfiability**, and want to satisfy as many clauses as possible.

A simple generator for the common option of 3 variables per clause is to take n variables and m clauses, with $m = 10n$ for reasonable difficulty. Each clause consists of randomly selected and flipped variables.

```

class Satisfiability3RandomInstance
{
    Vector<Vector<int>> clauses;
    int n;
public:
    Satisfiability3RandomInstance(int theN, int nc): n(theN),
        clauses(nc, Vector<int>(3))//3-sat
{//use n clauses for n variables
    for(int i = 0; i < nc; ++i)
    {
        Vector<int>& clause = clauses[i];
        for(int j = 0; j < clause.getSize(); ++j)
            //random variables and signs
            int variable = GlobalRNG().mod(n);
            clause[j] = variable * GlobalRNG().sign();
    }
}
int getSize() constreturn n;
int getClauseSize() constreturn clauses.getSize();
Vector<int> const& getClause(int i) const
{
    assert(i >= 0 && i < getClauseSize());
    return clauses[i];
}
};

```

The **bin packing problem** gives a set of items with weights and ∞ -many bins of capacity $>$ the max weight. Want to partition the items, so that \forall part \sum weights \leq the bin capacity, and the number of the used bins is minimal.

A simple generator uses random weights and capacity = 5.

```

class BinPackingRandomInstance
{
    Vector<double> weights;
    double binSize;
public:
    double getWeight(int i) const
    {
        assert(i >= 0 && i < weights.getSize());
        return weights[i];
    }
};

```

```

double getBinSize() const return binSize;
BinPackingRandomInstance(int n): binSize(5) //uniformly random weights
    for(int i = 0; i < n; ++i) weights.append(GlobalRNG().uniform01());
}

```

The **integer programming problem** is the linear programming problem (see the “Numerical Optimization” chapter), where every solution variable is discrete. **Rounding down** the corresponding linear program solution variables gives a suboptimal solution because an optimal one rounds down only some of them, but otherwise is a good starting solution for further improvement.

Can model many problems as integer programming, and a good solution approach is expressing a problem in terms of integer programming, for which well-engineered solvers handle problems of small to medium size. Many good commercial and open-source such solvers are currently available. This isn't discussed further.

These problems are NP-complete and involve finding some combinatorial object, including a permutation, a subset, or a partition, that gives the best solution.

16.4 Approximation Algorithms

Sometimes can get in polynomial time a solution that is suboptimal by at most a factor of C . Such approximation algorithms are usually greedy heuristics. E.g., for bin packing the **next fit** strategy is 2-approximate (Vazirani 2004). It works online:

- 1. If the next item doesn't fit in the current bin, close the bin, and open another one**
- 2. Put the item into the current bin**

Approximation algorithms are useful only when fast and simple. Unfortunately:

- Need to develop one, and prove it C -approximate \forall problem
- For many problems provably \nexists an approximation algorithm with a good enough C
- Many approximation algorithms have high-order polynomial runtimes
- Solutions found by heuristics are usually better

A better approximation algorithm for bin packing is **first-fit decreasing**. It sorts the items by weight and packs larger first. It's also greedy and packs fairly well:

```

template<typename PROBLEM>
Vector<Vector<int>> firstFitDecreasing(PROBLEM const p, int n)
{
    assert(n > 0);
    Vector<pair<double, int>> items(n);
    double binSize = p.getBinSize();
    for(int i = 0; i < n; ++i)
    {
        items[i].first = p.getWeight(i);
        assert(items[i].first <= binSize); //else bad problem
        items[i].second = i;
    }
    quickSort(items.getArray(), 0, n - 1,
              PairFirstComparator<double, int, ReverseComparator<double>>());
    Vector<Vector<int>> result(1); //start with one empty bin
    Vector<double> sums(1, 0);
    for(int i = 0; i < n; ++i)
    {
        bool fit = false;
        for(int bin = 0; bin < result.getSize(); ++bin)
            if(sums[bin] + items[i].first <= binSize)
            {
                fit = true;
                result[bin].append(items[i].second);
                sums[bin] += items[i].first;
                break;
            }
        else if(bin == result.getSize() - 1)
            //doesn't fit in last bin, add a new one
            result.append(Vector<int>());
            sums.append(0);
    }
}

```

```

        }
    }
    return result;
}
}

```

For mostly theoretical research-level summaries of the field see Gonzalez (2018).

16.5 Greedy Construction Heuristics

For many hard problems can get a good solution by constructing it using a greedy criteria. Some of these are approximation algorithms. E.g., for knapsack, picking items by decreasing profit/cost until the next such item exceeds capacity usually gives good results, though it's not an approximation algorithm.

```

template<typename PROBLEM>
Vector<pair<double, int>> getKnapsackRatios(PROBLEM const& p)
{
    int n = p.getSize();
    double capacity = p.getCapacity();
    Vector<pair<double, int>> ratios(n);
    for(int i = 0; i < n; ++i)
        ratios[i] = make_pair(p.getProfit(i)/p.getWeight(i), i);
    quickSort(ratios.getArray(), 0, n - 1,
              PairFirstComparator<double, int, ReverseComparator<double>>());
    return ratios;
}

template<typename PROBLEM>
Vector<bool> approximateKnapsackGreedy(PROBLEM const& p)
{
    double remainingCapacity = p.getCapacity();
    Vector<pair<double, int>> ratios = getKnapsackRatios(p);
    Vector<bool> solution(ratios.getSize(), false);
    for(int i = 0; i < ratios.getSize(); ++i)
    {
        int j = ratios[i].second;
        if(remainingCapacity > p.getWeight(j))
        {
            remainingCapacity -= p.getWeight(j);
            solution[j] = true;
        }
    }
    return solution;
}

```

Greedy algorithms are usually scalable to at least medium n and are often the method of choice.

16.6 Branch and Bound

For some problems with incrementally constructable solutions, can compute a **lower bound** on the rest of the solution. E.g., for the TSP, the cost of visiting the remaining cities \geq the cost of their MST + the cost of shortest edge from the last visited city to any unvisited one. B&B starts with a global lower bound ∞ and recursively enumerates all solutions by specifying one component at a time. Lower bounds allow to:

- Try the most promising component next
- Not consider a component further when the cost so far + the lower bound \geq the global lower bound

Choose A					
Next	B	C	D	E	
Cumulative Cost		6	3.61	5.1	7.81
MST	DC + ED	ED + EB	CB + EC	DC + CB	
MST Cost		8.61	10	7.85	7.21
Connect edge	CB	DC	DC	ED	
Connect edge cost		3.61	3.61	3.61	5
Lowerbound	18.21	17.21	16.55	20.02	
Pruned			Pruned		
Choose D					
Next	B	C	E		
Cumulative Cost		12.17	8.7	10.1	
MST	EC	EB	CB	ED	
MST Cost		4.24	5	3.61	5
Connect edge	DC	CB	EC	DC	
Connect edge cost		3.61	3.61	4.24	3.61
Lowerbound	20.02	17.31	17.95	15.82	15.82
Pruned			Pruned		
Choose C					
Next	B	D	E		
Cumulative Cost		7.21	7.21	7.85	
MST	ED	EB	DB	ED	
MST Cost		5	5	7.07	5
Connect edge	DC	CB	DC	DC	
Connect edge cost		3.61	3.61	3.61	3.61
Lowerbound	18.52	18.52	18.52	18.52	18.52
Pruned			Pruned		
Choose C					
Next	B	E			
Cumulative Cost		12.31	12.95		
Connect edge	EB	EB			
Connect edge cost		5	5		
Lowerbound	17.31	17.95			
Pruned					
Choose B					
Next	D	E			
Cumulative Cost		14.28	12.21		
MST	ED	ED			
MST Cost		5	5		
Connect edge	DC	CB	DC	DC	
Connect edge cost		3.61	3.61	3.61	3.61
Lowerbound	19.28	17.21	19.28	17.21	17.21
Pruned			Pruned		
Choose D					
Next	B	E			
Cumulative Cost		14.28	12.21		
MST	ED	EB	EB	ED	
MST Cost		5	5	5	5
Connect edge	DC	CB	DC	DC	
Connect edge cost		3.61	3.61	3.61	3.61
Lowerbound	19.28	17.21	19.28	17.21	17.21
Pruned			Pruned		
Choose B					
Next	E				
Cumulative Cost		17.31			
Pruned					
Choose E					
Next	D				
Cumulative Cost		17.21			
Pruned					

Figure 16.3: B&B calculations for the TSP in Figure 16.1. First it goes ADCB, then eventually finds optimal ACBE. Equally optimal ACDE is pruned.

With tight enough lower bounds, it prunes many suboptimal solutions early and finds the optimum relatively quickly. B&B is an **anytime algorithm**, i.e., stopping it before termination gives the best answer found so far, assuming it reached a single complete solution. To make the implementation generic, the problem data structure calculates lower bounds. One general rule in this and other such implementations is that the problem object must not have any modifiable state, so pass any state to the algorithm. The below implementation also has recursion removed. The basic unit of action is to undo a previous move, do the next move, and to create a new state by generating next moves.

```
template<typename PROBLEM> pair<typename PROBLEM::X, bool> branchAndBound(
    PROBLEM const& p, int maxLowerBounds = 1000000)
//require one complete solution before stopping
{
    typename PROBLEM::X best; //assumed to be default-constructable
    double bestScore = numeric_limits<double>::infinity();
    typename PROBLEM::INCREMENTAL_STATE is = p.getInitialState();
    bool foundCompleteSolution = false; //will guarantee complete solution
    struct BBState
    {
        Vector<pair<double, typename PROBLEM::Move>> moves;
        int prevMove;
    };
    Stack<BBState> states;
    bool start = true;
    while(!states.isEmpty() || start)
    {
        bool goNextLevel = start, isSolutionComplete = false;
        if(start) start = false;
        else //act on current level
        {
            BBState& level = states.getTop();
            int i = level.prevMove;
            //undo prev move if any
            if(i > 0)
                level.moves.pop_back();
            else
                start = true;
            if(isSolutionComplete)
                best = level.state;
            else
                for(int j = 0; j < level.moves.size(); ++j)
                {
                    BBState copy = level;
                    copy.prevMove = i;
                    copy.state = p.getNewState(is, level.moves[j].second);
                    if(copy.state.isComplete())
                        isSolutionComplete = true;
                    else
                        states.push(copy);
                }
        }
    }
}
```

```

    if(i != -1) p.undoMove(is, level.moves[i].second);
    ++i; //check next move if any
    if(i < level.moves.getSize() && level.moves[i].first < bestScore)
        //if not out of moves and not pruned do next move
        p.move(is, level.moves[i].second);
        if(p.isSolutionComplete(is))
            {//update best
                double score = p.evaluate(is);
                if(score < bestScore)
                {
                    best = p.extractSolution(is);
                    bestScore = score;
                } //update flags
                isSolutionComplete = true;
                foundCompleteSolution = true;
            }
            else goNextLevel = true;
        }
    }
if(goNextLevel && (!foundCompleteSolution || maxLowerBounds > 0))
    //setup next level
    BBState levelNext = {p.generateMoves(is), -1};
    int m = levelNext.moves.getSize();
    assert(m > 0); //else bad problem
    maxLowerBounds -= m;
    quickSort(levelNext.moves.getArray(), 0, m - 1,
              PairFirstComparator<double, typename PROBLEM::Move>());
    states.push(levelNext);
    //for a complete solution first undo current move as generate no more
    else if (!isSolutionComplete) states.pop(); //come back to prev level
}
return make_pair(best, maxLowerBounds > 0);
}

```

The runtime strongly depends on the quality of the lower bounds and can be exponential in the worst case.

The following code allows using B&B for TSP. But it's not scalable beyond $n = 100$ or so because the move list calculation takes $O(n^3)$ time due to the expensive MST lower bound calculation.

```

template<typename PROBLEM> double findMSTCost(PROBLEM const& instance,
    Vector<int> const& remPoints)
{
    int n = remPoints.getSize();
    if(n <= 1) return 0;
    GraphAA<double> g(n);
    for(int i = 0; i < n; ++i)
        for(int j = i + 1; j < n; ++j) g.addUndirectedEdge(i, j,
            instance.evalStep(remPoints[i], remPoints[j]));
    assert(validateGraph(g));
    Vector<int> parents = MST(g);
    double sum = 0;
    for(int i = 0; i < parents.getSize(); ++i)
    {
        int parent = parents[i];
        if(parent != -1)
            sum += instance.evalStep(remPoints[i], remPoints[parent]);
    }
    return sum;
}
template<typename PROBLEM> class BranchAndBoundPermutation
{
    PROBLEM const& p;
    int const n;
public:

```

```

typedef Vector<int> INCREMENTAL_STATE;
INCREMENTAL_STATE getInitialState() const return INCREMENTAL_STATE();
typedef Vector<int> X;
BranchAndBoundPermutation(int theN, PROBLEM const& theProblem): n(theN),
    p(theProblem){}
typedef int Move;
bool isSolutionComplete(INCREMENTAL_STATE const& permutation) const
    {return permutation.getSize() == n;}
double evaluate(INCREMENTAL_STATE const& permutation) const
{
    assert(isSolutionComplete(permutation));
    return p(permutation);
}
X extractSolution(INCREMENTAL_STATE const& permutation) const
{
    assert(isSolutionComplete(permutation));
    return permutation;
}
Vector<pair<double, Move>> generateMoves(
    INCREMENTAL_STATE const& permutation) const
{
    double sumNow = 0;
    Vector<bool> isIncluded(n, false);
    for(int i = 0; i < permutation.getSize(); ++i)
    {
        isIncluded[permutation[i]] = true;
        if(i > 0) sumNow += p.evalStep(permutation[i - 1], permutation[i]);
    }
    Vector<pair<double, Move>> result;
    for(int i = 0; i < n; ++i)
        if(!isIncluded[i])
        {
            Vector<int> remainder;
            for(int j = 0; j < n; ++j) if(j != i && !isIncluded[j])
                remainder.append(j);
            double lb = sumNow + (permutation.getSize() > 0 ?
                p.evalStep(permutation.lastItem(), i) : 0) +
                findMSTCost(p, remainder);
            result.append(pair<double, Move>(lb, i));
        }
    return result;
}
void move(INCREMENTAL_STATE& permutation, Move m) const
    {permutation.append(m);}
void undoMove(INCREMENTAL_STATE& permutation, Move m) const
    {permutation.removeLast();}
};

template<typename INSTANCE> Vector<int> solveTSPBranchAndBound(
    INSTANCE const& instance, int maxLowerBounds)
{
    return branchAndBound(BranchAndBoundPermutation<INSTANCE>(
        instance.points.getSize(), instance), maxLowerBounds).first;
}

```

For B&B for knapsack a simple lower bound mechanism is an extension of the greedy algorithm using fractional items to fill the remaining capacity with the best profit/cost items. When the greedy algorithm fails to pack an item due to excessive weight, it moves on to the remaining items to pack as many as possible, so get a strict improvement. But for a lower bound assume that can take a fraction of the skipped item in the amount of the remaining capacity. This is the best-case scenario and so is a valid lower bound.

```

template<typename PROBLEM> class BranchAndBoundKnapsack
{
    PROBLEM const& p; //order 1
    int const n; //order 2

```

```

Vector<pair<double, int>> const ratios;
public:
    struct INCREMENTAL_STATE
    {
        Vector<bool> subset;
        int current;
        double currentTotalProfit, currentTotalWeight;
    };
    INCREMENTAL_STATE getInitialState() const
    {
        INCREMENTAL_STATE is = {Vector<bool>(n, false), 0, 0, 0};
        return is;
    }
    typedef Vector<bool> X;
    BranchAndBoundKnapsack(PROBLEM const& theProblem): p(theProblem),
        n(p.getSize()), ratios(getKnapsackRatios(p)) {}
    typedef pair<int, bool> Move;
    bool isSolutionComplete(INCREMENTAL_STATE const& is) const
    {
        return is.current == n;
    }
    double evaluate(INCREMENTAL_STATE const& is) const
    {
        assert(isSolutionComplete(is));
        return p(is.subset);
    }
    X extractSolution(INCREMENTAL_STATE const& is) const
    {
        assert(isSolutionComplete(is));
        return is.subset;
    }
    Vector<pair<double, Move>> generateMoves(INCREMENTAL_STATE const& is) const
    {
        Vector<pair<double, Move>> result;
        for(int i = 0; i < 2; ++i)
        {
            double lb = is.currentTotalProfit,
                   remainingCapacity = p.getCapacity() - is.currentTotalWeight;
            int j = ratios[is.current].second;
            if(i) //current move
            {
                remainingCapacity -= p.getWeight(j);
                lb += p.getProfit(j);
            }
            if(remainingCapacity >= 0) //remaining moves if any
            {
                for(int k = is.current + 1; k < n; ++k)
                {
                    int jj = ratios[k].second;
                    if(remainingCapacity >= p.getWeight(jj))
                    {
                        remainingCapacity -= p.getWeight(jj);
                        lb += p.getProfit(jj);
                    }
                    else
                        //reached non-fitting item, use fractional
                        lb += remainingCapacity * ratios[k].first;
                        break;
                }
            }
            result.append(pair<double, Move>(-lb, make_pair(j, i)));
        }
    }
    return result;
}

```

```

void move(INCREMENTAL_STATE& is, Move const& m) const
{
    if (m.second)
    {
        is.subset[m.first] = m.second;
        is.currentTotalProfit += p.getProfit(m.first);
        is.currentTotalWeight += p.getWeight(m.first);
    }
    ++is.current;
}
void undoMove(INCREMENTAL_STATE& is, Move const& m) const
{
    if (m.second)
    {
        is.subset[m.first] = false;
        is.currentTotalProfit -= p.getProfit(m.first);
        is.currentTotalWeight -= p.getWeight(m.first);
    }
    --is.current;
}
};

template <typename INSTANCE> Vector<bool> solveKnapsackBranchAndBound(
    INSTANCE const& instance, int maxLowerBounds)
{
    return branchAndBound(BranchAndBoundKnapsack<INSTANCE>(instance),
        maxLowerBounds).first;
}

```

For TSP, B&B doesn't scale due to the $O(n^2)$ memory for storing all the moves. So an interesting version is **realtime A***, which only executes the best-lower-bound move. It's useful as a greedy construction heuristic.

```

template <typename PROBLEM> typename PROBLEM::X realtimeAStar(PROBLEM const& p)
//require one complete solution before stopping
typename PROBLEM::INCREMENTAL_STATE is = p.getInitialState();
do
{
    Vector<pair<double, typename PROBLEM::Move>> moves =
        p.generateMoves(is);
    assert(moves.getSize() > 0); //else bad problem
    int best = argMin(moves.getArray(), moves.getSize(),
        PairFirstComparator<double, typename PROBLEM::Move>());
    p.move(is, moves[best].second);
} while (!p.isSolutionComplete(is));
return p.extractSolution(is);
}

```

For TSP use it as follows:

```

template <typename INSTANCE> Vector<int> solveTSPRTAS(INSTANCE const&
    instance)
{
    return realtimeAStar(BranchAndBoundPermutation<INSTANCE>(
        instance.points.getSize(), instance));
}

```

For knapsack, convince yourself that the move selection and the result are exactly the same as for the greedy algorithm, but with much overhead inefficiency.

16.7 State Space Shortest Path Search with Lower Bounds

A **state space** is a huge implicit graph, with possible problem states as vertices, and actions to transition from one state to another as edges. Each edge cost ≥ 0 . A shortest path from the start state to the goal state forms the solution. A* is a generalization of Dijkstra's algorithm that prioritizes vertices by the known cost to them + the lower bound on the rest of the path to the goal. This way it doesn't consider all vertices in the graph from the start. Lower bounds and goal checks depend only on the current state and the path to it. The **open set** holds the considered nodes with the best known lower bounds, and the **closed set** the fully visited nodes with known path distances.

1. Put the start node into the open set with the lower bound priority
2. Until the open set is empty
 3. Take off the best-lower-bound node
 4. Update it with the path distance, and put it in the closed set
 5. If it's the goal, return the solution path
 6. Put its children in the open set; if already present, update the lower bound if lower
 7. Report no solution

For the open set use an indexed heap for efficient updating of the child lower bounds. For the closed set use a hash table from a state to the previous state to allow reconstruction of the solution path. The implementation assumes that the states have unique ids, and the caller maps between them and its own state representation. Pass the closed set to the problem to allow it to calculate lower bounds. E.g., to solve the TSP, the graph is a tree where any node has an edge to each untried city:

Choose A					
Next	B	C	D	E	
Cumulative Cost		6	3.61	5.1	7.81
MST	DC + ED	ED + EB	CB + EC	DC + CB	
MST Cost		8.61	10	7.85	7.21
Connect edge	CB	DC	DC	ED	
Connect edge cost		3.61	3.61	3.61	5
Lowerbound	18.21	17.21	16.55	20.02	

Choose D			Choose C			
Next	B	C	E	B	D	E
Cumulative Cost		12.17	8.7	10.1	7.21	7.21
MST	EC	EB	CB	ED	EB	DB
MST Cost		4.24	5	3.61	5	5
Connect edge	DC	CB	EC	DC	CB	DC
Connect edge cost		3.61	3.61	4.24	3.61	3.61
Lowerbound	20.02	17.31	17.95	15.82	15.82	18.52

Choose B			Choose D		
Next	D	E	B	E	
Cumulative Cost		14.28	12.21	14.28	12.21
Connect edge	ED	ED	EB	EB	
Connect edge cost		5	5	5	5
Lowerbound	19.28	17.21	19.28	17.21	

Choose E	
Next	D
Cumulative Cost	17.21

Figure 16.4: A* calculations for the TSP in Figure 16.1. First it goes AD, then tries ACD, and finally finds optimal ACBE.

For implementation allow the problem to access the closed set because some calculations may depend on it in some cases. The main resource is the memory for the closed and open states, so impose a budget on the total number of states. But because every generated state is remembered, this is also a limit on the number of calculated lower bounds, except for a minor fraction of recalculations.

```
template<typename PROBLEM> struct AStar
{ //closed set paths stored as parent pointer tree
    typedef typename PROBLEM::STATE_ID STATE_ID;
    typedef typename PROBLEM::HASHER HASHER;
    typedef ChainingHashTable<STATE_ID, STATE_ID, HASHER> P;
    class PredVisitor
```

```

{
    P& pred;
public:
    PredVisitor(P& thePred) : pred(thePred) {}
    STATE_ID const* getPred(STATE_ID x) const {return pred.find(x);}
    Vector<STATE_ID> getPath(STATE_ID x) const
    {
        Vector<STATE_ID> path;
        for(;;)
        {
            path.append(x);
            STATE_ID* px = pred.find(x);
            if(px) x = *px; //form path
            else break; //no pred
        }
        path.reverse(); //need path in travel not parent pointer order
        return path;
    }
};

static pair<Vector<STATE_ID>, bool> solve(PROBLEM const& p,
    int maxSetSize = 1000000)
//parent of current state as data
typedef pair<double, STATE_ID> QNode;
IndexedHeap<QNode,
    PairFirstComparator<double, STATE_ID>, STATE_ID, HASHER> pQ;
P pred;
PredVisitor v(pred);
bool foundGoal = false;
STATE_ID j = p.start(); //start has no predecessor
pQ.insert(QNode(p.remainderLowerBound(p.nullState(), j, v),
    p.nullState()), j);
while(!pQ.isEmpty() && pred.getSize() + pQ.getSize() < maxSetSize)
{
    pair<QNode, STATE_ID> step = pQ.deleteMin();
    j = step.second;
    if(j != p.start())
        pred.insert(j, step.first.second); //now know best predecessor
    if(p.isGoal(j, v))
    {
        foundGoal = true;
        break;
    } //subtract the last move's lower bound to get the exact distance
    double dj = step.first.first -
        p.remainderLowerBound(step.first.second, j, v);
    Vector<STATE_ID> next = p.nextStates(j, v);
    for(int i = 0; i < next.getSize(); ++i)
    {
        STATE_ID to = next[i];
        double newChildLowerBound = dj + p.distance(j, to, v) +
            p.remainderLowerBound(j, to, v);
        QNode const* current = pQ.find(to);
        if((current && newChildLowerBound < current->first) ||
            (!current && !pred.find(to))) //update if better or new
            pQ.changeKey(QNode(newChildLowerBound, j), to);
    }
} //form path to goal or best current partial solution
return make_pair(v.getPath(j), foundGoal);
}
};

```

A* is optimal for a given lower bound function in the number of considered states, on which the runtime depends (Russell & Norvig 2020). A* considers fewer states than B&B because:

- It expands the node with the best global lower bound, and not the best child of the current node

- The change key operation forces to consider only the best path to the current state so that the solutions based on suboptimal paths will never be considered further

But B&B wins in other aspects:

- The open/closed state data structures can get too large, and, if the problem isn't solved with the given memory limit, only a partial solution is returned. B&B needs $O(\text{the path length})$ memory, so can afford orders-of-magnitude more lower bound calculations. The found solution might be optimal but not provably.
- Making a move with A* isn't efficient relative to with B&B, which matters for cheap lower bounds.
- Setting up the problem data structure for A* is more difficult than for B&B.

So for combinatorial problems B&B is better as a solver that gives a good approximate solution, and A* is good for proving solution optimality for small n . But for something like Rubik's cube A* works better because depth-first search of B&B doesn't fit the problem as well as the breadth-first search of A*. Still, B&B works with some **iterative deepening** strategy, where limit the maximum depth in stages, using doubling on the limits.

For combinatorial problems determination of state IDs for A* is difficult because can't determine the ID by the position. A* assumes unique states, but this is problem-specific and often not intuitive to decide what should be unique. The state generally depends on:

- Which partial solution components are present
- The sequence in which those components are added

A* is good at dealing with (2). So the state should only be based on (1). To save memory, a bit set is usually a good representation, at least for a part of the state. This is used for TSP permutations below, augmented with the last element to distinguish rotations.

Remembering the full partial solution as state, as in B&B, is a bad idea because both use more memory and don't benefit from the change key operation. A more naïve representation is based on indexing each lower bound calculation from 0. Complete solutions are created from the last move and its predecessors in the closed set of A*. This gives a cheap, one-word state but doesn't benefit from the change key operation. So for problems where (2) is irrelevant, such as subset-based problems, only consider B&B.

A* doesn't seem useful for practical TSP solving. With a generous 10^7 set size limit it's usually able to solve random instances of size 35 but for 40 (the experiments use increments of 5) runs out of memory. B&B can prove optimality of 30-instance sizes, but not 35 (the solution is optimal though or close). So the following implementation is mostly a proof of concept. It doesn't need the visitor functionality, unlike the one using the naïve state representation. To compare, the latter roughly uses an order-of-magnitude more states and is able to solve instances of size 30 but not 35.

```
template<typename PROBLEM> struct AStartTSPProblem
{
    PROBLEM const& problem;
    typedef pair<Bitset<>, int> STATE_ID;
    STATE_ID nullState() const
        return make_pair(Bitset<>(problem.points.getSize()), -1);
    struct HASHER
    {
        DataHash<> h;
        HASHER(unsigned long long m): h(m) {}
        unsigned long long operator()(STATE_ID const& s) const
        {
            Vector<unsigned long long> storage = s.first.getStorage();
            storage.append(s.second);
            return h(storage);
        }
        //don't know a legitimate best first node
    } start() const {return nullState(); }
    Vector<int> convertStatePath(Vector<STATE_ID> const& path) const
        //first state start
        Vector<int> result;
        for(int i = 1; i < path.getSize(); ++i)
            result.append(path[i].second);
        return result;
    }
    Vector<int> findRemainder(STATE_ID const& id) const
    {
```

```

    Vector<int> result;
    for(int i = 0; i < id.first.getSize(); ++i)
        if(!id.first[i]) result.append(i);
    return result;
}
template<typename VISITOR>
bool isGoal(STATE_ID id, VISITOR const& dummy) const
{
    id.first.flip();
    return id.first.isZero();
}
template<typename VISITOR>
Vector<STATE_ID> nextStates(STATE_ID const& id, VISITOR const& dummy) const
{
    Vector<STATE_ID> result;
    Vector<int> remainder = findRemainder(id);
    for(int i = 0; i < remainder.getSize(); ++i)
    {
        STATE_ID to = id;
        to.first.set(remainder[i], true);
        to.second = remainder[i];
        result.append(to);
    }
    return result;
}
template<typename VISITOR> double remainderLowerBound(STATE_ID const& dummy,
    STATE_ID const& to, VISITOR const& dummy2) const
{
    return findMSTCost(problem, findRemainder(to));
}
template<typename VISITOR> double distance(STATE_ID const& j,
    STATE_ID const& to, VISITOR const& dummy) const
{
    return j == start() ? 0 : problem.evalStep(j.second, to.second);
}
};

```

Recursive best-first search (RBFS) uses O(the path length) space (assuming a bounded number of children in each) but revisits some states. It stores in each node the best known value of the best alternative subtree that is accessible from any ancestor node and forgets the nonroot nodes.

1. The current node = the root, with lower bound = 0 (always valid), and the alternative = ∞
2. Put all children in a priority queue with their lower bounds as priorities
3. Until found the goal, the root's value = ∞ when a solution doesn't exist, or exceeded computational budget
4. Take the best child from the queue
5. If have none, return the subtree with lower bound ∞
6. If the child's lower bound > the alternative, return the lower bound
7. Else recurse to the child, with alternative = min(the second best child's priority, the current node's alternative)
8. Put the child back on the queue with the updated lower bound

		Choose A			
Next	B	C	D	E	
Cumulative Cost		6	3.61	5.1	7.81
MST	DC + ED	ED + EB	CB + EC	DC + CB	
MST Cost		8.61	10	7.85	7.21
Connect edge	CB	DC	DC	ED	
Connect edge cost		3.61	3.61	3.61	5
Lowerbound	18.21	17.21	16.55	20.02	
Updated LB			17.31		
		Choose D			
Next	B	C	E		Choose C
Cumulative Cost		12.17	8.7	10.1	B D E
MST	EC	EB	CB		ED 7.21 7.21 7.85
MST Cost		4.24	5	3.61	EB DB
Connect edge	DC	CB	EC		DC 5 5 7.07
Connect edge cost		3.61	3.61	4.24	CB DC
Lowerbound	20.02	17.31	17.95		3.61
Updated LB				15.82 15.82	18.52
				17.21	
		Choose B		Choose D	
Next	D	E	B	D E	
Cumulative Cost		14.28	12.21	14.28	12.21
Connect edge	ED	ED	EB	EB	
Connect edge cost		5	5	5	5
Lowerbound	19.28	17.21		19.28	17.21
		Choose E			
Next	B				
Cumulative Cost				17.21	

Figure 16.5: RBFS calculations for TSP in Figure 16.1. First it goes AD, then ACB, then ACD, and, forgetting about ACB, finds optimal ACDE

The algorithm makes progress because known alternative lower bounds only increase. So no cycling is possible even if some lower bounds are exactly 0.

```
template<typename PROBLEM> struct RecursiveBestFirstSearch
{
    typedef typename PROBLEM::STATE_ID STATE_ID;
    typedef Stack<STATE_ID> P;
    P pred; //path to the goal, which is top
    PROBLEM const& p;
    enum{SUCCESS = -1, FAILURE = -2};
    typedef pair<double, STATE_ID> INFO; //lower bound and state
    bool foundGoal;
    Vector<STATE_ID> bestPath;
    class PredVisitor
    { //assume top of stack always current node, so pred is next
        Stack<STATE_ID>& pred;
    public:
        PredVisitor(P& thePred) : pred(thePred) {}
        STATE_ID const* getPred(STATE_ID dummy) const
        {
            Vector<STATE_ID>& storage = pred.storage;
            return storage.getSize() > 1 ? &storage[storage.getSize() - 2] : 0;
        }
        Vector<STATE_ID> getPath(STATE_ID dummy) const
        {
            Vector<STATE_ID> path;
            Stack<STATE_ID> s = pred;
            while(!s.isEmpty()) path.append(s.pop());
            path.reverse(); //need path in travel not parent pointer order
            return path;
        }
    };
}
```

```

};

double work(INFO state, double alternative, double pathCost,
    int& maxLowerBounds)
//stop if found goal, or out of moves, or exceed computation budget
    PredVisitor v(pred);
    if(p.isGoal(state.second, v)) return SUCCESS;
    Vector<STATE_ID> next = p.nextStates(state.second, v);
    if(next.getSize() == 0) return numeric_limits<double>::infinity();
    if(maxLowerBounds < next.getSize()) return FAILURE;
    maxLowerBounds -= next.getSize();
//sort children by lower bound
    Heap<INFO, PairFirstComparator<double, STATE_ID> > children;
    for(int i = 0; i < next.getSize(); ++i)
        children.insert(INFO(max(state.first, pathCost +
            p.distance(state.second, next[i], v) +
            p.remainderLowerBound(state.second, next[i], v)), next[i]));
    for(;;)
    {
        INFO best = children.deleteMin();
        //don't process remaining children if alternative better and
        //return the current best child value
        if(best.first > alternative) return best.first;
        //compute d before push best, else break visitor invariant
        double d = p.distance(state.second, best.second, v);
        pred.push(best.second);
        //as alternative use better of alternative and next best child
        best.first = work(best, children.isEmpty() ?
            alternative : min(children.getMin().first, alternative),
            pathCost + d, maxLowerBounds);
        if(best.first == SUCCESS) return SUCCESS;
        else if(best.first == FAILURE) return FAILURE;
        children.insert(best); //enqueue child with revised estimate
        pred.pop(); //undo move
    }
}
RecursiveBestFirstSearch(PROBLEM const& theProblem,
    int maxLowerBounds = 10000000): p(theProblem), foundGoal(false)
{
    pred.push(p.start());
    foundGoal = (work(INFO(0.0, pred.getTop()),
        numeric_limits<double>::infinity(), 0, maxLowerBounds) == SUCCESS);
    PredVisitor v(pred);
    bestPath = v.getPath(pred.getTop());
}
};

```

RBFS uses too little memory. The runtime should be similar to A* but is hard to analyze because much depends on the greedy choice success. Can store the values of the forgotten subtrees in an LRU cache, but this improvement is complicated to implement.

RBFS is less useful for TSP than A*, solving 20- but not 25-instance problems. The algorithm makes sense for problems with cheap lower bounds like Rubik's cube, solving which with A* breaks reasonable memory limits but not reasonable lower bound limits with RBFS. For more on solving such puzzles see Kopec et al. (2016).

16.8 Local Search

For most optimization problems similar solutions are of similar quality. So local search (also called **hill climbing**) starts with an initial solution, constructed randomly, by a heuristic, an approximation algorithm, or other means, and makes small improving changes until reaching a **local optimum**. The **moves** can be:

- **First-found improving**—useful when potentially all moves must be checked. Doesn't scale to large n because it's very wasteful to generate a large move list if the first move is found very quickly, unless generate incrementally.
- **Best**—most useful if efficiently computable by a special algorithm, and not by checking all steps.

- **Random improving**—simple and scales well for large n when most moves are improving. But to detect a local optimum at best can check if the last k moves weren't improving, with maybe an agnostic $k = 1000$ or a problem-dependent strategy that looks at the size of the neighborhood.

Moves may allow **incremental evaluation**, i.e., computing a candidate solution's quality from the current solution's quality and the move's properties in $O(n)$ time, even if the solution representation space and the move's runtime are $O(n)$. This makes random nonimproving moves cheap to reject.

The local search implementation here allows all move types and incremental evaluation. With best use maximum stall = 1. For random improving a good strategy is to count the number of possible moves N for the used operator and use maybe $O(N \lg(N))$ per coupon collector's problem. A tail estimate might also make sense—see Wikipedia (2019). Moves of 0 difference are allowed to prevent stalling on plateau landscapes.

```
template<typename PROBLEM> typename PROBLEM::X localSearch(PROBLEM const& p,
    typename PROBLEM::X x, int maxMoves, int maxStall = -1)
//the second value of the below is the score
typedef pair<typename PROBLEM::MOVE, double> MOVE;
for(int i = 0; maxMoves-- && (maxStall == -1 || i < maxStall); ++i)
{
    MOVE m = p.proposeMove(x);
    if(m.second <= 0)
    {
        i = -1; //reset counter on accept
        p.applyMove(x, m.first);
    }
}
return x;
}
```

3 general moves for common solution representations (the complexities are for random improving and a problem of size n):

- Permutation:
 - Remove an element and insert it between two others— $O(n)$. Doesn't seem very useful.
 - Swap two possibly adjacent elements— $O(1)$.
 - Reverse a part of the permutation array— $O(n)$. Equivalent to swapping two links, assuming symmetric cost of edges. The evaluation is $O(1)$ for symmetric problems.
 - Swap two adjacent elements— $O(1)$. Not popular, at least for TSP, and is a special case of the reversal.

```
template<typename EVALUATOR> class SymmetricPermutationProblemReverseMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<int>, typename EVALUATOR::INCREMENTAL_STATE> X;
    typedef pair<int, int> MOVE;
    SymmetricPermutationProblemReverseMove(EVALUATOR const& theE) : e(theE) {}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        Vector<int> m = GlobalRNG().randomCombination(2, x.first.getSize());
        if(m[0] > m[1]) swap(m[0], m[1]);
        return make_pair(make_pair(m[0], m[1]),
            e.evalReverse(m[0], m[1], x.first, x.second));
    }
    void applyMove(X& x, MOVE const& m) const
    {
        assert(m.first < m.second && m.first >= 0 &&
            m.second < x.first.getSize());
        x.second = e.updateIncrementalStateReverse(m.first, m.second, x.first,
            x.second);
        x.first.reverse(m.first, m.second);
    }
    double getScore(X const& x) const {return e(x.first);}
};

template<typename EVALUATOR> class PermutationProblemSwapMove
```

```

{
    EVALUATOR const& e;
public:
    typedef pair<Vector<int>, typename EVALUATOR::INCREMENTAL_STATE> X;
    typedef pair<int, int> MOVE;
    PermutationProblemSwapMove(EVALUATOR const& theE): e(theE) {}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        Vector<int> m = GlobalRNG().randomCombination(2, x.first.getSize());
        if(m[0] > m[1]) swap(m[0], m[1]); //unnecessary, but cleaner this way
        return make_pair(make_pair(m[0], m[1]),
                         e.evalSwap(m[0], m[1], x.first, x.second));
    }
    void applyMove(X& x, MOVE const& m) const
    {
        assert(m.first < m.second && m.first >= 0 &&
               m.second < x.first.getSize());
        x.second = e.updateIncrementalStateSwap(m.first, m.second, x.first,
                                                x.second);
        swap(x.first[m.first], x.first[m.second]);
    }
    double getScore(X const& x) const{return e(x.first);}
};


```

- Subset:

- Select i , and deselect j items— $O(i + j)$. A simple special case is to flip an item's selection.

```

template<typename EVALUATOR> class SubsetProblemFlipMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<bool>, typename EVALUATOR::INCREMENTAL_STATE> X;
    typedef int MOVE;
    SubsetProblemFlipMove(EVALUATOR const& theE): e(theE) {}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        int i = GlobalRNG().mod(x.first.getSize());
        return make_pair(i, e.evalStep(i, x.first, x.second));
    }
    void applyMove(X& x, MOVE const& i) const
    { //update incremental state
        assert(i >= 0 && i < x.first.getSize());
        x.second = e.updateIncrementalState(i, x.first, x.second);
        //do flip
        x.first[i] = !x.first[i];
    }
    double getScore(X const& x) const{return e(x.first);}
};


```

- Partition:

- Move i items from partition A to partition B and j items from B to A — $O(i + j)$. A simple case is to move one item from one partition to another. A convenient representation for partitions is a list of lists of items that belong together.

```

template<typename EVALUATOR> class PartitionProblemSwapMove
{
    EVALUATOR const& e;
public:
    typedef pair<Vector<Vector<int> >,
                typename EVALUATOR::INCREMENTAL_STATE> X;
    struct MOVE{int index, from, to;};
    PartitionProblemSwapMove(EVALUATOR const& theE): e(theE) {}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        assert(x.first.getSize() >= 2); //cls code breaks
    }
};


```

```

Vector<int> fromTo =
    GlobalRNG().randomCombination(2, x.first.getSize());
Vector<int> const& bin1 = x.first[fromTo[0]];
assert(bin1.getSize() > 0); //else bad problem
int index = GlobalRNG().mod(bin1.getSize());
MOVE m = {index, fromTo[0], fromTo[1]};
return make_pair(m, e.evalStep(index, m.from, m.to, x.first,
    x.second));
}
void applyMove(X& x, MOVE const& m) const
{//update incremental state
    assert(m.from >= 0 && m.from < x.first.getSize() &&
        m.to >= 0 && m.to < x.first.getSize() && m.index >= 0 &&
        m.index < x.first[m.from].getSize());
    x.second = e.updateIncrementalState(m.index, m.from, m.to, x.first,
        x.second);
    //do swap
    Vector<int>& bin1 = x.first[m.from];
    x.first[m.to].append(bin1[m.index]);
    bin1[m.index] = bin1.lastItem();
    bin1.removeLast();
    //remove bin1 if it became empty
    if(bin1.getSize() == 0)
    {
        bin1 = x.first.lastItem();
        x.first.removeLast();
    }
}
double getScore(X const& x) const{return e(x.first);}
};

```

For permutation problems such as TSP use local search as follows:

```

template<typename INSTANCE> Vector<int>
solveSymmetricPermutationLocalSearchReverse(INSTANCE const& instance,
    Vector<int> const& initial, int maxMoves)
{//use log(n) as local search stop limit
    int nPossibleMoves = initial.getSize() * initial.getSize()/2;
    return localSearch(SymmetricPermutationProblemReverseMove<INSTANCE>(
        instance), make_pair(initial, instance.getIncrementalState(initial)),
        maxMoves, int(10 * nPossibleMoves * log(nPossibleMoves))).first;
}
template<typename INSTANCE> Vector<int> solvePermutationLocalSearchSwap(
    INSTANCE const& instance, Vector<int> const& initial, int maxMoves)
{//use log(n) as local search stop limit
    int nPossibleMoves = initial.getSize() * initial.getSize()/2;
    return localSearch(PermutationProblemSwapMove<INSTANCE>(
        instance), make_pair(initial, instance.getIncrementalState(initial)),
        maxMoves, int(10 * nPossibleMoves * log(nPossibleMoves))).first;
}

```

For subset problems such as knapsack and maximum satisfiability use local search as follows:

```

template<typename INSTANCE> Vector<bool> solveSubsetLocalSearchFlip(
    INSTANCE const& instance, Vector<bool> const& initial, int maxMoves)
{//use log(n!) as local search stop limit
    return localSearch(SubsetProblemFlipMove<INSTANCE>(instance),
        make_pair(initial, instance.getIncrementalState(initial)), maxMoves,
        int(10 * initial.getSize() * log(initial.getSize()))).first;
}

```

For partition problems such as bin packing use local search as follows:

```

template<typename INSTANCE> Vector<Vector<int> >
solvePartitionLocalSearchSwap(INSTANCE const& instance,
    Vector<Vector<int> > const& initial, int maxMoves)
{//use log(n!) as local search stop limit
    int nPossibleMoves = initial.getSize() * initial.getSize()/2;

```

```

    return localSearch(PartitionProblemSwapMove<INSTANCE>(instance),
        make_pair(initial, instance.getIncrementalState(initial)), maxMoves,
        int(10 * nPossibleMoves * log(nPossibleMoves))).first;
}

```

In terms of object-oriented design, these methods are generic and not coupled to the problem, which allows reuse but sometimes forces inefficiency. To compare, B&B and A* are coupled to the problem. Methods that deal with constraints for which good penalty function aren't available must also be coupled to the problem, at least for the specific way of ensuring feasibility.

A move type forms a graph with vertices corresponding to solutions and having outgoing edges to vertices reachable by the allowable moves. The number of iterations of local search \leq the size of the longest improving path and is $O(\text{exponential in } n)$, so always limit the number of iterations. A move type is **complete** if the graph is strongly connected. Completeness allows visiting every solution. Visually, the graph is a mountainous terrain, with floor heights corresponding to solution qualities.

The **global optimum** is the best solution. Local search may get stuck in lower-quality local optima. Difficult landscape features that confuse local search include a:

- **Plateau**—a flat area where the solutions have about the same quality
- **Golf hole**—a local optimum that is much better than all nearby solutions

A solution's **neighborhood** is the set of all solutions reachable from it by a single move. Its size is the number of allowable moves. A neighborhood **contains** another if it contains its every solution. Local optimum in one neighborhood needn't be so in another not contained by it. Can use moves from several neighborhoods that don't contain each other and switch when the current one gets stuck. Or can use an expensive move type in a large neighborhood when a cheap one gets stuck in a contained neighborhood.

A **metaheuristic** is generally a strategy for avoiding getting stuck in a local optimum. Many have been proposed and popularized, but this chapter discusses few:

- Only those that guide local search deserve consideration in practice—this is a mostly unanimous conclusion from the research literature. For many methods, the top-performing variants incorporate local search in some ways—it's just too good to ignore.
- Only consider those that need very little from the user. A domain-specific random move and the ability to evaluate it incrementally is the minimum requirement. Other needs include mixing two solutions, create a perturbation that is hard to undo by a local move, undoing a done move incrementally, etc.
- Prefer simpler and more established algorithms that are used often and successfully. It's up to the research community to sort out the worthy ones and improve performance as much as possible. As a general trend, any improvement have been at most slight, so the user wants to be a late adapter of promising methods.
- Consider only simple and most logical parameter settings, unless the literature suggests otherwise. It's up to the research community and not the user to experiment with the myriad of appealing strategies. Any deviations from a "standard" configuration must be logically justified or empirically proven to lead to better performance to be used. E.g., if a parameter value is 1, trying 1.1 is silly—on any reasonable test set this will only give better performance by multiple testing (see the "Computational Statistics" chapter).
- Not all algorithms and variants allow easy parameter tuning with a "self-tuned" version that calculates reasonable defaults. Only algorithms that allow this will be considered—any tuning a user might do manually should be automatable.
- Not all appealing ideas lead to general algorithms—many strategies might sound reasonable, but the implementation would need substantial customization to various problems, enough to discourage general application.

16.9 Applying Local Search to Some Problems

The main difficulty is allowing efficient incremental evaluation to work with the common moves. For constrained problems like knapsack and bin packing use a small penalty for simplicity. Working with feasible solutions only prevents reuse of common moves but is likely to be a better strategy for practical problem solving. This isn't discussed further.

For TSP, evaluating reverse and swap moves needs some care to not make a silly mistake but is otherwise straightforward. The incremental state is the current evaluation, which is typically the minimal needed information. The implementation below and those for other problems in this section implement the interface expected by the moves for that representation, consisting of `evaluate*`,

```

updateIncrementalState*, getIncrementalState*, and operator().

template<typename PROBLEM> class TSPProxy
{
    PROBLEM const& p;
    double evalStep(Vector<int> const& permutation, int from, int to) const
        {return p.evalStep(permutation[from], permutation[to], to == 0);}

public:
    typedef double INCREMENTAL_STATE; //current score
    TSPProxy(PROBLEM const& theP): p(theP) {}
    INCREMENTAL_STATE updateIncrementalStateReverse(int i, int j,
        Vector<int> const& permutation, INCREMENTAL_STATE const& is) const
    { //take out edges i-1 to i and j to j + 1; add i-1 to j and i to j + 1
        int n = permutation.getSize();
        double imlFactor = i > 0 ? evalStep(permutation, i - 1, j) -
            evalStep(permutation, i - 1, i) : 0,
            jp1Factor = j + 1 < n ? evalStep(permutation, i, j + 1) -
            evalStep(permutation, j, j + 1) : 0;
        return is + imlFactor + jp1Factor;
    }
    double evalReverse(int i, int j, Vector<int> const& permutation,
        INCREMENTAL_STATE const& is) const
        {return updateIncrementalStateReverse(i, j, permutation, is) - is;}
    INCREMENTAL_STATE updateIncrementalStateSwap(int i, int j,
        Vector<int> const& permutation, INCREMENTAL_STATE const& is) const
    { //take out edges i-1 to i and i to i + 1, j-1 to j and j to j + 1
        //add edges i-1 to j and i to i + 1, j-1 to i and i to j + 1
        int n = permutation.getSize();
        double imlFactor = i > 0 ? evalStep(permutation, i - 1, j) -
            evalStep(permutation, i - 1, i) : 0,
            ip1Factor = i + 1 < n ? evalStep(permutation, j, i + 1) -
            evalStep(permutation, i, i + 1) : 0,
            jm1Factor = j > 0 ? evalStep(permutation, j - 1, i) -
            evalStep(permutation, j - 1, j) : 0,
            jp1Factor = j + 1 < n ? evalStep(permutation, i, j + 1) -
            evalStep(permutation, j, j + 1) : 0;
        return is + imlFactor + ip1Factor + jm1Factor + jp1Factor;
    }
    double evalSwap(int i, int j, Vector<int> const& permutation,
        INCREMENTAL_STATE const& is) const
        {return updateIncrementalStateSwap(i, j, permutation, is) - is;}
    INCREMENTAL_STATE getIncrementalState(Vector<int> const& permutation) const
        {return p(permutation);}
    double operator()(Vector<int> const& permutation) const
        {return getIncrementalState(permutation);}
};

For knapsack, to evaluate an item flip need extra incremental state due to the weight constraint, unlike
for TSP. To create some gradient penalize extra weight enough to exceed all profits.

```

```

template<typename PROBLEM> class KnapsackPenaltyProxy
{
    PROBLEM const& p;
    double profitLimit;
    double getScore(pair<double, double> const& is) const
    { //evaluate complete solution as minimization problem
        double score = -is.first, capacity = p.getCapacity();
        if(is.second > capacity) //simple penalty for exceeding capacity
            score += profitLimit * exp((is.second - capacity)/capacity);
        return score;
    }
public:
    typedef pair<double, double> INCREMENTAL_STATE; //first is profit
    KnapsackPenaltyProxy(PROBLEM const& theP): p(theP), profitLimit(0)
        {for(int i = 0; i < p.getSize(); ++i) profitLimit += p.getProfit(i);}

```

```

INCREMENTAL_STATE updateIncrementalState(int iToFlip,
    Vector<bool> const& subset, INCREMENTAL_STATE const& is) const
{
    bool selection = !subset[iToFlip];
    return INCREMENTAL_STATE(is.first + p.getProfit(iToFlip) *
        (selection ? 1 : -1), is.second + p.getWeight(iToFlip) *
        (selection ? 1 : -1));
}
double evalStep(int iToFlip, Vector<bool> const& subset,
    INCREMENTAL_STATE const& is) const
{//evaluate single step difference
    return getScore(updateIncrementalState(iToFlip, subset, is)) -
        getScore(is);
}
INCREMENTAL_STATE getIncrementalState(Vector<bool> const& subset) const
{
    assert(subset.getSize() == p.getSize());
    double totalProfit = 0, totalWeight = 0;
    for(int i = 0; i < subset.getSize(); ++i) if(subset[i])
    {
        totalProfit += p.getProfit(i);
        totalWeight += p.getWeight(i);
    }
    return INCREMENTAL_STATE(totalProfit, totalWeight);
}
double operator()(Vector<bool> const& subset) const
{
    return getScore(getIncrementalState(subset));
};

```

For minimum satisfiability incremental evaluation is a little tricky—the incremental state is the number of clauses satisfied so far. Also to map a variable to the list of clauses containing it. For a given flip, only these are worked on.

```

template<typename PROBLEM> class SatisfiabilityProxy
{
    PROBLEM const& p;
    Vector<Vector<int>> varToClauseMap;
    bool evaluateClause(Vector<bool> const& subset, Vector<int> const& clause,
        int overrideI = -1) const
    {
        bool value = false;
        for(int j = 0; j < clause.getSize(); ++j)
        {
            int i = abs(clause[j]);
            bool valVar = subset[i];
            if(i == overrideI) valVar = !valVar; //override means flip
            if(clause[j] < 0) valVar = !valVar; //negated variable
            value |= valVar; //sat is "and" of "or" clauses
        }
        return value;
    }
public:
    typedef int INCREMENTAL_STATE; //nSatisfied
    SatisfiabilityProxy(PROBLEM const& theP): p(theP),
        varToClauseMap(p.getSize())
    {//use n clauses for n variables
        for(int i = 0; i < p.getClausesSize(); ++i)
        {
            Vector<int> const& clause = p.getClauses(i);
            for(int j = 0; j < clause.getSize(); ++j)
            {//variables have signs
                int variable = abs(clause[j]);
                varToClauseMap[variable].append(i);
            }
        }
    }

```

```

        }

    }

INCREMENTAL_STATE updateIncrementalState(int iToFlip,
    Vector<bool> const& subset, INCREMENTAL_STATE const& is) const
{
    Vector<int> const& affectedClauses = varToClauseMap[iToFlip];
    int nSatisfiedOld = 0, nSatisfied = 0;
    for(int i = 0; i < affectedClauses.getSize(); ++i)
    {
        Vector<int> const& clause = p.getClause(affectedClauses[i]);
        nSatisfiedOld += evaluateClause(subset, clause);
        nSatisfied += evaluateClause(subset, clause, iToFlip);
    }
    return is + (nSatisfied - nSatisfiedOld);
}

double evalStep(int iToFlip, //evaluate single step difference
    Vector<bool> const& subset, INCREMENTAL_STATE const& is) const
{
    return -(updateIncrementalState(iToFlip, subset, is) - is);
}

INCREMENTAL_STATE getIncrementalState(Vector<bool> const& subset) const
{
    assert(subset.getSize() == varToClauseMap.getSize());
    int nSatisfied = 0;
    for(int i = 0; i < p.getClausesSize(); ++i)
        nSatisfied += evaluateClause(subset, p.getClause(i));
    return nSatisfied;
}

double operator()(Vector<bool> const& subset) const
{
    return -getIncrementalState(subset);
}
};

```

For bin packing, the incremental state is the current evaluation, as for TSP. A general partition need not correspond to an allowable packing, so use a penalty for overfilled bins.

```

template<typename PROBLEM> class BinPackingProxy
{
    PROBLEM const& p;
    double getBinScore(Vector<int> const& bin) const
    { //evaluate complete solution as minimization problem
        if(bin.getSize() == 0) return 0;
        double sum = 0;
        for(int i = 0; i < bin.getSize(); ++i)
            sum += p.getWeight(bin[i]);
        double score = 1, binSize = p.getBinSize();
        if(sum > binSize) //simple penalty for exceeding capacity
            score += exp((sum - binSize)/binSize);
        return score;
    }
public:
    typedef double INCREMENTAL_STATE; //current score
    BinPackingProxy(PROBLEM const& theP): p(theP) {}
    INCREMENTAL_STATE updateIncrementalState(int index, int from, int to,
        Vector<Vector<int>> const& bins, INCREMENTAL_STATE const& is) const
    {
        assert(index < bins[from].getSize());
        Vector<int> &bin1 = (Vector<int>&)bins[from],
            &bin2 = (Vector<int>&)bins[to];
        double partialScore = is - getBinScore(bin1) - getBinScore(bin2);
        //do move
        bin2.append(bin1[index]);
        bin1[index] = bin1.lastItem();
        bin1.removeLast();
        double score = partialScore + getBinScore(bin1) + getBinScore(bin2);
        //and do move
        bin1.append(bin2.lastItem());
    }
};

```

```

        bin2.removeLast();
        swap(bin1[index], bin1.lastItem());
        return score;
    }

    double evalStep(int index, int from, int to, Vector<Vector<int>> const& bins,
                   INCREMENTAL_STATE const& is) const
    {
        return updateIncrementalState(index, from, to, bins, is) - is;
    }

    INCREMENTAL_STATE getIncrementalState(Vector<Vector<int>> const& bins)
    const
    {
        double sum = 0;
        for(int i = 0; i < bins.getSize(); ++i) sum += getBinScore(bins[i]);
        return sum;
    }

    double operator()(Vector<Vector<int>> const& bins) const
    {
        return getIncrementalState(bins);
    }
}

```

16.10 Simulated Annealing

Metaheuristics are strategies for global exploration. In particular, they allow local search to escape local optima. SA works with random moves and accepts worsening ones with some probability that starts high and decreases until becoming very small and making the process equivalent to local search. The random motion gets the search into the landscape part with the global optimum, and the eventual local search finds it. The name is inspired by the physical annealing process of metal cooling, which gives a better structure when down slowly.

The score change Δf = the proposed score – the current score, and "temperature" T determine $\Pr(\text{accept a bad move})$ as accept if $e^{-\Delta f/T} > \text{uniform01}$. For a bad move $\Delta f > 0$. A simpler, equivalent condition is $\Delta f < T \times \text{exponential01}$ ($-\log(\text{uniform01})$ is an exponential variate; see the "Monte Carlo Algorithms" chapter). Every iteration multiplies T by a "cooling factor" to reduce it.

So the effective final temperature $T_{\text{last}} = T_0 \times \text{coolingFactor}^{n\text{Moves}}$. Any bad moves with Δf below this have a very high probability, so the implementation allows to control them by considering Δf as though of larger magnitude, specified by the caller.

```

template<typename PROBLEM> typename PROBLEM::X simulatedAnnealing(
    PROBLEM const& p, typename PROBLEM::X x, double T, double coolingFactor,
    double TCap, int maxMoves)

{
    assert(maxMoves > 0 && isfinite(T) && T > TCap && TCap >= 0 &&
           coolingFactor < 1 && coolingFactor > 0 && isfinite(p.getScore(x)));
    typedef pair<typename PROBLEM::MOVE, double> MOVE;
    for(; maxMoves--; T *= coolingFactor)
    {
        //comparison false if m.second == NaN
        MOVE m = p.proposeMove(x);
        if(m.second <= 0 || max(m.second, TCap) < T *
            GlobalRNG().exponential(1)) p.applyMove(x, m.first);
    }
    return x;
}

```

Setting the parameters to good values isn't trivial. Traditionally, the users had to try some possibilities and use whatever is best for their domain. Some form of grid search or another optimization method (see the "Numerical Optimization" chapter) is useful here. But have a reasonable self-tuning procedure based on the following:

- Assume that have an initial solution of random quality, so a random walk will keep it random.
- Initially should accept $\approx 90\%$ of moves of typical value. A reasonable strategy is to spend some small fraction of always-accepted initial moves, such as $\sqrt{(n\text{Moves})}$, to estimate the 95th % change value. If have domain knowledge, instead use the maximum possible change. Then calculate T_0 as $= \frac{\text{typical value}}{-\log(\text{the wanted probability})}$.
- The maximum number of iterations is the only control of runtime. Usually need many iterations to find the global optimum, so pick as many as patience allows. A value such as 10^6 is reasonable.

- Set the cooling factor so that most later iterations aren't doing only local search. I.e., don't want T to drop to values too close to 0 in few iterations, as would be the case with the typical suggested starting cooling factor of 0.9999* (though for some toy problems using $1 - \frac{10}{nMoves}$ isn't too far away from the value found using the below procedure). Pick T_{cap} as the 5th percentile change from the initial random moves. If have domain knowledge of a minimum possible change > 0, use that instead.

Consider the system "frozen" when the probability of accepting is $\frac{1}{nMoves}$, and set T_{last} based on

that. Then $coolingFactor = \left(\frac{T_0}{T_{last}} \right)^{\frac{1}{nMoves}}$. This is safe numerically.

```
template<typename PROBLEM> typename PROBLEM::X
    selfTunedSimulatedAnnealing(PROBLEM const& p, typename PROBLEM::X x,
    int maxMoves = 1000000)
{
    assert(maxMoves >= 9); //min for estimators to work
    int nEstimate = int(sqrt(maxMoves)); //reasonable for a good estimate
    typedef pair<typename PROBLEM::MOVE, double> MOVE;
    //T estimation stage
    Vector<double> values;
    for(int i = 0; i < nEstimate; ++i)
    { //make some random moves to get magnitude of changes
        MOVE m = p.proposeMove(x);
        double change = abs(m.second);
        if(isfinite(change) && change > 0)
        {
            values.append(change);
            p.applyMove(x, m.first);
        }
    }
    maxMoves -= nEstimate;
    if(values.getSize() < 3) //bad problem, use local search
        return localSearch(p, x, maxMoves, -1);
    double prFirst = 0.9, prLast = 1.0/maxMoves,
        deltaMax = quantile(values, 0.95), deltaMin = quantile(values, 0.05),
        T0 = -deltaMax/log(prFirst), TLast = -deltaMin/log(prLast),
        inverseRange = max(TLast/T0, numeric_limits<double>::min()),
        coolingFactor = pow(inverseRange, 1.0/maxMoves);
    return simulatedAnnealing(p, x, T0, coolingFactor, deltaMin, maxMoves);
}
```

For permutations problems such as TSP use simulated annealing as follows:

```
template<typename INSTANCE> Vector<int>
    solveSymmetricPermutationSimulatedAnnealingReverse(INSTANCE const& instance,
    Vector<int> const& initial, int maxMoves)
{
    return selfTunedSimulatedAnnealing(SymmetricPermutationProblemReverseMove<
        INSTANCE>(instance), make_pair(initial, instance.getIncrementalState(
        initial)), maxMoves).first;
}
template<typename INSTANCE> Vector<int>
    solvePermutationSimulatedAnnealingSwap(INSTANCE const* instance,
    Vector<int> const& initial, int maxMoves)
{
    return selfTunedSimulatedAnnealing(PermutationProblemSwapMove<
        INSTANCE>(instance), make_pair(initial, instance.getIncrementalState(
        initial)), maxMoves).first;
}
```

For subset problems such as knapsack and maximum satisfiability use simulated annealing as follows:

```
template<typename INSTANCE> Vector<bool> solveSubsetSimulatedAnnealing(
    INSTANCE const& instance, Vector<bool> const& initial, int maxMoves)
{
```

```

    return selfTunedSimulatedAnnealing(SubsetProblemFlipMove<INSTANCE>(
        instance), make_pair(initial, instance.getIncrementalState(initial)),
        maxMoves).first;
}

```

For partition problems such as bin packing use simulated annealing as follows:

```

template<typename INSTANCE> Vector<Vector<int>>
solvePartitionSimulatedAnnealing(INSTANCE const& instance,
Vector<Vector<int>> const& initial, int maxMoves)

return selfTunedSimulatedAnnealing(PartitionProblemSwapMove<INSTANCE>(
    instance), make_pair(initial, instance.getIncrementalState(initial)),
    maxMoves).first;
}

```

Some experiments with the self-tuned version show that both the probability of acceptance and the solution score decrease at a reasonable rate. An interesting behavior is observed for problems where a possible change value is always 1—effectively have a “nonparametric” simulated annealing where bad moves are done with some probability regardless of Δf , which is the best strategy for such problems. For problems with moves of varied Δf this isn’t a good strategy due to occasional very bad moves. But for moves of quality below of T_{cap} this again seems to be the best option in the absence of better knowledge. So at the first stage of the algorithm a well-tuned simulated annealing is done, and at the second stage, where parameter setting is hard, nonparametric annealing takes over.

A good question is whether can improve the hard-coded numbers and some other choices. Using the 90% initial acceptance probability with a “typical change value” is consistent with Talbi (2009) and many other sources. $-1/\log(0.9) \approx 9.5$ is rather large, but the literature doesn’t seem to have discovered a better strategy.

Using the maximum and the minimum seen values is a common advice. Though when the range is very large and most values are concentrated around a narrower subrange, can spend too much computation on rare change values that are too high or too low. But for self-tuning this is reasonable, particularly given that simulated annealing doesn’t seem to be very sensitive to the details of the cooling.

Substituting the maximum and the minimum with the 95th and 5th quantiles is justified by that these are probably the most extreme quantiles that are estimable with low enough variance with the chosen number of initial moves. It also adds a minor guard against some very wide ranges. The estimation is too dependent on the initial configuration though, so the initial temperature can, e.g., only represent penalized infeasible solutions. But the capping guards against this by doing the nonparametric annealing for lower temperatures. A very minor assumption is that the neighborhood is symmetric by effectively looking at Δf of reversals of improving moves, which it needn’t be, though this is. In general a self-tuning procedure can’t compete with one based on domain knowledge.

A multiplicative cooling factor is the most common but not the only option. Technically simulated annealing allows several moves for a given T value, which is the same as one move with a discretized T decrease. In general, with proper scale any function that decreases from 1 to 0 or some small value will work. E.g., can think of a linear or polynomial decrease, etc. But only with the exponential decrease of multiplication the acceptance probability of T_{cap} has a doubly-exponential behavior, which looks roughly linear at first and then slowly decreasing to the target value. Polynomial decrease schedules seem to have the advantage of going to an exact 0 so that don’t need to estimate the final T_{last} . But want to control T for a reasonable range of change values, avoiding both too fast of a decrease and spending too much time allowing higher change values.

The exponential distribution is also subject to some doubt. Would a fat-tail distribution such as Levy work better? Overall, the algorithm seems to benefit from small amounts of noise but not large ones. Using a fat-tail distribution would make it behave like the nonparametric annealing that rarely allows changes orders-of-magnitude larger than the current T .

Storing the best solution found so far has a minor score benefit and might dominate the runtime for incrementally computable moves.

Perhaps the best justification of the effectiveness of simulated annealing is that it lands in the deepest basin from which it can’t get out (Salamon et al. 2002). This is similar to what tabu search does deterministically (see the comments section). Per above reasoning about various choices and because the algorithm has been stable since its introduction in 1983, this is the final version of the implementation. In many cases, particularly when the user’s time is limited it’s reasonable that simulated annealing is both the first and the last procedure tried on a problem.

The theory of simulated annealing is somewhat peculiar. See van Laarhoven & Aarts (1987) for an over-

view and Michiels et al. (2007) for proofs. As presented, but with a slow-enough, special cooling schedule simulated is guaranteed to converge, but the number of considered solutions exceeds that of enumeration. For typical implementations with multiplicative temperature decrease don't have any theory, but practical performance is usually good. As any other metaheuristic, it's effectively an approximation algorithm without performance guarantees.

The original version of simulated annealing relied on the convergence of a simple mathematical process: at every temperature have a random walk which moves around the state space infinitely, resulting in some visitation probabilities. As T is decreased, the probabilities concentrate around the set of global minima. With a limited number of moves at each T and a slow decrease, the computational process is reasonably similar to the mathematical process. But visiting the same solution many times is very wasteful, so the behavior of the computational process is very different, as shown by the deepest basin idea.

For the mathematical process, any $T > 0$ works as well as any other, but for the computational process starting with a high T is a practical necessity. The modern version of the algorithm with one move at the same temperature but a slower decrease is both more robust against too many moves at the same T and eliminates this parameter altogether. Assuming simulated annealing isn't sensitive to some ϵ -band around the current T , and that the decrease is slow enough, both versions do the same thing. The final algorithm is better thought of as inspired by the mathematical process and not something that is trying to follow it.

16.11 Iterated Local Search

Simulated annealing is most suited for random moves, and **ILS** for best or first-improving. It iteratively finds a local minimum and **jumps** far in the landscape, storing the best found solution. The jump should be not easily reversible by local search. A **partial restart solution** doesn't lose all current solution components and is usually better than a **random restart solution**, which tends to be of average quality—the **central limit catastrophe**. All metaheuristics can be seen as trying to improve upon random restart in guiding local search. As such, random restart is a good baseline.

By default set the maximum jumps to 100 and subsequently to as many as patience allows.

```
template<typename PROBLEM> typename PROBLEM::X iteratedLocalSearch(
    PROBLEM& p, typename PROBLEM::X x, int maxBigMoves)
{
    typename PROBLEM::X bestX = x;
    double bestScore = p.getScore(bestX);
    while (maxBigMoves--)
    {
        x = p.localSearchBest(x);
        //update best
        double xScore = p.getScore(x);
        if (xScore < bestScore)
        {
            bestScore = xScore;
            bestX = x;
        }
        p.bigMove(x);
    }
    return bestX;
}
```

ILS is best suited when the local search is a specialized algorithm that has nothing to do with regular local search. The user may need much research to discover a good jump move. A simple alternative of doing a couple of random moves as a restart only leads to an overall algorithm that is worse than simulated annealing. So for simplicity, illustration, and benchmark results, the representation-specific implementations below use random restart as the jump.

For symmetric permutation problems:

```
template<typename EVALUATOR> struct SymmetricPermutationILSFromRandReverseMove
{
    EVALUATOR const& e;
    typedef Vector<int> X;
    int lsMoves;
    SymmetricPermutationILSFromRandReverseMove(EVALUATOR const& theE,
                                                int lsMaxMoves): lsMoves(lsMaxMoves), e(theE) {}
    X localSearchBest(X const& x)
```

```

    {return solveSymmetricPermutationLocalSearchReverse(e, x, lsMoves); }
double getScore(X const& x){return e(x);}
void bigMove(X& x)
    {GlobalRNG().randomPermutation(x.getArray(), x.getSize());}
};

template<typename INSTANCE>
Vector<int> solveSymmetricPermutationIteratedLocalSearch(INSTANCE const&
    instance, Vector<int> const& initial, int lsMaxMoves, int bigMoves)
{
    SymmetricPermutationILSFromRandReverseMove<INSTANCE> move(instance,
        lsMaxMoves/bigMoves);
    return iteratedLocalSearch(move, initial, bigMoves);
}
}

```

For subset problems:

```

template<typename EVALUATOR> struct SubsetILSFromRandFlipMove
{
    EVALUATOR const& e;
    typedef Vector<bool> X;
    int lsMoves;
    SubsetILSFromRandFlipMove(EVALUATOR const& theE, int lsMaxMoves):
        lsMoves(lsMaxMoves), e(theE) {}
    X localSearchBest(X const& x)
        {return solveSubsetLocalSearchFlip(e, x, lsMoves);}
    double getScore(X const& x){return e(x);}
    void bigMove(X& x)//simple restart
        {for(int i = 0; i < x.getSize(); ++i) x[i] = GlobalRNG().mod(2);}
};

template<typename INSTANCE> Vector<bool> solveSubsetIteratedLocalSearch(
    INSTANCE const& instance, Vector<bool> const& initial, int lsMaxMoves,
    int bigMoves)
{
    SubsetILSFromRandFlipMove<INSTANCE> move(instance, lsMaxMoves/bigMoves);
    return iteratedLocalSearch(move, initial, bigMoves);
}
}

```

Can also have an acceptance criteria to reject damaging jumps, but there seems to be little point because their goal is exploration.

16.12 Genetic Algorithms

Genetic algorithms are simple and popular due an early introduction into more accessible literature. But a lot of research and my personal experiments suggest that single-solution local-search-based methods are more scalable, need less from the user, and for typical combinatorial problems give better solutions for the same computational budget. So be forewarned before investing the time in this approach, though your problem may be different. But for continuous optimization (see the “Numerical Optimization” chapter) incremental efficiency of local jumps based on some random displacement disappears, and genetic algorithms are useful. They also seem to have a natural advantage for multiobjective optimization (discussed later in the chapter).

Maintain a set of solutions, and iteratively update it by:

- **Selecting** better solutions
- **Crossing over** of pairs of solutions
- **Mutating** individual solutions

This is inspired by the biological evolution process. The steps are easy for a problem such as knapsack that uses subsets for solution representation, but tricky for TSP with permutation solution representation (but see Gendreau & Potvin 2018). A generic recipe:

- Selection—first select the best solution. Then do **tournament selection**—for every remaining slot take the better of two randomly selected solutions.
- Crossover—this depends on the representation, but for subsets and multidimensional vectors of something a simple solution is **uniform crossover**. Pick some index i . Then for the first “child” take the first part until i from “parent” 1, and second from “parent” 2. Vice versa for the second “child”. One rule of thumb is that “children” should be similar to “parents”, i.e., good properties of the latter

are mostly preserved, like with the jump move of ILS.

- Mutation—modern genetic algorithms use local search instead of a more traditional random move with some small probability. So the resulting algorithm is called “**genetic local search**”. Other common names include “**genetic hybrid**” and “**memetic algorithm**”.

A generic implementation relies on the caller to specify crossover and local search operators.

```
template<typename PROBLEM> pair<typename PROBLEM::X, double>
geneticLocalSearch(PROBLEM const& p, int populationSize, int nLocalMoves,
int maxEvals)
{
    assert(maxEvals > populationSize && populationSize > 1);
    int n = 2 * (populationSize/2) + 1; //must be odd, first is best
    //and the rest evolve in pairs
    Vector<pair<double, typename PROBLEM::X>> population(n),
        populationNew(n);
    for(int i = 0; i < n; ++i) population[i].first =
        p.evaluate(population[i].second = p.generate());
    PairFirstComparator<double, typename PROBLEM::X> c;
    while((maxEvals -= (n - 1) * nLocalMoves) >= 0)
    { //elitism - keep best
        populationNew[0] = population[argMin(population.getArray(), n, c)];
        //tournament selection
        for(int i = 1; i < n; ++i)
        {
            int j = GlobalRNG().mod(n), k = GlobalRNG().mod(n), winner =
                c(population[j], population[k]) ? j : k;
            populationNew[i] = population[winner];
        }
        for(int i = 1; i + 1 < n; i += 2)
        { //crossover picked parents to create children
            p.crossover(populationNew[i].second, populationNew[i + 1].second);
            //local search children and evaluate
            for(int j = 0; j < 2; ++j)
            {
                populationNew[i + j].second =
                    p.localSearch(populationNew[i + j].second, nLocalMoves);
                populationNew[i + j].first =
                    p.evaluate(populationNew[i + j].second);
            }
        }
        population = populationNew;
    }
    int best = argMin(population.getArray(), n, c);
    return make_pair(population[best].second, population[best].first);
}
```

To use this for a subset problem such as knapsack need to specify the operators and decide how to allocate the evaluations. A reasonable strategy:

- If instance size n is large, and incremental moves are $O(1)$, use $O(n)$ local search moves such as $n/5$ to make the time cost about equal to sample generation and evaluation.
- If n is small, or local moves don't have efficient incremental evaluation, use $nMoves^{1/3}$ local moves.
- Population size is $\sqrt{\frac{nMoves}{nLocalMoves}}$. Same for the number of iterations.

```
template<typename FUNCTION> class GLSSubset
{
    FUNCTION const& f;
    int n;
public:
    typedef Vector<bool> X;
    GLSSubset(FUNCTION const& theF, int theN): f(theF), n(theN) {}
    X generate() const
    {
        Vector<bool> subset(n);
```

```

        for(int i = 0; i < n; ++i) subset[i] = GlobalRNG().mod(2);
        return subset;
    }

    void crossover(X& x1, X& x2) const
    { //uniform crossover
        assert(x1.getSize() == x2.getSize());
        for(int k = 0; k < x1.getSize(); ++k) if(GlobalRNG().mod(2))
            swap(x1[k], x2[k]);
    }

    X localSearch(X const& x, int nLocalMoves) const
    { return solveSubsetLocalSearchFlip(f, x, nLocalMoves); }
    double evaluate(X const& x) const{return f(x);}
};

template<typename FUNCTION> Vector<bool>
geneticLocalSearchSubset(FUNCTION const& f, int n,
int maxEvals = 10000000)
{
    assert(maxEvals >= n); //basic sanity check, really need 4 * nLocalMoves
    //based on incremental cost to get same generation and local with large n
    int nLocalMoves = max(n/5, int(pow(maxEvals, 1.0/3))),
    populationSize = int(sqrt(maxEvals * 1.0/nLocalMoves));
    //one generation + evaluation equivalent
    return geneticLocalSearch(GLSSubset<FUNCTION>(f, n), populationSize,
        nLocalMoves, maxEvals).first;
}
}

```

For permutations crossover is more difficult because must end up with valid child permutations. One simple option is **order crossover**: given x_1 and x_2 , put the second half of elements of x_1 in order of x_2 . Likewise, put the first half of elements of x_2 in order of x_1 . This is the basic version with starting position 0 and length $n/2$. A more general version is to make these random, as for uniform crossover. Permutations are the same no matter from which element they are listed, so can implicitly rotate by starting from a non-0 index.

```

template<typename FUNCTION> class GLSPermutation
{
    FUNCTION const& f;
    int n;
    bool isSymmetric;

public:
    typedef Vector<int> X;
    GLSPermutation(FUNCTION const& theF, int theN, bool theIsSymmetric):
        f(theF), n(theN), isSymmetric(theIsSymmetric)
        {assert(theN >= 4); //need 4 for crossover to work}

    X generate() const
    {
        X p(n);
        for(int i = 0; i < n; ++i) p[i] = i;
        GlobalRNG().randomPermutation(p.getArray(), n);
        return p;
    }

    void crossover(X& x1, X& x2) const
    { //order crossover of length in [2, n - 2] from random start rotation
        assert(x1.getSize() == n && x2.getSize() == n);
        int start = GlobalRNG().mod(n), length = 1 + GlobalRNG().mod(n - 3);
        Vector<bool> x1FromX2(n, false), x2FromX1(n, false);
        for(int i = 0; i < n; ++i)
        { //x1 second part from x2; x2 first part from x1
            int j = (start + i) % n; //implicit rotation start from start
            if(i < length) x2FromX1[x2[j]] = true;
            else x1FromX2[x1[j]] = true;
        }
        X x1Copy = x1; //need temp due to overlap
        for(int i = 0, i1 = (start + length) % n; i < n; ++i)
        { //fill x1 second part from x2 in its order
            int element = x2[(start + i) % n]; //iterate over x1 from start
            x1Copy[i] = element;
        }
        x1 = x1Copy;
    }
}

```

```

        if(x1FromX2[element])
        {
            x1[i1] = element;
            i1 = (i1 + 1) % n;//contiguous advance
        }
    }
    for(int i = 0, i2 = start; i < n; ++i)//over x2 from start
    {//fill x2 second part from original x1 in its order
        int element = x1Copy[(start + i) % n];
        if(x2FromX1[element])
        {
            x2[i2] = element;
            i2 = (i2 + 1) % n;//contiguous advance
        }
    }
}
X localSearch(X const& x, int nLocalMoves)const
{//for symmetric refer reverse move
    return isSymmetric ? solveSymmetricPermutationLocalSearchReverse(f, x,
        nLocalMoves) : solvePermutationLocalSearchSwap(f, x, nLocalMoves);
}
double evaluate(X const& x)const{return f(x);}
};

template<typename FUNCTION> Vector<int>
geneticLocalSearchPermutation(FUNCTION const& f, int n, bool isSymmetric,
int maxEvals = 10000000)
{
    assert(maxEvals >= n);//basic sanity check, really need 4 * nLocalMoves
//based on incremental cost to get same generation and local with large n
int nLocalMoves = max(n/5, int(pow(maxEvals, 1.0/3))),  

    populationSize = int(sqrt(maxEvals * 1.0/nLocalMoves));
//one generation + evaluation equivalent
return geneticLocalSearch(GLSPermutation<FUNCTION>(f, n, isSymmetric),
    populationSize, nLocalMoves, maxEvals).first;
}

```

For partition problems the trick is to convert the list of bins to an assignment, which is easily crossed over.

```

template<typename FUNCTION> class GLSPartition
{
    FUNCTION const& f;
    int n;
    void removeEmptyBins(Vector<Vector<int> & partition)const
    {
        for(int i = n - 1; i >= 0; --i) if(partition[i].getsize() == 0)
        {
            partition[i] = partition.lastItem();
            partition.removeLast();
        }
    }
public:
    typedef Vector<Vector<int> > X;
    GLSPartition(FUNCTION const& theF, int theN): f(theF), n(theN) {}
    X generate()const
    {//random assignment
        Vector<Vector<int> > partition(n);
        for(int i = 0; i < n; ++i) partition[GlobalRNG().mod(n)].append(i);
        removeEmptyBins(partition);
        return partition;
    }
    void crossover(X& x1, X& x2)const
    {//convert to assignments
        X* xs[2] = {&x1, &x2};//c++ doesn't allow arrays of references
    }
}

```

```

Vector<int> assignments[2] = {Vector<int>(n, -1), Vector<int>(n, -1)};
for(int k = 0; k < 2; ++k)
{
    X& x = *xs[k];
    for(int bin = 0; bin < x.getSize(); ++bin)
        for(int j = 0; j < x[bin].getSize(); ++j)
        {
            int item = x[bin][j];
            assert(item >= 0 && item < n);
            assignments[k][item] = bin;
        }
    //uniform crossover on assignments
    for(int item = 0; item < n; ++item)
        //check for bad input first--every element must be assigned
        assert(assignments[0][item] != -1 && assignments[1][item] != -1);
        if(GlobalRNG().mod(2))
            swap(assignments[0][item], assignments[1][item]);
    //convert back to bin vectors
    for(int k = 0; k < 2; ++k)
    {
        X& x = *xs[k];
        x = Vector<Vector<int> >(n);
        for(int item = 0; item < n; ++item)
            x[assignments[k][item]].append(item);
        removeEmptyBins(x);
    }
}
X localSearch(X const& x, int nLocalMoves) const
{
    return solvePartitionLocalSearchSwap(f, x, nLocalMoves);
}
double evaluate(X const& x) const{return f(x);}
};

template<typename FUNCTION> Vector<Vector<int> >
geneticLocalSearchPartition(FUNCTION const& f, int n,
int maxEvals = 10000000)
{
    assert(maxEvals >= n); //basic sanity check, really need 4 * nLocalMoves
    //based on incremental cost to get same generation and local with large n
    int nLocalMoves = max(n/5, int(pow(maxEvals, 1.0/3)));
    populationSize = int(sqrt(maxEvals * 1.0/nLocalMoves));
    //one generation + evaluation equivalent
    return geneticLocalSearch(GLSPartition<FUNCTION>(f, n), populationSize,
        nLocalMoves, maxEvals).first;
}

```

Unlike for single-solution algorithms, it's less clear why genetic algorithms do well. The best explanation seems to be that they follow a process similar to parallel ILS. E.g., for a subset problem index order doesn't matter, so uniform crossover essentially permutes the indices in some order and does **single-point crossover**—exchange the first and the second parts of the parents. For permutations only the starting point and length are random, and effectively single-point crossover is applied also. The result seems to be maintaining a diverse population of overall good quality. It's easy to derive an ILS jump from a crossover for the same representation—crossover with a random solution, but with a heavy bias toward the current solution, and optimize for efficiency.

Must maintain a diverse population, otherwise will get a **collapse** to a single solution. Tournament selection is better at it than other methods (Gendreau & Potvin 2018). With local search as mutation operation, collapse happens with probability 1, but with many local moves per mutation and a large population likely run out of budget first.

16.13 Some Performance Results

In these and any other comparisons the goal is to illustrate the application of some common algorithms to the problem and not to present whatever specialized state-of-the-art algorithms are available. Only some of the solvers are used for each problem. Some of the problems are very easy compared to others, and all algorithms perform about equally. The presented random instance generators are used, and only a single run is

used for illustration, so no statistical conclusions are possible. All algorithms are run on the same instance. The ILS solutions, where available, only serve as a baseline of random restart local search.

TSP is not too hard relative to its combinatorial space of all permutations, and simulated annealing with the reverse move is a clear winner.

Problem Size	100
Expected	7.07
Initial	50.01
LS Reversals	8.23
STSA Reversals	7.67
GE Reversals	9.79
ILS Reversals	7.86
LS Swaps	17.31
STSA Swaps	15.79
B&B	8.50
RTAS	8.57

Figure 16.6: Results for TSP.

Knapsack is an easy problem—the greedy algorithm loses only to B&B and very slightly.

Problem Size	1000
Initial	-235.06
Greedy	-405.45
LS Flips	-245.42
STSA Flips	-399.12
Genetic	-397.47
ILS	-271.29
B&B	-405.49

Figure 16.7: Results for knapsack

For 3-SAT, the generated instances seem easy. All metaheuristics get about the same score.

Problem Size	100 1000
Initial	-861
LS Flips	-967
STSA Flips	-970
Genetic	-971
ILS	-970

Figure 16.8: Results for 3-SAT

Bin packing is an easy problem—the greedy approximation algorithm does best.

Problem Size	1000
Initial	1000.00
Greedy	104.0
LS Swaps	109.00
STSA Swaps	111.00
Genetic	124.08

Figure 16.9: Results for bin packing; the genetic solution is infeasible but easily repaired

Genetic local search did worse than simulated annealing on knapsack and TSP, and about the same on 3-SAT and bin packing, given the same number of local moves. But even in this case and generally genetic algorithms are potentially much slower due to the inability to do incremental evaluation. The budget for local search algorithms looks at incremental evaluations and the budget for other algorithms looks at normal evaluations, which isn't exactly a fair comparison but the best available.

16.14 Problem-specific Preprocessing

For some types of problems can analyze a particular problem and reduce its size by using various properties, e.g., symmetries. This needs insight and can't be done automatically. E.g., for specific **train scheduling**, merging common routes, stations, and using other problem-specific information reduced the problem size enough to use brute force (Weihe 2001).

16.15 Multiobjective Optimization

Want to find a solution that optimizes several objectives. E.g., want to minimize time to destination, fuel use, and ticket probability when driving. A decision maker may want to see several possibilities instead of opti-

mizing some function of the objectives and getting a single answer.

Any solution not worse than any other in all objectives is optimal, unless strictly worse in some and equal in others. Optimal solutions form a possibly-infinitely-large **Pareto-optimal front**. It's harder to manage than a single solution. So it's best to optimize some utility function, such as a linear combination of the desired objectives, with logical weights that consider one objective at a time, weigh all equally, or require certain ones to have values from some range, etc. See Talbi (2009) and references therein for many more methods and concepts, particularly algorithms for working with all objectives at the same time.

16.16 Constraint Processing

For some optimization problems, solutions consist of variables with values \in some integral domain. A **constraint** is a set of disallowed value tuples. E.g., a Sudoku solution consists of 81 variables $\in [1, 9]$, and each box, row, and column must contain all the numbers. A solution consists of value assignments that don't violate the constraints. If \nexists a solution, the problem is **unsatisfiable**. Constraints that involve two variables are **binary**—the most efficient to represent and work with.

The simplest way to represent variables and allowed values is a vector of bit sets. Each variable is associated with a bit set of size = its domain, and bit i is set if the i^{th} value in the domain is allowed. The simplest way to represent binary constraints is an undirected graph with vertices corresponding to variables and edges to constraints between the involved vertices. The edge data has a constraint functor that checks if a value is allowed.

```
template<typename CONSTRAINT> struct ConstraintGraph
{
    typedef GraphAA<CONSTRAINT> GRAPH;
    GRAPH g;
    Vector<Bitset<>> variables;
    void addVariable(int domain)
    {
        g.addVertex();
        variables.append(Bitset<>(domain));
    }
    void addConstraint(int v1, int v2, CONSTRAINT const& constraint)
    {
        assert(v1 != v2);
        g.addUndirectedEdge(v1, v2, constraint);
    }
    void disallow(int variable, int value)
    {
        variables[variable].set(value, false);
    }
    bool hasSolution(int variable) {return !variables[variable].isZero();}
};
```

A value is allowed if any other variable can take on ≥ 1 values such that the pair of values doesn't violate the constraint between them:

```
bool isAllowed(int variable, int value, int otherVariable,
               CONSTRAINT const& constraint)
{
    for(int i = 0; i < variables[otherVariable].getSize(); ++i)
        if(variables[otherVariable][i] && constraint.isAllowed(variable,
                                                               value, otherVariable, i)) return true;
    return false;
}
```

To check a variable against another and **revise** its domain by removing disallowed values, check every set value. Let $d =$ the maximum domain size of any variable. Then the runtime of revising is $O(d^2)$.

```
bool revise(int variable, int otherVariable, CONSTRAINT const& constraint)
{
    bool changed = false;
    for(int i = 0; i < variables[variable].getSize(); ++i)
        if(variables[variable][i] &&
            !isAllowed(variable, i, otherVariable, constraint))
        {
            disallow(variable, i);
            changed = true;
        }
}
```

```

    return changed;
}

```

Revising every variable against every other variable with which it has a constraint until stop making revisions gives a correct final solution. **Simplified AC3 (SAC3)** makes this more efficient using the fact that after a revision a variable isn't revisable until its neighbors are revised:

1. Revise every variable with respect to its every neighbor, and enqueue it
2. Until the queue is empty
3. Dequeue a variable
4. \forall neighbor variable
5. Revise it with respect to the variable
6. Enqueue if revised

Can use any list data structure instead of a queue. Let n = the number of variables, and c = the number of constraints. Then each pass makes expected c/n and worst-case n revise calls. Because each variable can be revised $\leq d$ times, the worst-case runtime of the algorithm is $O(n^2d^3)$. The best case is $O(cd^2)$ when step (2) isn't needed, and the average is in-between, depending on the problem, or the same if the number of queue passes is $O(1)$.

```

bool SAC3Helper(int v, Queue<int>& q, Vector<bool>& onQ, bool isFirstPass)
{
    onQ[v] = false;
    for(typename GRAPH::AdjacencyIterator i = g.begin(v);
        i != g.end(v); ++i)
        //in the first pass revise the variables and in subsequent passes
        //the neighbors
        int revisee = i.to(), against = v;
        if(isFirstPass) swap(revisee, against);
        if(revise(revisee, against, i.data()))
        {
            if(!hasSolution(revisee)) return false; //problem unsatisfiable
            if(!onQ[revisee]) q.push(revisee);
            onQ[revisee] = true;
        }
    }
    return true;
}
bool SAC3()
{
    Queue<int> q;
    Vector<bool> onQ(g.nVertices(), true);
    for(int j = 0; j < g.nVertices(); ++j)
        if(!SAC3Helper(j, q, onQ, true)) return false;
    while(!q.isEmpty())
        if(!SAC3Helper(q.pop(), q, onQ, false)) return false;
    return true;
}

```

Can create a binary constraint graph from higher-order constraints as an approximation, by checking only one variable against the rest. E.g., the **AllDifferent constraint** requires a subset of variables to take on different values. It allows, e.g., to model Sudoku as a binary constraint problem:

```

struct AllDifferent
{
    LinearProbingHashTable<int, bool> variables;
    void addVariable(int variable) {variables.insert(variable, true);}
    struct Handle
    {
        LinearProbingHashTable<int, bool>& variables;
        bool isAllowed(int variable, int value, int variable2, int value2)
        const
        {
            if(variables.find(variable) && variables.find(variable2))
                return value != value2;
            return true;
        }
    }
}

```

```

    }
    Handle<LinearProbingHashTable<int, bool>& theVariables):
        variables(theVariables) {}
    } handle;
    AllDifferent(): handle(variables) {}
};

Use it to model Sudoku as follows:

```

```

struct Sudoku
{
    AllDifferent ad[3][9];
    ConstraintGraph<AllDifferent::Handle> cg;
    Sudoku(int values[81])
    {
        for(int i = 0; i < 81; ++i)
        {
            cg.addVariable(9);
            if(values[i])
            {
                cg.variables[i].setAll(false);
                cg.variables[i].set(values[i] - 1, true);
            }
            else cg.variables[i].setAll(true);
        }
        for(int i = 0; i < 9; ++i)
        {
            int rowStart = i * 9, columnStart = i,
                boxStart = i/3 * 27 + (i % 3) * 3;
            for(int j = 0; j < 9; ++j)
            {
                int rowMember = rowStart + j;
                int columnMember = columnStart + j*9;
                int boxMember = boxStart + j/3 * 9 + j % 3;
                ad[0][i].addVariable(rowMember);
                ad[1][i].addVariable(columnMember);
                ad[2][i].addVariable(boxMember);
                if(j == 8) continue;
                for(int k = j+1; k < 9; ++k)
                {
                    int boxMember2 = boxStart + k/3 * 9 + k % 3;
                    cg.addConstraint(rowMember, rowStart + k, ad[0][i].handle);
                    cg.addConstraint(columnMember, columnStart + k * 9,
                        ad[1][i].handle);
                    cg.addConstraint(boxMember, boxMember2, ad[2][i].handle);
                }
            }
        }
        cg.SAC3();
    }

void printSolution() const
{
    for(int i = 0; i < 81; ++i)
    {
        if(i % 9 == 0) cout << '\n';
        int count = 0, value = -1;
        for(int j = 0; j < 9; ++j)
        {
            if(cg.variables[i][j])
            {
                ++count;
                value = j + 1;
            }
        }
    }
}

```

```

    if(count > 1) cout << "x";
    else cout << value;
}
cout << endl;
};

}

```

SAC3 correctly eliminates values that violate the binary subconstraints but not the ones that don't and violate higher-order-tuple constraints. In this case can try B&B or backtracking that uses SAC3 to check the current value assignment, reduce domains of not-yet-selected variables, and select the next variable as the one with the smallest domain to maximize pruning. E.g., pure SAC3 can solve an easy Sudoku puzzle but makes no progress on medium and hard ones:

```

int easy[] =
{
    2,0,1,7,5,0,0,0,
    0,0,9,0,0,8,0,1,0,
    0,0,0,3,0,1,0,0,7,
    0,2,0,5,0,0,1,7,8,
    0,8,0,0,0,0,0,9,0,
    1,5,7,0,0,4,0,3,0,
    6,0,0,8,0,2,0,0,0,
    0,9,0,6,0,0,5,0,0,
    0,0,0,1,7,9,3,0,6
};

int medium[] =
{
    0,0,5,0,0,3,2,9,0,
    9,0,0,2,0,0,0,3,4,
    0,0,0,0,1,0,5,0,0,
    0,0,0,0,9,0,0,7,1,
    0,0,0,5,0,5,0,0,0,
    7,3,0,0,2,0,0,0,0,
    0,0,7,0,6,0,0,0,0,
    6,8,0,0,0,9,0,0,2,
    0,5,2,8,0,0,6,0,0
};

int hard[] =
{
    0,1,2,0,6,0,8,0,0,
    0,0,0,0,3,0,0,5,0,
    6,0,0,4,0,0,0,0,7,
    0,0,6,0,0,0,0,1,0,
    0,9,7,0,0,0,6,4,0,
    0,8,0,0,0,0,7,0,0,
    8,0,0,0,0,1,0,0,3,
    0,4,0,0,5,0,0,0,0,
    0,0,1,0,2,0,9,7,0
};

```

Figure 16.10: SAC3 final solutions of Sudoku puzzles of varying difficulty

16.17 Stochastic Problems

For problems where the objective function is noisy and want to find a solution with the best $E[\text{quality}]$, sample average/path approximation methods (see the “Numerical Optimization” chapter) are effective.

The presented algorithms aren't designed to handle noise, though some do so well. E.g., for simulated annealing, deciding if the generated solution is better using a single comparison is technically invalid because of the noise, but can still use it as is for small noise levels relative to the current solution's quality. For larger ones local search is essentially a random walk.

16.18 General Advice

The **no free lunch theorem (NFL)** (Burke & Kendall 2013) states that over all possible problem instances no algorithm does better than random search, ending discussion about which particular method is best in all cases. The main idea is that knowing local information is equally likely to be helpful or harmful over all problems (same for the NFL for learning; see the “General Machine Learning” chapter). But in practice rarely solve problems where local information is harmful. It can be only a little useful—at least want nearby solutions to be of similar enough quality. Also \forall problem some algorithms exploit problem-specific information better than others, and vice versa for other problems.

No algorithm is effective where all solutions except the best have similar quality. Some conclusions of the NFL and general experience:

- Use problem-specific information. Good lower bounds and local move types are what makes an algorithm efficient by allowing it to not consider low-quality solutions. Only such information gives an advantage over brute-force enumeration. Incremental evaluation and efficient problem representation also make a big difference.
- Efficient algorithms have a low modeling overhead. E.g., for the TSP B&B is better than A*.
- Well-designed algorithms eventually find the solution with probability 1, like random search, at least even under some unrealistic conditions. This tends to prevent wastefulness. The proofs are usually useful only as “moral support”, and detailed theory beyond that generally isn’t useful unless of help in making implementation decisions.
- Metaheuristics that enforce feasibility might not work well for problems with black-box constraints (not related to constraint processing), and even a feasible starting solution may be hard to find. Gradient-providing penalization is the best black-box strategy here.
- For very large instances usually only greedy algorithms and local search are feasible.
- Many problems have **phase transitions**, where instances randomly generated with certain parameters are easy, but adjusting the parameters beyond some thresholds creates practically unsolvable problems with ineffective problem-specific information.
- Many well-studied problems have complicated solution methods that outperform generic ones. E.g., for the TSP, **Kernighan-Lin heuristic** can solve million-point instances very well, and instances with < 10000 points can often be solved provably optimally by complicated algorithms (Applegate et al. 2007), though in both cases with considerable runtime.
- Some problems are known to be too difficult. E.g., for the **quadratic assignment problem**, which is a permutation problem, only very small instances have been solved exactly. Such problems are effectively a constructive proof of the NFL. When an algorithm claims to have done better on some instance than the best known solution, this doesn’t mean that the difference in the improvement is large or that it will do well on other common problems.

16.19 Implementation Notes

Most implementations have much originality, at least with the API choice, which was finalized only when applying the algorithms to various problems with different properties.

For exact algorithms the only novelty is the use of an indexed heap for A*.

For metaheuristics, the implementation of simulated annealing is original and took much experimentation to finalize all the details. For genetic algorithms the novelty is the formula for the population size. Otherwise follow the textbooks, though need careful selection due to a variety of choices.

Constraint processing functionality is fairly limited, though the choice of the main algorithm is original.

16.20 Comments

The estimation of T_{last} for simulated annealing is original, as is using T_{cap} . Without such a procedure any guess of the T range must be arbitrary, and give up control over small changes. Another approach based on neighborhood size is mentioned in van Laarhoven & Aarts (1987), but it seems to be clumsy and not beneficial for the user to give this information. Consult this reference for an insightful summary of the early attempts to make implementation decisions, but beware that they are based on the version where many moves are made at the same T value.

Simulated annealing seems to be easy to improve. The nonparametric option is one possibility. A commonly suggested variant is **threshold accepting**—assume the exponential variate is always 1. But it fails completely for problems where Δf is always the same, and good logical parameter tuning is impossible, unlike for probabilistic variants. Its only but important achievement is introducing the modern version of

simulated annealing with a single move at each T . Nonmonotonic temperature decrease schedules are also a possibility, with the idea to increase T if no progress is being made. I won't explore this further because the actual algorithms with all the details filled in aren't popular, but see the references in Talbi (2009) if curious.

The literature on genetic algorithms is very large. Talbi (2009) and Gendreau & Potvin (2018) are good entry points. Burke & Kendall (2013) and Schneider & Kirkpatrick (2006) are also useful. Early genetic algorithms dogmatically stuck to random mutations instead of local search. But even in nature, if local search replaced mutation, we would be superhuman.

Many other metaheuristics have been proposed (see Chopard & Tomassini 2018 for basic introductions, and Talbi 2009 and Gendreau & Potvin 2018 for details). Those I haven't discussed I ruled out based on the mentioned general principles. But some need further comment and are discussed next.

Tabu search—tweak local search to disallow certain solutions by remembering attributes of visited solutions or made moves. Use memory and not randomization to escape local optima. Some apparent flaws:

- It's unclear how to implement this generically. A simple suggestion is for each representation to store a list of reverse moves. But this only seems to be effective with best or first-improving moves, and a small-size list isn't enough to escape from local minima of large neighborhoods. E.g., with large n and a quadratic-size neighborhood even local search won't try all possible moves. But randomness always works. By the same reasoning don't have a clear extension to continuous problems without some explicit discretization.
- It doesn't work well with random moves, so won't scale to large n .
- For problems where the method is claimed to be effective, often with state-of-the-art performance, it's only so after substantial expert tuning. So it seems to be more of a method for experts that doesn't come with a self-tuned version. E.g., the method can get into cycles, which need another mechanism to fix (see Chopard & Tomassini 2018 for an example).
- It's hard to derive convergence properties.

Estimation of distribution algorithms—similar to the EM algorithm (see the "Machine Learning—Clustering" chapter).

1. Create a probabilistic model for generating solutions
2. Until convergence
3. Generate a population from the model
4. Do local search on each population member
5. Reestimate the parameters of the model from the best population members

This may seem appealing, at least because can't have population collapse, but the use of a model isn't free:

- Even for a subset representation where the model is a vector of Bernoulli parameters, don't model interdependence.
- For a permutation problem such as the TSP need $O(n^2)$ memory to store all pairs of distances, and operate along the lines of something like the **nearest neighbor construction heuristic** (start with any city, and go next to the closest unvisited one).
- For a partition problem representation is unclear (nor discussed in any of the main introductions). E.g., a list of possible partitions also leads to $O(n^2)$ memory and breaks down because renumbering the partitions doesn't change the solution but reassigned the distributions.

The most natural algorithm in this category is the **cross-entropy method** (Rubinstein & Kroese 2004). As is, it doesn't use local search and suggests a large, nonscalable choice of population size, but these are easily adjusted. **Ant colony optimization**, a more popular method but with different motivation, mimics the behavior of ants choosing next steps (Dorigo & Stützle 2004). It has more parameters, but allows incremental updates by generating one new solution at a time. Far more algorithmic tuning went into it than into cross-entropy method, but all of it seems easily transferable.

An interesting way to see a genetic algorithm is as an estimation-of-distribution algorithm with a nonparametric representation of the model. The crossover effectively does both the model update and the generation of a new population in a single step. It might seem that a good idea is to have a nonparametric model by maintaining a population and using it directly instead of the model, but any simple use is equivalent to having a model, and complicated use by, e.g., trying to match components doesn't work without excessive clumsiness. Genetic crossover seems to be the about only reasonable way to use a population directly.

Variable-neighborhood search uses several move types for local search. The basic algorithm uses moves from a large neighborhood when in a local optimum from a smaller, usually contained neighborhood.

It's clumsy to the user by requiring to define several types of moves, and doesn't work well with random moves because don't know when to switch. A version where do several steps of a move as a composite move needs a way to undo a move, specifying which is also clumsy for the user.

GRASP modifies a greedy algorithm for the problem to use randomness, e.g., by selecting the next component with probability proportional to the value or the rank of possible next step. The result is improved by local search. This is repeated, and a more advanced version keeps a set of good solutions and uses **path relinking**—i.e., explore the path between the constructed solution and a random one from the set (Resende & Ribeiro 2016). But:

- The basic version is hardly better than random restarting, except for something like knapsack where the greedy algorithm is very good
- The advanced version seems to do a clumsy genetic crossover—e.g., it's simple to implement for subsets, but not for permutations

Parejo et al. (2012) comprehensively review several implementations of metaheuristics. But these are mostly frameworks for researchers by researchers and not ready solutions for the users. E.g., based on a quick look at some of them, the code tends to have silly organization, with excessive object orientation and do-everything classes, and many methods and settings are included that shouldn't be used in practice.

16.21 Projects

- For maximum satisfiability implement a greedy algorithm that iteratively picks a variable and sets its value to what will satisfy most clauses. Can cycle through clauses in random order until make no change.
- Implement B&B for bin packing as an extension of the greedy first-fit decreasing. A simple bound function is to divide the remaining weight by the bin size.
- Consider various restarting strategies for A*. An obvious one is to complete the solution using real-time A*. A more complicated one is to use the partial solution as the starting solution for next round of A*. Repeat this until construct a complete solution. At least one move must be made per iteration, but usually several, depending on the branching factor and the memory limit. Study the quality of this approximation.
- For A* a simple way to save memory is to use the quality of an existing approximate solution as a cut-off bound and not enqueue moves whose lower bounds exceed it. Try this to see how much memory is saved. E.g., can use a metaheuristic first.
- Experiment with realtime A* to discover its usefulness as a greedy construction algorithm. Try it with TSP, followed by local search. Does this do better than simulated annealing?
- Make RBFS nonrecursive by also storing {Heap, alternative, path cost} in the move stack.
- Try the adjacent element swap move for the TSP. Here the neighborhood size is linear and not quadratic, unlike for other moves.
- Implement local search moves for combinations. Define simulated annealing and genetic algorithms based on this (the latter also needs a crossover). Find a problem to test them on.
- Implement first-improving moves in some of the presented common representations. One approach is to get a list of moves and randomize the order. A special local search would do that, and its termination criteria would be absence of improving moves. With moves of 0 quality be careful of cycling. How does it compare with random moves for small problems? What about for large ones? Does it make sense to start with random moves and switch to a move list when many random moves give no improvement? A more efficient approach is to use a move iterator. This avoids generating all moves, but prevents random order, so might have bias. At best can randomly choose iteration direction or use something like Gray code enumeration with the iterators to get some randomness. Both first-improving and best moves are friendly to cheap evaluation with expensive move, such as reversals for the TSP.
- Research specialized moves for the TSP such as those based on multiple reversals. Can these be extended to general strategies such as ILS?
- Try simulated annealing with a fat-tail distribution such as Levy. Use the same logic to pick parameters—i.e., by computing the corresponding probabilities. Also keep the best solution for extra security. Try this for numerical problems also (see the “Numerical Optimization” chapter).
- Investigate simulated annealing with relative quality change—does this allow to simplify parameter setting? Does performance improve?
- Implement tabu search for common problem representations using best or first-improving moves. Experiment on small and large problems to compare against simulated annealing.

- Study acceptance probability of simulated annealing on any problem using a sliding window of maybe 1000 moves. Looks at move value relative to current T . Does the probability evolution for both large-value and small-value moves look reasonable?
- Implement random search and random-restart local search as baseline methods. ILS that serves as the latter is only missing for partition problems. Do some experiments to show that the presented metaheuristics give an improvement over these.
- Use the presented algorithms to solve some other problems such as Rubik's cube. A few good ones are from chess:
 - Place n queens on an $n \times n$ board so that they don't attack each other
 - Find the smallest number of knights to cover an $n \times n$ board

16.22 References

- Applegate, D. L., Bixby, R. E., Chvatal, V., & Cook, W. J. (2007). *The Traveling Salesman Problem: a Computational Study*. Princeton University Press.
- Burke, E. K., & Kendall, G. (Eds.). (2013). *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer.
- Chopard, B., & Tomassini, M. (2018). *An Introduction to Metaheuristics for Optimization*. Springer.
- Dorigo, M., & Stützle , T. (2004). *Ant Colony Optimization*. MIT Press.
- Gendreau, M., & Potvin, J. Y. (Eds.). (2018). *Handbook of Metaheuristics*. Springer.
- Gonzalez, T. F. (2018). *Handbook of Approximation Algorithms and Metaheuristics*. CRC.
- Kopec, D., Pileggi, C., Ungar, D., & Shetty, S. (2016). *Artificial Intelligence and Problem Solving*. Stylus Publishing, LLC.
- Michaels, W., Aarts, E., & Korst, J. (2007). *Theoretical Aspects of Local Search*. Springer.
- Parejo, J. A., Ruiz-Cortés, A., Lozano, S., & Fernandez, P. (2012). Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3), 527–561.
- Resende, M. G., & Ribeiro, C. C. (2016). *Optimization by GRASP*. Springer.
- Rubinstein, R. Y., & Kroese, D. P. (2004). *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Springer.
- Russell, S. J., Norvig, P. (2020). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- Salamon, P., Sibani, P., & Frost, R. (2002). *Facts, Conjectures, and Improvements for Simulated Annealing*. SIAM.
- Sipser, M. (2013). *Introduction to the Theory of Computation*. Cengage Learning.
- Schneider, J., & Kirkpatrick, S. (2006). *Stochastic Optimization*. Springer.
- Talbi, E. G. (2009). *Metaheuristics: from Design to Implementation*. Wiley.
- van Laarhoven, P. J., & Aarts, E. H. (1987). *Simulated annealing: Theory and applications*. Springer.
- Vazirani, V. V. (2004). *Approximation Algorithms*. Springer.
- Weihe, K. (2001). On the differences between “practical” and “applied”. In *Algorithm Engineering* (pp. 1–10). Springer.
- Wikipedia (2019). Coupon collector's problem. https://en.wikipedia.org/wiki/Coupon_collector's_problem. Accessed November 16, 2019.

17 Large Numbers

17.1 Introduction

Need arbitrary-precision arithmetic when the built-in types aren't enough—e.g., for cryptography and scientific computation. A simple implementation of a large number class is presented.

Some main themes:

- Most algorithms are a formalization of high school math
- Operation complexity is measured in terms of the number of digits of the inputs

17.2 Representation

A number consists of a sign and a vector of “digits” in some base. For unambiguous, simple, and efficient representation:

- The least significant digit is at index 0.
- Let $w \leq$ the largest available word size, assumed to be `unsigned long long`. The base $B = 2^{w/2}$ because the product of two $w/2$ bit numbers fits into a w -bit word. Algorithms assume that B is a power of two. Use `uint_32` as the largest compatible base word.
- No leading 0's, except when the number = 0.
- -0 and 0 are allowed—simpler than correcting -0 after every operation.

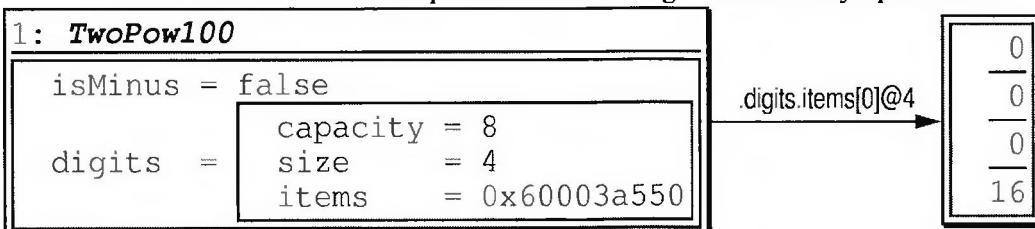


Figure 17.1: Memory layout of a large number representing 2^{100}

```
class Number : public ArithmeticType<Number>
{
    bool isMinus;
    typedef uint32_t DIGIT;
    typedef unsigned long long LARGE_DIGIT;
    enum(BASE_RADIX = numeric_limits<DIGIT>::digits);
    Vector<DIGIT> digits;
    DIGIT getDigit(int i) const { return i < nDigits() ? digits[i] : 0; }
    Number(int size, DIGIT fill) : digits(size, fill), isMinus(false) {}
    void trim() { while(nDigits() > 1 && isZero()) digits.removeLast(); }

public:
    typedef DIGIT DIGIT_TYPE;
    int nDigits() const { return digits.getSize(); }
    DIGIT& operator[](unsigned int i) { return digits[i]; }
    DIGIT const& operator[](unsigned int i) const { return digits[i]; }
    void appendDigit(DIGIT const& digit) { digits.append(digit); }
    bool isZero() const { return digits.lastItem() == 0; }
    bool isPositive() const { return !isMinus && !isZero(); }
    bool isNegative() const { return isMinus && !isZero(); }
    void negate() { isMinus = !isMinus; }
    Number abs() const { return isMinus ? -*this : *this; }
    bool isOdd() const { return digits[0] % 2; }
    bool isEven() const { return !isOdd(); }
    Number(): isMinus(false), digits(1, 0) {} // default is 0
    // convenience constructor for single-digit numbers
    explicit Number(long long x): isMinus(x < 0), digits(1, std::abs(x))
        (assert(std::abs(x) <= numeric_limits<DIGIT_TYPE>::max())); }
```

Additional constructors come from decimal digit conversion (discussed later in the chapter). The comparisons handle also allow absolute value comparing.

```

bool absLess(Number const& rhs) const
{ //more digits larger, else MSD digit decides
    if(nDigits() != rhs.nDigits()) return nDigits() < rhs.nDigits();
    for(int i = nDigits() - 1; i >= 0; --i)
        if(digits[i] != rhs[i]) return digits[i] < rhs[i];
    return false;
}
bool absEqual(Number const& rhs) const
{ //more digits larger, else MSD digit decides
    if(nDigits() != rhs.nDigits()) return false;
    for(int i = 0; i < nDigits(); ++i)
        if(digits[i] != rhs[i]) return false;
    return true;
}
bool operator<(Number const& rhs) const //handle 0 == -0
{ return (isMinus && !rhs.isMinus && !isZero()) || absLess(rhs); }
bool operator==(Number const& rhs) const //handle 0 == -0
{ return (isMinus == rhs.isMinus && isZero()) && absEqual(rhs); }

```

17.3 Addition and Subtraction

A **full adder** adds digits a , b , and $carry$ and returns the sum and the new carry. E.g., $7 + 8$ with carry 0 is 5 with carry 1. Using it add two digit vectors digit by digit, propagating the carry.

$$\begin{array}{r}
 a \quad 048 \\
 + \\
 b \quad 057 \\
 \hline
 \text{sum} \quad 105 \\
 \text{carry} \quad 0110
 \end{array}$$

Figure 17.2: Example of carry propagation in addition

The result is at most one digit longer than either addend and trimmed if that digit = 0.

```

static DIGIT fullAdder(DIGIT a, DIGIT b, bool& carry)
{
    LARGE_DIGIT sum = LARGE_DIGIT(a) + b + carry;
    carry = sum >> BASE_RADIX;
    return sum;
}
static Number add(Number const& a, Number const& b)
{ //O(|a|+|b|)
    int n = max(a.nDigits(), b.nDigits());
    Number result(n + 1, 0);
    bool carry = 0;
    for(int i = 0; i < n; ++i)
        result[i] = fullAdder(a.getDigit(i), b.getDigit(i), carry);
    result[n] = carry;
    result.trim();
    return result;
}

```

Subtraction of two positive numbers assumes that the first is greater and places the result in it. It propagates the carry like addition from least to most significant digits. This differs from the high school left-to-right.

$$\begin{array}{r}
 a \quad 105 \\
 - \quad 057 \\
 \hline
 result \quad 048 \\
 carry \quad 0110
 \end{array}$$

Figure 17.3: Example of carry propagation in subtraction

```

static void sub(Number& a, Number const& b)
{//O(|a| + |b|)
    bool carry = 0;
    for(int i = 0; i < a.nDigits(); ++i)
    {
        LARGE_DIGIT digit = LARGE_DIGIT(b.getDigit(i)) + carry;
        carry = a[i] < digit;
        a[i] -= digit; //implicitly mod B
    }
    a.trim();
}

```

Operators “+” and “-” are implemented on top of these functions and look at the sign:

```

Number operator-() const
{
    Number result = *this;
    result.negate();
    return result;
}
Number& operator+=(Number const& rhs)
{
    if(isMinus == rhs.isMinus)
        //if same sign add
        digits = add(*this, rhs).digits;
        return *this;
    }
    else return *this -= -rhs; //else subtract
}

Number& operator-=(Number const& rhs)
{
    if(isMinus == rhs.isMinus)
        //if same sign subtract
        if(absLess(rhs))
        {
            Number temp = rhs;
            sub(temp, *this);
            temp.negate();
            *this = temp;
        }
        else sub(*this, rhs);
        trim();
        return *this;
    }
    else return *this += -rhs; //else add
}

```

17.4 Shifts

Shifts multiply or divide by a power of two, shifting words and bits separately for efficiency, like for bit set. The runtime is $O(n)$.

```

Number& operator>>=(unsigned int k)
{

```

```

int skipDigits = k/BASE_RADIX, last = nDigits() - skipDigits,
    skipBits = k % BASE_RADIX;
if(skipDigits > 0) //first apply whole-digit shifts
    for(int i = 0; i < nDigits(); ++i)
        digits[i] = i < last ? digits[i + skipDigits] : 0;
if(skipBits > 0) //then bit shifts
{
    DIGIT carry = 0, tempCarry;
    for(int i = last - 1; i >= 0; --i)
    {
        tempCarry = digits[i] << (BASE_RADIX - skipBits);
        digits[i] = (digits[i] >> skipBits) | carry;
        carry = tempCarry;
    }
}
trim(); //in case introduced 0's
return *this;
}
Number operator<=(unsigned int k)
{//first make space for extra digits
    int skipDigits = k/BASE_RADIX, skipBits = k % BASE_RADIX;
    for(int i = 0; i < skipDigits + 1; ++i) digits.append(0);
    if(skipDigits > 0) //apply whole-digit shifts
        for(int i = nDigits() - 1; i >= 0; --i)
            digits[i] = i < skipDigits ? 0 : digits[i - skipDigits];
    if(skipBits > 0) //then bit shifts
    {
        DIGIT carry = 0;
        for(int i = skipDigits; i < nDigits(); ++i)
        {
            DIGIT tempCarry = digits[i] >> (BASE_RADIX - skipBits);
            digits[i] = (digits[i] << skipBits) | carry;
            carry = tempCarry;
        }
    }
    trim(); //in case bit shift not large enough to fill in all the space
    return *this;
}
}

```

17.5 Multiplication

Use the $O(|a||b|)$ high school method. Reduce number \times number to a sequence of number \times digit. The result of the latter is at most one digit larger than the number. E.g., $98 \times 99 = 99 \times 8 + (99 \times 9) \ll_{10} 1$. Also, 9×9 is 1 with carry 8 and 99×9 is computed digit-by-digit to get $011 + 880 = 891$. Implementing some version of which with string-of-digits number representations is a popular interview question.

```

static DIGIT digitMult(DIGIT a, DIGIT b, DIGIT& carry)
{
    LARGE_DIGIT prod = LARGE_DIGIT(a) * b;
    carry = prod >> BASE_RADIX;
    return prod;
}
static Number mult(Number const& a, DIGIT const& b)
{
    Number result(a.nDigits() + 1, 0);
    bool addCarry = 0;
    DIGIT multCarry = 0, newMultCarry;
    for(int i = 0; i < a.nDigits(); ++i)
    {
        result[i] = fullAdder(digitMult(a[i], b, newMultCarry), multCarry,
            addCarry);
        multCarry = newMultCarry;
    }
}

```

```

        result[a.nDigits()] = fullAdder(0, multCarry, addCarry);
        result.trim();
        return result;
    }

Number& operator*=(Number const& rhs)
{ // O(|a| * |b|); multiply by one digit at a time
    Number const& a = *this, b = rhs;
    Number product(a.nDigits() + b.nDigits(), 0);
    for(int j = 0; j < b.nDigits(); ++j)
        product += mult(a, b[j]) << BASE_RADIX * j;
    product.isMinus = a.isMinus != b.isMinus; // negative if different signs
    product.trim();
    return *this = product;
}

```

17.6 Division

a/b computes the quotient q and the remainder r using long division. Assuming both positive:

1. Set $r = a$
2. Normalize r and b to make MSD of $b \geq \frac{B}{2}$
3. While $r < b$
4. Find $p =$ the largest power of base B such that the result $s = pb \leq r$
5. Find max k such that $ks \leq r$
6. $r = rs$
7. $q += kp$
8. Renormalize r

It may seem that (5) needs binary search, but can do it with $O(1)$ multiplications. Theorem (Brent & Zimmermann 2011): Let z be the most significant digit of s, x and y the two most significant digits of r . Form

$$\text{guess } g = \begin{cases} 1, & \text{if } nDigits(r) = nDigits(s) \\ \frac{xB+y}{z}, & \text{otherwise} \end{cases} \text{. Then } k \leq g \leq k+2.$$

Proof: Because b is normalized, $z \geq \frac{B}{2}$. When the number of digits is the same, $k = 1$ because $z \leq x \leq B-1$. Otherwise, $g - k$ is smallest (the first case) or largest (the second case) if the remaining digits of a are largest and of b smallest in the first case, vice versa in the second, and z is smallest. In the first case due to the extra digits of a , $g - k \leq \frac{1}{z} = 0$. In the second, $k = \frac{xB+y}{z+1}$, $gz + r_1 = k(z+1) + r_2$, and $g = k + \frac{k+r_2-r_1}{z} \leq k+2$ because $\frac{k}{z} < 2$ and $r_2 - r_1 < z$. \square

So division first shifts r and b to make $z \geq \frac{B}{2}$ and shifts back r after it's calculated.

```

static DIGIT findK(Number const& s, Number const& b)
{ // O(|a|), find k such that 0 <= k < BASE and kb <= s < (k + 1)b;
    DIGIT guess = s.digits.lastItem() / b.digits.lastItem();
    if(s.nDigits() > b.nDigits()) guess = (s.digits.lastItem() *
        (1ull << BASE_RADIX) + s[s.nDigits() - 2]) / b.digits.lastItem();
    while(s < mult(b, guess)) --guess; // executes <= 2 times
    return guess;
}

static Number divide(Number const& a, Number const& b1, Number& q)
{ // O(|a| * |b|)
    assert(!b1.isZero());
    q = 0;
    Number b = b1.abs(), r = a.abs(); // first normalize
    int norm = BASE_RADIX - lgFloor(b.digits.lastItem()) - 1;
    r <<= norm;
    b <<= norm;
    for(int i = r.nDigits() - b.nDigits(); i >= 0; --i)

```

```

    {
        int shift = i * BASE_RADIX;
        Number s = b << shift;
        DIGIT k = findK(r, s);
        q += mult(Number(1) << shift, k); //q += pk
        r -= mult(s, k);
        //both q and r negative if different signs
        q.isMinus = r.isMinus = a.isMinus != b1.isMinus;
        return r >>= norm; //renormalize
    }
    Number& operator%=(Number const& rhs)
    {
        Number quotient(0);
        return *this = divide(*this, rhs, quotient);
    }
    Number& operator/=(Number const& rhs)
    {
        Number quotient(0);
        divide(*this, rhs, quotient);
        return *this = quotient;
    }
}

```

Need $O(|a||b|)$ time, as for multiplication.

17.7 Conversion to Decimal

Conversion to another base B_2 reduces to division and has the same complexity. Repeatedly divide by B_2 and store the moduli, then reverse them. In practice only want to convert to decimal and use a string as the result, so add the minus sign if needed. This is a common interview question for integer numbers.

```

string toDecimalString() const
{
    string result;
    Number r = *this;
    while(!r.isZero())
    {
        Number q(0);
        result.push_back('0' + divide(r, Number(10), q)[0]);
        r = q;
    }
    if(isMinus) result.push_back('-');
    reverse(result.begin(), result.end());
    return result;
}
Number(string const& decimals)
{
    assert(decimals.length() > 0);
    int firstDigit = 0;
    if(decimals[0] == '-') //only accept n-dash for minus
    {
        assert(decimals.length() > 1);
        isMinus = true;
        firstDigit = 1;
    }
    Number result(0);
    for(int i = firstDigit; i < decimals.length(); ++i)
    {
        if(i == firstDigit) assert(decimals[i] != '0'); //disallow MSD = 0
        else result *= Number(10);
        assert(decimals[i] >= '0' && decimals[i] <= '9');
        result += Number(decimals[i] - '0');

    }
    digits = result.digits;
}

```

}

17.8 Exponentiation

E.g., $x \times x \times x \times x \times x \times x \times x \times x$ contains reusable square subproducts: $x^n = x^{n \% 2} (x^2)^{n/2}$. This leads to an efficient algorithm with $\lg(n)$ multiplications. Start with x and n , then the next iteration uses x^2 and $n/2$, etc. Modular exponentiation also reduces by the modulus after every multiplication to avoid large subproducts for efficiency.

```
Number power(Number const& t, Number n)
{
    Number x = t, result(1);
    for(;;)
    {
        if(n.isOdd()) result *= x;
        n >>= 1;//cheap division by 2
        if(n.isZero()) break;
        x *= x;
    }
    return result;
}
Number modPower(Number const& t, Number n, Number const& modulus)
{
    assert(!modulus.isZero());
    Number x = t, result(1);
    for(;;)
    {
        if(n.isOdd())
        {
            result *= x;
            result %= modulus;
        }
        n >>= 1;//cheap division by 2
        if(n.isZero()) break;
        x *= x;
        x %= modulus;
    }
    return result;
}
```

17.9 Lg

Need $O(1)$ time because only the highest bit matters.

```
int lg() const
    {return BASE_RADIX * (nDigits() - 1) + lgFloor(digits.lastItem());}
```

17.10 Integer Square Root

Newton's method (see the "Numerical Algorithms—Working with Functions" chapter) computes the correct result (Brent & Zimmermann 2011). For efficiency it starts with a log-based upper bound:

```
Number sqrtInt(Number const& t)
//start with a good guess
    Number x(Number(1) << (1 + t.lg()/2));
    for(;;)
    {
        Number y = (x + t/x)/Number(2);
        if(y < x) x = y;
        else return x;
    }
}
```

Usually need $O(1)$ iterations to converge, so the runtime is effectively $O(\text{division})$.

17.11 Greatest Common Divisor

The **Euclidean algorithm** subtracts the smaller number from the larger until the smaller = 0. For efficiency divide with remainder r instead, so when $a > b$, $\text{GCD}(a, b) = \text{GCD}(b, r = a \% b)$. The **extended version** computes x, y such that $\text{GCD}(a, b) = ax + by$ by retaining all quotients. Because after each iteration the GCD remains the same when a becomes r , $\text{GCD}(a, b) = \text{GCD}(r, b) = rx + by = (a - bq)x + by = ax + b(y - qx)$, and when b becomes 0, $\text{GCD}(r, b) = r \times 1 + b \times 0$. The runtime is $O(|a||b|)$ (Cormen et al. 2009).

```
Number extendedGcdR(Number const& a, Number const& b, Number& x, Number& y)
{
    if (!b.isPositive())
    {
        x = Number(1);
        y = Number(0);
        return a;
    }
    Number q, r = Number::divide(a, b, q), gcd = extendedGcdR(b, r, y, x);
    y -= q * x;
    return gcd;
}

Number extendedGcd(Number const& a, Number const& b, Number& x, Number& y)
{
    assert(a.isPositive() && b.isPositive());
    return a < b ? extendedGcdR(b, a, y, x) : extendedGcdR(a, b, x, y);
}

Number gcd(Number const& a, Number const& b)
{
    Number x, y;
    return extendedGcd(a, b, x, y);
}
```

17.12 Modular Inverse

The inverse of $a \% n$ exists only if $1 = \text{gcd}(a, n) = ax + ny$. It's x because $(ax + ny) \% n = ax \% n$.

```
Number modInverse(Number const& a, Number const& n)
{
    assert(a.isPositive() && a < n);
    Number x, y;
    extendedGcd(a, n, x, y);
    if (x.isNegative()) x += n; //adjust range if needed
    return x;
}
```

The runtime is $O(|a||n|)$.

17.13 Primality Testing

Fermat's theorem: If n is prime, $\forall a$ such that $1 < a < n$ and $\text{gcd}(a, n) = 1$, $a^{n-1} \% n = 1$. But it also holds for composite **Carmichael numbers** such as 561, so using its converse with various a as a test fails. **Miller-Rabin algorithm:**

1. Factor $n - 1$ as $2^c d$ such that d is odd
2. $x = a^d \% n$
3. c times
4. Square x mod n
5. If the square = 1, but not $x = 1$ or $x = n - 1$, n is composite because for a prime p $x^2 = 1 \pmod{p} \leftrightarrow (x+1)(x-1) = 0 \pmod{p} \leftrightarrow x = 1 \pmod{p}$ or $x = -1 \pmod{p}$.

It's proven (Welschenbach 2005) that the answer:

- *composite* is correct.
- *not composite* is wrong with probability $< \frac{1}{4}$ when $1 < a < n$ is chosen randomly. This probability decreases with the size of n , as does the number of tests to make it $\approx 2^{-100}$.

```
bool provenComposite(Number const& a, Number const& n)
{
    Number ONE = Number(1), oddPart = n - ONE;
```

```

int nSquares = 0;
while(oddPart.isEven())
{
    oddPart >>= 1;
    ++nSquares;
}
Number x = modPower(a, oddPart, n);
for(int i = 0; i < nSquares; ++i)
{ //if x2 is 1 x must have been 1 or -1 if n is prime
    Number x2 = modPower(x, Number(2), n);
    if(x2 == ONE && x != ONE && x != n - ONE) return true;
    x = x2;
}
return x != ONE;
}

```

Each test needs $\leq \lg(n)$ squarings of a , with the total cost $O(\lg(n)y^2)$ for random $a \in [1, n]$ and y digits. The algorithm first does trial division by small primes < 50 . Doing so until ≈ 2000 seems optimal experimentally (Schneier 1995) but complicates the code a little (i.e., would use the sieve of Eratosthenes; see the "Miscellaneous Algorithms" chapter). For efficiency the implementation chooses $a \in [2, \min(n, B - 1)]$.

```

bool isPrime(Number const& n)
{
    n.debug();
    if(n.isEven() || n < Number(2)) return false;
    int smallPrimes[] = {3,5,7,11,13,17,19,23,29,31,37,41,43,47};
    for(int i = 0; i < sizeof(smallPrimes)/sizeof(int); ++i)
    {
        Number p = Number(smallPrimes[i]);
        if(n == p) return true;
        if((n % p).isZero()) return false;
    } //Miller-Rabin if trial division was inconclusive
    int nTrials = 1;
    int sizes[] = {73,105,132,198,223,242,253,265,335,480,543,627,747,927,
        1233,1854,4096}, nTests[] = {47,42,35,29,23,20,18,17,16,12,8,7,6,5,4,
        3,2};
    for(int i = 0; i < sizeof(sizes)/sizeof(*sizes); ++i)
        if(n.lg() < sizes[i])
        {
            nTrials = nTests[i];
            break;
        }
    while(nTrials--)
    { //use single-digit exponents for efficiency
        Number::DIGIT_TYPE max = numeric_limits<Number::DIGIT_TYPE>::max();
        if(provenComposite(Number(GlobalRNG().inRange(2, (Number(max) < n ?
            max : int(n[0])) - 1)), n)) return false;
    }
    return true;
}

```

To generate a random n -bit prime, until the result is prime, generate a random n -bit number, and set its highest and lowest bits. By the Riemann's hypothesis, $\forall n$ $O(1/n)$ numbers are prime, so $E[\text{the number of primality tests}] = O(n)$. The result isn't cryptographically secure if the random generator isn't.

17.14 Rationals

Can portably extract the integer base and the exponent out of a double:

```

pair<long long, int> rationalize(double x)
{//only support the usual binary (not decimal)
    assert(numeric_limits<double>::radix == 2 &&
        numeric_limits<double>::digits <= numeric_limits<long long>::digits);
    int w = numeric_limits<double>::digits, e;
    x = frexp(x, &e); //normalize x into [0.5, 1]
    long long mantissa = ldexp(x, w); //find x^53
}

```

```

    return make_pair(mantissa, e - w);
}

```

A rational consists of a large number numerator and denominator and reduces them by their GCD after every operation. The complexity expressions assume that both have the same number of digits for simplicity.

```

struct Rational: public ArithmeticType<Rational>
{
    Number numerator, denominator;
    Rational(Number const& theNumerator = Number(0),
              Number const& theDenominator = Number(1)): numerator(theNumerator),
              denominator(theDenominator)
    {
        assert (!denominator.isZero());
        reduce();
    }
    Rational(double x): denominator(1), numerator(1)
    {
        pair<long long, int> mantissaExponent = rationalize(x);
        numerator = Number(mantissaExponent.first);
        int e = mantissaExponent.second;
        if(e < 0) denominator <<= -e;
        else if(e > 0) numerator <<= e;
    }
    void reduce()
    {
        Number g = gcd(numerator, denominator);
        numerator /= g;
        denominator /= g;
    }
    bool isZero() const {return numerator.isZero(); }
    bool isMinus() const
        {return numerator.isNegative() != denominator.isNegative(); }
};

```

Addition and subtraction are $O(n^2)$ due to GCD to keep the sizes small. But arguably don't need GCD for cheap operations.

```

Rational operator-() const
{
    Rational result = *this;
    result.numerator.negate();
    return result;
}
Rational& operator+=(Rational const& rhs)
{
    numerator = numerator * rhs.denominator + rhs.numerator *
        denominator;
    denominator *= rhs.denominator;
    reduce();
    return *this;
}
Rational& operator-=(Rational const& rhs) {return *this += -rhs; }

```

Multiplication and division are $O(n^2)$.

```

Rational& operator*=(Rational const& rhs)
{
    numerator *= rhs.numerator;
    denominator *= rhs.denominator;
    reduce();
    return *this;
}
Rational& operator/=(Rational const& rhs)
{
    assert (!rhs.isZero());
}

```

```

    numerator *= rhs.denominator;
    denominator *= rhs.numerator;
    reduce();
    return *this;
}

```

Other useful operations are `lg` and evaluation to some precision:

```

int lg() const {return numerator.lg() - denominator.lg();}

Number evaluate(Number const& scale = Number(1))
{return numerator * scale / denominator;}

```

In many computations, can simulate rationals because a fixed-precision real number is an integer divided by a normalizing factor. This allows, e.g., to calculate π to n digits as $\text{floor}(10^n\pi)$, using power series for $4\text{atan}(1)$ and giving the integer several extra digits to avoid round-off errors. But in general don't try to simulate floating point arithmetic using rational arithmetic—e.g., a sequence of multiplications can lead to combinatorial explosion that rapidly exhaust memory. A better alternative is arbitrary-precision floating point arithmetic (see Brent & Zimmermann 2010).

17.15 Implementation Notes

Other than some basic conventions, the implementation of a large number follows textbook descriptions.

Construction of a rational from a floating-point value is a new idea. Perhaps interval arithmetic is more useful here (see the “Numerical Algorithms—Introduction and Matrix Algebra” chapter).

17.16 Comments

For multiplication, after ≈ 100 digits **Karatsuba multiplication** is the fastest up to some value. The asymptotically best known method uses the fast Fourier transform (see the “Numerical Algorithms—Working with Functions” chapter) and is very complicated. Computations with millions of digits use it, but for such sizes few other algorithms are feasible.

17.17 Projects

- Implement other bit operations such as “&”
- Experiment to find best largest prime for trial division during primality testing
- Research and implement arbitrary-precision floating point arithmetic

17.18 References

- Brent, R. P., & Zimmermann, P. (2010). *Modern Computer Arithmetic*. Cambridge University Press.
 Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
 St Denis, T. (2006). *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic*. Syngress.
 Schneier, B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley.
 Welschenbach, M. (2005). *Cryptography in C and C++*. Apress.

18 Computational Geometry

18.1 Introduction

This chapter describes the most useful data structures for points in many dimensions, particularly for finding nearest neighbors. These fall under the field of computational geometry, which is otherwise about computation of drawable objects such as a convex hull. Some original implementation details are given for the VP tree.

The chapter also introduces algorithms for the computation of such objects their inherent robustness issues when implemented with floating point arithmetic. See the references for more.

18.2 Distances

Many algorithms and data structures assume metric distances between points. **Euclidean distance** is the most useful and, when squared, $= \sum(\text{individual dimension component distances})$, which is **computable incrementally** in $O(1)$ time per component. E.g., to compare two distances can compute the square of one, and stop computing the square of the other if the incremental result is larger.

```
template<typename VECTOR> class EuclideanDistance
{
    static double iDistanceIncremental(VECTOR const& lhs, VECTOR const& rhs,
        int i) //add on a component
    {
        double x = lhs[i] - rhs[i];
        return x * x;
    }
    static double distanceIncremental(VECTOR const& lhs, VECTOR const& rhs,
        double bound = numeric_limits<double>::infinity())
    { //compute distance up to a bound
        assert(lhs.getSize() == rhs.getSize());
        double sum = 0;
        for(int i = 0; i < lhs.getSize() && sum < bound; ++i)
            sum += iDistanceIncremental(lhs, rhs, i);
        return sum;
    }
public:
    struct Distance
    { //metric functor
        double operator()(VECTOR const& lhs, VECTOR const& rhs) const
        { return sqrt(distanceIncremental(lhs, rhs)); }
    };
    struct DistanceIncremental
    { //incremental functor that returns distance squared
        double operator()(VECTOR const& lhs, VECTOR const& rhs) const
        { return distanceIncremental(lhs, rhs); }
        double operator()(VECTOR const& lhs, VECTOR const& rhs, int i) const
        { return iDistanceIncremental(lhs, rhs, i); }
        double operator()(double bound, VECTOR const& lhs, VECTOR const& rhs)
            const { return distanceIncremental(lhs, rhs, bound); }
    };
};
```

An efficient point dictionary and some related data structures are hierarchies represented as trees where each subtree covers a part of the volume, with the root covering the whole. Given a distance function, the **hierarchy distance** is the closest distance between a query point and any point in the subtree.

18.3 VP Tree

Given a function that computes a metric distance between keys, can implement:

- Map operations
- ***k*-nearest neighbor query (*k*-NN)**— find *k* closest elements of a given key *x*

- **Distance query**—find all elements within a specified distance from x . These enable, e.g., finding all strings with edit distance from a given string ≤ 2 .

A VP tree (abbreviated from “**vantage point**”; also called “**post office**” tree) picks an object as the root and some radius r for it. All objects at distance $\leq r$ go in the left subtree, and $> r$ in the right. Use $r =$ the distance to the first object inserted into the subtree of the root, making the object the left child. To prevent numerical issues do an ϵ -minded comparison by sending slightly larger distances to the left (see the “Numerical Algorithms—Introduction and Matrix Algebra” chapter). To improve pruning during search, every node stores its maximum distance to any node in the subtree.

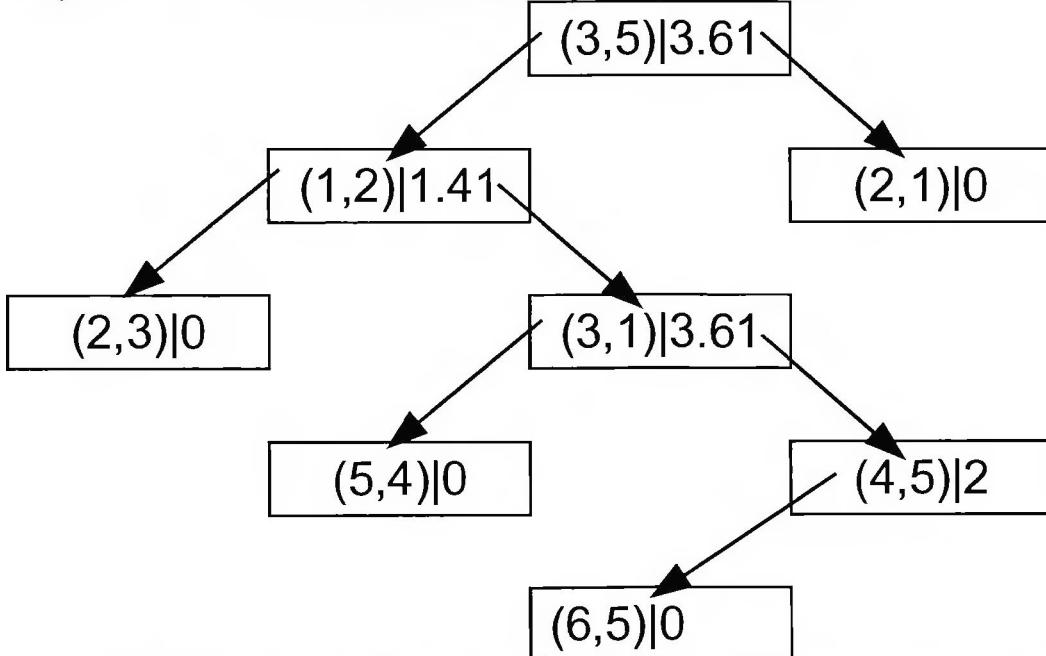


Figure 18.1: Structure of a 2D VP tree with Euclidean data—the left part is the point, and the right the left-child distance

Theorem (Samet 2006): Given a, b, c in a metric space, where $r_1 \leq d(a, b) \leq r_2$, and $d(a, c) = k$, $\max(0, k - r_2, r_1 - k) \leq d(b, c)$. This lower-bounds child distances to a query point when a is the key of the current node, b of the child, and c of the query. So also maintain in each node the subtree radius, i.e., the furthest-descendant distance.

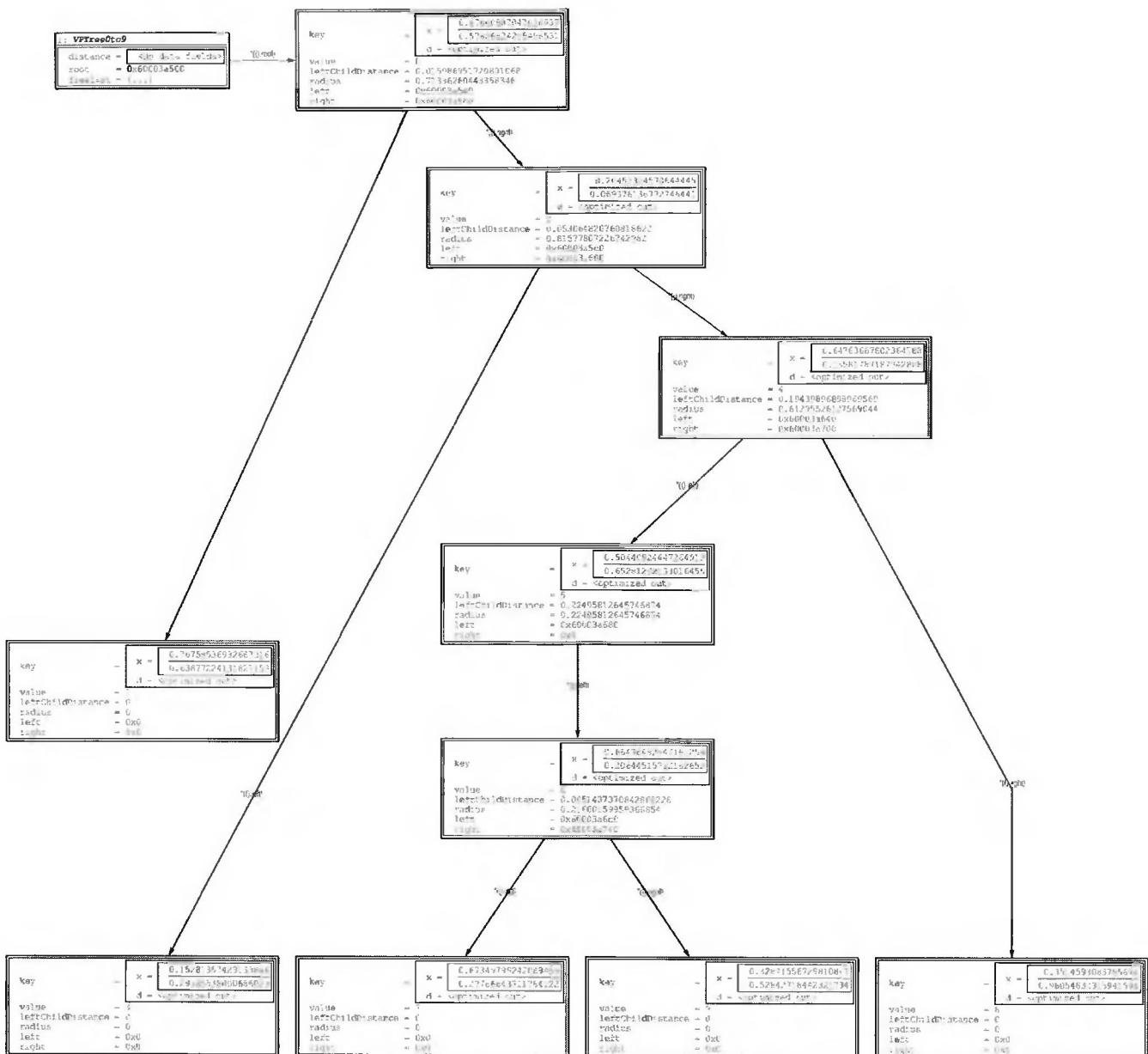


Figure 18.2: Memory layout of a VP tree

```

template< typename KEY, typename VALUE, typename DISTANCE> class VpTree
{
    DISTANCE lowerBound;
    static double bound(double keyDistance, double rLow, double rHigh)
        {return max(0., max(keyDistance - rHigh, rLow - keyDistance));}
    struct Node
    {
        KEY key;
        VALUE value;
        double leftChildDistance, radius;
        Node *left, *right;
        Node(KEY const& theKey, VALUE const& theValue): key(theKey), left(0),
            right(0), value(theValue), leftChildDistance(0), radius(0) {}
        double leftChildBound(double keyDistance)
            {return bound(keyDistance, 0, leftChildDistance);}
        double rightChildBound(double keyDistance)
            {return bound(keyDistance, leftChildDistance, radius);}
    }* root;
    Freelist<Node> f;
public:
    typedef DISTANCE DISTANCE_TYPE; //update doc!
    DISTANCE const& getDistance() {return lowerBound;}
    typedef Node NodeType;

```

```
    bool isEmpty() const {return !root;}  
};
```

The constructors are similar to those of a binary search trees, except for needing a distance functor instead of a comparator:

```
Node* constructFrom(Node* n)  
{  
    Node* tree = 0;  
    if(n)  
    {  
        tree = new(f.allocate())Node(n->key, n->value);  
        tree->leftChildDistance = n->leftChildDistance;  
        tree->radius = n->radius;  
        tree->left = constructFrom(n->left);  
        tree->right = constructFrom(n->right);  
    }  
    return tree;  
}  
VpTree(DISTANCE const& theDistance = DISTANCE()): root(0),  
lowerBound(theDistance) {}  
VpTree(VpTree const& rhs): lowerBound(rhs.lowerBound)  
{root = constructFrom(rhs.root);}  
VpTree& operator=(VpTree const& rhs) {return genericAssign(*this, rhs);}
```

Find follows the tree definition by going to the left child if the node distance to the query not ϵ -larger and right otherwise.

```
VALUE* find(KEY const& key) const  
{  
    Node* n = root;  
    while(n && key != n->key) n = !isELess(n->leftChildDistance,  
        lowerBound(key, n->key)) ? n->left : n->right;  
    return n ? &n->value : 0;  
}
```

Insert on the path down updates the radius in every resulting parent node if the new node becomes a left child:

```
void insert(KEY const& key, VALUE const& value)  
{  
    Node **pointer = &root, *n;  
    while((n = *pointer) && key != n->key)  
    {  
        double d = lowerBound(key, n->key);  
        n->radius = max(n->radius, d);  
        if(!n->left) n->leftChildDistance = d; //will make left child  
        pointer = &(!isELess(n->leftChildDistance, d) ? n->left : n->right);  
    }  
    if(n) n->value = value; //equality--assign new value  
    else *pointer = new(f.allocate())Node(key, value);  
}
```

The height isn't controlled because ! \exists a good way to balance—this and the runtime are discussed later in the chapter. So the tree can be unbalanced, but, as with binary trees, random-order insertion leads to a good expected performance. To remove a node, due to the defined hierarchies, can use only weak deletion (not implemented here).

A distance query uses the child bounds to prune nodes farther than a specified radius—they can't be in the subtree:

```
Vector<NodeType*> distanceQuery(KEY const& key, double radius) const  
{  
    Vector<NodeType*> result;  
    distanceQuery(key, radius, result, root);  
    return result;  
}  
void distanceQuery(KEY const& key, double radius, Vector<Node*>& result,  
    Node* n) const
```

```

{
    if(!n) return;
    double d = lowerBound(n->key, key);
    if(d <= radius) result.append(n);
    if(n->leftChildBound(d) <= radius)//first go left if not pruned
        distanceQuery(key, radius, result, n->left);
    if(n->rightChildBound(d) <= radius)//then right if not pruned
        distanceQuery(key, radius, result, n->right);
}

```

The % of nodes are checked that don't need to be depends on the quality of the bounds. In the worst case such queries check all nodes in $O(n)$ time.

A k -NN query uses a max heap of k closest nodes and a generic helper function that finds the distance to the k^{th} closest element, which is the heap's minimum:

```

template<typename NODE> struct QNode
{
    NODE* n;
    double d;
    bool operator<(QNode const& rhs) const{return d > rhs.d; }
    static double dHeap(Heap<QNode>& heap, int k)
    {
        return heap.getSize() < k ?
            numeric_limits<double>::max() : heap.getMin().d;
    }
};

```

A k -NN query does branch and bound of the tree, using the heap to keep k closest found neighbors. Pruning and child selection use bounds of the currently furthest node. The found neighbors are heap-sorted in order of closeness.

```

Vector<NodeType*> kNN(KEY const& key, int k) const
{
    Heap<HEAP_ITEM> heap;
    kNN(root, key, heap, k);
    Vector<Node*> result;//heap-sort found nodes in by distance
    while(!heap.isEmpty()) result.append(heap.deleteMin().n);
    result.reverse();
    return result;
}
NodeType* nearestNeighbor(KEY const& key) const
{
    assert(!isEmpty());
    return kNN(key, 1)[0];
}
typedef QNode<Node> HEAP_ITEM;
void kNN(Node* n, KEY const& key, Heap<HEAP_ITEM>& heap, int k) const
{
    if(!n) return;
    //replace furthest n in heap with the current n if it's closer
    HEAP_ITEM x = {n, lowerBound(key, n->key)};
    if(heap.getSize() < k) heap.insert(x);
    else if(x.d < HEAP_ITEM::dHeap(heap, k)) heap.changeKey(0, x);
    //expand closer child first
    double lb = n->leftChildBound(x.d), rb = n->rightChildBound(x.d);
    Node* l = n->left, *r = n->right;
    if(lb > rb)//go to smaller-lower-bound node first to reduce the chance
    {//of going to the other one by placing closer nodes on the heap
        swap(lb, rb);
        swap(l, r);
    }
    if(lb <= HEAP_ITEM::dHeap(heap, k)) kNN(l, key, heap, k);
    if(rb <= HEAP_ITEM::dHeap(heap, k)) kNN(r, key, heap, k);
}

```

To extend a VP tree for external memory, put more keys and pointers in each node, sorted by distances

to the parent node.

18.4 k-d Tree

Multidimensional points comparable in each dimension support a **range query**—find all items whose keys for the specified dimensions are in a given range. A *k*-d tree is a binary tree that branches on each dimension in turn. Some but not all dimension keys may be equal.

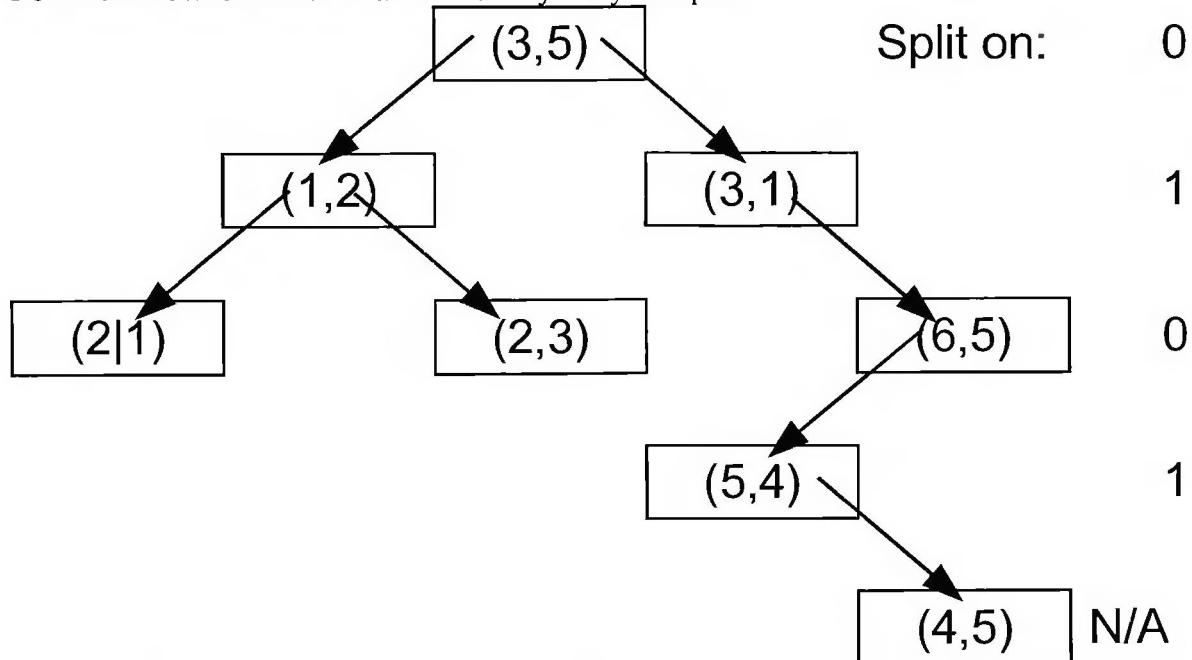


Figure 18.3: Structure of a 2D k-d tree with Euclidean data

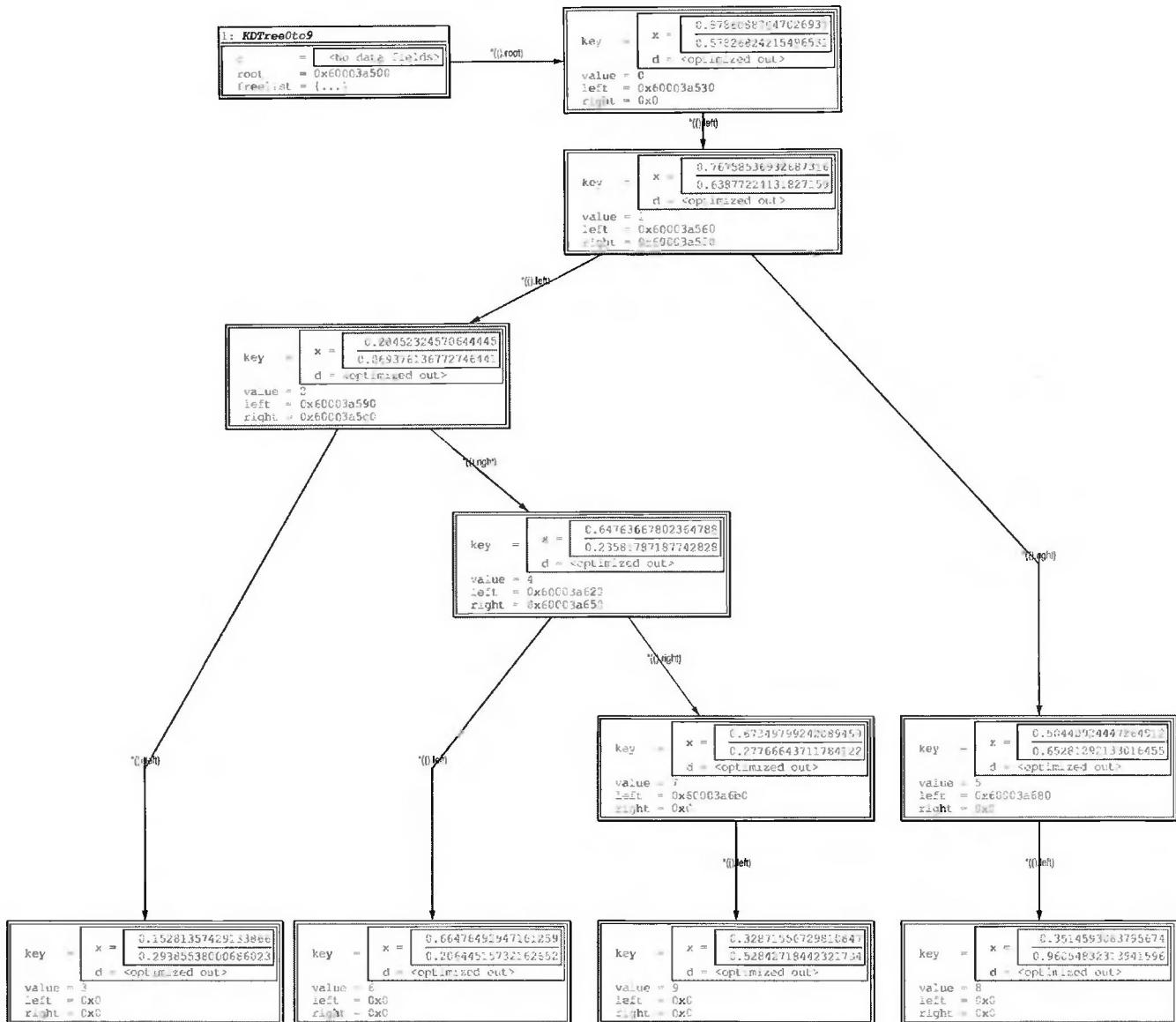


Figure 18.4: Memory layout of a 2D k-d tree

```

template<typename KEY, typename VALUE,
        typename INDEXED_COMPARATOR = LexicographicComparator<KEY>> class KDTree
{
    INDEXED_COMPARATOR c;
    struct Node
    {
        KEY key;
        VALUE value;
        Node *left, *right;
        Node(KEY const& theKey, VALUE const& theValue): key(theKey),
            value(theValue), left(0), right(0) {}
    } * root;
    Freelist<Node> f;
    int D;
    Node* constructFrom(Node* n)
    {
        Node* tree = 0;
        if(n)
        {
            tree = new(f.allocate()) Node(n->key, n->value);
            tree->left = constructFrom(n->left);
            tree->right = constructFrom(n->right);
        }
        return tree;
    }
}

```

```

public:
    typedef Node NodeType;
    bool isEmpty() const {return !root; }

    KDTTree(int theD, INDEXED_COMPARATOR const& theC = INDEXED_COMPARATOR()): 
        root(0), c(theC), D(theD) {}

    KDTTree(KDTTree const& rhs): c(rhs.c){root = constructFrom(rhs.root);}

    KDTTree& operator=(KDTTree const& rhs) {return genericAssign(*this, rhs);}
};

```

Find and insert are like the unbalanced-binary-search-tree ones, except for the dimension cycling. The reusable `findPointer` also computes the parent of the found node for a 1-NN query.

```

Node** findPointer(KEY const& key, Node*& parent) const
{
    Node* n, **pointer = (Node**) &root; //cast for const
    parent = 0;
    for (int i = 0; (n = *pointer) && !c.isEqual(key, n->key); 
          i = (i + 1) % D)
    {
        parent = n;
        pointer = &(c(key, n->key, i) ? 
                    n->left : n->right);
    }
    return pointer;
}

VALUE* find(KEY const& key) const
{
    Node *n = *findPointer(key, n);
    return n ? &n->value : 0;
}

void insert(KEY const& key, VALUE const& value)
{
    Node *dummy, **pointer = findPointer(key, dummy);
    if(*pointer) (*pointer)->value = value;
    else *pointer = new(f.allocate())Node(key, value);
}

```

For VP and k -d trees with items inserted in random order, $E[\text{the height}] = O(\lg(n))$. The worst-case height is $O(n)$. Usually this isn't a problem because if point values in a single dimension are in random order, and D is constant relative to n , $E[\text{the height}] = O(\lg(n))$. A better measure of performance is how effective a query is along every dimension it specifies. E.g., for a 2D k -d tree balanced on the x -coordinate and not on the y -coordinate, a range query on (x, y) is the same as linear scan for y on the result of a range query on x . Rotations aren't supported, but insertion can partially rebuild to maintain balance using amortized or expected $O(\lg(n)^2)$ time (see the "Miscellaneous Algorithms and Techniques" chapter). Can support deletions, but inefficiently; use weak deletions if needed.

A range query does depth-first search that checks if the considered nodes are inside the specified box $[l, u]$. If a node isn't, one of its children isn't, which allows pruning.

```

Vector<NodeType*> rangeQuery(KEY const& l, KEY const& u,
                               Vector<bool> const& dimensions) const
{
    Vector<Node*> result;
    rangeQuery(l, u, dimensions, result, root, 0);
    return result;
}

void rangeQuery(KEY const& l, KEY const& u, Vector<bool> const& dimensions,
                  Vector<Node*>& result, Node* n, int i) const
{
    if(!n) return;
    bool inRange = true; //check if current node in range
    for (int j = 0; j < D; ++j)
        if(dimensions[j] && (c(n->key, l, j) ||
                                c(u, n->key, i))) inRange = false;
    if(inRange) result.append(n);
    int j = (i + 1) % D; //only check range for the wanted dimensions
}

```

```

    if(!(dimensions[i] && c(n->key, l, i)))
        rangeQuery(l, u, dimensions, result, n->left, j);
    if(!(dimensions[i] && c(u, n->key, i)))
        rangeQuery(l, u, dimensions, result, n->right, j);
}

```

The worst-case runtime is $O(n^{(D-1)/D})$ for a balanced tree, which is optimal for $O(n)$ space. On average for small D the runtime is $O(\lg(n))$ (Samet 2006). Intuitively, a k -d tree is most useful for small D because for large have many variables to partition on, and use each only few times.

A **partial match query** finds all nodes with the key matching only at the specified dimensions. Reduce it to a range query where $l = u$. An **interval query** (not implemented here) finds all intervals that contain a given point, as a 2D range query $[x, \max] \times [\min, x]$, where \min/\max are maintained explicitly or replaced by a suitable $-\infty$ and ∞ . This generalizes to rectangles and hyperrectangles.

A k -NN query works with a metric distance, but for efficiency additionally assumes an incrementally computable distance, which forces the distance function to have the form $\sum_{0 \leq i < D} g(x[i], y[i])$. The algorithm is a B&B, similar to that of the VP tree, but uses unnormalized distances of the form $d(x, y)$ and updates them incrementally on the search path. Initially the partial key = x , corresponding to the 0 distance to the tree. When the search goes left, the partial key of the left node doesn't change, and of the right node sets its current coordinate to that of the current node. Symmetrically, for going right, the partial key of the right node doesn't change, and of the left node sets its current coordinate to that of the current node. So $d(x, \text{the partial key})$ is the best possible lower bound on the hierarchy distance. It's nondecreasing on the search path. Distance comparison is incremental.

```

typedef QNode<Node> HEAP_ITEM;
template<typename DISTANCE> void kNN(Node* n, KEY const& key,
    Heap<HEAP_ITEM>& heap, int k, int i, KEY& partial,
    double partialDistance, DISTANCE const& distance) const
{
    double best = HEAP_ITEM::dHeap(heap, k);
    if(n && partialDistance < best)
        //update partial distance
        double newPartialDistance = distance(key, n->key, i) -
            distance(key, partial, i);
        if(heap.getSize() < k)
        {
            HEAP_ITEM x = {n, distance(key, n->key)};
            heap.insert(x);
        }
        //use new partial distance to check for a cut again
        else if(newPartialDistance < best)
            //incremental calculate-compare
            double d = distance(best, key, n->key);
            if(d < best)
            {
                HEAP_ITEM x = {n, d};
                heap.changeKey(0, x);
            }
        }
    int j = (i + 1) % D;
    //swap children for best order
    Node *l = n->left, *r = n->right;
    if(!c(key, n->key, i)) swap(l, r);
    kNN(l, key, heap, k, j, partial, partialDistance, distance);
    //set partial component to the n component, use the n
    //as temporary storage
    swap(partial[i], n->key[i]);
    kNN(r, key, heap, k, j, partial, newPartialDistance, distance);
    swap(partial[i], n->key[i]);
}
}

```

For $k = 1$ putting on the queue the parent of where the query would be inserted improves pruning. But don't do it for $k > 1$ due to potentially reinserting the parent into the heap when its size $< k$.

```

template<typename DISTANCE> Vector<NodeType*> kNN(KEY const& key, int k,
DISTANCE const& distance) const
{
    Heap<HEAP_ITEM> heap;
    KEY partial = key;
    kNN(root, key, heap, k, 0, partial, 0, distance);
    Vector<Node*> result;//heap-sort found nodes in by distance
    while(!heap.isEmpty()) result.append(heap.deleteMin().n);
    result.reverse();
    return result;
}
template<typename DISTANCE> NodeType* nearestNeighbor(KEY const& key,
DISTANCE const& distance) const
{
    assert(!isEmpty());
    Node* parent, *result = *findPointer(key, parent);
    if(result) return result;//found equal-value node, d = 0
    Heap<HEAP_ITEM> heap;//put parent on heap
    HEAP_ITEM x = {parent, distance(key, parent->key)};
    heap.insert(x);
    KEY partial = key;
    kNN(root, key, heap, 1, 0, partial, 0, distance);
    return heap.getMin().n;
}

```

A distance query is similar to a k -NN query. Build a partial key that bounds all nodes in the current subtree. It starts as the query x , and is updated when going left or right means that all descendants are at least some partial distance away based on the corresponding dimension. Because work with incremental distance, the partial radius is the square of the wanted radius for Euclidean distance.

```

template<typename DISTANCE> Vector<NodeType*> distanceQuery(KEY const& x,
double partialRadius, DISTANCE const& distanceIncremental) const
{
    Vector<Node*> result;
    KEY partial = x;
    distanceQuery(x, partialRadius, result, root, 0, distanceIncremental,
                  partial, 0);
    return result;
}
template<typename DISTANCE> void distanceQuery(KEY const& x,
double partialRadius, Vector<Node*>& result, Node* n, int i,
DISTANCE const& distanceIncremental, KEY& partial,
double partialDistance) const
//first try to prune subtree
if(!n || partialDistance > partialRadius) return;
if(distanceIncremental(n->key, x) <= partialRadius)
    result.append(n);
i = (i + 1) % D;
Node* nodes[] = {n->left, n->right};
for(int j = 0; j < 2; ++j)
    //apply partial to right subtree if x[i] on the left side of n and
    //to left if on the right; equality not a problem
    bool applyPartial = c(x, n->key, i) == (j == 1);
    double dDelta = 0;
    if(applyPartial)
    {
        dDelta = distanceIncremental(x, n->key, i) -
            distanceIncremental(x, partial, i);
        swap(partial[i], n->key[i]);//use n as temp storage
    }
    distanceQuery(x, partialRadius, result, nodes[j], i,
                  distanceIncremental, partial, partialDistance + dDelta);
    if(applyPartial) swap(partial[i], n->key[i]);
}

```

}

For external memory have nodes of size B with $k - 1$ keys and k pointers, fill them to capacity, and insert any new node between the appropriate keys.

18.5 Problems in High Dimensions

Range and k -NN queries are slower for large D because \forall metric distance, as the dimension $\rightarrow \infty$, the difference between the distances to any object's nearest and furthest neighbors $\rightarrow 0$, so bounds become ineffective for pruning, forcing the queries to look at almost every node.

Can reduce runtime if willing to get approximate answers: multiply the lower bounds by $(1 + c)$ for some small constant c before a pruning check. Then the found neighbors are at most a factor of $(1 + c)$ away from the nearest. But for implementation and practical use it's unclear how to pick c , what precise performance is gained, and which data structure is best for this.

18.6 Data Structures for Geometric Objects

The simplest solution is to have a bounding box around an object, and store its endpoints in a k -d tree. E.g., represent intervals or rectangles as multidimensional points.

Can also represent the box by its centroid, and store the endpoints in the node item. This makes queries such as k -NN more efficient.

Alternatively, represent objects as images with each cell containing a list of pointers to objects that touch it and having at most one pointer, assuming no objects.

18.7 Points

Can represent a **multidimensional point** as a vector, but usually know D at compile time, so it's more efficient to use an array. Algorithms should be implemented to work with both where possible—as do Euclidean distance and the trees. The resulting point implements several arithmetic operators for convenience, using the usual vector arithmetic:

```
template<typename KEY, int D = 2>
class Point: public ArithmeticType<Point<KEY, D>>
{
    KEY x[D];
public:
    static int const d = D;
    KEY& operator[](int i){assert(i >= 0 && i < D); return x[i];}
    KEY const& operator[](int i) const{assert(i >= 0 && i < D); return x[i];}
    int getSize() const{return D;}
    Point(){for(int i = 0; i < D; ++i) x[i] = 0;}
    Point(KEY const& x0, KEY const& x1)
    {
        assert(D == 2); //to prevent accidents for D > 2
        x[0] = x0;
        x[1] = x1;
    }
    bool operator==(Point const& rhs) const
    {
        for(int i = 0; i < D; ++i) if(x[i] != rhs.x[i]) return false;
        return true;
    }
    Point& operator+=(Point const& rhs)
    {
        for(int i = 0; i < D; ++i) x[i] += rhs.x[i];
        return *this;
    }
    Point& operator*=(double scalar)
    {
        for(int i = 0; i < D; ++i) x[i] *= scalar;
        return *this;
    }
    friend Point operator*(Point const& point, double scalar)
```

```

    {
        Point result = point;
        return result *= scalar;
    }
    Point& operator=(Point const& rhs) {return *this += rhs * -1;}
    Point operator-() {return *this * -1;}
    double friend dotProduct(Point const& a, Point const& b)
    {
        double dp = 0;
        for(int i = 0; i < D; ++i) dp += a[i] * b[i];
        return dp;
    }
};

typedef Point<double> Point2;

```

18.8 Geometric Primitives

The sign of the area of a triangle, given by three 2D points, tells whether the points turn left, i.e., are in counter-clockwise (CCW) order. When the area = 0, the points don't turn, but usually consider this a left turn.

The area of the triangle (a, b, c) in CCW order = $\det \begin{bmatrix} 1 & a.x & a.y \\ 1 & b.x & b.y \\ 1 & c.x & c.y \end{bmatrix}$:

```

double triangleArea(Point2 const& a, Point2 const& b, Point2 const& c)
    {return (b[0] - a[0]) * (c[1] - a[1]) - (b[1] - a[1]) * (c[0] - a[0]);}
bool ccw(Point2 const& a, Point2 const& b, Point2 const& c)
    {return triangleArea(a, b, c) >= 0;}//true if the points turn left

```

The result can be wrong for very thin triangles due to numerical cancellation in the subtraction. A **double** has 52 bits of precision, the subtraction doesn't change the required precision, and the multiplication doubles it, so if every term has ≤ 26 bits, the result is correct. Otherwise, to avoid numerical errors can use large-number rationals:

```

Rational robustTriangleArea(Point2 const& a, Point2 const& b, Point2 const& c)
{
    return (Rational(b[0]) - Rational(a[0])) * (Rational(c[1]) -
        Rational(a[1])) - (Rational(b[1]) - Rational(a[1])) * (Rational(c[0]) -
        Rational(a[0]));
}
bool robustCcw(Point2 const& a, Point2 const& b, Point2 const& c)
    {return !robustTriangleArea(a, b, c).isMinus();}

```

18.9 Convex Hull

Given a set of 2D points, a convex hull is a subset of them such that a "rubber band" put over it encloses the set:

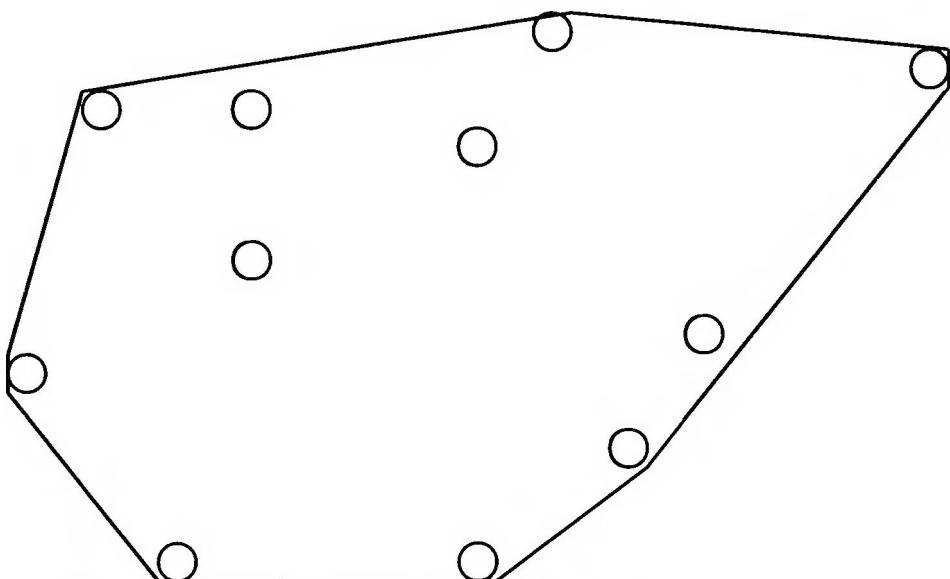


Figure 18.5: A set of points and its convex hull

To compute it:

1. Sort the points by the x -coordinate, breaking ties by the y -coordinate
2. Put the first two points on the hull
3. Compute the upper hull: \forall other point
4. Put it on the hull
5. If the last three points make a left turn, remove the second-last point
6. From the one-before-last to the first point do the same, and remove the duplicate last added point—computes the lower hull

```

void processPoint(Vector<Point2>& hull, Point2 const& point)
{
    hull.append(point);
    while(hull.getSize() > 2 && ccw(hull[hull.getSize() - 3],
        hull[hull.getSize() - 2], hull[hull.getSize() - 1]))
    {
        hull[hull.getSize() - 2] = hull[hull.getSize() - 1];
        hull.removeLast();
    }
}
Vector<Point2> convexHull(Vector<Point2>& points)
{
    assert(points.getSize() > 2);
    quickSort(points.getArray(), 0, points.getSize() - 1,
        LexicographicComparator<Point2>());
    //upper hull
    Vector<Point2> result;
    result.append(points[0]); //initialize with the first two points
    result.append(points[1]);
    for(int i = 2; i < points.getSize(); ++i) processPoint(result, points[i]);
    //lower hull, remove leftmost point which is added twice
    for(int i = points.getSize() - 2; i >= 0; --i)
        processPoint(result, points[i]);
    result.removeLast();
    return result;
}

```

With floating point arithmetic points on a line may end up on the wrong side, or a too sharp left turn may become a right turn, destroying the topological structure. Can use rational arithmetic to avoid this (not implemented here; see also the projects section). The runtime is $O(n \lg(n))$, but the bottleneck is the $O(n)$ turn tests due to constant factors. For random points $E[\text{the hull size}] = O(\lg(n))$ (De Berg et al. 2008).

18.10 Plane Sweep

This technique solves many problems. E.g., the convex hull calculation sorts the points on the x -coordinate, processes them from left to right, and then from right to left. Each point is an **event**, processing which puts

some invariant on the space until the next event.

E.g., consider finding the area of possibly overlapping buildings on a picture, each represented by its two upper coordinates:

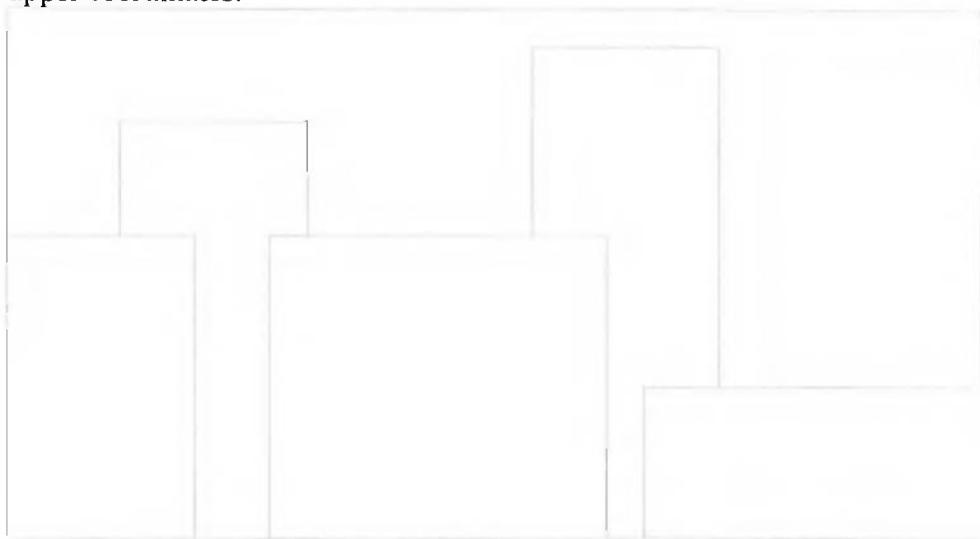


Figure 18.6: Example of the building area problem

A simple solution is to sort on the x -coordinate and process from left to right, maintaining the current height. When processing an event point, increment the total volume by the current height \times the distance to the previous event. To keep track of the current maximum height, use a map of “open” buildings. A building opens when its left coordinate is an event and closes when the right one is. An STL-based solution might look like this:

```
struct RoofCorner
{
    double x, y;
    bool isLeft;
    RoofCorner(double theX, double theY, bool theIsLeft): x(theX), y(theY),
        isLeft(theIsLeft){} //comparison on x + give priority to left over right
    bool operator<(RoofCorner const& rhs) const
        {return x == rhs.x ? isLeft > rhs.isLeft : x < rhs.x;}
};

double buildingArea(vector<RoofCorner> corners)
{
    sort(corners.begin(), corners.end());
    double result = 0, currentHeight = 0, lastX = corners[0].x;
    map<double, int> openBuildings; //indexed and sorted by height, note that
    //don't have numerical issues with double key as no calculations are done
    for(unsigned int i = 0; i < corners.size(); ++i)
    {
        double x = corners[i].x, y = corners[i].y;
        result += currentHeight * (x - lastX);
        lastX = x;
        //manage count of open buildings
        openBuildings[y] += corners[i].isLeft ? 1 : -1;
        if(openBuildings[y] == 0) openBuildings.erase(y);
        //current height is that of tallest open building
        currentHeight = openBuildings.size() > 0 ?
            openBuildings.rbegin()->first : 0;
    }
    return result;
}

void testBuildingArea()
{
    vector<RoofCorner> points;
    points.push_back(RoofCorner(0, 1, true));
    points.push_back(RoofCorner(2, 1, false));
    points.push_back(RoofCorner(1, 2, true));
    points.push_back(RoofCorner(3, 2, false));
}
```

```

    assert(buildingArea(points) == 5);
}

```

This is a complicated interview question that most developers will struggle with and take at least an hour to solve correctly, so interviewers should avoid it.

18.11 Comments

Multidimensional data structures and computational geometry are large topics. The main problems with the former are lack of balancing ability and performance with large D . So many other algorithms have been proposed—see Samet (2006).

A VP tree is presented with a user-picked node distance in other sources. The idea of using the first-in-inserted-node distance seems original. It's scaled correctly with every distance and performs well on my experiments for machine learning (see the "Machine Learning—Classification" chapter).

An interesting data structure for high-dimensional data nearest-neighbor search is **locality-sensitive hashing (LSH)** (Samet 2006). Despite the promising idea, it's unclear how to set it up to be useful as a black box, even for distance functions with known good hash functions, such as Euclidean (presented in some other references). My attempts to use it for machine learning failed (see the comments for the "Machine Learning—Classification" chapter). It also needs some parameters beyond a $(1 + c)$ tolerance factor on the distance to the actual best, which have no logical meaning to the user, and occasionally doesn't return a nearest neighbor at all.

For computational geometry the main issue is robust and efficient computation. Large-number arithmetic is a simple but inefficient solution. A more efficient but much more complicated one is using **floating point filters** (Mehlhorn & Näher 1999; see also <https://cs.nyu.edu/yap/book/egc/>). Do floating-point computation with error bounds, and switch to large number arithmetic only if can't rule out errors.

For many more algorithms see de Berg et al. (2008). Perhaps the main omission in this chapter is that of triangulation algorithms, whose results are useful in solving partial differential equations by finite element methods (see the comments in the "Numerical Algorithms—Working with Functions" chapter). But a useful implementation also needs robust arithmetic. Computation geometry is more mathematical than computational, as surveyed by a recent theoretical book Goodman et al. (2017). Computer graphics is almost completely unrelated.

18.12 Projects

- Implement an interval query for a k -d tree
- Implement interval arithmetic (see the comments section in the "Numerical Algorithms—Introduction and Matrix Algebra" chapter), and do convex hull calculations with it. In case of uncertainty about the turn can drop the point almost on a line, so don't need exact arithmetic. For many specialized algorithms see Ratschek & Rokne (2003).
- Investigate numerical behavior of radius query. It is necessary to adjust some values by epsilon to make sure sufficiently nearby items are always found?

18.13 References

- de Berg, M., Cheong, O., & Van Kreveld, M., Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer.
- Goodman, J. E., O'Rourke, J., & Toth, C. D. (2017). *Handbook of Discrete and Computational Geometry*. CRC.
- Mehlhorn, K., & Näher, S. (1999). *LEDA: a Platform for Combinatorial and Geometric Computing*. Cambridge University Press.
- Ratschek, H., & Rokne, J. (2003). *Geometric Computations with Interval and New Robust Methods: Applications in Computer Graphics, GIS and Computational Geometry*. Horwood.
- Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

19 Error Detection and Correction

19.1 Introduction

Errors in communicated data are possible, e.g., during network routing, for which codes to detect and correct errors are useful. Same for files storage on CD, DVD, etc. Only the main algorithms are discussed—many more are part of various standards. See the references for further details.

19.2 Binary Polynomials

An important mathematical model for several discussed algorithms is arithmetic with binary polynomials. It's efficient implementation is a needed building block.

A bit string with highest set bit i set represents a polynomial with binary coefficients degree $i + 1$. E.g., byte 00001101 represents $x^3 + x^2 + 1$. Multiword arithmetic with binary polynomials is more efficient than with numbers due to not needing carries for addition and other properties:

Expression	Implementation
$p + q$ or $p - q$	$p \wedge q$
px	$p \ll 1$
p/x	$p \gg 1$ with remainder $p \& 1$
pq	term by term multiplication like with numbers
pp	squaring each term because cross terms cancel out
p/q and $p \% q$	long division by repeatedly subtracting largest $qx^m < p$ from p

Table 1: How to implement operations on binary polynomials p and q

19.3 Polynomials Over Finite Fields

A **field** is an abstract algebra structure where \forall element \exists an additive inverse and \forall every nonzero element \exists multiplicative inverse—e.g., the reals. A **finite field** has finitely many elements—e.g., integers mod p for a prime p . A useful object is polynomials with coefficients from a finite field—e.g., binary polynomials.

For implementation it's convenient to have finite fields with 2^m elements, particularly with $m = 1$ (binary polynomials) or 8 (discussed later in the chapter), corresponding respectively to bits and bytes.

An **irreducible polynomial** of degree m doesn't divide any other polynomial of a smaller degree. A **primitive polynomial** is an irreducible polynomial that divides $x^n + 1$ for $n=2^m-1$ and not for smaller n (Rorabaugh 1996). Do arithmetic with binary polynomials as usual, but mod a particular primitive polynomial. For $m = 1$ one such polynomial is 1011 ($x^3 + x + 1$). A primitive polynomial is for polynomials like a prime number for numbers.

19.4 Error Detection

E.g., sending data over a network can flip some bits in error. Adding k bits of redundancy to a message m allows to detect an error with probability $1 - 2^{-k}$ because a random message produces random bits. Some naive approaches to implement this:

- Interpret m as a large number, and find the remainder of dividing it by a k -bit prime—gives almost this probability but is inefficient.
- Use the universal hash function of the “Hashing” chapter's Theorem B—more efficient but gives worse probability. Using several hashes makes the failure probability arbitrary small though.

For convenience assume k -bit words. In practice and in the implementation here $k = 32$ for convenience. This way the number of redundancy bits is fixed for the user. The **CRC** algorithm appends to the message k 0's, and returns the k -bit remainder of the result's binary polynomial division by a picked $(k + 1)$ -bit polynomial. Though not necessary for the algorithm, want the polynomial to be primitive so that the remainder depends all bits of the data. The implementation ignores the polynomial's $(k + 1)^{\text{th}}$ bit = 1 and uses p = its k lower bits.

1. Append k 0's to m
2. $w = k$ most significant bits of m
3. While m has more bits
4. $w = (w \ll 1) \wedge$ the next bit of m
5. If the MSB of the previous $w == 1$, $w \wedge= p$
6. Return w

m	Augmented	Bit	w
- 1 1	0 0 0	1	1 0 0 0
p - 1 0	0 1		0 0 1
		1	0 1 0
	- 1 0 1 0		0 0 1
	- 1 0 0 1	0	0 1 1
	0 1 1		

Figure 19.1: Long division example with $m = 11$, $p = 1001$, $k = 3$, and the corresponding algorithm calculations

Since the last k bits = 0, m bits affect w only as **control bits** for the xor-with- p -or-not decisions. So load m only in the control bits, not in w . This eliminates needing to augment m with k 0's:

1. $w = 0$
2. While m has more bits
3. $w = (w \ll 1)$
4. If (the MSB of the previous $w \wedge$ the next bit of $m == 1$), $w \wedge= p$
5. Return w

Bit	w after a loop pass
N/A	'000
1	'001
1	'011

It's more efficient to process bytes. The next 8 control bits and p decide the next 8 xor-with- p -or-not decisions. $\text{MSD}_8(m) \wedge \text{MSD}_8(w)$ would be the control byte, except must adjust the controls for p to make it so. The previous algorithm effectively works on $c = w \wedge m$, with both extended as needed to the size. Also $c = w \wedge (0 \wedge m)$, so precomputation can assume $w = 0$ and run the previous algorithm on $0_{\text{extended}} \wedge m_{\text{extended}}$. Because xor is cumulative, for a given p and \forall byte, precompute the xors into a single constant, and use control bytes.

1. Compute the constants
2. $w = 0$
3. While m has more bytes
4. $w = w \ll 8$
5. $w \wedge= \text{constant}[\text{the MSD byte of the previous } w \wedge \text{ the next byte of } m]$
6. Return w

The 32-bit implementation can use any p to get $\Pr(\text{can detect}) = 1 - 2^{-k}$, but 0xFA567D89 is good at detecting all errors with a particular Hamming distance (Koopman 2002). The calculation can be online, with $O(n)$ total runtime.

```
class CRC32
{
    uint32_t polynomial, constant[256];
public:
    CRC32(uint32_t thePolynomial = 0xFA567D89u) : polynomial(thePolynomial)
    {
        for(int i = 0; i < 256; ++i)
        {
            constant[i] = polynomial & 1;
            polynomial = (polynomial >> 1) ^ (i > 7 ? 0xEDB88320 : 0x00000000);
        }
    }
}
```

```

        constant[i] = i << 24; //make extended i
        for(int j = 0; j < 8; ++j) constant[i] =
            (constant[i] << 1) ^ (constant[i] >> 31 ? polynomial : 0);
    }
}

uint32_t hash(unsigned char* array, int size, uint32_t crc = 0)
{
    assert(numeric_limits<char>::digits == 8);
    for(int i = 0; i < size; ++i)
        crc = (crc << 8) ^ constant[(crc >> 24) ^ array[i]];
    return crc;
}
};

```

19.5 Channels and Codes

A **binary symmetric channel (BSC)** flips each m bit with probability p . If $p \neq \frac{1}{2}$, the channel gives random bits, and $p > \frac{1}{2}$ is equivalent to $p < \frac{1}{2}$, with all bits flipped. The number of flipped bits in a codeword is the **Hamming distance** between it and the received one. For a nonbinary alphabet the distance = the number of changed symbols.

Can correct some errors using redundancy, as for error detection. A **(n, k) code** has n -symbol codewords for k -symbol messages, with $n \geq k$. A code's **rate** $R = k/n$. Codes are almost always **block codes** where encode/decode a large block of size n at a time for efficiency. A very simple code is **repetition**—copy each bit $n - 1$ times, creating a $(n, 1)$ code.

The **capacity** C of a channel = the mutual information between the input and the output. E.g., for BSC, $C = 1 - H(p)$, where H is the binary entropy ($H(p) = p\lg(p) + (1-p)\lg(1-p)$; see the "Compression Algorithms" chapter). In the worst case, $p = \frac{1}{2}$, $H(p) = 0$, and communication is impossible.

Noisy channel coding theorem (Moon 2021): With $R \geq C$ reliable error correction is impossible; otherwise for large enough $n \exists$ codes that allow error correction with arbitrarily small $\Pr(\text{can't decode})$.

So must pick $R < C$; it's usually easy to estimate C for the considered channel model using simulation. Another important idea (from the proof) is that random codes have the desired performance. But decoding random codes is computationally prohibitive—need to find the codeword nearest in terms of the Hamming distance to the received word, which takes time exponential in n without further knowledge about the code. So practical codes have some structure that allows efficient decoding but can't reach the performance of the theorem.

The simplest way to correct errors is to detect with CRC, and resend any corrupt messages. This works well only for high-capacity channels with tiny p , so that $E[\text{the number of resent messages}]$ is small. But, e.g., DVD surface scratches introduce errors, which resending can't fix, and C can be low, so need proper correcting codes.

∃ two general strategies for decoding:

- Worst-case—suppose the Hamming distance between any two codewords $\geq d$ for some d . Any received word becomes the nearest codeword, so can correct $\leq \frac{d-1}{2}$ errors.
- Probabilistic—find the most likely codeword. For certain codes can correct far more errors than suggested by d , but don't have any guarantees about succeeding in decoding a particular word.

For the nearest-codeword logic on BSC, $\Pr(\text{failed to decode}) \leq \Pr(\text{any error corrupted} > \frac{d-1}{2} \text{ symbols}) = 1 - \text{BinomialCDF}\left(\frac{d-1}{2}, n, p\right)$ (Moon 2021). If $pn < \frac{d-1}{2}$, using normal approximation and bounding the tail (see the "Computational Statistics" chapter), the probability is exponentially small in $\left(\frac{d-1}{2}\right)^2$ (see the "Computational Statistics" chapter).

The nearest-codeword logic is a special case of maximum likelihood when all codewords are equally likely. Can usually decode beyond d because the expected distance is much smaller than the worst-case d . But usually the only way to measure performance is simulation, which can be very slow, particularly for low R , which requires larger n and higher computational resources.

Both can be wrong even if they successfully come up with a codeword because the true codeword may differ. This is perhaps best detected by putting some efficient error-detection logic such as CRC inside each

block to give more bits to the certainty of correction and not the correction itself.

The **singleton bound**: The distance d between codewords of an (n, k) code satisfies $d \leq n - k + 1$. Proof: Two (k, k) codewords are different in ≥ 1 symbol, and adding another $n - k$ symbols makes $d \leq n - k + 1$. \square

A (n, k) code is **linear** if encoding is equivalent to interpreting m of size k as a symbol vector and multiplying it by a code generator matrix G of size $k \times n$, using the appropriate arithmetic. A code is **systematic** if the first $k \times k$ submatrix of $G = I$. For systematic codes the message is part of the codeword, and the rest is effectively parity check bits—this improves efficiency and only store the $(n - k) \times k$ nonidentity part. Practical codes are linear and systematic.

Computation efficiency matters—e.g., can get high transmission speed if can encode/decode faster—this may outweigh any R gain due to better codes. Also, because codes are usually implemented in hardware, prefer codes with hardware-friendly implementations and parallelizability.

An effective way to use an error-correcting code is **refreshing**—periodically decode and encode the data. This corrects bit errors after they are introduced, but before there are too many to correct.

19.6 Finite Field Computation

With regular arithmetic \exists a field with $m = 8$ or another nonprime value—need other ways to construct it. Binary polynomials of degree $< m$ allow construction of such a **Galois field**, called $\text{GF}(2, m)$. For $m \leq$ word size such polynomials are most efficiently represented as integers. This is a somewhat unusual concept—binary polynomials serve as numbers—in particular as coefficients to other polynomials. They aren't numbers by not mapping directly to $[0, 2^{m-1}]$, but the number of them is the same, and have equivalents of 0 (additive identity) and 1 (multiplicative identity) (these match the numbers actually). The behavior is no different than using normal polynomials with coefficients modulo a prime. But instead of dealing with the largest prime that fits a given word, can work with the whole word range and keep the properties of polynomials with field coefficients.

Unlike with primes, field arithmetic isn't directly efficient for multiplication and inversion. But from abstract algebra know that every nonzero element is a power of some element which isn't 0 or 1 such as x (2 in the field) and is usually called **alpha**. Because m is usually small, precomputed tables map from the power representation and back.

To compute the tables, given $a = x$ and a primitive polynomial p , the first element is 0, and the next $n - 1$ are $a^i \% p$ for $0 \leq i < n - 1$. Incremental calculation avoids the modulus: $x' = x^{i-1} \ll 1$, so subtract p from x' if $x' \geq n$.

```
class GF2mArithmetic
{
    int n;
    Vector<int> el2p, p2el;
public:
    int one() const {return 1;}
    int alpha() const {return 2;}
    GF2mArithmetic(int primPoly)
    { // m is the highest set bit of polynomial
        int m = lgFloor(primPoly);
        assert(m <= 16); // avoid using too much memory
        n = twoPower(m);
        el2p = p2el = Vector<int>(n - 1); // 0 has no corresponding power
        p2el[0] = 1; // a^0 = 1, a^(n-1) also 1 so don't store it, and
        // implicitly el2p[1] = 0
        for(int p = 1; p < n - 1; ++p)
        { // calculate a^p from a^(p-1)
            int e = p2el[p - 1] << 1; // multiply by x
            if(e >= n) e = sub(e, primPoly); // reduce if needed
            el2p[e - 1] = p;
            p2el[p] = e;
        }
    }
    int elementToPower(int x) const
    {
        assert(x > 0 && x < n);
        return el2p[x - 1];
    }
}
```

```

int powerToElement(int x) const
{
    assert(x >= 0 && x < n - 1);
    return p2el[x];
} // both + and - just not
int add(int a, int b) const{return a ^ b;}
int sub(int a, int b) const{return add(a, b);}
int mult(int a, int b) const
{ // add in power basis and convert back
    return a == 0 || b == 0 ? 0 :
        powerToElement((elementToPower(a) + elementToPower(b)) % (n - 1));
}
int div(int a, int b) const
{ // subtract in power basis and convert back
    assert(b != 0);
    return a == 0 ? 0 : powerToElement((elementToPower(a) + (n - 1) -
        elementToPower(b)) % (n - 1));
}
};

```

19.7 Polynomials over Galois Field Elements

The next step is to define polynomials over coefficients from a finite field. This doesn't need any special devices other than doing all arithmetic in the field, and all operations use high-school algebra. As with large numbers (see the "Large Numbers" chapter):

- The lower-order coefficients are in the beginning of the storage vector
- Represent 0 by a vector of size 1 containing the 0 coefficient
- Trim leading 0's after every operation if needed.

```

template<typename ITEM, typename ARITHMETIC>
struct Poly: public ArithmeticType<Poly<ITEM, ARITHMETIC> >
{
    Vector<ITEM> storage;
    ARITHMETIC ari;
public:
    int getSize() const{return storage.getSize();}
    int degree() const{return getSize() - 1;}
    Poly(ARITHMETIC const& theAri, Vector<ITEM> const& coefs = //default is 0
        Vector<ITEM>(1, 0)): ari(theAri), storage(coefs)
    {
        assert(getSize() > 0);
        trim();
    }
    static Poly zero(ARITHMETIC const& theAri){return Poly(theAri);}
    ITEM const& operator[](int i) const{return storage[i];}
    void trim()
    {
        while(getSize() > 1 && storage.lastItem() == 0) storage.removeLast();
    }
    Poly& operator+=(Poly const& rhs)
    { //add term-by-term, no carry
        while(getSize() < rhs.getSize()) storage.append(0);
        for(int i = 0; i < min(getSize(), rhs.getSize()); ++i)
            storage[i] = ari.add(storage[i], rhs[i]);
        trim();
        return *this;
    }
    Poly& operator-=(Poly const& rhs)
    { //subtract term-by-term, no carry
        while(getSize() < rhs.getSize()) storage.append(0);
        for(int i = 0; i < min(getSize(), rhs.getSize()); ++i)
            storage[i] = ari.sub(storage[i], rhs[i]);
        trim();
        return *this;
    }
};

```

```

Poly& operator*(ITEM const& a)
{
    for(int i = 0; i < getSize(); ++i)
        storage[i] = ari.mult(storage[i], a);
    trim();
    return *this;
}
Poly operator*(ITEM const& a) const
{
    Poly temp(*this);
    temp *= a;
    return temp;
}
Poly& operator<<=(int p)
{
    assert(p >= 0);
    if(p > 0)
    {
        for(int i = 0; i < p; ++i) storage.append(0);
        for(int i = getSize() - 1; i >= p; --i)
        {
            storage[i] = storage[i - p];
            storage[i - p] = 0;
        }
    }
    return *this;
}
Poly& operator>>=(int p)
{
    assert(p >= 0);
    if(p >= getSize()) storage = Vector<ITEM>(1);
    if(p > 0)
    {
        for(int i = 0; i < getSize() - p; ++i) storage[i] = storage[i + p];
        for(int i = 0; i < p; ++i) storage.removeLast();
    }
    return *this;
}
Poly& operator*=(Poly const& rhs)
// multiply each term of rhs and sum up
{
    Poly temp(*this);
    *this *= rhs[0];
    for(int i = 1; i < rhs.getSize(); ++i)
    {
        temp <<= 1;
        *this += temp * rhs[i];
    }
    return *this;
}
static Poly makeX(ARITHMETIC const& ari)
// x = 1 * x + 0 * 1
{
    Vector<ITEM> coefs(2);
    coefs[1] = ari.one();
    return Poly(ari, coefs);
}
Poly& reduce(Poly const& rhs, Poly& q)
// quotient-remainder division, similar to numbers
{
    assert(rhs.storage.lastItem() != 0 && q == zero(ari));
    Poly one(ari, Vector<ITEM>(1, ari.one()));
    while(getSize() >= rhs.getSize())
    // field guarantees exact division
    {
        int diff = getSize() - rhs.getSize();
        ITEM temp2 = ari.div(storage.lastItem(), rhs.storage.lastItem());
        for(int i = 0; i < diff; ++i)
            storage.append(0);
        storage[diff] = temp2;
    }
    trim();
    return *this;
}

```

```

        assert(storage.lastItem() ==
               ari.mult(temp2, rhs.storage.lastItem()));
        *this -= (rhs << diff) * temp2;
        q += (one << diff) * temp2;
    }
    return *this;
}
Poly& operator%=(Poly const& rhs)
{
    Poly dummyQ(ari);
    return reduce(rhs, dummyQ);
}
bool operator==(Poly const& rhs) const
{
    if(getSize() != rhs.getSize()) return false;
    for(int i = 0; i < getSize(); ++i) if(storage[i] != rhs[i]) return false;
    return true;
}
ITEM eval(ITEM const& x) const
{//Horner's algorithm
    ITEM result = storage[0], xpower = x;
    for(int i = 1; i < getSize(); ++i)
    {
        result = ari.add(result, ari.mult(xpower, storage[i]));
        xpower = ari.mult(xpower, x);
    }
    return result;
}
};

```

A **formal derivative** of a polynomial is the same as the usual derivative, but using field arithmetic. I.e., ax^p becomes apx^{p-1} . Polynomial powers aren't field elements, so add a p times.

```

Poly formalDeriv() const
{
    Vector<ITEM> coefs(getSize() - 1);
    for(int i = 0; i < coefs.getSize(); ++i)
        for(int j = 0; j < i + 1; ++j)
            coefs[i] = ari.add(coefs[i], storage[i + 1]);
    return Poly(ari, coefs);
}
};

```

19.8 Reed–Solomon Codes

Let $t = n - k$. RS codes satisfy the singleton bound directly and can correct $t/2$ errors, so assume odd k . The m symbols are interpreted as values in some finite field that form polynomial coefficients. For simplicity assume that field is GF(2, 8), so that the coefficients are bytes. This also fixes the block length n to 255. k can be any odd value $< n$ depending on application-specific knowledge about how much correction will be needed; 223 is a typical choice. The primitive polynomial choice for the field makes little difference, so use the standard 301, corresponding to $ax^2 + 1$ where a is binary polynomial $x + 1$ (Moon 2021).

For setup compute the **generator polynomial** $g(x) = \prod_{0 \leq i < t} (x - a_i)$; a can be any element $\neq 0$ and $\neq 1$; here use 2.

```

class ReedSolomon
{
    int n, k;
    GF2mArithmetic ari;
    typedef Poly<unsigned char, GF2mArithmetic> P;
    typedef Vector<unsigned char> V;
    P generator;

public:
    ReedSolomon(int theK = 223, int primPoly = 301): ari(primPoly),

```

```

generator(ari, V(1, 1)), k(theK),
n(twoPower(lgFloor(primPoly)) - 1)
{
    assert(k < n && numeric_limits<unsigned char>::digits == 8);
    P x = P::makeX(ari);
    for(int i = 0, aPower = ari.alpha(); i < n - k; ++i)
    {
        generator *= (x - P(ari, V(1, aPower)));
        aPower = ari.mult(aPower, ari.alpha());
    }
    assert(generator.getSize() == n - k + 1);
}
};


```

To encode a block:

1. Create a polynomial $m(x)$ from the message coefficients
2. $c(x) = (m(x) \ll t) + (m(x) \ll t) \% g(x)$
3. Return the coefficients of c , padded with trimmed 0's to maintain block length if necessary

```

V encodeBlock(V const& block) const
{
    assert(block.getSize() == k);
    P c(ari, block); //init c
    c <= (n - k); //make space for code
    c += c % generator; //add code
    //beware of poly trim if block is 0
    while(c.storage.getSize() < n) c.storage.append(0);
    return c.storage;
}


```

An arbitrary message must be broken up into whole blocks, which are padded with 0's as necessary.

```

V lengthPadBlock(V block)
{
    assert(block.getSize() < k);
    block.append(block.getSize());
    while(block.getSize() < k) block.append(0);
    return block;
}
pair<V, bool> lengthUnpadBlock(V block)
{
    assert(block.getSize() == k);
    while(block.getSize() >= 0 && block.lastItem() == 0) block.removeLast();
    bool correct = block.getSize() >= 0 &&
        block.lastItem() == block.getSize() - 1;
    assert(correct);
    if(correct) block.removeLast();
    return make_pair(block, correct);
}


```

The decoding is more complicated:

1. Calculate the syndrome polynomial $s(x)$. $s[i] = c(a^i)$, for the corresponding a^i of g . If $s(x) = 0$, no errors are detected (but are possible as for CRC).
2. Find the error locator $\Lambda(x)$ and the error evaluator $\Omega(x)$ polynomials (discussed later). Normalize both by dividing them by $\Lambda[0]$. If $\Lambda(x) = 0$, report failure.
3. Find the roots of $\Lambda(x)$ by evaluating it at every nonzero field elements. Report failure if found none. Given a root r , the power representation of its inverse r^{-1} is the corresponding error location, and the error value $= -\Omega(r^{-1})/\Lambda'(r^{-1})$, where Λ' is the formal derivative of Λ .
4. Correct the errors in c using $c[i] = c[i] + \text{error}[i]$.
5. If $c(x) \% g(x) = 0$, report failure.

Field arithmetic is used in all steps. For proofs of correctness of each step see Moon (2021).

```

pair<V, bool> decodeBlock(V const& code) const
{//calculate syndrome polynomial
    assert(code.getSize() == n);
    P c(ari, code);


```

```

int t = n - k, aPower = ari.alpha();
V syndromes(t);
for(int i = 0; i < t; ++i)
{
    syndromes[i] = c.eval(aPower);
    aPower = ari.mult(aPower, ari.alpha());
}
P s(ari, syndromes);
if(s == P::zero(ari)) //no error if yes
{//take out check data and restore trimmed 0's
    c >>- t;
    while(c.storage.getSize() < k) c.storage.append(0);
    return make_pair(c.storage, true);
}//find locator and evaluator polys
pair<P, P> locEv = findLocatorAndEvaluator(s, t);
if(locEv.first == P::zero(ari)) return make_pair(code, false);
//find locator roots
V roots;
for(int i = 1; i < n + 1; ++i)
    if(locEv.first.eval(i) == 0) roots.append(i);
if(roots.getSize() == 0) return make_pair(code, false);
//find error values
P fd = locEv.first.formalDeriv();
V errors;
for(int i = 0; i < roots.getSize(); ++i) errors.append(ari.sub(0,
    ari.div(locEv.second.eval(roots[i]), fd.eval(roots[i]))));
//correct errors
while(c.storage.getSize() < n) c.storage.append(0);
for(int i = 0; i < roots.getSize(); ++i)
{
    int location = ari.elementToPower(ari.div(ari.one(), roots[i]));
    assert(location < c.getSize());
    c.storage[location] = ari.add(c.storage[location], errors[i]);
}
if(c % generator != P::zero(ari)) return make_pair(code, false);
c >>= t;
return make_pair(c.storage, true);
}

```

$\Lambda(x)$ and $\Omega(x)$ are found by a modification of the extended Euclidean algorithm:

1. Start with $\Lambda_0 = 1$, $\Lambda_{-1} = 0$, $\Omega_0 = s(x)$, $\Omega_{-1} = x^t$
2. While $\text{degree}(\Omega) \geq t/2$
3. $\Omega_{i+1} = \Omega_{i-1} \% \Omega_i$, with quotient q
4. $\Lambda_{i+1} = \Lambda_{i-1} - q\Lambda_i$

```

pair<P, P> findLocatorAndEvaluator(P const& syndromePoly, int t) const
{
    P evPrev(ari, V(1, ari.one())), ev = syndromePoly,
    locPrev = P::zero(ari), loc = evPrev;
    evPrev <= t;
    while(ev.degree() >= t/2)
    {
        P q(ari);
        evPrev.reduce(ev, q);
        swap(ev, evPrev);
        locPrev -= q * loc;
        swap(loc, locPrev);
    }//normalize them
    if(loc != P::zero(ari))
    {
        int normalizer = ari.div(ari.one(), loc[0]);
        loc *= normalizer;
        ev *= normalizer;
    }
}

```

```

    return make_pair(loc, ev);
}

```

RS codes are linear because polynomial operations correspond to matrix multiplication. They also allow correcting t **erasures** (i.e., errors with known location), or a mixture of errors and erasures (Moon 2021). So many applications, such as item bar codes, only use erasure decoding.

RS codes aren't as effective at correcting random bit errors because with large n only can guarantee correcting d symbol errors. But if **burst errors** that affect multiple consecutive bits are likely (e.g., CD/DVD disk scratches), RS codes do well.

19.9 Bounds on Fixed-alphabet Minimum-distance Codes

RS codes meet the singleton bound by increasing the alphabet with n so that each symbol uses $\lceil \lg(n) \rceil$ bits. Many other codes work directly with bits or other fixed-size symbols, independent of n . Asymptotically, as $n \rightarrow \infty$, all minimum-distance codes are subject to some bounds (Moon 2021; Guruswami 2010). Let $b = \frac{d}{n}$.

Then for binary symbols (these extend to larger alphabets using generalized entropy) asymptotically:

- **Singleton:** $R \leq 1 - b$
- **Plotkin:** $R \leq 1 - 2b$. Strictly better than the singleton bound. That RS codes meet the latter isn't a contradiction because alphabet size $\rightarrow \infty$ makes better bounds collapse to singleton.
- **Gilbert-Varshamov:** $R \geq 1 - H(b)$. A lower bound, but is conjectured to be an upper bound up to $o(1)$ (Guruswami 2010). The expression is essentially identical to the one giving the capacity of BSC.

For BSC, as $n \rightarrow \infty$, the fraction of errors is exactly p (e.g., by Hoeffding's inequality), i.e., need $b > 2p$ to guarantee correction. So the effective capacity for minimum-distance codes = $1 - H(2b)$. This is below C , and no reliable communication is possible for $p \geq \frac{1}{4}$, which isn't the case for general codes. So all fixed-alphabet minimum-distance codes are flawed asymptotically. But for n up to 1000 or so, some binary codes have good performance (such as BCH; see the "Comments" section). Maximum likelihood decoding is able to decode beyond d , and modern codes are based on it.

19.10 Boolean Matrices

Boolean matrix algebra is a building block for probabilistic codes. The behavior is the same as for regular matrices (see the "Numerical Algorithms—Introduction and Matrix Algebra" chapter) with Boolean coefficients, but the implementation is more memory-efficient because use a bit set as storage. And get bit-parallelism for some operations.

```

class BooleanMatrix: public ArithmeticType<BooleanMatrix>
{
    int rows, columns;
    int index(int row, int column) const
    {
        assert(row >= 0 && row < rows && column >= 0 && column < columns);
        return row + column * rows;
    }
    Bitset<> items;
public:
    BooleanMatrix(int theRows, int theColumns): rows(theRows),
        columns(theColumns), items(theRows * theColumns)
        {assert(items.getSize() > 0);}
    int getRows() const {return rows;}
    int getColumns() const {return columns;}
    bool operator()(int row, int column) const
        {return items[index(row, column)];}
    void set(int row, int column, bool value = true)
        {items.set(index(row, column), value);}
    BooleanMatrix operator*=(bool scalar)
    {
        if(!scalar) items.setAll(false);
        return *this;
    }
    friend BooleanMatrix operator*(bool scalar, BooleanMatrix const& a)
    {

```

```

        BooleanMatrix result(a);
        return result *= scalar;
    }

    friend BooleanMatrix operator*(BooleanMatrix const& a, bool scalar)
    {return scalar * a;}
    BooleanMatrix& operator+=(BooleanMatrix const& rhs)
    { // and - are both xor
        assert(rows == rhs.rows && columns == rhs.columns);
        items ^= rhs.items;
        return *this;
    }

    BooleanMatrix& operator-=(BooleanMatrix const& rhs){return *this += rhs; }

    BooleanMatrix& operator*=(BooleanMatrix const& rhs)
    { // the usual row by column
        assert(columns == rhs.rows);
        BooleanMatrix result(rows, rhs.columns);
        for(int i = 0; i < rows; ++i)
            for(int j = 0; j < rhs.columns; ++j)
            {
                bool sum = false;
                for(int k = 0; k < columns; ++k)
                    sum ^= (*this)(i, k) * rhs(k, j);
                result.set(i, j, result(i, j) ^ sum);
            }
        return *this = result;
    }

    Bitset<> operator*(Bitset<> const& v) const
    { // matrix * vector
        assert(columns == v.getSize());
        Bitset<> result(rows);
        for(int i = 0; i < rows; ++i)
            for(int j = 0; j < columns; ++j)
                result.set(i, result[i] ^ ((*this)(i, j) * v[j]));
        return result;
    }

    friend Bitset<> operator*(Bitset<> const& v, BooleanMatrix const& m)
    {return m.transpose() * v; }

    static BooleanMatrix identity(int n)
    {
        BooleanMatrix result(n, n);
        for(int i = 0; i < n; ++i) result.set(i, i);
        return result;
    }

    BooleanMatrix transpose() const
    {
        BooleanMatrix result(columns, rows);
        for(int i = 0; i < rows; ++i)
            for(int j = 0; j < columns; ++j) result.set(j, i, (*this)(i, j));
        return result;
    }

    bool operator==(BooleanMatrix const& rhs)
    {
        if(rows != rhs.rows || columns != rhs.columns) return false;
        return items == rhs.items;
    }
};

```

19.11 Low-density Parity-check Codes

Given m , here interpreted as a column vector, a linear code uses a systematic generator matrix G to compute codeword $w = Gm$. With $G = [X|I]^T$ ("|" is the augmentation notation), $H = [I|X]$ is a corresponding **parity-check matrix** such that:

- $HG = 0$

- \forall code word $w, Hw = 0$

LDPC codes are binary, created by designing H , and deriving G by extracting X . A sparse parity-check matrix A is created and transformed into H using the fact that G or H transformed by column permutations or elementary row operations produce the same code (up to a bit permutation corresponding to the column permutation). The resulting H and G aren't sparse, so discard H and use A for decoding.

Let $t=n-k$, and w_c and w_r be such that $\frac{n}{w_r} = \frac{t}{w_c}$, with no-remainder division (this defines a **regular**

LDPC code; Moon 2021). Typically set $w_c = 3$ (smaller values are ineffective and larger not necessary), and compute w_r from the above equation.

To create A :

1. Create A' —in row r for $0 \leq r < \frac{t}{w_c}$, set w_r consecutive bits from position rw_r .
 2. Stack w_c permutations of A' , such that the first is the identity permutation, and the rest are random.

The resulting A isn't necessarily of full rank, meaning that some rows are linearly dependent. So during Gaussian elimination with column pivoting during creation of H , no column can have a 1 in the worked-on row. So the row is has all 0's, and is skipped; this is why use column pivoting instead of the more common row pivoting. So R can be a little larger than designed, which needs to be accounted for during picking R . Alternatively can repeat generation until have full rank, but may need many repeats. A hybrid strategy is to repeat until have some minimum number of columns. These strategies aren't considered further here.

Another trick is that whenever columns are swapped during pivoting, the same swap must be applied to A.

The 0 rows are then removed from H , leading to a $t' \times n$ matrix H' . It's systematic so G is derived from it and is a $n' \times k'$ matrix, for $k' = n - t'$. Despite row removal, have $AG = 0$, so A is a valid parity-check matrix for G .

```
"a" a
1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0
0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1
0 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 "g" g
0 0 1 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 1
1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1
0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0
0 0 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 0
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 1
"hi" h' 0 1 0 0 0 1 1 0
1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 0 0 1 0 1 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 0 1 0 0 0 1 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0
```

Figure 1: Example calculation of g from A with $n = 20$ and $k = 7$.

```

class LDPC
{
    BooleanMatrix a, q; // sparsity of A not exploited here
    Bitset<> extractMessage(Bitset<> const &code) const

```

```

{
    int k = getNewK(), n = code.getSize();
    Bitset<> message(k);
    for(int i = 0; i < k; ++i) message.set(i, code[i + n - k]);
    return message;
}
public:
    int getNewK() const {return g.getColumns();}
    LDPC(int n, int k, int wc = 3): a(n - k, n), g(n, n - k)
    {
        int t = n - k, wr = n / (t / wc);
        assert(t % wc == 0 && n % wr == 0 && wc * n == wr * t && t < n);
        //create a
        for(int r = 0; r < t / wc; ++r) //the first section
            for(int c = 0; c < wr; ++c) a.set(r, c + r * wr);
        Vector<int> perm(n);
        for(int c = 0; c < n; ++c) perm[c] = c;
        for(int i = 1; i < wc; ++i) //other sections as permutations of the
        //first
        GlobalRNG().randomPermutation(perm.getArray(), n);
        for(int r = 0; r < t / wc; ++r)
            for(int c = 0; c < wr; ++c)
                a.set(r + i * t / wc, perm[c + r * wr]);
        //create H from A
        BooleanMatrix h = a;
        int skip = 0;
        for(int r = 0; r < t; ++r)
        //find column with 1, if not return
            int cNow = r - skip, c = cNow;
            for(; c < n; ++c) if(h(r, c)) break;
            if(c == n) ++skip; //all-0 row
            else if(c != cNow) //swap columns cNow and c
                for(int rb = 0; rb < t; ++rb)
                {
                    bool temp = h(rb, cNow);
                    h.set(rb, cNow, h(rb, c));
                    h.set(rb, c, temp);
                    //same for a
                    temp = a(rb, cNow);
                    a.set(rb, cNow, a(rb, c));
                    a.set(rb, c, temp);
                }
                for(int rb = 0; rb < t; ++rb)
                    if(rb != r && h(rb, cNow))
                        for(c = cNow; c < n; ++c)
                            h.set(rb, c, h(rb, c) ^ h(r, c));
        //remove 0 rows from h
        int tProper = t - skip, delta = 0;
        BooleanMatrix hNew(tProper, n);
        for(int r = 0; r < tProper; ++r)
        //nonzero rows have correct identity part set
            while(!h(r + delta, r) && r < tProper) ++delta;
            for(int c = 0; c < n; ++c) hNew.set(r, c, h(r + delta, c));
        //create g from h
        int kProper = n - tProper;
        g = BooleanMatrix(n, kProper);
        for(int r = 0; r < n; ++r)
            for(int c = 0; c < kProper; ++c)
                if(r < tProper) g.set(r, c, hNew(r, tProper + c)); //x part
                else g.set(r, c, r - tProper == c); //identity part
        assert(a * g == BooleanMatrix(t, kProper));
    }
    Bitset<> encode(Bitset<> const& message) const

```

```

    {
        assert(message.getSize() == getNewK());
        return g * message;
    }
};

```

Decoding for BSC needs knowing p . But the algorithm isn't sensitive to the exact value, so heuristically assume $C = R$, and use numerical equation solving to find the corresponding p .

Decoding uses maximum likelihood (see the "Computational Statistics" chapter). Given that a bit can only take on two values, compute log of the likelihood ratio \forall bit. A priori,

$$I_{\text{initial}}(b) = \ln \left(\frac{\Pr(\text{bit} = 1 | \text{received } b)}{\Pr(\text{bit} = 0 | \text{received } b)} \right) = \ln \left(\frac{1-p}{p} \right) (b ? 1 : -1).$$

Keep a "matrix" nu (implemented by a hash table) that has "partial" likelihoods in positions where A has set bits. For a given r , let c' be c such that $A[r, c]$ is set. Also, let $I_c(c') = \tanh \left(\frac{nu[r, c'] - I[r]}{2} \right)$ be "conditional likelihoods" (see Moon 2021 for the mathematics). Then to decode:

1. Set all nu entries = 0
2. Set all I entries = $I_{\text{initial}}(\text{code}[c])$
3. $\text{corrected} = \text{code}$
4. Until $A \times \text{corrected} \neq 0$ or run out of iterations
5. \forall row r of A
6. $\text{temp} = \prod I_c(c')$
7. $\forall c' \quad nu[r, c'] = -2 \tanh^{-1}(\text{temp}/I_c(c'))$
8. \forall column c of A
9. $I[c] = I_{\text{initial}}(\text{code}[c]) + \sum nu[r, c]$
10. $\text{corrected}[c] = (I[c] > 0)$

```

struct H0Functor //for numerical solving for p
{
    double hValue;
    double H(double p) const { return p > 0 ? p * log2(1/p) : 0; }
    double operator() (double p) const { return H(p) + H(1 - p) - hValue; }
    H0Functor(double theHValue) : hValue(theHValue) {}

};

double pFromCapacity(double capacity) const //solver guaranteed to succeed
    { return solveFor0(H0Functor(1 - capacity), 0, 0.5).first; }

unsigned int uIndex(unsigned int r, unsigned int c) const
    { return r * a.getColumns() + c; }

pair<Bitset<>, bool> decode(Bitset<> const &code, int maxIter = 1000,
    double p = -1) const
{
    int n = a.getColumns(), k = getNewK(), t = a.getRows();
    assert(code.getSize() == n && maxIter > 0);
    Bitset<> zero(k), corrected = code;
    if(a * code == zero) return make_pair(extractMessage(code), true);
    if(p == -1) p = pFromCapacity(1.0 * k/n); //find p if not given
    double const llr1 = log((1 - p)/p); //initialize l
    Vector<double> l(n);
    for(int i = 0; i < n; ++i) l[i] = llr1 * (code[i] ? 1 : -1);
    LinearProbingHashTable<unsigned int, double> nu; //initialize nu
    for(int r = 0; r < t; ++r) for(int c = 0; c < n; ++c) if(a(r, c))
        nu.insert(uIndex(r, c), 0);
    while(a * corrected != zero && maxIter-- > 0) //main loop
    { //update nu
        for(int r = 0; r < t; ++r)
        {
            double temp = 1;
            for(int c = 0; c < n; ++c) if(a(r, c))
                temp *= tanh((*nu.find(uIndex(r, c)) - l[c])/2);
            for(int c = 0; c < n; ++c) if(a(r, c))
            {

```

```

        double *nuv = nu.find(uIndex(r, c)), product = temp/
            tanh((*nuv - l[c])/2), value = -2 * atanh(product);
            //set numerical infinities to heuristic 100
            if(!isfinite(value)) value = 100 * (product > 0 ? -1 : 1);
            *nuv = value;
    }
} //update l and the correction
for(int c = 0; c < n; ++c)
{
    l[c] = llrl * (code[c] ? 1 : -1);
    for(int r = 0; r < t; ++r) if(a(r, c))
        l[c] += *nu.find(uIndex(r, c));
    corrected.set(c, l[c] > 0);
}
bool succeeded = maxIter > 0;
return make_pair(succeeded ? extractMessage(corrected) : code,
    succeeded);
}

```

As presented, an iteration of decoding takes $O(nt)$. Sparse matrix algebra cuts it down to $O(n)$. Here need a sparse matrix data structure that allows both row and column iterations.

As for the RS code implementation, instead of sending message blocks of specific bit size, e.g., can use 2 bytes (more generally $\lceil \lg(k) \rceil$ bits) to encode the length of a message block in bits and set the unused bits to 0.

19.12 Implementation Notes

The implementations follow the textbooks. The only creativity is in the algorithm selection. A mistake was adding extensive tests much later, which resulted in bugs.

19.13 Comments

☰ many other useful channel models. Some of the most important ones (Moon 2021):

- **Binary erasure channel (BEC)**—unlike for BSC, bits don't flip but become unrecognizable with some probability.
- **Gaussian channel**—data transmission uses high- and low-current signals to respectively represent "1" and "0". Errors are better modeled by a normal distribution with some variance. Decoding Gaussian output directly before converting back to binary is somewhat more effective due to not losing information by discretizing, but the resulting codes can be decoded only in hardware where this information is known.

Nonlinear codes have been explored much less and don't have any advantages.

Another frequently presented way to decode RS is **Berlekamp–Massey algorithm**. It's much more complicated though and has about the same performance, so no reason to prefer it. RS codes are popular due to efficient erasure decoding. Nonbinary LDPC have been proposed and compete with RS here, but without as much advantage as for binary codes.

Some other important codes include (Moon 2021):

- **BCH**—a generalization of RS that decouples block size from the symbol size by having different input and output fields. Usually consider only binary symbols. Don't even meet Gilbert-Varshamov bound (Guruswami 2010; see this for a detailed performance analysis). But with small n for the corresponding rates these are among the best-rate codes and are also simple and efficient. Though LDPC with larger n get better rates, for some applications BCH is enough.
- **Turbo**—their patent expired (Wikipedia, 2016b). They are much more complicated than LDPC, have slower the decoding iterations, and aren't easy to design. But they do better with low-capacity channels. Current LDPC research is attempting to close that gap.

For more on **iterative codes** (i.e., LDPC and turbo) see Johnson (2009). In particular, regular LDPC may not be the best possible, and an irregular construction which also prevents cycles of size 4 (a particular configuration of parity bits) is recommended. Current LDPC research focuses on deterministic constructions to improve efficiency. Also, have a way to use A to encode directly, without computing dense G , so that all operations are with sparse matrices (Moon 2021; Johnson 2009); this is complicated and not implemented here.

Most other codes are obsolete despite being included in many older standards and protocols. Most of

those that are still discussed in textbooks were the best-performing when invented, at least for special cases.

19.14 Projects

- Allow to construct a CRC from a stream.
- Research 64-bit polynomials for CRC, and implement 64-bit CRC based on one.
- Change Boolean matrix multiplication to use the cross-product form (see the “Numerical Algorithms—Introduction and Matrix Algebra” for details). This allows the final addition of matrices to be bit-parallel, with overall performance $O(n^3/w)$.
- Use a buffer to simulate decoding blocks from a stream with a low error rate. Extend this to also include compression and cryptography algorithms from the corresponding chapters.
- For LDPC, generate H' several times, and take the best result. Investigate how many repetitions are useful theoretically.
- Implement a sparse Boolean matrix. Consider reusing the general sparse matrix code (see the “Numerical Algorithms—Introduction and Matrix Algebra” chapter), but don't forget that need binary algebra.

19.15 References

- Guruswami, V. (2010). Introduction to Coding Theory. Course notes, <http://www.cs.cmu.edu/~venkatg/teaching/codingtheory/>. Accessed October 16, 2016.
- Johnson, S. J. (2009). *Iterative Error Correction: Turbo, Low-Density Parity-Check and Repeat-Accumulate Codes*. Cambridge.
- Koopman, P. (2002). 32-bit cyclic redundancy codes for internet applications. *International Conference on Dependable Systems and Networks* (pp. 459–468). IEEE.
- Moon, T. K. (2021). *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley.
- Rorabaugh, C. B. (1996). *Error Coding Cookbook: Practical C/C++ Routines and Recipes for Error Detection and Correction*. McGraw-Hill.
- Williams, R. (1993). A painless guide to CRC error detection algorithms. http://www.repairfaq.org/filipg/LINK/F_crc_v3.html. Accessed May 18, 2013.
- Wikipedia (2016a). Reed–Solomon error correction. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction. Accessed October 1, 2016.
- Wikipedia (2016b). Turbo code. https://en.wikipedia.org/wiki/Turbo_code. Accessed October 16, 2016.
- Wikipedia (2018). Cyclic redundancy check. https://en.wikipedia.org/wiki/Cyclic_redundancy_check. Accessed September 23, 2018.

20 Cryptography

20.1 Introduction

This chapter is only a basic introduction to some main ideas of cryptography. It completes the typical pipeline of compression, encryption, and error coding before communicating a file. Cryptography is very wide field and need very complicated implementations, so no attempt is made to get into the details. The references cited don't even cover latest developments and standards. If need cryptographic functionality, use a good library.

Want to secure various **communication protocols** such as two parties exchanging secret messages. Protocols are usually modeled by interactions between several parties. The conventional ones are:

- **Alice, Bob, Carol, David**, etc.—communicating parties.
- **Eve**—can see all messages.
- **Mallory**—can see and modify all messages. Though Mallory can drop or garbage all messages, want to ensure that Mallory can't profit from hacking them.
- **Trent**—a trusted authority able to confirm a party's identity.

Every party is assumed to know the protocol, and security relies on a **password key** that is known only to the authorized parties, and without which it's infeasible to get any useful information about the communication. E.g., someone wanting to read your email may know your service provider and username but not the password.

20.2 File Encryption

Many unsecure methods were used historically to **encrypt** a file so that can't **decrypt** it without the key. Julius Caesar rotated the alphabet by 3 so that *a* became *d*, *x a*, etc. In "The Dancing Men", letters corresponded to dancing figures, and Sherlock Holmes used the *etnorias* heuristic to break the code, substituting most likely letters for most repeated figures assuming the most common is *e*, the second most common *t*, etc., and backtracking wrong guesses.

Let r be a random bit string of size n . Given an n -bit message m , $\text{code}(m) = m \wedge r$, and $\text{message}(\text{code } c) = c \wedge r$. Can't decode c without r because every n -bit m is equally likely. This **one-time pad** is impractical because if use r for another message, $c_1 \wedge c_2 = m_1 \wedge m_2$, which leaks information.

A **stream cipher** initializes a secure random number generator with a sufficiently random seed and produces a stream of random bytes to form r . Guessing the future bytes from the past ones is infeasible if the encryption doesn't reuse the same seed. Using a checksum before encryption allows detecting bit errors in the encrypted file with high probability. To encrypt using RC4 and CRC:

1. Create a RC4 from a key and a Xorshift-transformed sequence from the current time
2. Calculate and append a CRC to the file
3. Encrypt the result with the stream
4. Append the time to it

Use a seed in addition to a password so that, e.g., repeated messages aren't recognized as such. It's safe to use time or a counter as seed if not reusing time-key combinations.

```
void applyARC4(uint32_t seed, Vector<unsigned char> temp,
               Vector<unsigned char>& data)
{
    for(int i = 0; i < temp.getSize(); ++i)
        temp[i] ^= (seed = xorshiftTransform(seed));
    ARC4 arc4(temp.getArray(), temp.getSize());
    for(int i = 0; i < data.getSize(); ++i) data[i] ^= arc4.nextByte();
}

Vector<unsigned char> simpleEncrypt(Vector<unsigned char> data,
                                      Vector<unsigned char> const& key)
{
    uint32_t seed = time(0), s = sizeof(int);
    CRC32 crc32;
    Vector<unsigned char> theSeed = ReinterpretEncode(seed, s), crc =
        ReinterpretEncode(crc32.hash(data.getArray()), data.getSize(), s);
    for(int i = 0; i < s; ++i) data.append(crc[i]);
```

```

applyARC4(seed, key, data);
for(int i = 0; i < s; ++i) data.append(theSeed[i]);
return data;
}

```

Decryption:

1. Read the seed
2. Remove its bytes
3. Read and decrypt the rest using the seed
4. Read the CRC
5. Remove its bytes
6. Make sure the file matches the CRC

```

pair<Vector<unsigned char>, bool> simpleDecrypt(Vector<unsigned char> code,
    Vector<unsigned char> const& key)
{
    assert(code.getSize() >= 8);
    enum{s = sizeof(uint32_t)};
    Vector<unsigned char> seed, crc;
    for(int i = 0; i < s; ++i) seed.append(code[code.getSize() + i - 4]);
    for(int i = 0; i < s; ++i) code.removeLast();
    applyARC4(ReinterpretDecode(seed), key, code);
    for(int i = 0; i < s; ++i) crc.append(code[code.getSize() + i - 4]);
    for(int i = 0; i < s; ++i) code.removeLast();
    CRC32 crc32;
    return make_pair(code, crc32.hash(code.getArray(), code.getSize()) ==
        ReinterpretDecode(crc));
}

```

Both need $O(n)$ time.

Though this simple algorithm is convenient and probably good enough for simple applications, the more complicated **AES** (St Denis 2006) is the de facto standard for encryption with a password.

20.3 Key Length

A key should have a negligible probability of being guessed or enumerated. E.g., it may be feasible for a massively parallel system to check all 64-bit keys if can test each quickly. A key should be such that the most efficient attack needs an equivalent of brute-force enumeration of 128-bit keys, which is infeasible even in future. Beware:

- Using shorter keys is more efficient, but handling the data is usually the bottleneck.
- Every year hardware gets faster and research produces more efficient attacks.
- Some data must be secure for many years, so a key length chosen now must be secure in future.
- The value of the data and who wants it matter. E.g., protecting your credit card from small criminals needs less security than protecting top-secret information from other governments with code breakers and supercomputers working for years.

Can enumerate typical 8-character passwords, even if all letters are equally likely. Though in practice trying to break a key using brute force is hard even for small keys. Need to be able to recognize properly decrypted data from garbage. E.g., the data may be in an unknown language. For file passwords a simple heuristic is to check if all characters are ASCII—a fast and accurate test in many cases.

Character type	Number of possible passwords	Bits of security	Time to break at 10^9 passwords per second	Length needed for 128 bit security
Digits	10^8	27	0.13 seconds	39
Lowercase letters	26^8	38	275 seconds	27
Mixed case + digits	62^8	48	3.26 days	21
Keyboard nonspace characters	94^8	52	52.1 days	20
Bytes	256^8	64	585 years	16

Users tend to pick passwords consisting of concatenations of dictionary words, each with < 16 bits of

security. Rules like *must contain at least one nonletter* don't increase the bits per character by much because users will enter exactly one, and it's much more efficient to enumerate 6-letter-and-1-digit tuples than 7-letter-or-digit tuples. A rule such as lock after 10 consecutive bad attempts is effective at preventing such hack attempts (but this may be a nuisance to the user if the goal of the hack is to lock).

A secure and easy-to-remember password is a memorable or a shocking phrase, e.g., *When_Under_a_Gun_Say_Merry_Christmas_To_911*. A secure random number generator with a high-entropy seed generates the most secure keys.

20.4 Key Storage

Need to store or remember a key. If it's lost, the encrypted data is unrecoverable. Might find a key with access to the decryption device because running it in a debugger may show the key. E.g., keys have been recovered from hardware devices by measuring heat emissions or freezing RAM chips. An application can't prevent such attacks.

The easiest way to get a key is to bribe someone or use **social engineering**. E.g., (Mitnick & Simon 2002): You get a traffic ticket, and in your city police officers attend training that takes priority over ticket court appearance. You call the precinct and pretend to be a lawyer needing to subpoena the officer who gave you the ticket and ask when he is unavailable. The helpful clerk gives you the dates. You ask for a hearing on one of them. The fair judge agrees. You show up and the officer doesn't—ticket canceled.

20.5 Cryptographic Hashing

Hash function h is **cryptographically secure** if computing:

- $h(x)$ from x is easy
- Any bits of x from $h(x)$ is hard
- y such that $h(y) = h(x)$ is hard
- y and z such that $h(y) = h(z)$ is hard

These force h to output ≥ 128 -bit results and be much less efficient than a universal hash function. **SHA3 algorithm** is the recommended choice (see the NIST website for details).

20.6 Key Exchange

Two parties who don't know each other don't have a common key. A common model is that Alice and Bob exchange messages, and Eve reads them. In another model, Mallory intercepts messages and pretends to be Bob when talking to Alice and Alice when talking to Bob, exchanging keys with each.

RSA algorithm (see St Denis 2006 for details) gives every party a **private key**, known only to the party, and a **public key**. A message is encrypted using the public key and decrypted using the private key. RSA doesn't suffer from the man-in-the-middle attack if Trent publishes the public key. E.g., web browsers have lists of trusted authorities such as Verisign. "Authority not recognized, do you want to proceed?" means fraud or that the website doesn't want to pay and acts as its own authority.

When designing a dedicated client and server, give the server a single private key and hard-code the public key into the client.

20.7 Other Protocols

Assume that every party wants to not be cheated and cheat every other party.

Protocol	Goal	Algorithm idea
Authentication	Verify user identity and protect stored passwords.	<ol style="list-style-type: none"> 1. ∀ account store the username, a long random string, and a hash of the password concatenated with a random string. 2. To authenticate, retrieve the user's random string, calculate the password hash, and check if it matches the stored result.
Commitment	Alice wants to commit a decision without revealing it to Bob, and Bob doesn't want Alice to change	<ol style="list-style-type: none"> 1. Bob sends Alice a long random string. 2. Alice appends her decision to it,

	it after committing.	encrypts the result with a random key, and sends it to Bob. 3. When it's time to reveal the decision, Alice sends Bob her key. Bob decrypts and verifies his random string.
Secret sharing	Give a secret to m parties, so that any k can determine it.	<ol style="list-style-type: none"> 1. Pick a prime $p > \max(k, \text{the largest possible secret})$. 2. Generate a random polynomial of degree $k - 1$. The coefficient of the 0^{th} term is the message. 3. Give party i the polynomial $\% p$, evaluated at i. 4. To construct the secret, k parties solve the corresponding k equations for the coefficients.

20.8 Implementations Notes

Implementing cryptography well in C++ in a portable way needs functionality from beyond the STL. In particular, need a secure source of random numbers. So despite my plan to make this a much larger chapter, I scaled back many implementations after realizing this when experimenting with the AES. Can encapsulate the cryptography to require a secure random seed as input, at least in the form of a black-box functor, which would allow good implementations of many of the algorithms. I chose to not take this route and instead developed a simple stream cipher based on a password-based seeding strategy.

20.9 Comments

Cryptography is a wide field. Due to special cases and efficient hard problem solvers, must take great care to make sure that a randomly generated seed or other information doesn't fall into those by accident. For well-established protocols such as RSA, the easy cases are well-known, though ignored by many sources. So the implementations are very complicated by necessity.

Can't rule out \forall protocol that special cases don't exist because they can be discovered in future. Theoretically, to prevent easy solving of hard problems must prove absence of efficient randomized or fixed-parameter algorithms or perhaps show that a phase transition occurs with high probability, etc (see the "Optimization" chapter). Security mostly relies on the belief that such cases are unlikely. Many protocols use extra scrambling, in case some issues will be discovered. E.g., for hashing, older **MD5** and **SHA1** algorithms are considered broken despite strong beliefs in their security years ago. **SHA3** was developed because the currently safe **SHA2** seemed likely to be broken soon.

The quality of the random generator and the seed is critical. Using password + time as a seed is a good heuristic, but ideally have a special-device random source. The latter can be the only choice because many algorithms such as secure hash functions work without needing a password.

The ability to update a protocol is very important. Germany might as well have lost WW2 because the British were able to break their Enigma. It was an improvement of the dancing men cipher that wouldn't yield to brute force human attacks but yielded to one of the first computers. Even experts in the field don't know for sure if the existing protocols are safe. At best can create libraries of robust implementations, and actively patch those with known attacks. A good protocol must resist attacks by the white hat community for years to be considered usable. Many are motivated to spend time breaking proposed protocols because success gives minor fame and good job opportunities.

A data communication/storage pipeline usually includes all of compression, error detection, encryption, and error correction. Use this order because encrypted data doesn't compress well, want to detect tampering with encrypted data, and want error correction to recover all encrypted data in case of errors to enable decryption.

20.10 Projects

- Improve portability in the decryption code by not assuming specific word sizes.

20.11 References

- Mitnick, K. D., & Simon, W. L. (2001). *The Art of Deception: Controlling the Human Element of Security*. Wiley.
- Schneier, B. (1995). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley.
- St Denis, T. (2006). *Cryptography for Developers*. Syngress.

21 Computational Statistics

"All models are wrong, but some are useful" – George Box

"It is easy to lie with statistics, but easier to lie without them" – Frederick Mosteller

"Never give an estimator without giving a confidence set" – Larry Wasserman

"To consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem" – Sir Ronald Fisher

"Anyone who conducts an argument by appealing to authority isn't using his intelligence; he is just using his memory" – Leonardo da Vinci

"There is no more common error than to assume that, because prolonged and accurate mathematical calculations have been made, the application of the result to some fact of nature is absolutely certain" – Alfred North Whitehead

"It is utterly implausible that a mathematical formula should make the future known to us, and those who think it can would once have believed in witchcraft" – Jacob Bernoulli

"It isn't what we don't know that gives us trouble, it's what we know that ain't so." – Will Rogers

21.1 Introduction

The material in this chapter can be considered useful mathematical statistics without the mathematics. The focus is on procedures for analyzing data and an overview of the theory needed to understand them. A basic prerequisite is a course in mathematical statistics. I emphasize estimation and confidence intervals and deemphasize hypothesis testing. Procedures with minimal assumptions are preferred where available, and practical use of the generic bootstrap procedure is discussed substantially, with some original material.

The main idea of statistics that drives all the details is that if data points follow some distribution, then a function of them follows another distribution, discovering which allows all kinds of inference.

21.2 Estimands

Probability theory describes behavior of known random phenomena. Statistics is the inverse process—estimate properties of a **data generator** from the data it generates. Need the following for this to make sense (Westfall & Henning 2013):

- Have a **model**—need scientific judgment to pick one, and it needn't correspond to the true generator even closely—work under the what if it does.
- Have n iid data points, assumed to be generated according to the model—often already available, but sometimes need to do experiments to collect it. Non-iid data is also allowed in some cases.
- Want to know the model's **parameters** to get its complete specification, but often only interested in some particular ones—and use the data to figure them out. The parameters needn't be part of any mathematical specification of the model, though they usually are.

E.g. (specific methods are discussed later in this or the "Machine Learning—Regression" chapters):

- Want the current median income of a country, so estimate it from survey data (not the mean because don't want billionaires to skew the value)
- Want to value a house, and, assuming a linear regression model use past sales data to estimate partial contributions of factors such as property tax and ratings of nearby schools
- For house valuing above instead have a random forest model where for every regression tree estimate both its node structure and parameters of each node

The "what if the model is true" rarely is, but usually leads to only a small approximation error (discussed fully in the "General Machine Learning" chapter) when not, and "true" has no deeper meaning. Any model, no matter how nonsensical, can be estimated from any data, unless the data is impossible according to the model. So at worst estimate a bad model well. A model is abstract and not related to reality—even for the median income example the iid assumption needn't be true because survey respondents may boast or hide

the truth. Still, usually get useful models as measured by their prediction quality (see the “General Machine Learning” chapter). The goal is to determine at least qualitative behavior—something true in the model is likely to be true in reality.

So it's OK to rely on approximate models. In many other aspects of life they are routinely used—e.g., laws are an approximation to some ideal justice system, roads regulations (particularly speed limits) can be improved, but they are good enough. Also it's OK to use approximate models approximately, e.g., by simplifying calculations or making minor assumptions. In the examples above:

- For the median income assume a distribution with some median, which is always true for a continuous distribution (and most discrete ones; but in general the latter can have a discontinuity at the median).
- For the random forest model of house prices, assume the mechanism closely matches a random forest of regression trees with some structure and node values.
- For the linear regression know that it's usually more approximate than the random forest, but still useful due to simplicity. Though the derivation of the calculations is often stated without any probabilistic assumptions, it's a special case of applying maximum likelihood (discussed later in the chapter) to normally-distributed errors.

In some cases can deduce a model from domain properties. E.g., in physics a new relevant parameter is likely to be revolutionary, so it's probably irrelevant. Also, e.g., if deal with memoryless data, consider the exponential distribution model.

Parameters can be **structural** as in the random forest model—most generally they are any information needed to specify a particular data generator. For regression trees they are the split points—defined conditionally for a given node on those of its ancestors. Some important classes of parameters:

- **Unidentifiable**—not estimable, e.g., when square normal samples, lose at least the sign of the mean. Typically can estimate only parameters that are aggregates and represent typical behavior, such as a mean, quantiles, or a split value of a tree node. Despite the availability of many algorithms, statistical analysis isn't about what you want, but about what can you get with the information you have.
- **Nuisance**—not in the wanted subset of parameters, but on which estimating the subset depends. E.g., when estimate the mean and its confidence interval, usually also need to estimate the variance.

Don't need statistics when know the wanted parameters. Otherwise it allows, as much as possible from the availability of the data, to:

- Estimate values of parameters—targets of estimation are also called **estimands**
- Rule out unlikely values with reasonable probability, i.e., out of many inferences mistakes will be rare—this is done using confidence intervals and hypotheses tests (both discussed later in the chapter)

Sometimes also want to know functions of estimands such as a difference. Here the best choice of the estimands can depend on the estimators—e.g., if both the difference of means and of medians are equally useful, then pick the one with the best estimator for the situation. To avoid silly uses it's best to first think about how would use θ if knew it exactly, and then use any estimate in the same way but potentially with adjustments.

For most problems can't do any estimation without making some assumptions, and some are more reasonable than others. Many algorithms make several unavoidable heuristic choices, and can be themselves seen as heuristics. In particular, instead of starting with the assumption that all problems are solvable, which isn't the case, it's best to assume that none are, and discover those that are, together with the assumptions that make it so. Same for algorithms in numerical methods, optimization, and machine learning, in all of which need some special structure for useful solutions.

The most interesting case for statistical inference is when a specific distribution is assumed on the data, given a complete or partial specification of the parameters. A typical assumption is that given θ , the data $\sim \text{normal}(\theta, \text{known or unknown variance})$. This leads to the **log-likelihood** = $LL(\{x_i\} | \{\theta\}) = \sum_{0 \leq i < n} \log(\text{PDF}(x_i | \{\theta\}))$. This is a joint, unnormalized PDF of the data, transformed with the logarithm for mathematical and numerical convenience, through which a lot of inference is done.

When no specific distribution is assumed, can **resample** by drawing new data from it uniformly, leading to the **empirical distribution function (EDF)** $F(x) = \sum_i \frac{x > S_i}{n}$. A lot of inference can be done from this also, particularly without significant further assumptions.

It makes no sense to study without further restrictions (such as with sample size n) statistics like length of a confidence interval because it $\rightarrow 0$ for reasonable constructions—the parameter doesn't exist.

21.3 Estimators

An estimator f is a function of the data whose value is intended to be close to θ . As such it itself is random, with dependence on the data, and has a **sampling distribution** S . Properties of S determine the quality of an estimator. In particular it's never the case (except for discrete parameters or artificial data) that $\hat{\theta} = f(\text{data})$ is exactly θ . At best, when S is narrow enough, as measured, e.g., by its standard deviation, $\hat{\theta}$ can match θ to a few decimal places (and definitely not to the full precision of the calculation).

Several general principles help derive good estimators. Each creates a concentration with a certain "gravitational pull" toward θ . As the running example for some of these, consider estimating the Bernoulli parameter p . $LL(\{x_i\}|p) = \text{count}(1)\log(p) + (n - \text{count}(1))\log(1-p)$.

- **Plug-in principle**—if $\theta = g(\text{some distribution } T)$ for some function g , then $f = g(\text{EDF(data sampled from } T))$ is a **plug-in estimator** (Wasserman 2004). E.g., the sample mean and median are plugin. For the Bernoulli example, $\bar{x} = \frac{\text{count}(1)}{n}$.
- The **method of moments**—some parameters of the data are functions of parameters that can be estimated using the plug-in principle, in particularly all moments (e.g., $E[X]$, with random variable X corresponding to the generation mechanism). In case of several simple parameters, such as the normal with unknown mean and variance, this leads to a system of equations. But the method isn't general.
- **Maximum likelihood**—find $\arg\max_{\theta} LL(\{x_i\}|\{\theta\})$. E.g., if $\{x_i|\{\theta\}\} \sim \text{normal}$, $\hat{\theta} = \bar{x}$ is the estimate (see Westfall & Henning 2013 for some examples of derivations). If replace the normal by Laplace, get the median instead. S of maximum likelihood estimators $\sim \text{normal}$ asymptotically under some **regularity conditions** (Wasserman 2004; "regularity" means advanced math is needed to identify the exceptions). Commonly solve for the maximum by setting the log-likelihood derivative to 0. For the Bernoulli example, $\frac{d}{dp} LL(\{x_i\}|p) = \frac{\text{count}(1)}{p} - \frac{(n - \text{count}(1))}{1-p} = \frac{\text{count}(1) - np}{p(1-p)}$, setting which to 0 leads to \bar{x} .
- **Distance-based estimation**. Pick a distance function d between the assumed distribution of the generator output, given the parameters, and the EDF(the data). Then find the d -minimizing parameters. E.g., d can be Kolmogorov-Smirnov in 1D and for $D > 1$ star discrepancy based on the smallest hypercube containing the data; for discrete distributions can use chi-squared distances (all discussed later in the chapter). But despite the intuitive appeal, this is computationally problematic and hasn't been used much. If curious, see Basu et al. (2011) and references therein.
- **Bayesian estimation**—assume a belief state where θ and not the observed data is random. Given a reasonably justified (often called **objective**) **prior distribution** about θ before seeing the data and the data, derive a **posterior distribution** with the belief state that reflects all known information (see the "Machine Learning—Classification" chapter for an example of using Bayes theorem). Then, as in maximum likelihood, chose the maximum-PDF point as $\hat{\theta}$. E.g., for the Bernoulli example can use $\frac{1 + \text{count}(1)}{2 + n}$. This is based on a beta-binomial setup with a beta(1, 1) prior (Wikipedia 2019a).

The advantages include computational stability (here never divide by 0) and absence of excessive optimism when one of the counts is very small.

- **Minimum description length (MDL) principle**—seek the shortest representation of the data and the generator, given the parameters. A number x can be represented to any fixed precision using $O(\log(x))$ bits using, e.g., gamma code (see the "Compression" chapter). Any constant factors drop out for selecting the best code. Compared to maximum likelihood, have an extra term $\sum \log(\theta_i)$. Compared to Bayesian estimation, look for a code and not a prior, which can be easier. See Rissanen (2011) for details. This, as the equivalent Bayesian estimation, has advantages for $D > 1$.
- Any of the above, but on functions of the data given $\{\theta\}$ such as $\text{sign}(x_i - \theta)$ or some range buckets. This allows to relax distribution assumptions on S . But for transforming data to ranks, as done by some procedures, this doesn't work—ranks are a global property based on all the data, and as such they can't be assigned likelihood based on a single observation and the model.

In principle, there are sets of parameters that these criteria converge to. But a solution needn't exist, be unique, match the true parameter in the limit, or be tractable computationally. Still, with specific estimators, these problems usually don't occur. But in general must always validate an estimator regardless of which principle was used to derive it.

Given θ and $\hat{\theta}$, can, at least theoretically, assign a loss $L(\theta, \hat{\theta})$ to quantify the cost of not finding θ

exactly. Usually use the **mean-squared error (MSE)** loss, where $L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2$. An asymmetric L also makes sense when underestimation is worse than overestimation, or vice versa. Only require that:

- $L(\theta, \theta) = 0$
- $L(\theta, \hat{\theta} \neq \theta) > 0$
- L is convex or quasiconvex

The first two are self-explanatory, and the last one is based on that larger mistakes shouldn't cost less than smaller ones. Sometimes insist on convexity due to its mathematical properties—e.g., sums of convex functions are convex. Scaling L by a constant doesn't impact any conclusions, so choose the most convenient scale. Commonly assume that minor changes to L make little difference to any conclusions based on it. It's practically impossible to predetermine all consequences of using a particular $\hat{\theta}$, so any reasonable strategy should work.

Need domain knowledge to determine the exact appropriate L , but usually using nonstandard L that are relevant to the user prevents applying generic useful algorithms. A particular inference may be inappropriate if the end user's L is very different from the standard ones. There may not be analytical techniques to handle unusual L , but simulation or numerical optimization will usually do. I.e., use it to derive an estimator that minimizes some aggregate risk measure (discussed later), as the sample average does for MSE-based ones under some conditions. This is complicated (e.g., need to find some estimator representation, etc.) and not discussed further.

Values of $\hat{\theta}$ are controlled by S , and can define **risk** $R(\theta) = E_{\hat{\theta}}[L(\theta, \hat{\theta})]$. The expectation is easy to work with but otherwise not special—could have used the median or the 90th percentile to get another reasonable definition. With L_1 loss (the absolute difference) R is actually the expected error—but even if estimated from data, this isn't as useful as a confidence interval. Still, when using a high percentile for risk instead of the expectation, such a quantity may be easier to interpret.

Some desirable properties of estimators aren't based on risk. An estimator is:

- **Consistent** if $\hat{\theta} \rightarrow \theta$ as $n \rightarrow \infty$, i.e., getting increasingly more samples makes $\hat{\theta}$ accurate in the limit. Though statisticians drool for consistency like programmers for optimization, finite- n behavior can be very different. A consistent estimator isn't necessarily a good one in practice, but an inconsistent one needs a good excuse to be considered. Beware that "asymptotic" in statistics means as $n \rightarrow \infty$, which isn't the same as the "O" notation that ignores constant factors. Though asymptotic approximations are usually reasonably accurate, like Taylor series in calculus, all procedures with only asymptotic known good properties must be checked by simulation.
- **(Mean-) unbiased** if $E[\hat{\theta}] = \theta$; the expectation is over S . Median-unbiasedness is defined analogously and sometimes more important, and can't have both if S is asymmetric. Unbiasedness is regardless of θ , but a biased estimator can have a different bias for every θ . $\hat{\theta}$ is over- or under-estimated according to S , and the bias is usually only a small nudge. Bayesian- and MDL-based estimators are almost always slightly biased, but rarely enough to pay attention to. For the Bernoulli example x is unbiased, and the Bayesian estimator has bias 0 for $p = 0$, and a positive one otherwise (Wikipedia 2019a). Something like a ratio of estimates can't generally be unbiased. Median bias is easier to fix because it's unaffected, e.g., by monotone functions of the estimate.
- **Invariant**—transforming the data by a reasonable set of functions, such as change of measurement units, should transform $\hat{\theta}$ in the same way (many sources use the word "equivariant", reserving "invariant" for no change, but the meaning is usually clear). While maximum-likelihood estimators have this property (Kiefer 1987), Bayesian- and MDL-based ones usually don't. But it's reasonable to insist on approximate invariance, i.e., that a transformation leads to only a minor difference. For the Bernoulli example both x and the Bayesian estimator are invariant to scaling and shifting.

Estimators derived from general principles are usually consistent but on case-by-case basis corrected to get some of the other properties, which however are rarely violated substantially. Corrections for computational stability are also sometimes made, and usually have a Bayesian interpretation.

For unbiased estimators, using variance as the performance measure matches the MSE loss, because $MSE = bias^2 + variance$ (Wasserman 2004; see the "General Machine Learning" chapter for a similar relationship for other L).

For convergence deal with limits of functions, not numbers, and several slightly different definitions make sense. All are with respect to the asymptotic sampling distribution $S(n)$ when $n \rightarrow \infty$ (Wasserman 2004):

1. **L_2 (in quadratic mean):** $E[(\hat{\theta} - \theta)^2] \rightarrow 0$
2. **With probability 1 (strong):** $\Pr(\hat{\theta} \rightarrow \theta) = 1$
3. **In probability (weak):** $\Pr(|\hat{\theta} - \theta| < \epsilon) \rightarrow 1$

4. **In distribution:** the distribution of $g(\hat{\theta} - \theta) \rightarrow$ some limiting distribution, for some normalizing function g (such as $1/\sqrt{n}$)

Consistency is defined using(3). (1) and (2) imply (3), and (3) implies (4), and all other implications don't hold in general. In particular, consistency doesn't imply asymptotic unbiasedness. To make this happen need a stronger version of consistency based on L_2 convergence for the wanted conclusion.

At a given θ , let $b(\theta)$ be the bias, and $FI(x_0, \theta)$ **Fisher information** for a single observation (x_0 without loss of generality; FI is 2nd derivative of LL ; see Wasserman 2004 for some examples). Under regularity conditions have **Cramer-Rao inequality** (Lehmann & Casella 1998; Korostelev & Korosteleva 2011): for any f that uses n data points, $\text{var}(S) \geq \frac{1+b'(\theta)^2}{nFI(x_0)}$. E.g., consider the constant estimator $f(\text{data}) = 42$. At $\theta = 42$ it's perfect, but $b(\theta) = 42 - \theta$, and $b'(\theta) = -1$. So, as expected, variance is bounded by 0. As the CLT, the bound is for an absolute quantity and isn't relative to $\hat{\theta}$ or θ .

Some special cases escape the regularity conditions, though this is usually ignored. One is when θ is on the boundary—e.g., when have n samples from $\text{uniform}(0, \theta)$, and want to estimate θ . Here $\hat{\theta} = \frac{n}{n-1} \max(x_i)$ is an unbiased estimator (Korostelev & Korosteleva 2011), where the variance is $O(n^{-2})$.

The inequality leads to a number of conclusions:

- Because $b(\theta)^2 \geq 0$, this is a limit on MSE
- For an unbiased f , $b'(\theta) = 0$, giving a minor simplification to the formula
- As long as $b'(\theta)$ is bounded away from -1 (such as when b is Lipschitz with constant < 1), the standard deviation of S is $O(n^{-1/2})$. Because the bias is usually $O(n^{-1})$ (Wasserman 2004), this is essentially the working precision of $\hat{\theta}$. MDL methods encode estimates to this precision (Rissanen 2011).
- All estimators have efficiency limits (at least for the regular cases) on variance of S .

The inequality is rarely an equality—only **exponential families** are an exception (Kiefer 1987). E.g., for the mean of the normal distribution with known variance, the sample mean satisfies the inequality exactly, but not with unknown variance. The significance of the bound is mainly in the existence of a performance limit, and an asymptotic $O(n^{-1/2})$ standard deviation of S .

The bound has a lot of generality—even tree split values are estimated in some way. E.g., can use the bootstrap to calculate the sample variance of a specific node's value, and don't expect a decrease better than $O(n^{-1})$. (As an implementation detail, resample the data that got to the node. Do this for every node for a tree that is estimated from the whole data.)

Looking at bias and variance is supported by **Chebyshev's inequality** (Wasserman 2004): For a distribution with a finite μ and a finite $\sigma > 0$, $\Pr(|x - \mu| > k\sigma) < \frac{1}{k^2}$. To apply to estimation replace μ by θ . In particular, if the bias \leq a fixed fraction of σ , and $\sigma \rightarrow 0$, the estimates zoom in on θ . So convergence in MSE, also called convergence in L_2 , means convergence in risks for many other L .

But the reverse bound isn't generally true—for the Cauchy distribution, which is S for the sample mean of Cauchy variates, having an arbitrary small variance needn't bound other L . Need extra information, such as one of the following:

- $L \geq$ a function of the variance. E.g., if use MSE^2 , it decreases at least as $O(n^{-2})$ but is a silly choice. This suggests that L such as linex (discussed later in the chapter) are useful for estimate computation but not necessarily for ranking. MSE is useful because it's related to the standard deviation, which is related to the length of the universally applicable Wald construction of confidence intervals (discussed later in the chapter). Still, e.g., for something like the sample median confidence interval length depends on other estimates, so picking L based on the interval construction doesn't work in general.
- Assume a specific S , such as the normal, where all quantile locations are determined or bounded by a function of the standard deviation of S . Any L is quasiconvex and > 0 , so even in tails of size a have at least a partial contribution of size $a(L(\text{left location}(a, \text{variance}) + L(\text{right location}(1 - a, \text{variance})))$ to R . And can take the maximum of this over all a . E.g., if these location-specific L are linear in the standard deviation, they are also upper-bounded by $O(n^{-1/2})$ asymptotically. Nonlinear dependence leads to different asymptotics, which are hard to calculate, but even with an unknown but finite rate L can't decrease arbitrarily fast in n .

The asymptotic behavior can be somewhat different from finite- n behavior:

- Consistency means that $R \rightarrow 0$ (need some regularity conditions such as Lipschitz L)
- Cramer-Rao bound is met by more estimators because the constants converge

An interesting property of estimators is being based on a **sufficient statistic**—i.e., on a function of the sample that contains all information in it with regard to $L\theta L$ (technically must be able to factor $L\theta L$ in a certain way—see Westfall & Henning 2013 for some examples). E.g., for estimating the mean and the variance of the normal distribution, the sum and the sum of squares are as good as the sample. But in general need a parametric model to exploit sufficiency. At best get to use:

- All sample order statistics (i.e., the sorted data). This is the case for the sample median. For $D > 1$ can't sort, so only have the data.
- Unique value counts for discrete distributions.

The best general conclusion is to use all the data in some way, which is a more basic requirement than sufficiency with respect to some distribution. E.g., the trimmed mean, which is a known good estimator, isn't sufficient for any common distribution, though it uses all data smartly.

Estimators usually lead to derived estimators such as confidence intervals and hypotheses tests, which aim to quantify some aspects of S better than its variance does. But on their own they have no meaning due to being unnecessary if θ is known.

When picking an estimator, want the final choice to:

- Satisfy any external requirements—particularly if the result is intermediate and is passed on to another calculation. Some estimators, like the sample mean, have special theory (like the CLT or finite-sample inequalities, discussed later in the chapter), getting which may matter more than other factors.
- Be statistically efficient, as discussed later in the chapter.
- Have the general qualities of a useful algorithm.

An interesting question is what the estimand is when the conditions of an estimator aren't satisfied. Such **descriptive statistics** are often useful on their own, without reference to underlying parameters. E.g., the trimmed mean is neither the mean nor the median for an asymmetric distribution. Still, for comparisons any location measure (discussed later in the chapter) is as good as any other, and the trimmed mean is usually the most efficiently estimable statistic, so consider using it first.

Given samples from a known distribution, simulation will work for checking properties of any estimator (including confidence intervals and tests).

Any conclusions based on the data are as good as the data. Just as testing software can't guarantee absence of bugs, a statistical procedure based on iid data can't guarantee a bulletproof estimate. But even in law statistical results are admissible evidence, though not conclusive on their own. Statistical procedures work under specific conditions, which may not be what you want, even if others interpret the results the way you want. Often don't have formally stated conditions for correctness, leading to certain mental discomfort and potentially a long research. If several sources don't discuss such conditions, they probably don't exist, at least in general form. Current proof tools of statistical theory are limited to mostly smooth problems and asymptotic results, and it's a research challenge to produce more useful theorems. Practical application of certain procedures such as the bootstrap (discussed later in the chapter) must settle for folk knowledge of working well in practice.

21.4 Finding Most Efficient Estimators

The main questions are regarding the:

1. Existence and uniqueness of a lowest-risk estimator
2. The ability to recognize a best estimator
3. The ability to find a best estimator

Any answer must at least be restricted to a particular S or a data distribution, and in general each distinct distribution among uncountably many can lead to a different answer. But pick a presumably representative S , which usually happens to be convenient mathematically or in some other way, and heuristically (in many cases unreasonably) assume that the conclusions hold in general, at least for the wanted application. Commonly $S =$ the normal, potentially transformed by some function.

Even with these restrictions, the general case is problematic, so focus on unbiased estimators first. Here for (2) have the **Lehmann-Scheffe theorem** (Kiefer 1987): an estimator is lowest-risk unbiased $\forall \theta$ when using convex L if it's:

- A function of a sufficient statistic
- **Complete**—a mathematical requirement that essentially makes an unbiased estimator unique (Samaniego 2010)

But best unbiased needn't be best overall. E.g., for the Bernoulli example for $p = 0.5$, the Bayesian estimator has 0 bias and lower variance (see Wikipedia 2019a for the formula) than \bar{x} , which is complete (Kiefer

1987), so Lehmann-Scheffe applies. And for nearby θ the MSE of the Bayesian estimator is also lower until far enough to the tails. This Bayesian estimator is by far not unique—e.g., can even have fractional increments is desired, and all of these are invariant to location shifts and scaling. A complete estimator based on a sufficient statistic may not exist, in which case neither does a single uniformly best unbiased estimator. Existence is essentially limited to the classical parametric normal-based estimators.

L can be a very asymmetric function such as **linex**, where $L(d) = e^{ad} - ad - 1$, for $d = \hat{\theta} - \theta$, and a is a scale parameter (not 0 and can be negated to flip the graph), usually 1 (Samaniego 2010; see also Franses et al. 2017 for some other common choices). But the best unbiased estimator doesn't change because it doesn't depend on L . So restricting to unbiased needs a good justification. Still, determining an accurate L for the domain is practically impossible, so work with an approximation, which allows to optimize approximately.

In the general case, the main issue is that, as for the Bayesian estimator in the Bernoulli example, can have different risk for different values of θ . So aggregate risk over all θ . Commonly use the:

- **Maximum risk**—the worst-case protection.
- **Bayes risk**—given some belief-state distribution over θ , this = $E_\theta[R(\theta)]$. E.g., a delta PDF on the maximum-risk value leads to the maximum risk, so this is a more general case. Also can use **objective distributions**—e.g., for bounded ranges (i.e., due to domain constraints) the uniform, and for unbounded ranges a fat-tailed one like Cauchy (be careful to make sure the expectation is defined).

With maximum risk a general theorem of game theory (Nash equilibrium) shows existence of a **minimax estimator**:

- I.e., with the smallest maximum risk.
- It may be randomized—i.e., produce different $\hat{\theta}$ from the same data (like for rock-paper-scissors, randomization leads to a safe strategy)
- The game is that have a distribution with unknown parameters and n , and pick an estimator. Then nature picks θ (then get the data, but this is irrelevant here).

Using randomization looks suspicious because different analyses of the same data set produce different results. But the data should be a much bigger source of randomness. Think of this as associating random information with every particular data set. For scientific reproducibility generating another data set will also generate another random information. But still it's unclear how to validate someone's analysis.

Minimax estimators often don't exist or aren't obvious even in simple cases. E.g., for the Bernoulli example \bar{x} isn't minimax, though a slightly more complicated Bayesian estimator $\frac{n\bar{x} + \sqrt{n}/2}{n + \sqrt{n}}$ is (Shao 2003). Min-

imax estimators are hard to find. One useful theorem is that Bayes estimators with constant risk are minimax (Kiefer 1987; this is how the above estimator was found). An interesting idea is to mimic the logic of approximation algorithms and define c -minimax estimators, which are at most a constant factor c away from the minimax performance, but this hasn't been explored in the literature.

Bayes risk cheats in a sense that the optimal estimator will be a Bayes estimator that uses the risk distribution as the prior. But a prior alone doesn't lead to a unique estimator because need to weigh the data in some way. E.g., with the beta-binomial any beta(*count, count*) is a uniform prior.

Using the maximum risk has a certain worst-case satisfaction, and avoids cheating such as putting a delta PDF on the smallest-risk value. But Bayes risk is also useful:

- Estimators with the same maximum risk can have different risks at all other points, and minimax estimators may be inadmissible (discussed later). Still, estimators with large risk differences for different θ are suspicious, so handling such cases isn't practically relevant. Perhaps the best way to think of these is the same as for interpolation with Chebyshev or minimax polynomials in numerical analysis (see the “Numerical Algorithms—Working with Functions” chapter)—the former win on average but lose a little on maximum.
- Bayes risk is easy to estimate by simulation, using a multi-delta PDF on randomly sampled or selected points. In easy cases, such as for the normal mean or median, might only work with a single representative θ value—here 0 (and disallow silly estimators such as $f(\{x_i\}) = 0$).
- The choice of L is also subjective, and the overall effect of it and the aggregation method matters.

E.g., for linex to minimize aggregate risk would pick a smaller value over a larger one, as much as uncertainty allows. Here for the normal model with known σ , \bar{x} is inadmissible in favor of $\bar{x} - \frac{a\sigma^2}{2n}$ (Samaniego

2010). Numerical work can be helpful with unusual L . E.g., can evaluate an estimator for specific L and aggregation function for a specific data distribution. This is easiest when a single θ is representative, e.g., assuming a normal distribution around it, but works with any R . This even allows to search for the best esti-

mator numerically in some function representation, e.g., weights for a linear combination of order statistics with fixed n . Might even be able to generalize the results analytically to find a good estimator.

Finding and even recognizing the best estimator is difficult and often impossible, even in the unbiased case. So consider the easier comparing estimators on risk. Instead of aggregating, can define a Pareto frontier (see the “Optimization” chapter) of estimators in terms of risk—those on it are **admissible**. Using inadmissible estimators needs a good explanation, but admissibility doesn’t automatically imply quality—e.g., the constant estimator is admissible.

Using the MSE the sample variance is inadmissible because instead using the maximum likelihood $1/n$ divisor in its formula loses variance more than gains bias² (Samaniego 2010). But using this divisor is essentially a known mistake due to the external constraint that estimate variance not to discover the true one, but to get a confidence interval based on the t -distribution (discussed later in the chapter), which assumes the unbiased estimate. Interestingly, the resulting sample standard deviation is biased (Westfall & Henning 2013), but this doesn’t matter. So the mistake attribution has nothing to do with some preference for unbiasedness. E.g., in EM clustering (see the “Machine Learning—Clustering” chapter) use the biased maximum likelihood estimator to make the theory work. Also in the implementation of the BC bootstrap confidence interval (discussed later in the chapter) use the Bayesian estimator of the Bernoulli parameter instead of the mean to avoid range bounds better than by rounding out-of-bound values. Generally, an unbiased estimator can violate range constraints, and fixing this introduces a small bias. In some cases good unbiased estimators don’t exist (see Lehmann & Casella 1998 for an example).

Properties external to risk such as unbiasedness and invariance are still desirable even if not mandated by external constraints, and should consider giving up a bit of risk to gain them:

- Getting accurate confidence intervals for unbiased estimators is easier—e.g., for the bootstrap (discussed later in the chapter) the fastest normal-based methods assume the bias is small and don’t bother correcting it. This is important because an estimator should always come with a confidence interval.
- Certain types of invariance help if later transform the estimates in some way.
- Both provide some regularization (for more on this concept see the “General Machine Learning” chapter) with respect to the choice of L to ensure low aggregate risk. Closely satisfying these properties also has the same effect.

Quantitative comparisons are more meaningful than qualitative ones, so typically compare using **relative aggregate risk ratio**—this is also unit-free. In the asymptotic case when look at the MSE and biases are small enough to be ignored, the S variance ratio of two estimators is **asymptotic relative efficiency (ARE)** (Wikipedia 2016e). Equivalently, can use the sample-size-needed-to-get-equal-variance ratio—this also works well for hypothesis tests, confidence intervals, and other L . These are theoretically known for many estimator comparisons. E.g., when several functions of data estimate the same parameter—e.g., the sample mean, the sample median, and the sample trimmed mean (discussed later in the chapter) estimate the mean of a symmetric distribution (where it exists). Though can have finite- n ratios, particularly computed by simulation, asymptotic ones are easier to derive analytically and still reasonably predictive.

In practice want an estimator that is rarely too bad and usually good for typical data. Estimators that do very well in rare cases and poorly otherwise can be useful only in applications with domain knowledge in favor of the former. Same for confidence intervals and tests.

21.5 Some Peculiarities of Asymptotics

In principle asymptotic approximation is straightforward—e.g., the CLT essentially shows everything about it. But some surprises include:

- The possibility and effectiveness of higher-order approximations—e.g., the CLT can be made more accurate using third and forth moments (see Dasgupta 2008 for details).
- Some mathematical challenges—e.g., consistency doesn’t imply stronger convergence. These usually aren’t statistically important and won’t be considered further.

For the CLT the approximation error is $O(1/\sqrt{n})$ by the **Berry-Essen theorem** (depending on the third moment, Dasgupta 2008)—the same rate as the estimation error. Approximations with such error are **first-order accurate**. Including more moments brings down the approximation error by a factor of \sqrt{n} . E.g., a **second-order accurate** approximation has approximation error $O(1/n)$ but still $O(1/\sqrt{n})$ estimation error.

Because the estimation error doesn’t go down, and higher-order moments or other information must exist and be estimated, so use them rarely. Second-order may be better overall than first-order, but this depends on the particular case and must be verified by simulation. The approximation error is only for the asymptotic object, and may not be observed with finite n , just like Taylor series needs $x \rightarrow 0$ to cancel out

the effects of the particular unknown function being approximated and force it to a known polynomial. As with Taylor series, using arbitrary many terms for finite- n approximation can diverge, though the practical at-most-second-order accuracy is never at risk.

21.6 Evaluating the Normal CDF

The normal is a common sampling distribution, particularly asymptotically, as seen from the CLT. But sometimes using $z = 2$ for confidence intervals isn't good enough. So need to calculate z that corresponds to a certain confidence level, and vice versa, particularly when adjust for multiple testing (discussed later in the chapter).

The calculation reduces to evaluating the error function, available from `<cmath>` since C++ 11. See Wikipedia (2014c) for some simple, reasonable approximations if curious, but the standard implementation ought to be better. Need to scale the argument though, and the one- and the two-sided cases are a bit different:

```
double approxNormalCDF(double x)
{
    double uHalf = erf(abs(x)/sqrt(2))/2;
    return 0.5 + (x >= 0 ? uHalf : -uHalf);
}

double approxNormal2SidedConf(double x) {return 2 * approxNormalCDF(x) - 1;}
```

To find z corresponding to a particular confidence, use the inversion method. This is guaranteed to work because CDFs are monotonic. Efficiency is also good because the normal has thin tails, so rarely go beyond few standard deviations.

```
double find2SidedConfZ(double conf)
{
    assert(conf > 0 && conf < 1);
    return invertCDF([](double x){return approxNormalCDF(x);}, 0.5 + conf/2);
}
```

21.7 Evaluating the T-Distribution CDF

The t -distribution with $v=n-1$ degrees of freedom models the mean of normal samples of size n , adjusting for estimating σ^2 (Wikipedia 2018a). Though for $n > 30$ the normal approximation is considered accurate, may want smaller n or higher accuracy.

For $v \geq 3$, the **Hill approximation** converts the t value to a z value and has the worst-case $O(10^{-5})$ error (with "0" hiding a small constant). $z=w+\frac{w^3+3w}{b}-\frac{(4w^7+33w^5+240w^3+855w)}{10b(b+0.8w^4+100)}$, where

$w=\sqrt{a \log\left(1+\frac{t^2}{n}\right)}$, $a=n-0.5$, and $b=48a^2$ (Brophy 1987). For $v \leq 2$, $TCDF(t,1)=\frac{1}{2}+\frac{1}{\pi} \operatorname{atan}(t)$, and $TCDF(t,2)=\frac{1}{2}+\frac{t}{2\sqrt{2+x^2}}$ (Yerukala et al. 2013; the slightly simpler **Gleason approximation** they recommend has much larger errors for $v=3$).

```
double approxTCDF(double t, int v)
{//Hill's method, always < 10^-4?
    assert(v > 0);
    if(v == 1) return 0.5 + atan(t)/PI(); //Cauchy
    if(v == 2) return 0.5 + t/2/sqrt(2 + t * t); //also exact
    double a = v - 0.5, b = 48 * a * a, w = sqrt(a * log(1 + t * t/v)), w27[6];
    w27[0] = w * w;
    for(int i = 1; i < 6; ++i) w27[i] = w * w27[i - 1];
    double z = w + (w27[0] + 3) * w/b - (4 * w27[5] + 33 * w27[3] +
        240 * w27[1] + 855 * w)/(10 * b * (b + 0.8 * w27[2] + 100));
    return approxNormalCDF(t > 0 ? z : -z);
}

double approxT2SidedConf(double x, int v) {return 2 * approxTCDF(x, v) - 1;}
```

To find a t value corresponding to a particular confidence, use the inversion method, as for the normal:

```
double find2SidedConfT(double conf, int v)
{
    assert(conf > 0 && conf < 1 && v > 0);
```

```

    return invertCDF([v] double x) {return approxTCDF(x, v);}, 0.5 + conf/2);
}

```

21.8 More on Confidence Intervals

Another good measure of estimator accuracy is the size of an associated **confidence interval**. More generally have a **confidence set**, which is a disjoint interval, or a **confidence region** for multidimensional parameters, but these are less useful. Defined by sample-dependent bounds l and u , a confidence interval is such that $\Pr(\theta \in [l, u] | \text{sample}) \geq 1 - \alpha$. Almost always use $\alpha = 0.05$, leading to a 95% confidence interval. This error tolerance is imposed on statistical methods externally as the largest error commonly acceptable scientifically. Intuitively, a confidence interval is where the data “thinks” θ is, and different data have a different “opinion”. Often only look for $u - l$ or $\max(u - \hat{\theta}, \hat{\theta} - l)$ as a single measure of accuracy of $\hat{\theta}$. Any uncertainty is due to the data, though sometimes also due to randomization in the estimator (e.g., it may be based on simulation). In frequentist statistics the concern is about long-term properties of procedures and not their individual results. Though there are bad constructions of confidence intervals, frequentist validation is never in question.

E.g., if know the variance of S , f is unbiased, and assume it computes the mean, Chebyshev's inequality gives $(l, u) = \hat{\theta} \pm \sigma \sqrt{\frac{2}{\alpha}}$. Even if estimate σ from data, and f has a small bias or the result is very different from

the mean, this is usually a very conservative interval because Chebyshev's inequality is usually very loose. So, if can estimate the variance of S , often make a stronger assumption that S is normal, and use $(l, u) = \hat{\theta} \pm 2s$. This is **Wald interval**. In practice many $\hat{\theta} \sim \text{normal}$ or near-normal. Can estimate the variance using the bootstrap if can't calculate it directly.

Because $(-\infty, \infty)$ is a valid but useless interval, want confidence intervals to be “short”, as usually measured by one of:

- Expected length—for 1-sided intervals use distance to the wanted boundary from θ . For more general regions use appropriately defined volume. Usually 1-sided intervals aren't the right choice and are often best computed by reducing to two-sided intervals, so little further consideration is given to them. Median length might be more appropriate in some cases, but hasn't been studied.
- **Accuracy**—probability of coverage of various values of $\hat{\theta}$ given θ . This allows, among other things, to define admissibility by probabilistic set containment (deterministic containment is impossible due to the random nature of confidence intervals)—a confidence interval is **uniformly more accurate** than another if $\forall \hat{\theta} \neq \theta$ it's more accurate. In few special cases get a **uniformly most accurate (UMA)** interval (i.e., one-sided intervals for single-parameter exponential families, like the binomial; Shao 2003).

Some other desirable requirements can be placed on confidence sets but aren't essential:

- Want a connected, and for $D > 1$, even convex region. But for something like a flight of a bird around an obstacle have a bimodal distribution, where a two-piece confidence set makes sense.
- For 2-sided intervals want a symmetric interval with $\leq \alpha/2$ error on each side. For $D > 1$ want this \forall variable.

Fortunately the general definition usually induces these automatically:

- Coverage tends to be balanced due to diminishing returns $\partial \alpha / \partial \text{length}$ when α is small. For $D > 1$, can define balance maybe along a particular direction.
- From the above any holes are cheaper to close than extending to the sides further.
- Only for something like the bird flight (see the “Monte Carlo Algorithms” chapter), this logic doesn't work because central regions are unlikely.

An interval is very likely to enclose $\hat{\theta}$, which is usually a maximum-likelihood point, and can act as its error estimate. But in general confidence intervals and estimates are mildly and not directly related. E.g., the estimator = the centroid or the most frequently included point of a confidence set is consistent. But don't have a finite-sample relation.

Despite preferring equal α from statistical efficiency point of view, might want to allow one side to have larger error due to asymmetric costs in the application. But most APIs (including those in this chapter) don't support this. Can get the same effect by computing two intervals corresponding to the wanted α and merging them. Still, this solution is potentially computationally expensive, and with randomized constructions run into minor peculiarities.

Using expected length as the shortness measure helps avoid holes, but with accuracy the shape makes no difference. The former is also easier to study with simulation because the latter is mostly a relative com-

parison measure. Neither follows the general strategy of comparing estimators based on some loss function and risk, but some intuitive loss functions are proposed later in the chapter. Given a control, expected length can be considered as risk and compared as for estimators—using minimax, admissibility, and other θ -dependent quantities. Still, this is best used with simulations using Bayes risk with representative points. The literature mostly checks optimality based on accuracy.

Because UMA intervals are available in few cases, like for estimators, can consider restricted classes by imposing (Shao 2003):

- Unbiasedness—for confidence intervals this is based on accuracy, and bias = the most frequently included point – θ . As for estimators, accidental small bias is possible and makes little difference. To calculate it using simulation, generate many confidence intervals, and compute the resulting histogram using a left-to-right pass through the sorted endpoints (this is a good interview question). The middle of the highest-count region is the estimate of the most frequently included point.
- Invariance—with respect to one-to-one transformations with respect to θ , particularly translation and scaling. This is slightly more general than for estimates.

As with estimators, these properties make intuitive sense but needn't result in a gain—e.g., a slightly biased interval can cover both true and false values better. But many classical confidence interval constructions are **UMA unbiased (UMAU)** (Shao 2003), including the:

- T -distribution interval for the sample mean
- Sign-test based interval for the sample median

Expected length and accuracy are related by the **Ghost-Pratt identity** (Casella 1996), which applies to volumes with holes allowed and finite length. E.g., it means that UMAU intervals have the shortest expected length among unbiased intervals.

$\alpha < 0.05$ don't work as well in general:

- Any minor assumption deviations, such as sufficiently fast convergence to normal for CLT-based intervals, make the interval approximate also. Any approximation error is also usually larger in the tails, i.e., with small a . So not taking $1 - \alpha = 99.9\%$ or so doesn't introduce false confidence.
- If the computed interval misses θ , it's unlikely to do so by much. So if use the size of a confidence interval as an error estimate of $\hat{\theta}$, $\alpha = 0.05$ or a nearby value gives the same order-of-magnitude result.

Smaller α usually don't grow the interval considerably. E.g., for the CLT-based intervals $\alpha = 0.0027$ leads to a 50% longer interval than for $\alpha = 0.05$ ($z = 2$ vs $z = 3$). So for large n and minimal-assumption intervals such as the quantile-based ones (discussed later in the chapter), using $\alpha \leq 0.01$ might make sense.

Not for every estimator asymptotically $S \sim \text{normal}$ —e.g., it can be multimodal. But even here a normal-based interval is useful as a rough estimate of width—for a consistent estimator, $u - l \rightarrow 0$. Don't need confidence intervals for intermediate steps where the estimate will be plugged in somewhere else.

$\hat{\theta}$ gets majority values $\in (S_\alpha, S_{1-\alpha})$, but this isn't a confidence interval for θ —it only bounds values of $\hat{\theta}$:

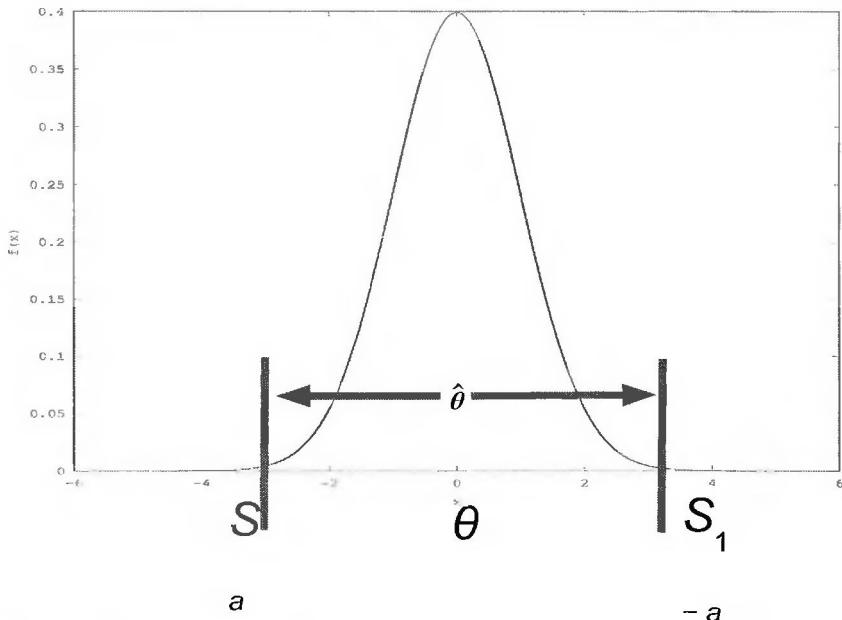


Figure 21.1: S bounds

The best (exact) confidence interval for θ is $(\theta - (S_{1-\alpha} - \hat{\theta}), \theta + (\hat{\theta} - S_\alpha)) = (\hat{\theta} + \theta - S_{1-\alpha}, \hat{\theta} + \theta - S_\alpha)$ (Lunneborg 2000). On the left/right border θ is respectively underestimated/overestimated. I.e., if $\hat{\theta} = S_\alpha$, with

respect to other samples $(\theta - (S_{1-a} - S_a), \theta)$ forms a correct interval; so does $(\theta, \theta + (S_{1-a} - S_a))$ when $S_{1-a} = \hat{\theta}$. Don't know θ , but this construction is useful for comparing various confidence interval constructions theoretically and by simulation.

Confidence intervals only give information about θ within the scope of the chosen model, which may not be a good approximation. E.g., confidence interval for the median always makes sense. ARE and other comparison measures are also only relevant with respect to a model. Only testing the model with different data shows how assumption violations affect confidence intervals. E.g., a confidence interval for a linear regression coefficient only measures the estimation error. It tells nothing about the impact of the variable due to potential impact of other variables (see the "Machine Learning—Regression" chapter), even if the prediction error is very small. The focus of statistics is on properties of estimators of parameters of a model, regardless of its predictive usefulness. If the computed confidence intervals are too wide, need more data, and should have planned the sample size better (discussed later in the chapter).

Given confidence intervals of correct a , and pick the one with the shortest expected length. Here, an equivalent of admissibility is probabilistic inclusion—a contained interval is clearly worse. But if a procedure oversteps a but a little bit, may still want to select it if the produced interval are much shorter than those of the competition. This can be done with a loss function, but a standard one such as $l = \text{function}(\text{length}) + \text{attained } a$ (Kiefer 1987) is neither popular nor intuitive.

An intuitive **normal loss** is given by $l = E[\text{length}] / (\text{z-score corresponding to the attained } a)$. The length can be made relative by dividing over that of the exact interval if the latter known or obtained by simulation. This loss has some issues:

- It pays no attention to the requested a —e.g., for intervals based on the normal theory, a 95% interval is as good as a 99.73% and 1.5 longer one regardless of which was requested
- No penalty for asymmetric coverage—want to have $a/2$ on each side to avoid bias, and to downgrade procedures that try to reduce length by shifting to the more generous side
- The normal distribution is unfair to very small attained a because they only go few more standard deviations

The last issue is solved by using **Chebyshev loss** $l = \frac{E[\text{length}]}{\sqrt{1/a_{\text{attained}}}}$, based on inverting the Chebyshev inequality. For asymmetric coverage the best option seems to be $l = \frac{E[\text{length left}]}{\sqrt{0.5/a_{\text{Left attained}}}} + \frac{E[\text{length right}]}{\sqrt{0.5/a_{\text{Right attained}}}}$; the side length is from θ and, as an additional minor penalty, = 0 if θ isn't covered on that side. Still, this asymmetric Chebyshev loss is only a good measure of efficiency of giving up a for expected length.

A proper loss function must take into account that, despite the fact the selection of a is arbitrary, it's nevertheless to be respected. Chebyshev loss suggests to make the penalty multiplicative. So exponential = $l = E[\text{length}] \exp\left(\max\left(\frac{a_{\text{attained}}}{a_{\text{target}}} - 1, 0\right)\right)$ seems reasonable, though it or any other is hardly the single number to look at. The asymmetric version can be computed as the Chebyshev one.

21.9 Finite-sample Bounds for the Mean

The CLT is asymptotic, and, even without rare events, convergence to a normal can take a while; need $n \geq 600$ in some simple cases (Wilcox 2016). If the data $\in [0, 1]$, have bounds on μ that hold $\forall n$. More generally, for $x_i \in [a, b]$, map x_i into $[0, 1]$, compute the bound, and map that to $[a, b]$, which works by the linearity of expectation.

Hoeffding inequality: Let $\Delta(a) = \sqrt{\frac{\ln(1/\alpha)}{2n}}$. Then with probability $\geq 1 - \alpha$ have any one of the:

- Upper interval: $\mu \leq \bar{x} + \Delta(a)$
- Lower interval: $\mu \geq \bar{x} - \Delta(a)$
- Two-sided interval: $\mu \in \bar{x} \pm \Delta(a/2)$

The two-sided interval follows from the other two and the union bound (a version of Bonferroni inequality, discussed later in the chapter). The lower bound follows from the upper bound using the transformation $\mu = 1 - \bar{\mu}$.

Hoeffding bounds are data-independent and can be used as a stopping criteria. They are used mostly theoretically due to the simple expression, particularly in machine learning (see the "General Machine Learning" chapter). Finite-sample bounds are the best that can be hoped for, but don't have them without

assumptions such as boundedness, because otherwise very rarely a huge value such as 10^9 can occur.

Per numerical studies in Anguita et al. (2013), the **original Hoeffding inequality** gives the best bound. The upper-bound Δ is implicitly given by the solution of $\left(\left(\frac{1-\bar{x}-\Delta}{1-\bar{x}}\right)^{1-\bar{x}}\left(\frac{\bar{x}+\Delta}{\bar{x}}\right)^{\bar{x}}\right)^n = a$. They don't give implementation details, but here are a few tricks:

- The lower-bound is symmetric using $\mu = 1 - \bar{x}$
- Take logs to get $\log\left(1 - \frac{\Delta}{1-\bar{x}}\right)(1-\bar{x}) + \log\left(1 + \frac{\Delta}{\bar{x}}\right)\bar{x} = \frac{\log(a)}{n}$ —to remove the expensive power function, the corner cases of $\bar{x} = 0$ or 1 , and improve numerical stability.
- Equation-solve for $\Delta \in [0, 1 - \bar{x} - \epsilon]$ or $[0, \bar{x} - \epsilon]$ to get respectively the upper or the lower bound, where ϵ is the numerical precision limit. If $\bar{x} \notin (\epsilon, 1 - \epsilon)$, the corresponding bound is 0.

```
struct HoefFunctor
{
    double ave, a;
    int n;
    double getTerm(double t, double d) const
    { //limit is 0 safely approached
        if(d < 0) assert(d <= t);
        double result = log(1 + d/t) * t;
        return isfinite(result) ? result : 0;
    }
public:
    double operator()(double d) const
    {
        assert(d >= 0 && d <= 1);
        return (getTerm(1 - ave, -d) + getTerm(ave, d)) * n - log(a);
    }
    static pair<double, double> conf(double ave, int n,
        double confidence = 0.95)
    {
        assert(ave >= 0 && ave <= 1 && confidence > 0 && confidence < 1 &&
            n > 0);
        HoefFunctor f = {ave, (1 - confidence)/2, n};
        double e = numeric_limits<double>::epsilon();
        double upperD = ave < 1 - e ? solveFor0(f, 0,
            1 - ave - e).first : 0;
        f.ave = 1 - ave;
        double lowerD = ave > e ? solveFor0(f, 0,
            ave - e).first : 0;
        return make_pair(ave - lowerD, ave + upperD);
    }
};
```

21.10 Confidence Intervals for Common Location Measures

A **location measure** is any linear function of the quantiles. The mean, median, and many other estimators fall in this category.

For the mean typically use the t -distribution-based interval, where for small samples instead of the normal use t with $n - 1$ degrees of freedom. This is exact for normal samples and in practice most useful for small n from a bounded distribution.

```
pair<double, double> getTConf(IncrementalStatistics const&s, double a = 0.05)
{
    assert(s.n > 1 && a > 0 && a < 1);
    double ste = s.getStandardErrorSummary().stdev() *
        find2SidedConfT(1 - a, s.n - 1);
    return make_pair(s.getMean() - ste, s.getMean() + ste);
}
pair<double, double> getTConf(Vector<double> const& data, double a = 0.05)
{
    IncrementalStatistics s;
```

```

    for(int i = 0; i < data.getSize(); ++i) s.addValue(data[i]);
    return getTConf(s, a);
}

```

The median is more generally the 50th percentile quantile. Logically, the k^{th} quantile is such that fraction k of the data is no larger and $1 - k$ no smaller. For sample quantiles this doesn't lead to a unique value. So for the sample median of even n the convention is to average two middle values, though any in-between value satisfies the logical property. For a general sample quantile average the lower and the upper value when k lands on a discontinuity of the EDF. As a convention, for $k \notin [0, 1]$ the result is $\pm\infty$; in particular this helps with confidence intervals.

```

double quantile(Vector<double> data, double q, bool isSorted = false)
{
    assert(data.getSize() > 0);
    if(q < 0) return -numeric_limits<double>::infinity();
    else if(q > 1) return numeric_limits<double>::infinity();
    int n = data.getSize(), u = q * n, l = u - 1;
    if(u == n) u = l; //check corner cases
    else if(u == 0 || double(u) != q * n) l = u; //and border values
    if(!isSorted)
    {
        quickSelect(data.getArray(), n, u);
        if(l != u) quickSelect(data.getArray(), u, l);
    }
    return (data[l] + data[u])/2;
}

double median(Vector<double> const& data, bool isSorted = false)
{
    return quantile(data, 0.5, isSorted);
}

```

To get nonparametric confidence intervals for quantile estimates, invert the **one-sample sign test**. The inversion technique is discussed generally later in the chapter. The idea of the sign test is that given n samples from a distribution with a known k^{th} quantile θ , expect kn of them $< \theta$. So the null distribution is binomial(n, k) \approx normal($kn, nk(1 - k)$). Then given $z = 2$ (or another reasonable value), solve to get $\theta \in (\text{quantile}_{k-d}, \text{quantile}_{k+d})$, with $d = z \sqrt{\frac{k(1-k)}{n}}$ (Gibbons & Chakraborti 2020; they use an optional continuity correction). E.g., for data = {1, 2, ..., 99, 100}, the sample median = 50.5, with $(l, u) = (41, 60)$. For the mean, the normal interval = 50.5 ± 5.8 . This is expected because the median is less efficient for the normal distribution. For the trimmed mean (discussed later), get estimate 50.5 ± 7.7 .

```

pair<double, double> quantileConf(Vector<double> const& data, double q = 0.5,
    bool isSorted = false, double z = 2)
{
    double d = z * sqrt(q * (1 - q) / data.getSize());
    return make_pair(quantile(data, q - d, isSorted),
        quantile(data, q + d, isSorted));
}

```

Beware that for discrete distributions quantiles can be undefined—e.g., for binomial with $p = 0.5$ the median is neither 0 nor 1. A confidence interval still makes sense, though it will be $[0, 1]$ even in the limit.

It's wrong to think of the sign test logic as using only the signs of the data and loosing too much information. The data points are random and individually carry information only as part of a sample. Any statistic gets only what it needs to calculate some concentration such as the mean or the median. Properties of S determine usefulness.

Consider the c sample trimmed mean (Wilcox 2016): take out $\lfloor cn/2 \rfloor$ smallest and largest values, and return the mean of the rest. Typically $c = 0.2$. Given an array of data, can compute it in expected $O(n)$ time with two calls to quickselect to isolate the middle from the sides. (An online calculation is also possible—like for the median, but instead of two heaps use three dynamic sorted sequences).

```

double trimmedMean(Vector<double> data, double c = 0.2,
    bool isSorted = false)
{
    int n = data.getSize(), trim = c * n;
    assert(n > 0 && c >= 0 && c < 0.5);
    if(!isSorted)
    {

```

```

        quickSelect(data.getArray(), n, n - trim - 1);
        quickSelect(data.getArray(), n - trim - 1, trim);
    }
    double sum = 0;
    for(int i = trim; i < n - trim; ++i) sum += data[i];
    return sum / (n - 2 * trim);
}

```

$c = 0$ is equivalent to the mean, and $c = 0.5$ to the median. For Cauchy the sample trimmed mean is more efficient at estimating the **location parameter** than the sample median (StackExchange 2016a). For symmetric distributions the sample mean, the sample median, and the sample trimmed mean estimate both the mean and the median, which are the same. Asymptotically the sample trimmed mean \sim normal (Wilcox 2016), and can estimate its standard deviation analytically. In particular, **Windsorization** of a sample is similar to trimming, except instead of dropping the a tails, replace them by copying the retained extreme values. Then $\text{SE}(\text{sample trimmed mean}) = \frac{\text{SE}(\text{the Windsorized sample})}{1 - 2c}$.

```

double trimmedMeanStandardError(Vector<double> data, double c = 0.2,
    bool isSorted = false)
{
    int n = data.getSize(), trim = c * n;
    assert(n > 0 && c >= 0 && c < 0.5);
    if(!isSorted)
    {
        quickSelect(data.getArray(), n, n - trim - 1);
        quickSelect(data.getArray(), n - trim - 1, trim);
        //Windsorize tails
        for(int i = 0; i < trim; ++i) data[i] = data[trim];
        for(int i = n - trim; i < n; ++i) data[i] = data[n - trim - 1];
        IncrementalStatistics s; //calc regular se of values
        for(int i = 0; i < n; ++i) s.addValue(data[i]);
        return s.getStandardErrorSummary().stddev() / (1 - 2 * c);
    }
    pair<double, double> trimmedMeanConf(Vector<double> const& data, double z = 2)
    {
        double tm = trimmedMean(data), ste = trimmedMeanStandardError(data) * z;
        return make_pair(tm - ste, tm + ste);
    }
}

```

This estimate works well in practice for $c = 0.2$, but is less accurate for much smaller or much larger values—particularly for the median (Wilcox 2016).

Trimming helps with efficiency because the accuracy of the sample mean depends on averaging, and trimming drops observations that reduce accuracy by being too large/small—i.e., accuracy of the sample mean is sensitive to fat-tail distributions.

For the sample median, relative to the sample mean, the ARE is (Gibbons & Chakraborti 2020):

- ∞ for Cauchy
- $1.5/\pi$ for the normal
- $1/3$ for uniform (the worst case)

The sample mean is always efficient for bounded distributions, as shown by finite-sample inequalities. For a continuous unknown distribution, the sample trimmed mean or the sample median are a safer choice.

For asymmetric distributions the mean \neq the trimmed mean \neq the median, so need to decide which measure of location to use. E.g., for the housing prices, prefer the median, but this decision is based on domain knowledge and not any particular estimator properties. Even with the same data, might want different parameters depending on the action—e.g., given income data, want median income for welfare and mean income for tax estimation. Also, when exponentially large values are exponentially unlikely, the mean may be large, and the median small. Here a large mean is irrelevant because large numbers are effectively impossible. For something like salary, the mean is informative to employers but the median to employees because few become CEOs. For Bernoulli with $p \neq 0.5$, only the mean makes sense—the median is 0 or 1, and the trimmed mean is biased with respect to the mean. Finally, many distributions don't have good location measures in a sense that no single number represents typical behavior well. The trimmed mean and other locations which match the median for symmetric distributions are called **pseudomedians**.

21.11 Outliers and Robust Inference

Almost all statistical procedures assume iid data. But practical data may contain non-iid observations due to various **contamination** sources, such as mistakes in data reporting or measurements. Such data are **outliers**. Need extra information to distinguish an outlier from a sample from a fat-tail distribution, i.e., one of:

- Domain knowledge—impossible values are outliers—e.g., age = -1, etc.
- A strong prior belief in the given model—e.g., if the data is normal with known parameters, values that are many standard deviations away are exponentially unlikely. These are suspected outliers. But the model can be a bad approximation, which is usually more often so for the tails. So only confirmed-good models should be considered for outlier identification.

Outliers can be interesting on their own, e.g., they can be interpreted as newsworthy anomalies (see the "Machine Learning—Other Methods" chapter for more on this), but this isn't considered further. The main ways to reduce the effect of outliers in estimation:

- Identify and remove suspected outliers before applying estimation procedures. This is sometimes suggested but is an ill-posed problem with no good specific general strategy or ways to measure performance.
- Use **robust estimators** on which outliers have a limited effect. This is the preferred strategy, and estimators that are efficient for fat-tail distributions are usually already robust, as defined later. E.g., for estimating the mean of a symmetric distribution, the sample trimmed mean and the sample median are robust. If want to average age in a normal distribution model, the "-1" would have a strong bias effect in a small sample. But with the sample trimmed mean or the sample median get only a small impact on the estimate, such as shifting the sample median to the next smaller observation in the sorted order.

In this logic only outliers with very large/small values are worth guarding against because those with similar values to the other data can't have much impact. More formally, consider **influence**—how much impact a single observation can have. E.g., for the sample mean the influence = ∞ because an observation = ∞ would make it so. But for the sample median the influence is finite. The sample median remains finite with many ∞ values—up to 50% of the data. So its **breakdown point** = 50%. Similarly, for c -trimmed mean, the breakdown is c . Breakdown matters for small samples more than for large ones—e.g., even with 10% contamination a fraction of samples $n = 20$ will be unusable even with a mildly robust estimator (Huber & Ronchetti 2009). I was once on a jury that needed to agree on the amount of damages to be awarded, before knowing anything about robust statistics. So naturally I suggested using the mean, and everyone agreed. One juror gave \$0, and another \$10 million, which overall were outliers that lead to a higher amount that would be awarded if the median was used instead (at least \$1 million difference).

If gain robustness, can lose stability. E.g., for the binomial with $p = 0.5$ the sample median is very robust, but gives 0 or 1 for p no matter how large n is. Something similar can happen for the sample trimmed mean.

Forgetting about outliers, another advantage of robust estimators is reduced **sensitivity to model specification**—e.g., when the data \sim some distribution close to but not exactly normal. Even with slight deviations from distribution assumptions a robust estimator can be much more valid and/or efficient. In case of a fat-tail distribution it most certainly will be. See Wilcox (2016) for many examples. But the usually more involved computations of robust estimators are inconvenient—e.g., generally can't compute incrementally, which is important for large data sets.

E.g., the sample standard deviation is inefficient with fat-tail distributions. A robust alternative is the **normalized median absolute deviation (MADN)**. Take the median of absolute differences to the median, and normalize to make the value equal to the standard deviation of the normal distribution (Wilcox 2016):

```
pair<double, double> medianMADN(Vector<double> data)
{ //scaled to match normal std dev
    assert(data.getSize() > 2);
    double med = median(data);
    for(int i = 0; i < data.getSize(); ++i) data[i] = abs(data[i] - med);
    return make_pair(med, 1.4826 * median(data));
}
```

For normal data the efficiency is only 37%, but the breakdown is 50%, and for other distributions efficiency can be much higher.

21.12 Functions of Estimates

To make inference about $g(\theta)$ for some g , it's usually best to estimate it directly. But sometimes this is problematic.

For confidence intervals, a useful tool is the **delta method** (Wasserman 2004): Asymptotically, if $\hat{\theta} \sim \text{normal}\left(\theta, \frac{\sigma^2}{n}\right)$, and $g'(\theta)$ exists and $\neq 0$, $g(\hat{\theta}) \sim \text{normal}\left(g(\theta), \frac{|g'(\theta)|\sigma^2}{n}\right)$. So for estimation multiply s by $|g'(\theta)|$. This also works for $D > 1$ using ∇g and the covariance matrix.

If $g'(\theta) = 0$, $g''(\theta)$ determines the asymptotic behavior, and the limiting distribution is chi-squared (Das-Gupta 2008). More generally, use a Taylor series expansion of g , taken as far as needed. The property is asymptotic, so need simulation to test specific cases, particularly given that for g such as exponentiation the finite-sample distribution clearly isn't normal.

In other cases reason manually. E.g., consider $g(x) = \begin{cases} x, & \text{if } x < 0 \\ 2x, & \text{else} \end{cases}$, and $f = \text{sample mean}$. At $x = 0$ g isn't differentiable, so can't use the method directly. But can look at the distributions of nearby values of θ to deduce the behavior of S as transformed by g . If the true mean = 0, $g(\bar{x}) \sim$ distribution of which the left half is normal, and the right half stretched normal (this may seem like but isn't a 50% mix of $\text{normal}(0, 1)$ and $\text{normal}(0, 2)$).

For $g(x) = |x|$, even simulation of S fails to give a confidence interval that will contain $\theta = 0$ 95% of the time because it's on the boundary.

Sometimes want **transformation invariance**—i.e., transforming the data is equivalent to transforming the confidence intervals or other estimates in the same way. But a procedure either has it or doesn't, and it makes little sense to select a procedure based on this property.

21.13 Measuring Algorithm Runtime

Typically run the algorithm several times, take some measurements, and compute the wanted statistics. This works reasonably well in many cases, but must be careful. Some sources of noise:

- Randomization in the algorithm
- Bias of the clock (variance not as important)
- Variance of the system load

In all cases the noise only adds to the runtime, so the distribution is asymmetric. Even if the clock is accurate, want to find out $E[\text{algorithm's runtime}]$, but get to observe algorithm + system runtime. And the system can introduce substantial delays with a small probability, i.e., the system runtime potentially has a fat tail distribution.

A simple procedure to get better measurements is to run several batches such that each has enough runs to overcome noise from the clock and the randomization in the algorithm. Use the sample trimmed mean for the runtime of a batch. Divide it by the number of runs in a batch to estimate typical runtime. This estimates algorithm's runtime + typical system's runtime, avoiding extreme, less likely cases of major system delays. Hopefully the differences across systems will be proportional to their speed, and relative differences in performance will be only due to algorithm difference.

The 90% quantile may be a better location measure in some cases because having it under control keeps most users happy from a semi-worst-case view. But if this matters, want to compare for **stochastic dominance**, i.e., one alternative outperforms the rest in all quantiles of the distribution, and not just a particular one such as the median. Only for single-parameter distributions the mean difference usually implies stochastic dominance—e.g., even for the normal a larger variance adds more mass to the tails so that comparing the mean and the 90% quantile can give different results.

21.14 Correlation Analysis

Want to detect various relationships. **Pearson correlation** detects linear relationships using the sample correlation coefficient $r = \frac{1}{n-1} \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{s_x s_y}$ to estimate the linear correlation coefficient ρ . The answer

$\in [-1, 1]$, with 1 meaning a perfect linear relationship, and -1 a perfect negative-linear one. Values away from these mean only lack of a linear relationship—e.g., for samples from $y = x^2$, $\rho = 0$. $\rho \geq 0.95$ is considered strong.

```
double PearsonCorrelation(Vector<pair<double, double>> const& a)
{
    int n = a.getSize();
    assert(n > 1);
    IncrementalStatistics x, y;
    for(int i = 0; i < n; ++i)
```

```

    {
        x.addValue(a[i].first);
        y.addValue(a[i].second);
    }
    double covSum = 0;
    for(int i = 0; i < a.getSize(); ++i)
        covSum += (a[i].first - x.getMean()) * (a[i].second - y.getMean());
    covSum /= n - 1;
    double result = covSum/sqrt(x.getVariance() * y.getVariance());
    return isfinite(result) ? result : 0; //check for div by 0
}

```

Computing a confidence interval for ρ analytically requires assumptions. A reasonably accurate asymptotically normal interval works well if first do a transformation. In particular (Wikipedia 2019b), for data from bivariate normal with correlation ρ , asymptotically $\tanh^{-1}(r) \sim \text{normal}\left(\tanh^{-1}(\rho), \frac{1}{n-3}\right)$.

```

pair<double, double> PearsonCorrelationConf(double corr, int n, double z = 2)
{
    assert(0 <= corr && corr <= 1 && n > 3);
    double stat = atanh(corr), std = 1/sqrt(n - 3);
    return make_pair(tanh(stat - z * std), tanh(stat + z * std));
}

```

The adjustment by 3 isn't coming from the mathematics but is known to work better numerically (Lehmann & Romano 2005; see also Efron & Tibshirani 1993 for an example calculation). The transformation is asymptotically correct only for the bivariate normal distribution (Wilcox 2016), but this hasn't prevented widespread general use.

Because variables may be related monotonically but not linearly, **Spearman correlation** first transforms them into ranks and then calculates Pearson correlation. Instead can calculate rank mean and variance more directly, but this gets complicated when have ties.

```

Vector<double> convertToRanks(Vector<double> a)
{//create index array, sort it, and convert indices into ranks
    int n = a.getSize();
    Vector<int> indices(n);
    for(int i = 0; i < n; ++i) indices[i] = i;
    IndexComparator<double> c(a.getArray());
    quickSort(indices.getArray(), 0, n - 1, c);
    for(int i = 0; i < n; ++i)
        {//rank lookahead to scan for ties, then change a entries
            int j = i;
            while(i + 1 < n && c.isEqual(indices[j], indices[i + 1])) ++i;
            double rank = (i + j)/2.0 + 1;
            for(; j <= i; ++j) a[indices[j]] = rank;
        }
    return a;
}

double SpearmanCorrelation(Vector<pair<double, double> > a)
{
    Vector<double> x, y;
    for(int i = 0; i < a.getSize(); ++i)
    {
        x.append(a[i].first);
        y.append(a[i].second);
    }
    x = convertToRanks(x), y = convertToRanks(x);
    for(int i = 0; i < a.getSize(); ++i)
    {
        a[i].first = x[i];
        a[i].second = y[i];
    }
    return PearsonCorrelation(a);
}

```

Even in the presence of ties, asymptotically $r \sim \text{normal}\left(\rho, \frac{1}{n-1}\right)$, which is considered accurate for $n \geq 30$ (Gibbons & Chakraborti 2020).

```
pair<double, double> SpearmanCorrelationConf(double corr, int n, double z = 2)
{
    assert(0 <= corr && corr <= 1 && n > 1);
    double std = 1/sqrt(n - 1);
    return make_pair(corr - z * std, corr + z * std);
}
```

Correlation analysis is rarely useful unless it's exploratory. With any correlation measure must have a clear idea of what it's measuring. In general, no single number gives a good measure of degree of dependence. Usually want to have a predictive model, which is best done using regression. Though, e.g., in finance, historical correlations between securities are very useful for calculating risks of portfolios, but here pay no attention to general dependence. Pearson and Spearman correlations aren't robust—e.g., even rotating the data a little changes Pearson correlation substantially (Wilcox 2016). See also Vexler et al. (2016) for more on correlation. Any interpretation of a high correlation must not be causal—e.g., ice cream consumption and murder rates are correlated, but the causal factor for both is the summer.

Also notice that the confidence bands in both cases are data-agnostic, so even with $n = 100$ will get 95% confidence band $\approx r \pm 0.2$, which is rather wide.

An alternative to Spearman correlation is **Kendall's correlation** (Wikipedia 2021), which for every observation counts the number of **concordant pairs**—where for every other observation both x and y are smaller or larger than the reference observation. The result = the total number of concordant pair/Choose($n, 2$). Some differences:

- The computation is $O(nlg(n))$ using sorting
- The result is more robust than Spearman
- The normal approximation is better for the resulting binomial

A common adjustment to the formulas is for where the mean is known. E.g., for time series data such as stock returns, a common model is to difference the data and look at daily changes, which have a known mean 0 in the simplest model or some estimated long-term trend mean. The Pearson formula is easy to adjust—plug in the known means and remove a degree of freedom from the normal confidence for each known mean.

For Spearman this is more difficult—need to use the rank of the known mean but exclude that rank from the calculation. The distribution also changes, and a general way to adjust the normal approximation, if any is needed, is to be studied. For Kendall a simple heuristic, which is less accurate for other methods, is to include the known mean as an observation, and, perhaps optionally, adjust the calculation to not use it with other data points as reference observations.

21.15 The Bootstrap

Can estimate accuracy of a black-box estimator f with only minor assumptions by resampling the data set. EDF $F(x)$ approximates the real distribution T . In 1D have **Dvoretzky-Kiefer-Wolfowitz inequality**: $\Pr(\max_x |(T(x) - F(x))| > \epsilon) \leq 2e^{-2n\epsilon^2}$ for error $\epsilon > 0$, and such that this p -value ≤ 0.5 (DasGupta 2008). For vector-valued f don't have a multivariate DKW, only asymptotic convergence of F to T , but this is good enough for the bootstrap.

Bootstrap creates samples $f_i = f(\text{resample}_i)$, where resample_i consists of n samples from F , from a distribution B . $B \approx S$ when $F \approx T$ (unless f is problematic). So treat f_i as iid from S , and use them to find a confidence interval for θ :

1. **b times**
2. **Pick n random items from the data with replacement**
3. **$f_i \leftarrow f(\text{the resample})$**
4. **Calculate a confidence interval for $\hat{\theta}$ from $b f_i$ values, using one of the methods discussed later**

If f is such that increasing n doesn't magnify the error of sampling from F instead of T , heuristically $B \approx S$ is becoming arbitrarily accurate as $n \rightarrow \infty$. In general, expect the bootstrap to work when simulation works. It will also fail when simulation does:

- Let $f = g(\text{sample mean})$, where g is the step function, and the true mean = 0. Then $g(\text{sample mean}) \sim \text{Bernoulli}(0.5)$ regardless of n , and no confidence interval shrinks with n .

- If $\text{variance}(S) = \infty$, such as when use mean of Cauchy samples, have $B \approx S$, but variance-based confidence intervals are invalid (quantile-based intervals are valid but useless).

As discussed later, no method reliably detects when the bootstrap gives wrong confidence intervals. As for maximum likelihood, need regularity conditions. Like with maximum likelihood, in practice rarely can or bother to check these.

All confidence interval methods work with a resampler. For single-sample f , use the following:

```
template<typename FUNCTION, typename DATA = double> struct BasicBooster
{
    Vector<DATA> const& data;
    Vector<DATA> resample;
    FUNCTION f;
    BasicBooster(Vector<DATA> const& theData, FUNCTION const& theF = FUNCTION())
        :data(theData), f(theF), resample(theData) {assert(data.getSize() > 0);}
    void boot()
    {
        for(int i = 0; i < data.getSize(); ++i)
            resample[i] = data[GlobalRNG().mod(data.getSize())];
    }
    double eval() const{return f(resample);}
    //for the bootstrap-t interval
    BasicBooster cloneForNested() const{return BasicBooster(resample, f);}
};
```

Assume that S is normal. Then calculate the standard deviation from B to form the **bootstrap normal confidence interval**. 25 resamples suffice, and 200 is recommended for better accuracy (Efron & Tibshirani 1993).

```
template<typename BOOTER> double bootstrapStde(BOOTER& booter, int b = 200)
{
    assert(b > 2);
    IncrementalStatistics s;
    for(int i = 0; i < b; ++i)
    {
        booter.boot(); //resample
        s.addValue(booter.eval());
    } //beware - get standard deviation here not standard error
    return s.stdev();
}
template<typename BOOTER> pair<double, double> bootstrapNormalInterval(
    BOOTER& booter, int b = 200, double z = 2)
{
    double q = booter.eval(), stde = bootstrapStde(booter, b);
    return make_pair(q - z * stde, q + z * stde);
}
```

E.g., on a particular run on a sample of 1000 uniform01 values, with f = the sample mean, both the CLT and the bootstrap normal interval gave $\hat{\theta} = 0.493 \pm 0.019$, with 95% confidence. With f = the sample median, the nonparametric interval gave $\hat{\theta} = 0.512^{+0.035}_{-0.029}$, and the bootstrap normal interval 0.512 ± 0.032 .

This actually calculates the Wald interval, but without needing mathematical derivations for the standard errors of unknown f . It has many advantages over more accurate intervals discussed later:

- The accuracy is usually good enough—for small n the estimation error of the normal assumption is small, and for large it's a good approximation because most estimators compute **concentrations** like the mean or the median.
- The computation is orders-of-magnitude more efficient than that of the more accurate intervals. For large n and expensive f it might be the only computationally feasible solution.
- The more accurate intervals are theoretically so only for smooth f , whereas the bootstrap is more useful for black-box user-defined f , which may not be smooth at all.
- It's easy to explain and leads to a clear interpretation of estimate + error.
- It's easy to make adjustments for multiple testing.

Still, the normality assumption is somewhat inaccurate for skewed S —e.g., for the mean of the lognormal distribution. Asymptotically, an interval based on Chebyshev's inequality, using $z = \sqrt{1/a}$, is consistent for

the sample mean \bar{S} with finite variance because $s \rightarrow \sigma$. But it's known to be conservative, so use a **mixed Chebyshev interval** with $z = cz_{\text{normal}} + (1-c)\sqrt{1/a}$. This small correction to the normal interval makes up for several issues:

- In the general case, such as when have multiple samples, n is unavailable, and can't use the t -distribution directly
- The normal is only an approximate model, particularly for small n —the Wald interval needn't be even asymptotically correct as f might not have a CLT
- Bootstrap slightly underestimates s (Efron & Tibshirani 1993) by using the maximum likelihood estimate
- The true interval is likely to be asymmetric
- f might be slightly biased

This is effective even with a small mix using $c = 0.9$. E.g., for the usual 95% confidence get $z \approx 2.21$, which matches the t -distribution with 10 degrees of freedom. For 99.73%, corresponding to $z = 3$, for $c = 0.9$ get $z \approx 4.62$. With $c = 0.8$, get $z \approx 2.46$, which matches the t -distribution with 6 degrees of freedom, and for 99.73% get $z \approx 6.25$.

```
double getMixedZ(double a = 0.05, double c = 0.9)
{
    assert(a > 0 && a < 1 && b > 0 && b < 1);
    double z = find2SidedConfZ(1 - a), chebZ = sqrt(1/a),
    return z * c + chebZ * (1 - c);
}

template<typename BOOTER> pair<double, double> bootstrapMixedInterval(
    BOOTER& booter, int b = 200, double a = 0.05)
//use 100 mix
{
    assert(b > 2 && a > 0 && a < 1);
    return bootstrapNormalInterval(booter, b, getMixedZ(a));
}
```

Asymptotically $f(F) \rightarrow f(T) = \theta$, assuming f is consistent. Also $f(F) \approx \hat{\theta}$ for a reasonable f because using a data set consisting of several copies of the original data should make no difference unless f considers n in some way. Technically, f can duplicate the data for its own use as much as needed, and given that it doesn't means little if anything is gained. This is the only deviation from the idea of doing all computations in the "bootstrap world" $F \approx T$ due to its substantial computational savings; otherwise would need to simulate a large number of samples to compute $f(F)$. Recall that the theoretically ideal interval for θ is $(l, u) = (\hat{\theta} + \theta - S_{1-a}, \hat{\theta} + \theta - S_a)$. Plug in B for S and $f(F)$ for θ . Assuming $f(F) \approx \hat{\theta}$ (which is actually exact if f is a plug-in estimate; Lunneborg 2000), get $(l, u) = (2\hat{\theta} - B_{1-a}, 2\hat{\theta} - B_a)$. Called the **pivotal interval** (per Wasserman 2004; or **basic**, **reversed**, and **reflected** in other sources), it effectively assumes only $f(T, n) - \theta \approx f(B, n) - \hat{\theta}$.

Though $b = 1000$ is considered enough for such tail-quantile-based intervals (Efron & Tibshirani 1993), with modern computing power 10000 seems better per Chihara & Hesterberg (2018), and 3000 is a good golden middle per my experiments (discussed later). A smaller number such as 200 for the normal interval is no longer good enough because the standard deviation is easier to estimate than tail quantiles. E.g., with $b = 100$, it's pointless to ask for a 99.5% quantile-based interval. The number of distinct resamples = $\binom{2n-1}{n}$ and exceeds any practical choice of b . So with some large enough b all distinct values will have been tried in about equal proportion, and increasing it would improve neither accuracy nor resolution. So $b \rightarrow \infty$ gains nothing (despite being important for theory), and n limits the accuracy.

Though the pivotal interval is consistent (with regularity conditions, as for other intervals), it's not reasonable to expect that $\hat{\theta} \approx \theta$ because the later is fixed, and the former is random. But the differences should match reasonably well in that if $\hat{\theta}$ swings from θ in the same direction as the quantiles B^* do from S^* . Still, practical performance of this interval is poor for skewed S —Hesterberg (2014) describes it as "wrong pivot, forward" to account for its going in the wrong direction to correct the skew from the undue influence of $\hat{\theta}$ in the calculations.

A better pivot uses the standard deviation, leading to the **bootstrap-t interval**. The name is based on that the pivot looks like the t statistic, but otherwise has nothing to do with it. Think of the pivotal interval as $(l, u) = (\hat{\theta} - (B - \hat{\theta})_{1-a}, \hat{\theta} - (B - \hat{\theta})_a)$, where adjust the computed values of B by $\hat{\theta}$ right away. To improve it, use $(l, u) = (\hat{\theta} - s(B)P_{1-a}, \hat{\theta} - s(B)P_a)$, where P is the distribution of $p(\text{resample}) = \frac{f(\text{resample}) - \hat{\theta}}{\text{std}(f(\text{resample}))}$. Can

compute the resample-specific standard deviations analytically only in special cases, such as when $f = \text{the mean}$ (but this technique is nonetheless useful—see Wilcox 2016 for some examples). In the general case apply bootstrap recursively—i.e., compute the resample-specific standard deviations using a nested bootstrap that uses the resamples as the data. I.e., create a “subbootstrap world” for the bootstrap world. This is computationally demanding, so use $b = 1000$, and $b_2 = 50$ for the nested bootstraps. To keep track of the quantiles use two heaps of size $t \approx ba$ to store only the tails with the extreme values. The resource use is $O(fbb_2\ln(t))$ time and $O(n + t)$ space.

This dramatically improves performance both theoretically and practically. The interval is consistent like the pivotal one, and using resample-specific standard deviations makes a big difference, as verified by simulations. Theoretically (Efron & Tibshirani 1993), bootstrap- t is second-order accurate for smooth f —i.e., the one-sided coverage = $a/2 + o(n^{-1})$. The normal and the pivotal intervals are only first-order accurate—i.e., the one-sided coverage = $a/2 + o(n^{-1/2})$. These refer to asymptotic approximation error, and for the normal interval implicitly assume extra conditions such as that a CLT holds for f .

Unfortunately, bootstrap- t is known to be unstable for small n , enough so for Efron & Tibshirani (1993) to recommend against its use, except for location measures such as the sample mean and quantiles. My simulations (discussed later) show two instabilities:

- In length—e.g., for Pearson correlation for $n = 5$
- In coverage—e.g., for Spearman correlation and normal standard deviation for $n = 5$ and 10

A simple solution is to use the conservative (for the sample mean) Chebyshev inequality to cap the computed pivots to $[1, \text{Chebyshev } z]$. Here 1 is the z value corresponding to the maximal $a = 1$, and still get intervals of ≤ 1 standard error. This dramatically improves performance in problematic cases and makes the method safe in general. It's unclear whether a better general capping strategy exists.

```
template<typename BOOTER> pair<double, double> bootstrapTIntervalCapped(
    BOOTER& booter, int b = 1000, int b2 = 50, double confidence = 0.95)
{
    assert(b > 2);
    double a = (1 - confidence)/2;
    int tailSize = b * a;
    if(tailSize < 1) tailSize = 1;
    if(tailSize > b/2 - 1) tailSize = b/2 - 1;
    //max heap to work with the largest value
    Heap<double, ReverseComparator<double>> left;
    Heap<double> right; //min heap for the smallest value
    double q = booter.eval();
    IncrementalStatistics s;
    for(int i = 0; i < b; ++i)
    {
        booter.boot(); //resample
        double valStat = booter.eval();
        if(isfinite(valStat)) s.addValue(valStat);
        else continue;
        BOOTER booter2 = booter.cloneForNested();
        double stdeInner = bootstrapStde(booter2, b2),
               value = (valStat - q)/stdeInner;
        if(isnan(value)) continue; //possible division by 0 OK due to cap
        if(left.getSize() < tailSize)
        { //heaps not full
            left.insert(value);
            right.insert(value);
        }
        else
        { //heaps full - replace if more extreme
            if(value < left.getMin()) left.changeKey(0, value);
            else if(value > right.getMin()) right.changeKey(0, value);
        }
    }
    //beware - get standard deviation here not standard error
    //protect against bad data
    double normalZ = 1, chebZ = sqrt(1/(2 * a)), stde = s.n > 2 ? s.stdev() :
        numeric_limits<double>::infinity(), leftZ = right.getSize() > 0 ?
        min(max(right.getMin(), normalZ), chebZ) : chebZ, rightZ =
        left.getSize() > 0 ? min(max(-left.getMin(), normalZ), chebZ) : chebZ;
```

```

    return make_pair(q - stde * leftz, q + stde * rightz);
}

```

Another way to get a second-order interval is using **bootstrap calibration** (Boos & Stefanski 2013), that works with any interval:

1. Instead of calculating the interval from the original b resamples, create b_2 resamples of B , and calculate the interval on each of them, with the wanted a .
2. Of the b_2 intervals, some fraction will be to the right and some to the left of θ , failing to contain it. So repeat (1) with adjusted side target a to get the wanted ones using numerical binary search.
3. Compute the primary interval from the original b resamples, but with the adjusted a .

Some implementation details that depend on the used interval:

- The range for the binary search, which is reduced to equation solving for 0. Can use exponential search, but in the intervals discussed below have good bounds.
- What to do if don't have a solution in the above range (the endpoints don't have different signs). A continuous behavior is to take an upper bound a if enough, and a lower bound one otherwise.

A simple method, very similar in concept to the bootstrap- t interval, is the **calibrated Chebyshev interval**. It uses calibration with the Chebyshev-inequality interval. Unlike the normal interval, this gives an automatic search range as $[a_{\text{target}}, 1]$, which are the bounds of the capped bootstrap- t . Also use its $b = 1000$ and $b_2 = 50$ as defaults. Because the evaluations and not the binary search are the main computation, both intervals take the same time to compute. Need $O(b)$ memory.

Though pivoting and calibration are in principle equivalent, as pointed out in major sources, my simulation slightly favors the former. Nevertheless the idea is useful for better understanding of bootstrap- t and calibration:

- They don't share the breakdown of the uncapped bootstrap- t for $n = 5$ or 10.
- The reason for this breakdown seems to be that the subsample variances are a bit smaller than the full-sample ones, which leads to slightly longer intervals than necessary. So it's nested estimation and not the pivot that is problematic.

Technically can have $\hat{\theta} \notin (l, u)$ for many intervals, but this is unlikely because some resamples ought to lead to more extreme values than the sample, and an occurrence is a potential diagnostic of something gone wrong. The normal-based, the original and the capped bootstrap- t , and the calibrated Chebyshev intervals don't have this issue.

The other major construction of more-accurate-than-the-normal intervals is based on the ability to find a transformation that makes S look normal. In particular (Efron & Tibshirani 1993), this leads to the **percentile interval**— $(l, u) = (B_a, B_{1-a})$. It needs the same b as the pivotal interval and is also first-order accurate. It's the method of choice for many users:

- Deceptively simple—no analytical thinking is needed to use it, including distribution calculations, but it and not the pivotal interval is backward
- It performs better than the pivotal in practice by swinging the right way for skewed S
- It has some nice properties such as respecting range constraints and transformation invariance

There is no relation between the derivations of pivotal and percentile intervals—they are symmetric to each other around $\hat{\theta}$ by accident. But this shows that percentile is consistent for symmetric S , in addition to when a normalizing transformation exists.

Coverage isn't related to invariance, and if want to transform a confidence interval, it's not a good idea to compute it in the first place. Even the normal interval isn't invariant. Can trim computed intervals to meet range constrains—the trimming amount is typically too small to either increase length or reduce coverage noticeably. So the benefit of invariance is minor.

For quantile-based intervals, compared to using the quantile function to interpolate nearby values in the general case, rounding to get more extreme values is a simple, slightly conservative choice that also preserves transformation invariance.

Can improve by adding bias correction (the **BC interval**) and skewness correction (the **BCa interval**). See Efron & Tibshirani (1993) and Efron & Hastie (2016) for details. The latter is second-order accurate for smooth statistics but unfortunately only available as a black box for single-sample f (and not multi-sample ones like the difference of medians). Also the calculation of its “acceleration” uses the **jackknife** which doesn't work for nonsmooth f like the sample median, and so isn't robust in getting 2nd order. Percentile has poor coverage for small samples (Hesterberg 2014), and BC strictly improves it. Asymmetric tail sizes of BC and BCa may be problematic if b isn't enough to estimate the corresponding quantiles well, but from my simulations this doesn't seem to be an issue for both length and coverage.

Calibration also effective with the percentile interval; $b = 1020$ and $b_2 = 98$ as default are optimal choice

for a budget of 100000 evaluations (Boos & Stefanski 2013). The search range = [0, 1]; 1 is the largest value to produce a valid interval due to the symmetry constraints of the percentile interval. The performance is better than that of BC and BCa. The memory is $O(bb_2)$, which far exceeds that of the calibrated Chebyshev interval, which gets away with only storing the nested variances.

```

class BTPCFunctor
{
    Vector<Vector<double>> sets;
    bool findLeft;
    double q, aTarget;
    pair<double, double> makeInterval(int i, double a) const
    {//percentile
        assert(0 <= i && i < sets.getsize());
        double b = sets[i].getsize();
        int left = max<int>(0, a/2 * b),
        right = min<int>(b - 1, (1 - a/2) * b);
        return make_pair(sets[i][left], sets[i][right]);
    }
    double operator() (double a) const
    {//returns < 0 if under target
        int missCount = 0, b = sets.getsize();
        for(int i = 0; i < b; ++i)
            {//make sure that don't confuse miss left and miss right
                pair<double, double> conf = makeInterval(i, a);
                missCount += (findLeft ? q < conf.first : conf.second < q);
            }
        return aTarget/2 - missCount * 1.0/b;
    }
public:
    BTPCFunctor(double theQ, double a): q(theQ), aTarget(a), findLeft(true) {}
    void addSet(){sets.append(Vector<double>());}
    void addValue(double value){sets.lastItem().append(value);}
    void closeSet()
        {quickSort(sets.lastItem().getArray(), sets.lastItem().getsize());}
    void flipSide(){findLeft = !findLeft;}
    double findA() const
    {
        double left = 0, right = 1, yLeft = this->operator()(left);
        return haveDifferentSign(yLeft, this->operator()(right)) ?
            solveFor0(this, left, right).first :
            yLeft < 0 ? left : right;
    }
};

template<typename Booter> pair<double, double>
bootstrapDoublePercentileInterval(Booter& booter, int b = 1020,
int b2 = 98, double confidence = 0.95)
{
    assert(b > 2 && confidence > 0 && confidence < 1);
    double q = booter.eval();
    BTPCFunctor f(q, 1 - confidence);
    Vector<double> values(b);
    for(int i = 0; i < b; ++i)
    {
        booter.boot();//resample
        values[i] = booter.eval();
        Booter booter2 = booter.cloneForNested();
        f.addSet();
        for(int j = 0; j < b; ++j)
        {
            booter2.boot();//resample nested
            f.addValue(booter2.eval());
        }
        f.closeSet();
    }
}

```

```

    }
    quickSort(values.getArray(), b);
    int left = max<int>(0, f.findA()/2 * b);
    f.flipSide(); //default is left, now find right
    int right = min<int>(b - 1, (1 - f.findA()/2) * b);
    //cap at q for containment
    return make_pair(min(q, values[left]), max(q, values[right]));
}

```

Based on simulations below both double percentile and capped bootstrap-*t* intervals perform well. Either is a good choice for large computation budgets, though identifying the best of them needs further study. The latter seems safer for black-box *f* due to its consistency, though can also argue that the former has transformation-invariance-based robustness against unknown unknowns.

Distribution and Statistic	Pivotal				Percentile				Normal			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Normal Mean	0.030	0.035	-0.025	0.250	0.030	0.034	-0.025	0.248	0.029	0.034	-0.007	0.251
LogNormal Mean	0.002	0.152	-0.127	0.250	0.008	0.118	-0.127	0.294	0.003	0.131	-0.095	0.278
Levy Median	0.012	0.189	0.240	0.368	0.030	0.032	0.240	0.312	0.004	0.043	0.449	0.289
Normal Stdev	0.027	0.064	-0.087	0.278	0.001	0.109	-0.087	0.242	0.011	0.079	-0.067	0.253
2-Normal Mix Mean	0.027	0.034	-0.027	0.242	0.032	0.036	-0.027	0.254	0.027	0.033	-0.009	0.243
Normal Error Corr	0.135	0.001	0.001	0.245	0.072	0.011	0.001	0.290	0.094	0.003	0.028	0.268
Normal Error Spearman	0.052	0.000	0.422	0.175	0.000	0.038	0.422	0.093	0.005	0.000	0.465	0.096
Average Ranks	8.857	3.000	4.286	8.000	2.571	5.000	5.571	5.286	3.571			
Mixed	BC				BCa				Bootstrap-t			
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL	
0.018	0.021	0.098	0.217	0.033	0.034	0.026	0.255	0.033	0.035	-0.024	0.256	0.022
0.031	0.112	0.000	0.268	0.013	0.111	-0.099	0.317	0.027	0.088	-0.006	0.360	0.011
0.003	0.036	0.589	0.289	0.021	0.040	0.158	0.310	0.020	0.039	0.165	0.306	0.026
0.006	0.062	0.033	0.238	0.021	0.072	-0.093	0.272	0.031	0.053	-0.057	0.280	0.025
0.01	0.023	0.095	0.203	0.033	0.038	0.027	0.260	0.033	0.039	-0.024	0.264	0.024
0.076	0.001	0.136	0.245	0.045	0.020	0.066	0.284	0.037	0.021	0.109	0.280	0.028
0.002	0.000	0.619	0.066	0.016	0.017	0.224	0.224	0.014	0.024	0.273	0.229	0.023
2.857	8.000	1.286	7.429	2.000	6.286	6.857	4.429	7.000	2.571	8.714	7.286	
Bootstrap-t-capped	Percentile Double				Chebyshev Double							
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL	
0.023	0.023	0.073	0.232	0.025	0.027	0.041	0.239	0.024	0.027	0.038	0.236	
0.011	0.064	0.210	0.356	0.020	0.066	0.151	0.363	0.014	0.067	0.178	0.363	
0.020	0.031	0.626	0.378	0.028	0.031	0.274	0.317	0.034	0.047	0.273	0.372	
0.024	0.027	0.161	0.264	0.031	0.033	0.045	0.268	0.050	0.024	0.122	0.284	
0.024	0.027	0.070	0.244	0.029	0.032	0.054	0.261	0.028	0.033	0.035	0.256	
0.028	0.017	0.266	0.282	0.027	0.020	0.260	0.284	0.030	0.021	0.221	0.287	
0.022	0.008	0.239	0.229	0.019	0.042	0.191	0.238	0.037	0.006	0.110	0.238	
2.000	8.286	5.286	5.143	6.143	6.857	5.571	6.000	8.143				

Figure 21.2: Performances of several bootstrap intervals on several tasks with $n = 30$. The correlation function is $f(x) = x - 0.1x^2 + N(0, 0.1^2)$, with $x \sim$ standard uniform. The metrics are a_{left} (aL), a_{right} (aR), the length percentage relative to the exact interval ($\%L$), and the asymmetric Chebyshev loss based on relative lengths (ACL). The exact values and the exact intervals are calculated by simulation with 10^7 samples, and the bootstraps were runs 5000 times each. The b values where the defaults, but for pivotal and the noncalibrated percentile 3000 was selected to increase the computation by about a factor of 15 from the normal and to the second-order intervals to allow computation-based selection).

For larger n all intervals perform similarly on all problems—the asymptotic normality kicks in reasonably well. For feasible simulations only tested bootstrap-*t* among the second-order methods. Mixed gets smaller error rates than the rest at the expense of slightly longer intervals. More importantly, pivotal does worse than the normal in coverage, which in turn does as well as bootstrap-*t* in both coverage and length.

Distribution and Statistic		Pivotal				Percentile				Normal					
		aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL		
Normal Mean		0.023	0.024	0.001	0.154	0.025	0.024	0.001	0.157	0.021	0.023	0.019	0.151		
LogNormal Mean		0.011	0.044	-0.009	0.177	0.016	0.036	-0.009	0.165	0.012	0.039	0.009	0.171		
Levy Median		0.019	0.062	0.010	0.216	0.026	0.023	0.010	0.158	0.019	0.029	0.034	0.162		
Normal Stdev		0.022	0.033	-0.003	0.166	0.019	0.035	-0.003	0.168	0.020	0.032	0.014	0.167		
2-Normal Mix Mean		0.021	0.019	0.001	0.142	0.020	0.021	0.001	0.144	0.019	0.021	0.019	0.144		
Normal Error Corr		0.039	0.012	0.000	0.170	0.032	0.018	0.000	0.160	0.033	0.012	0.016	0.160		
Normal Error Spearman		0.037	0.011	0.028	0.170	0.017	0.034	0.028	0.169	0.024	0.018	0.046	0.152		
Normal Broken Line(Mean)		0.027	0.025	0.000	0.162	0.027	0.028	0.000	0.166	0.026	0.022	0.030	0.159		
Average Ranks		5.125	2.500	5.000	5.375	2.500	5.500	2.625	5.125	4.125	5.125	4.125			
Mixed		BC				BCa				Bootstrap-t					
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL			
-0.013	0.018	0.128	-0.129	0.025	0.025	-0.000	0.157	0.024	0.025	0.000	0.156	0.022	0.023	0.030	0.154
0.007	0.025	0.115	0.152	0.019	0.033	-0.008	0.163	0.027	0.027	0.005	0.165	0.022	0.023	0.044	0.157
0.012	0.018	0.143	0.141	0.024	0.024	0.004	0.156	0.025	0.023	0.005	0.155	0.026	0.022	0.074	0.167
0.010	0.018	0.124	0.137	0.024	0.033	-0.004	0.169	0.028	0.027	-0.001	0.166	0.023	0.025	0.034	0.160
0.014	0.011	0.126	0.124	0.022	0.020	-0.001	0.144	0.021	0.020	-0.001	0.143	0.018	0.018	0.031	0.138
0.023	0.007	0.126	0.147	0.029	0.020	0.000	0.158	0.027	0.021	0.002	0.156	0.025	0.019	0.035	0.155
0.017	0.011	0.156	0.135	0.028	0.024	0.012	0.163	0.026	0.025	0.014	0.162	0.025	0.022	0.031	0.158
0.014	0.013	0.136	0.136	0.027	0.027	-0.001	0.164	0.027	0.028	-0.001	0.166	0.026	0.022	0.072	0.166
	1.000	7.000	1.000	5.625	1.825	4.875	5.000	2.625	4.125	2.500	5.875	3.375			

Figure 21.3: Performance with $n = 1000$, 3000 times eachUsing $n = 5$ is a good stability test:

Distribution and Statistic		Pivotal				Percentile				Normal					
		aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL		
Normal Mean		0.082	0.083	-0.172	0.345	0.083	0.085	-0.172	0.348	0.075	0.077	-0.143	0.342		
LogNormal Mean		0.010	0.302	-0.341	0.299	0.015	0.266	-0.341	0.343	0.010	0.273	-0.303	0.329		
Levy Median		0.016	0.296	####	####	0.032	0.031	####	####	0.006	0.048	####	####		
Normal Stdev		0.113	0.143	-0.275	0.393	0.0000	0.353	-0.275	0.186	0.004	0.202	-0.196	0.286		
2-Normal Mix Mean		0.066	0.079	-0.191	0.313	0.078	0.105	-0.191	0.353	0.064	0.082	-0.163	0.324		
Normal Error Corr		0.389	0.006	-0.771	0.416	0.156	0.000	0.77	0.896	0.109	0.001	1.451	0.639		
Normal Error Spearman		0.305	0.002	0.389	0.384	0.305	0.000	0.389	1.030	0.223	0.001	0.689	0.699		
Average Ranks		8.143	2.000	3.286	7.714	2.000	5.143	5.571	4.000	3.429					
Mixed		BC				BCa				Bootstrap-t					
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL			
0.060	0.063	-0.053	0.338	0.077	0.091	-0.168	0.349	0.076	0.091	-0.143	0.358	0.024	0.027	1.170	0.493
0.006	0.250	-0.231	0.334	0.014	0.268	-0.346	0.344	0.017	0.249	-0.273	0.394	0.002	0.103	3.607	1.920
0.004	0.043	###	###	0.032	0.189	1.361	1.436	0.032	0.189	1.361	1.436	0.004	0.034	###	###
0.001	0.181	-0.111	0.280	0.033	0.314	-0.398	0.306	0.035	0.306	-0.414	0.306	0.132	0.023	2.683	0.864
0.048	0.064	-0.074	0.312	0.076	0.114	-0.188	0.359	0.080	0.120	-0.159	0.382	0.014	0.029	1.288	0.473
0.103	0.001	1.705	0.675	0.043	0.007	3.159	1.186	0.045	0.008	2.794	1.107	0.035	0.011	###	###
0.222	0.001	0.869	0.758	0.002	0.309	1.149	0.230	0.000	0.309	1.152	0.056	0.305	0.000	0.768	0.699
	3.857	6.429	3.571	7.714	4.429	4.857	7.286	5.143	5.286	2.000	9.286	8.857			
Bootstrap-t-capped		Percentile Double				Chebyshev Double									
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL			
0.025	0.028	0.485	0.343	0.044	0.048	0.156	0.355	0.027	0.030	0.467	0.350				
0.002	0.135	0.168	0.423	0.007	0.184	-0.100	0.407	0.002	0.137	0.159	0.427				
0.004	0.033	###	###	0.062	0.059	###	###	0.013	0.042	###	###				
0.098	0.062	0.101	0.420	0.067	0.306	-0.452	0.333	0.104	0.061	0.096	0.419				
0.015	0.031	0.429	0.301	0.038	0.075	0.130	0.379	0.016	0.034	0.412	0.312				
0.074	0.005	2.783	1.049	0.127	0.012	1.523	1.219	0.131	0.004	1.182	0.535				
0.305	0.000	0.696	0.732	0.305	0.000	0.997	1.506	0.280	0.000	0.774	0.761				
	2.714	8.000	5.714	6.000	5.571	8.286	3.857	7.000	6.429						

Figure 21.4: Performance with $n = 5$, 30000 times each; the "###" means orders-of-magnitude larger numbers than the rest; BC and BCa do well on Levy median accidentally

For $n = 10$ the instability of bootstrap-t for the correlation coefficient doesn't disappear though is substantially reduced. The fat-tailed issue doesn't disappear either for most intervals.

Distribution and Statistic		Pivotal				Percentile				Normal					
		aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL		
Normal Mean		0.051	0.051	-0.078	0.297	0.051	0.049	-0.078	0.296	0.049	0.047	-0.057	0.295		
LogNormal Mean		0.003	0.230	-0.245	0.274	0.010	0.191	-0.245	0.325	0.005	0.202	-0.202	0.307		
Levy Median		0.007	0.226	###	1.618	0.042	0.022	###	2.470	0.003	0.034	###	###		
Normal Stdev		0.062	0.100	-0.179	0.346	0.001	0.223	-0.179	0.232	0.012	0.143	-0.145	0.297		
2-Normal Mix Mean		0.042	0.048	-0.090	0.276	0.051	0.058	-0.090	0.304	0.042	0.050	-0.069	0.285		
Normal Error Corr		0.239	0.001	0.164	0.281	0.102	0.005	0.164	0.417	0.119	0.000	0.315	0.352		
Normal Error Spearman		0.105	0.000	0.599	0.238	0.007	0.009	0.599	0.197	0.007	0.0000	0.718	0.131		
Average Ranks		8.571	2.286	4.143		7.571	2.143	4.429	5.857	5.571	4.000				
Mixed	BC				BCa				Bootstrap-t						
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
0.036	0.034	0.043	0.278	0.051	0.049	-0.075	0.297	0.052	0.050	-0.063	0.302	0.026	0.022	0.247	0.275
0.002	0.180	-0.117	0.305	0.012	0.178	-0.216	0.343	0.021	0.155	-0.118	0.399	0.005	0.088	1.050	0.746
0.002	0.028	###	###	0.025	0.054	0.983	0.640	0.026	0.053	1.004	0.644	0.006	0.017	###	###
0.006	0.123	-0.055	0.286	0.029	0.161	-0.226	0.318	0.040	0.132	-0.202	0.335	0.047	0.033	0.655	0.447
0.030	0.036	0.029	0.266	0.054	0.061	-0.086	0.315	0.061	0.066	-0.069	0.337	0.030	0.038	0.251	0.326
0.107	0.0000	0.457	0.355	0.042	0.016	0.527	0.412	0.034	0.015	0.593	0.392	0.015	0.019	2.386	0.604
0.007	0.000	0.898	0.134	0.007	0.010	0.568	0.194	0.007	0.010	0.675	0.207	0.105	0.001	0.594	0.580
	3.857	6.143	2.714	7.714	2.429	4.429		7.286	4.286	6.143	2.714	8.714	8.286		
Bootstrap-t-capped		Percentile Double				Chebyshev Double									
aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL				
0.026	0.023	0.237	0.274	0.030	0.029	0.181	0.288	0.028	0.026	0.198	0.279				
0.005	0.093	0.193	0.398	0.012	0.106	0.058	0.395	0.006	0.097	0.173	0.405				
0.005	0.017	###	###	0.035	0.024	###	2.446	0.011	0.030	###	###				
0.046	0.037	0.249	0.354	0.052	0.120	-0.183	0.349	0.111	0.033	0.159	0.365				
0.030	0.037	0.228	0.318	0.041	0.046	0.198	0.356	0.033	0.041	0.191	0.327				
0.030	0.015	0.901	0.435	0.020	0.023	1.643	0.533	0.034	0.020	0.773	0.440				
0.013	0.001	0.604	0.213	0.007	0.019	2.163	0.397	0.007	0.001	1.356	0.253				
	2.286	8.000	6.000	5.143	6.714	7.143		3.857	7.857	7.714					

Figure 21.5: Performance with $n = 10, 15000$ times each

Based on the results, simplicity, computational efficiency, and symmetric error interpretation:

- Mixed seems to be a good default choice, particularly for very small or large n . Respectively, likely the normal approximation is accurate enough, or f computations are demanding enough to force small b .
- Should invest into the second-order accuracy of capped bootstrap- t or double percentile. If needed can trim mixed and capped bootstrap- t to respect domain range constraints.
- Noncalibrated percentile-based intervals don't seem useful. But BC almost always outperforms percentile and wins in expected length among all considered intervals. The latter is perhaps because the bias factor swings into one tail and extends it less than the other one is chopped.

If enough resamples from a fat-tail distribution have the largest value, no f can do it justice. It may be tempting to "fix" normal-based intervals using a robust scale estimate such as MADN, but these are less stable despite being robust. E.g., for $\{0, 0, 0, 1, 1\}$ the mean is more meaningful than the median, and with the latter needn't have second-order accuracy. Also for a robust estimator like the sample median, the bootstrap estimate of variance will have an unbounded influence function because a small fraction of resamples can be problematic. So in some cases using MADN might make sense, but this hasn't been explored in the literature.

Second-order accuracy is only for smooth statistics (Shao & Tu 1995), but even for nonsmooth ones capped bootstrap- t and calibrated seem to improve the approximation error even without second-order accuracy. Two-sided coverage is actually second-order accurate for all intervals, though this seems to make little difference.

Sometimes f is biased, possibly substantially. The bootstrap allows to estimate the bias using $\text{bias}(f) = E[f] - \theta \approx E_B[f] - \hat{\theta}$, which is a consistent estimate. It may seem better to use $E_B[f]$ as the estimate of θ instead of $\hat{\theta}$, but the adjusted estimate will have larger variance and may not be robust to bootstrap assumptions. Sometimes it's suggested to use the bias a diagnostic—a rule of thumb is that for best results $\text{bias} \leq 0.25 \times \text{standard deviation}$ (Efron & Tibshirani 1993; Chihara & Hesterberg 2018 recommend 0.1). But if f is reasonable, it doesn't seem worth it to check this—it's harder to estimate the bias than the standard deviation. So don't even calculate it for normal-based intervals. The argument that $f(F) \approx \hat{\theta}$ loses

value for bias estimation. May want to use $f(F)$ with data replicated maybe 100 times (though b replications make sense if f is computable in $O(n)$ time). But if f is strongly biased, there is no point using it, so don't even bother to compute confidence intervals. BC and second-order intervals automatically correct for bias (for some it's the median bias), though each with a different mechanism.

Second-order accuracy of some intervals suggests that the bootstrap is better than analytical procedures, which typically have first-order accuracy, for small n (Wikipedia 2014a). But it's computationally expensive and subject to extra Monte Carlo error. Use analytical techniques where they apply, and the bootstrap is best reserved for when no good ones do. For some problems bootstrap works better than for others, and it's best to check whether its application was already studied for a particular problem.

Can extend bootstrap to deal with multisample functionals. E.g., to compute a confidence interval for the difference of medians of two samples, resample from both, and output the difference of the resample medians.

```
template<typename FUNCTION, typename DATA = double> struct MultisampleBooter
{
    Vector<Vector<DATA>> const* > const& data;
    Vector<Vector<DATA>> > resample;
    FUNCTION f;
    MultisampleBooter(FUNCTION const& theF = FUNCTION()): f(theF) {}
    void addSample(Vector<DATA> const& theData)
    {
        data.append(&theData);
        resample.append(theData);
    }
    void boot()
    {
        for(int i = 0; i < data.getSize(); ++i)
            for(int j = 0; j < data[i].getSize(); ++j)
                resample[i][j] = data[i][GlobalRNG().mod(data[i].getSize())];
    }
    double eval() const
    {
        assert(data.getSize() > 0);
        return f(resample);
    }
    MultisampleBooter cloneForNested() const
    { //for the bootstrap-t interval
        MultisampleBooter clone(f);
        for(int i = 0; i < data.getSize(); ++i)
            clone.addSample(*data[i]);
        return clone;
    }
};
```

Need multiple testing adjustment (discussed later in the chapter) if produce several confidence intervals. For one-sided intervals, compute the two-sided interval at $\alpha/2$, and use the wanted half. Might also increase b to get better resolution.

As with any Monte Carlo algorithm, sometimes repeating gives somewhat different results. The default b values are selected so that the error due to random sampling is much less than the variability in the data (Efron & Tibshirani 1993). So if can afford computationally, pick b as maybe 10 or 100 times larger to reduce sampling variability to relatively negligible. But $b > 10^6$ is excessive. As a validation heuristic for a published analysis, can rerun the analysis several times, and see if the original result is among the typical cases. This works like a hypotheses test.

21.16 When Does Bootstrap Work?

Though want to know when particular confidence intervals work, a more basic question is about when $B \rightarrow S$. And this is only meaningful asymptotically, which needn't rule out good finite-sample behavior. Even here don't have useful general conditions. The main results (Shao & Tu 1995) guarantee that bootstrap is consistent when:

- f has a Berry-Essen-like theorem (see Hjorth 1994 for an easy proof). For f = sample mean, the basic idea is that the mean, the standard deviation, and the third moment are consistently estimated by their EDF counterparts. So the Berry-Essen theorem applies to the EDF as well, and using this can

show that the t statistics of the real distribution and the EDF converge. Though the theorem is known only for the mean, any f with asymptotically normal S will be fine in practice.

- f is smooth enough. Need Frechet or Hadamard differentiability, and a minor restriction on the influence function of f . But even the sample median doesn't have such differentiability, so a black-box f certainly wouldn't. But the bootstrap usually works anyway. Perhaps the basic numerical requirement that f is Lipschitz at θ and nearby values is enough to avoid smoothness issues where a fast-enough continuous drop looks like a step function. So while smoothness allows to prove consistency for particular f , it's not useful in practice.

Another set of conditions (Davison & Hinkley 1997; originally due to Bickel & Freedman 1981) work with the set of distributions N that surround T . In particular, for a certain function G related to f , $\forall A \in N$:

- $G_{A,n}$ must converge uniformly weakly
- The mapping from A to $G_{A,n}$ must be continuous

Unfortunately, they are impossible to verify for black-box f even with good domain knowledge. So for black-box f have no useful theory at all—only know that common f are unlikely to have no issues. But examples of failure are also numerous (see Shao & Tu 1995 for particular examples; Santana 2009 has a few updates):

- When the estimated parameter is on the boundary (Andrews 2000). A typical example is the maximum—the bootstrap fails to find its standard deviation.
- When f is nondifferentiable at θ .

That $B \rightarrow S$ needn't mean that bootstrap will work. E.g., for $g(x) = x^2$, applying g to the sample mean is smooth, so $B \rightarrow S$. But with $\mu = 0$, any confidence interval will miss out the boundary. Need also that simulation will work. In general confidence interval constructions assume a bit more than consistency conditions need, such as that S has finite variance, though failures for the latter are generally also failures for the former.

21.17 Hypothesis Tests

Sometimes only want to see whether a certain meaningful value of θ , such as 0, is ruled out by data. Rule out any value \notin a confidence interval:

```
bool confIncludes(pair<double, double> const& interval, double value)
    {return interval.first <= value && value <= interval.second;}
```

E.g., given two algorithms, can check a confidence interval on the difference of some performance metric for 0, which if present doesn't rule out equivalent performance as far as the data is concerned.

Can use a different mechanism to construct a test. That θ has the wanted value is the **null hypotheses**. In useful cases it's a **point hypothesis** about a single specific value of θ , but doesn't have to be. In this case the null is called **simple**, otherwise **composite**. The **alternative hypotheses** is the complement of the null, though the union needn't be the full sample space. Typically the null is that the considered claim is false and the alternative that it's true.

Tests usually have $O(n)$ runtime. A generic hypothesis test:

1. Pick a statistic—e.g., the sample mean
2. Define its distribution (called the **null distribution**) when calculated on the sample, assuming the null is true—e.g., normal with unknown variance, centered at θ
3. Calculate the **p-value** = $\Pr(\text{can have a same or more in-the-tail value than the sample statistic})$ —e.g., the tail position of the z-score
4. If it's small enough, reject the null, and assume the alternative is true—e.g., check whether the **p-value** $\leq \alpha = 0.05$
5. Else not enough evidence to reject the null, so assume the alternative isn't proven

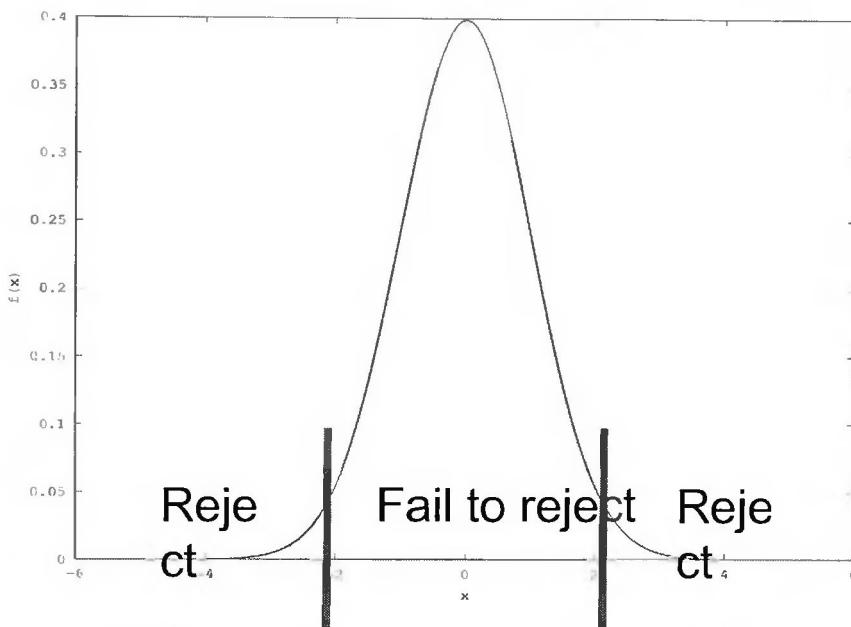


Figure 21.6: Hypothesis-test decision regions for the null

Another way to see a null distribution is based on grouping the data. E.g., if a population is split in half based on a coin toss, this will cause some random difference in statistics of the subpopulations. This difference is noise unless the split is based on some variable. Given a null distribution of the noise, a test tells whether the difference is more than expected due to noise. This is similar to arguing by contradiction—if the null poorly explains the data, it can't be true so the alternative must be. In terms of the language “accept claim” = “reject null” = “value ruled out by data”.

So far testing using confidence interval membership checking and hypotheses tests is operationally the same. A test that isn't derived from a confidence interval can usually be converted to one. Shift the data or its transformation by a value such that this is equivalent to shifting the parameter. I.e., reverse the logic of making tests out of confidence intervals. E.g., the confidence interval for quantile estimates was derived by inverting the sign test. But this doesn't always work—e.g., don't have a good way to invert the permutation test for a correlation coefficient (discussed later in the chapter).

A two-sided test actually does a three-way comparison. The hypothesis tested is that of inequality, but if the statistic is too extreme, know if it's too large or too small. A two-sided test is equivalent to two one-sided tests at $\alpha/2$ (which makes sense with Bonferroni correction, discussed later in the chapter). So given 100% interval for the p -value, and $\alpha = 5\%$, 95% goes to equality, 2.5% to one tail, and 2.5% to the other. An asymmetric allocation to the tails is also possible. If a statistic lands into either tail, reject the null with 95%, and conclude in case of comparison that one alternative has statistically superior value at 5% level.

Similar to maximum likelihood for constructing estimates, have **likelihood ratio tests** (Wasserman 2004): $2(LL(\text{MLE}(\text{model}, \text{data})) - LL(\text{MLE}(\text{model}, \text{data}, \text{null region}))) \sim \text{chi-squared}(d)$, where $d = \text{dimension}(\text{full parameter set}) - \text{dimension}(\text{parameter set when restricted to the null region})$. E.g., $d = 1$ for testing whether $\theta = 0$. Need regularity conditions for the logic to work in all cases (DasGupta 2008), and the one-sided case isn't clear. See also Self & Liang (1987) for some extensions.

E.g., the t -test, inverting which gives the corresponding confidence interval for the mean, can be shown to use the likelihood ratio statistic, but with exact and not the asymptotic distribution (Westfall & Henning 2013). Beware that like maximum likelihood estimates, likelihood ratio tests can perform poorly despite nice asymptotic properties (Kiefer 1987).

21.18 Comparing Tests

Despite the mathematical equivalence with confidence intervals, tests are somewhat different and come with different terminology. The same reasoning as for confidence intervals suggests $\alpha = 0.05$. Smaller values make sense in some case, but beware than p -values are highly sensitive to minor assumption deviations when very small due to typically large approximation errors in the tails.

A test makes mistakes over a sequence of decisions:

- **Type I**—falsely rejecting the null—e.g., jailing the innocent. Same for confidence intervals, though not called that way.
- **Type II**—falsely failing to reject the null—e.g., freeing the guilty. Related to how short the inverted confidence interval is.

Type I error is more important because from scientific point of view don't want to accept many false claims even if to do so must judge many others as not significant. So tests try to control it explicitly, i.e., if reject based on whether $p\text{-value} \leq \alpha$, decide wrong in $\leq \alpha$ percent of tests. Don't want to use a test that doesn't, at least enforce it asymptotically or under some conditions. Type II error can't be controlled and depends on the alternative and n . The null is selected to be more expensive to reject than the alternative as a scientific requirement. A test works the same way if these were to be swapped by the user.

Both errors are related through the **power of a test** = $\Pr(\text{decide to reject given } \theta)$:

- Control of type I error means $\max_{\theta \in \text{null}} \text{power}(\theta) \leq \alpha$
- Type II error = $1 - \min_{\theta \in \text{alternative}} \text{power}(\theta)$. In the worst case this $\rightarrow \alpha$, so at best get to control type II error when have some indifference zone close to the null. Picking n needed for a specific error bound is power analysis (discussed later in the chapter).

Given two tests that control type I error, want a more powerful one over the alternative set. This can be considered as a risk function using 0-1 loss for falsely failing to reject the null, so almost get the same comparison theory as for estimation. An admissibility relationship is formed when one test is **uniformly more powerful**, and in special cases (mostly for 1-sided tests) get **uniformly most powerful (UMP)** tests. These are defined with respect to particular null and alternative. Only consider power over the alternative for comparison—the null power must only stay $\leq \alpha$. So already for a single θ don't have a single aggregate risk number.

One useful device in showing UMP satisfaction is that likelihood ratio tests have it when the ratio is monotone. This works mostly for one-sided tests (and equivalence tests, discussed in the comments; Shao 2003).

Tests and intervals formed from one another generality inherit whatever optimality properties the original one had, such as being most powerful (Kiefer 1987). E.g., a test where the null is simple that is also UMP $\forall \theta$ leads to a UMA interval (Shao 2003). This holds only for nonrandomized tests, though the inversion works regardless (but loses continuity with randomization, unless use common random numbers for the inversion, as for permutation tests, as discussed later in the chapter).

Don't have a concept of minimax for tests unlike for estimators because power is usually very small closest to the null (but see Lehmann & Romano 2005). So power analysis only works with a representative point given by the indifference region. A Bayesian prior that puts a thin tail on the indifference region also makes sense, but isn't particularly useful. But with a specific indifference alternative can consider power on it as risk, and compare as for estimators.

Because UMP tests are rare, consider the common restrictions as for estimators, though they only enlarge the set of special cases:

- Unbiased tests—here (at least for a simple null) the bias = the lowest-power point $- \theta$. Try to enforce that the true value is rejected less often than a false one, but a biased test can have higher power on both. Don't have a corresponding variance concept for both tests and confidence intervals to allow to form the MSE.
- Invariant tests—with respect to one-to-one transformations—e.g., translation and scaling. **UMPU (UMP unbiased)** and **UMPI (UMP invariant)** tests are usually the same (Shao 2003).

Can compute test ARE in several ways, and the most common one essentially reduces to that of estimators. I.e., against a specific alternative consider $\frac{n(\text{test1, alternative, power target})}{n(\text{test2, alternative, power target})}$ as $\theta \rightarrow \theta_{\text{null}}$ (DasGupta 2008). This reduces to comparing the asymptotic variances of the underlying estimators.

21.19 Using Tests Effectively

Given that confidence intervals, when reported, always carry strictly more information compared to test results, are tests useful for anything other than confidence interval construction by inversion? When the only purpose of a confidence interval is testing, pure tests have some advantages in general:

- In some cases the statistic on the inverted confidence interval isn't meaningful, so the test result is more interpretable. E.g., if want to compare two algorithms on unrelated inputs to decide which one is better, something like the percentage of better performance isn't meaningful. Need some overall grade, which as θ must exist for a test result to make sense. But in general don't have enough information to scale the grades on each task.
- The computation can be more efficient—e.g., the sign test uses $O(n)$ time and $O(1)$ extra space and is online, while the sample median calculation is less efficient and not online.
- The p -value can be reported as the result and compared against α when needed. E.g., for important decisions might require a smaller or a larger α than for less important ones, and some domains

might have established a . This might make the implementation easier, though in some cases, such as with Nemenyi's test (discussed later in the chapter), it's more convenient and computationally efficient to pass a as a parameter. E.g., can calculate the critical value of a statistic, such as $z\text{-score} = 2$, and pass that to avoid evaluating the distribution.

- When many results are reported, multiplicity adjustment (discussed later in the chapter) works somewhat better for tests than for confidence intervals, leading to higher power.

Still, in most cases it's best to report confidence intervals. Among the above, only committing to a is a potential problem. But for a different a can usually recalculate the reported interval. Given that a are usually standard, and good reports contain information needed to replicate the findings, this rarely an issue.

A confidence interval doesn't give a p -value. But for interpretation purposes, if can calculate confidence \forall given a , can use binary search to put the null value on the border, and the p -value = the corresponding a . Beware that the result is meaningless though.

A typical use case for tests is deciding whether a new alternative is statistically better than the established one. This serves as a check against introduction of useless alternatives by interested parties. But this approach is far from flawless. Intuitively, testing seems unsatisfactory because it decides what θ isn't, whereas the goal is to discover it. In fact, a basic scientific question is whether testing is useful at all:

- A rejection of the null that rules out the 0 difference doesn't rule out an ϵ difference. E.g., the original sign test was used to show that the male/female birth proportion isn't 50/50 at any reasonable a due to $p\text{-value} = 2^{-82}$, but the proportion is now estimated to be $\approx 53/47$.
- Can fail to reject because didn't look at enough data to have enough power.

With confidence intervals neither happens:

- Lack of membership in an interval still comes with the interval, which tells where θ should be—far or near. A confidence interval and an estimate give some idea of how close θ is to the wanted value, and often deem a close-enough wanted value as accurate enough despite not being contained in an interval.
- Membership comes with an interval length, so can readily see whether the interval is too long to have come from enough data. An interval tells the complete story without needing a specific null value to test against, and a short interval gives practically complete information about θ .

But it's wrong to rule out tests as useless in all cases, as is done by some sources. Significance matters when supplemented by external information. So tests are useful with extra domain knowledge. In particular, for some domains, such as government approvals and legal cases, any difference is meaningful, and for some others θ can be hard to estimate. E.g., consider whether drinking coffee all day makes you feel worse than alternating coffee and water. Get the data by pairing coffee and mixed days. A test result of "this feel better on average" is reasonably useful, but it's hard to establish how much better.

Some examples where tests are useful (the specific tests are discussed later in the chapter):

- For proposal of new drugs and experimentally validated scientific idea the new object must be significantly better than the established one in an important way—this serves as a basic interview for promotion to further study and consideration though can be often answered using confidence intervals though. Also courts sentence the guilty party at a later trial, and drug companies do follow-up studies to decide whether it's profitable to invest in a new "significantly better" drug.
- Distribution-match tests are good for unit tests of random number generators. I.e., do many tests and see if the failure fraction is close to a (and can make judgment on the fraction later for approximate tests and generators).
- The sign test is used in regression tree construction.
- Friedman ranks are a good way to compare algorithms on many inputs, even if the associated significance tests aren't considered.

An approximate confidence interval is usually better than a less approximate test. E.g., don't use symmetry-based rank tests such as **Wilcoxon signed rank** and **Wilcoxon-Mann-Whitney** (see Gibbons & Chakraborti 2020 for details) because the sample trimmed mean, which also benefits from symmetry, leads to both an estimate and a confidence interval. (Can compute a confidence interval by inverting a rank test, but the corresponding **Hodges-Lehmann estimate** needs impractical $O(n^2)$ time for the obvious implementation. See also StackExchange (2016b) for some comparisons with the same-breakdown 29% sample trimmed mean). Rank tests are also sensitive to asymmetry (Wilcox 2016).

Often attempts are made to attach meaning to a p -value, usually wrongly. p -value = the minimum a such that the null is rejected. Interpretationally, a p -value is a computational artifact, only to be compared eventually against some a . The extreme value property is a computational definition—the statistic is random, and a p -value simply a rank of its extremeness, but this isn't useful interpretationally. Doing anything other than comparing against a is a mistake.

When a p -value is calculated, eventually use it to accept or reject a null. It doesn't matter how this decision is made. Many sources say that must pick the α before seeing the p -value, but this is false—only prevent interested parties from rejecting at 0.06 when the domain uses 0.05. Given the decision it doesn't matter what α was used. So it's fine to reject if $p < 0.001$, fail to reject no matter what if $p > 0.2$, and think about other factors to set α otherwise. Also can assign α buckets. E.g., for $0.05 \leq p < 0.001$ give normal funding for a validation study, and for $0.001 \leq p$ give extended funding, etc. This multiway decision is perhaps the closest correct way to consider a p -value as some “strength of evidence”, which it technically isn't. With confidence intervals can also compute several, and use the one appropriate for the logic such as the funding allocation above. E.g., given one's history of rejections, can compute a “personal effective α ” = max rejected p -value. But this is frequentism over the wrong sequence. If infinitely more tests are done, this will increase to match the actual used α .

A mistake is thinking of a p -value as a probability that the null is true. In frequentist statistics this is impossible—the estimation mistake is only with respect to the particular sample. Need accurate prior information and Bayesian statistics (discussed later in the chapter) to find out this probability. The p -value is only a part of the posterior, and at best tells relative rankings—for two randomly selected hypotheses which one is more likely. A small p -value gives a large set of α that lead to rejection of the null, so it's might be interpretationally useful based on the assumption that many such α are used—i.e., a very low p -value will satisfy most decision makers. Perhaps any intuitive happiness about low p -values only makes sense from this point of view. But neither idea seems to be operationally useful. A good practice is to reject at a specific α but report the p -value in case the reader would want to use another α .

Another mistake is thinking of a p -value as a probability of making a mistake. E.g., consider a hypothetical situation where test 20 true and 80 false claims (these are really percentages) at 0.05, and the test has 80% power. Then accept ≈ 16 true claims and 4 false claims. Here 25% of accepted claims are wrong, but it can be less or more depending on the distribution of the set of nulls that are tested. In psychology it has been claimed that >50% of published hypotheses accepted at 0.05 are false. So most tested hypotheses are false. More concerning is the possibility that most criminals are never caught, and a large fraction of the population is innocent, assuming the “beyond reasonable doubt” is equivalent to $\alpha = 0.05$.

It's tempting to think of something like $p = 10^{-6}$ or the one from the male/female ratio as definitely true. But need further context:

- “Impossible” events such as repeated lottery wins happen occasionally because of multiple testing (discussed later in the chapter; see Hand (2014) for many examples)
- Among millions of patterns it's hard to distinguish random and causal ones—even machine learning algorithms can't do that because need information other than the data
- Among law suits based on statistical patterns, an “infringement” such as hiring discrimination will appear by chance if decide randomly

Many tests make assumptions which are hard to verify and so ignored in practice. Then many properties such as type I error control are invalidated. As with confidence intervals, usually assume that the model of the test has some approximation error, but the conclusions are still close enough to be useful. So strict control of type I error is less important than it seems, though still wanted. In particular, different calculation methods can result in different p -values, so be careful about which assumptions to trust.

Beware that significance determination isn't a final result. In a court of law it's admissible evidence to be considered with other factors. Automatic final reliance on a significance level makes sense only when agreed upon before the test is done—e.g., for acceptance of new drugs by FDA. For scientific inference a low p -value only suggests consideration for further investigation. A correct way to see a test is the same way as an exam, e.g., for a driver's license—those that pass aren't necessarily good drivers, and those who don't aren't necessarily bad ones, but the former are reasonably deemed worthy of a license, and the latter aren't.

There is a certain desire for “exact procedures”, i.e., those that guarantee requested α and, in case of tests, have maximum power. But get them in only few cases. What procedure is exact isn't intuitively clear—the theoretical confidence interval construction that knows θ and S is. E.g., consider x of variates from the normal distribution. The t -test is exact, but the confidence interval based on it isn't because it involves σ , which is estimated by s , and $s^2 \sim \chi^2$ with some degrees of freedom. So for fixed x , s values jump around. So confidence intervals computed from different samples have different lengths, though the theoretical construction has a specific length. But for the test the t -statistic has a distribution that doesn't depend on σ . Same for the confidence interval for the median—once the empirical distribution is inverted by the quantile lookup, the found values remove any exactness. So in general, assumption-free procedures aren't exact, and inverting exact tests doesn't lead to exact confidence intervals.

An interesting dilemma occurs when several studies give different p -values. Some solutions:

- To use the smallest one need to adjust for multiple testing (discussed later in the chapter).

- An unadjusted “primary” p -value is uniformly distributed, so the smallest one of k has a beta distribution (see the “Random Number Generation” chapter). So instead of multiplicity adjustment, can calculate this distribution and invert it to get the adjusted p -value.
- Pick an arbitrary study, either at random, or using some data-free information such as whichever has the most trustworthy design or the largest sample size.
- If possible, instead of the all of the above, combine the data, and do a single test. This avoids low-power issues such as having several bad studies that fail to reject the null instead of a single one whose statistic has a narrow enough confidence interval to reject. Combination methods are more generally covered by **meta-analysis** (discussed in the comments).
- Treat the studies like interviews—whoever passed one round is advanced to another one for further testing. This allows new studies to ignore previous studies, which is the point of new studies. But it’s unclear how to treat studies with some significant p -values and some not—this depends on other factors. With confidence intervals and estimates this isn’t much of an issue.

Intuitive understanding of tests is error-prone because the human mind seeks simpler explanations of what is happening. Some mistakes not yet discussed:

- Many textbooks describe comparison or value tests as checking whether $\theta = 0$, which for natural experiments is almost surely false due to being different in some decimal place. But test whether $\theta = 0$ is ruled out by data, as with a confidence interval. The point of a test is to decide this for the data available at the moment.
- Any cost/profit assignment based on p -value or even a makes no sense. Profit from accepted true claims, lose a lot from accepted false claims, lose a little from failing to accept a true claim, and have no loss or profit for failing to reject a false claim. Assuming that all accepted claims are false and all failed-to-reject ones are true gives a valid but useless bound. Using the exact type I and type II error doesn’t help improve this post-decision. Can only get a better estimate using Bayesian logic if have a good prior. Otherwise at best can study these costs, and set appropriate a and power goals for a specific domain.
- “Tests with low power are common, and result in too many false nulls not being rejected”—true, but this isn’t a fault of tests. If a test is based on a small sample and doesn’t reject the no-difference null, the experiment itself was poorly done. It would equivalently produce a wide confidence interval. The common advice to make the estimated power ≥ 0.8 is only a guideline for a better experiment. Though opportunistic testing based on ad-hoc-available data is useful, should precede it by some power analysis.

For some other common mistakes see Greenland et al. (2016).

21.20 Validation of Studies

Any statistical result is only probably correct. At best can trust the procedure that produced the result, aware that it can be wrong this time around. Estimates, confidence interval length, and p -values all have distributions whose variance decreases with n but rarely becomes negligibly small.

E.g., for the CLT-based confidence intervals for the mean, the length is a function of s^2 , which under the normal model $\sim \text{chi-squared}$. So over many intervals the length will occasionally be very large. For a fat-tail distribution this is even more of a problem. The asymptotic rate also has a constant that is subject to some distribution, and rare order-of-magnitude swings are possible. Even for bounded distributions where Hoeffding inequality gives nuisance-parameter-free intervals that always have the same length, the primary statistic leads to some variation. Yet none of this prevents trusting the result of a reasonable procedure. E.g., Monte Carlo integrals are assumed to be accurate up to CLT limits, though occasionally they won’t be.

Bayesian logic (discussed later in the chapter) also doesn’t show correctness of a particular result, only a what-if-like belief in such result based on some initial assumptions and the data.

Still, the result is considered statistical evidence and never fully confirmed, except by external information. Even the U.S. Supreme Court ruled that statistical evidence (such as a fingerprint match) has no meaning by itself (Greenland et al. 2016), and other factors matter. E.g., if someone flipped a coin to make it heads 20 times in a row, it’s far more likely that they have special training or the coin is biased than that they have some supernatural control over the outcome.

For any published result have publication bias—only significant p -values and short confidence intervals are generally accepted, so the post-publication a is higher due to preselection, as previously explained for p -values.

A somewhat arbitrary judgment is eventually made about whether something is confirmed, disproved, or remains in question. Based on statistics alone, several convincing studies are usually needed to claim

truth so that any experiment and experimenter factors are removed. And usually want external data to support this. Disproved claims are usually those that fail to be confirmed in more trustworthy subsequent studies. So scientific theories based on experiments are considered established facts when they fail to be refuted long enough, but mistakes happen and even long-standing theories can be refuted eventually or made more specific. E.g., even Newtonian mechanics were discovered to be only an approximation and not the truth, given relativity.

Also, once a good study is done, won't get funding for further studies on the same subject. In Bayesian terms, such study acts as prior information.

Mathematics is valid unconditionally but usually not applicable to reality unless the assumptions hold exactly, which is never technically true. So experiments are needed to make sure that the conclusions are only slightly affected by assumption deviations in the domain.

The legal evidence criteria that a study must be done by a reasonable method to be considered (see the "Computer Law" chapter) is perhaps the best intuitive justification of approximate procedures. I.e., if another expert would have done it, they would have used a similar-quality method. So get enough credibility to present the result to a decision maker. For much further discussion see Beisbart & Saam (2018).

21.21 Comparing Matched Pairs

A **matched pairs experiment** is when have paired observations (e.g., the performances of two algorithms on a set of benchmark tasks). Want to discover if the better one, if any, is significantly so. Matched pair/tuple tests are convenient for computer simulations where an algorithm is run with different parameters.

The **sign test** assumes that the observations come from a continuous distribution. But continuity is a weak assumption because can make a discrete distribution continuous by adding a bit of PDF to make connections. So effectively make the **general position assumption** that observations don't follow any exact pattern. Conceptually, adding a bit of noise to each observation will satisfy this in all cases, so the sign test is completely distribution-free.

The idea is that given equal performance on n "games", the distribution of who "won" is binomial($n/2, n$), and a significant deviation rejects this. Count ties as draws, and for the discreteness of the binomial with an odd number of ties must drop one tied pair. But for the approximation $\text{binomial}(n/2, n) \approx \text{normal}(n/2, n/4)$ draws don't cause issues. Simplifying the algebra, $z\text{-score} = |\text{win difference}|/n$.

```
bool signTestAreEqual(double winCount1, double winCount2, double z = 2)
    {return abs(winCount1 - winCount2)/sqrt(winCount1 + winCount2) <= z;}
```

Beware that the discretization in converting to normal can make a difference, e.g., the computed z can be exactly 2, which may have a substantial effect on the percentage of true null rejections. With particular a the procedure will only have the correct type I error guarantee for a slightly larger a , but this approximate test is useful in the same way because the arbitrariness of the approximation is subsumed into that of the a choice. For a given data need to count the wins:

```
pair<double, double> countWins(Vector<pair<double, double>> const& data)
{
    int n1 = 0, n2 = 0;
    for(int i = 0; i < data.getSize(); ++i)
    {
        if(data[i].first < data[i].second) ++n1;
        else if(data[i].first > data[i].second) ++n2;
        else{n1 += 0.5; n2 += 0.5;}
    }
    return make_pair(n1, n2);
}
bool signTestPairs(Vector<pair<double, double>> const& data, double z = 2)
{
    pair<double, double> wins = countWins(data);
    return signTestAreEqual(wins.first, wins.second, z);
```

The sign test's normal approximation is asymptotic and inaccurate for very small n . Here should use binomial inversion with dropping of an odd tie.

The sign test is as far as can go with making as few assumptions as possible. Generally can't do statistical inference without making assumptions, and it's better to use existing assumption-based methods while recognizing their limitations than to have no method at all to approach a problem. ARE to the t -test is the same as that of the median to the mean.

If don't have symmetry, the sign test considers the median of the differences, which \neq the mean. So given two distributions with the same mean, one skewed left, and the other right, with $n \rightarrow \infty$ the sign test will report a significant difference in the medians. So the idea that for testing difference of means it assumes strictly less than normality and is therefore most general is deceptive. But this doesn't matter because should instead find a confidence interval for such difference.

In general, as for estimators, ! \exists a universally good test \forall problem, even a simple one such as comparing paired alternatives. Need to consider domain properties of the data, and pick tests accordingly. As a general technique, compare a method to a copy of itself to see possible deviations in any test-only metrics—this helps to train yourself to not pay too much attention to noise, which the human mind to too eager to interpret as a meaningful pattern.

21.22 Multiple Comparisons

Working with several estimates is tricky because of **multiple testing/multiple comparison**. Typically a conclusion based on rejecting the null if $p\text{-value} < 0.05$ is accepted. But a conclusion based on 100 studies, where for each $p\text{-value} < 0.05$, can't be accepted at $\alpha = 0.05$. E.g., consider:

- Simulating 100 same-size normal samples—the smallest-mean sample will underestimate the true mean, and the largest-mean one will overestimate, just by chance. And their confidence intervals won't have the promised coverage.
- Sending buy/sell investment advice to 1024 people for 10 days, so that after each day the wrong-prediction half is dropped. In the end, to a single person it will seem that the predictor is always right. Here 10/10 isn't the right estimate of performance.

The overall error rate is **family-wise error rate (FWER)**; it applies to both confidence intervals and p -values. A simple general method for making corrections for multiple testing is provided by the **Bonferroni inequality** (also called **Bonferroni correction, union bound, or Boole inequality**, in various versions such as subtracting both sides from 1 or using De Morgan laws):

- $\Pr(\text{event}_0 \cup \dots \cup \text{event}_{k-1}) \leq \sum \Pr(\text{event}_i)$.
- $\Pr(\text{event}_0 \cap \dots \cap \text{event}_{k-1}) \geq 1 - \sum(1 - \Pr(\text{event}_i))$.

In particular (Westfall et al. 2011), for k events:

- Confidence intervals—to have $1 - \alpha$ overall confidence, each needs to have $1 - \alpha/k$ confidence. E.g., for the normal distribution find the corresponding z :

```
double find2SidedConfZBonf(int k, double conf = 0.95)
{
    assert(k > 0);
    return find2SidedConfZ(1 - (1 - conf) / k);
}
```

- p -values—adjust p_i to kp_i .

Beware that the normal model is an approximation, and its quality is usually worst in the tails, which are very flat. Multiplicity adjustment for it makes little difference to the results, increasing sensitivity to the choice of the normal model. But this still works well in practice, perhaps because multiplicity correction is conservative.

For p -values can do better. If fail to reject some null with limited correction, need less correction for others. In particular, the **closed testing principle** leads to **Holm procedure** (Westfall et al. 2011):

1. Sort the p -values
2. Adjusted p -value_i = min{1, max{adjusted p -value_{i-1}, p -value_i × (k - i)}}

```
void HolmAdjust(Vector<double>& pValues)
{
    int k = pValues.getSize();
    Vector<int> indices(k);
    for(int i = 0; i < k; ++i) indices[i] = i;
    IndexComparator<double> c(pValues.getArray());
    quickSort(indices.getArray(), 0, k - 1, c);
    for(int i = 0; i < k; ++i) pValues[indices[i]] = min(i, max(i > 0 ?
        pValues[indices[i - 1]] : 0, (k - i) * pValues[indices[i]]));
}
```

It's strictly less conservative than Bonferroni.

But FWER control is too conservative with large k . In preliminary studies can dramatically cut down false rejections by instead controlling the **false discovery rate (FDR)**—the proportion of true nulls that are rejected. Then can use independent data for subsequent validation. Can even split the data in half and use

two rounds—the first with FDR, and the second with FWER.

Benjamini-Hochberg procedure controls FDR and is similar to Holm's (Westfall et al. 2011):

1. Sort the p -values
2. For i in descending order, adjusted p -value _{i} = min(adjusted p -value _{$i+1$} , p -value _{i} × $k/(i + 1)$)

```
void FDRAdjust(Vector<double>& pValues)
{
    int k = pValues.getSize();
    Vector<int> indices(k);
    for(int i = 0; i < k; ++i) indices[i] = i;
    IndexComparator<double> c(pValues.getArray());
    quickSort(indices.getArray(), 0, k - 1, c);
    for(int i = k - 1; i >= 0; --i) pValues[indices[i]] = min(i < k - 1 ?
        pValues[indices[i + 1]] : 1, pValues[indices[i]] * k/(i + 1));
}
```

This assumes no negative dependence between the hypotheses, which can't be verified in practice, but isn't a problem because FDR is only useful for preliminary analysis.

21.23 Comparing Matched Tuples

For $k > 2$ alternatives usually want to find out which pairs of alternatives are different. A simple solution is to use several sign tests, with adjustments for multiplicity. But **Nemenyi test** is more powerful. It uses **Friedman ranks**—generalization of the sign test wins. First convert observations to ranks in each domain i . Ranks are unique with a continuous distribution, but in practice this isn't always true. Handle ties from continuity violation by averaging tied ranks. Calculate $\forall 0 \leq j < k, r_j = \sum_i r_{ij}$ = rank sum for domain j .

```
Vector<double> FriedmanRankSums(Vector<Vector<double> > const& a)
{//a[i] is vector of responses on domain i
    assert(a.getSize() > 0 && a[0].getSize() > 1);
    int n = a.getSize(), k = a[0].getSize();
    Vector<double> alternativeRankSums(k);
    for(int i = 0; i < n; ++i)
    {
        assert(a[i].getSize() == k);
        Vector<double> ri = convertToRanks(a[i]);
        for(int j = 0; j < k; ++j) alternativeRankSums[j] += ri[j];
    }
    return alternativeRankSums;
}
```

Asymptotically, \forall alternatives i and j , $\frac{|r_i - r_j|}{\sqrt{nk(k+1)/6}} \sim \text{normal}$ (Gibbons & Chakraborti 2020; convince

yourself that this matches the sign test for $k = 2$ (some sources erroneously divide by 12 instead of 6). Note that use the analytical pooled variance instead of the estimated one. A minor assumption is that the observations come from a continuous distribution (Gibbons & Chakraborti 2020), but a violation of it only leads to ties. The variance calculation is for nontied observations, but ties can only reduce variance and so don't increase the type I error.

```
double NemenyiAllPairsPValueUnadjusted(double r1, double r2, int n, int k)
{
    return 1 - approxNormalCDF(abs(r1 - r2)/sqrt(n * k * (k + 1)/6.0));
}
```

A different way to use this procedure is to calculate the significant average rank difference needed for 95% acceptance. Though don't get p -values, this works for any pair with pairwise adjustment. To increase power, sometimes can reduce the number of preplanned comparisons in advance. A typical case is comparing all alternatives with a control alternative. Then adjust only for k hypotheses, assuming all vs control.

```
double findNemenyiSignificantAveRankDiff(int k, int n, bool forControl = false,
    double conf = 0.95)
{//Invert rank sum formula
    int nPairs = k * (k + 1)/2;
    double q = sqrt(nPairs/(3.0 * n)),
    z = find2SidedConfZBonf(forControl ? k : nPairs, conf);
    return q * z/n;//for rank average, not sum
}
```

Holm procedure needs computing all $\frac{k(k-1)}{2}$ *p*-values. The resulting matrix has dummy 1's on the diagonal (to signal nonsignificance).

```
Matrix<double> RankTestAllPairs(Vector<Vector<double>> const& a,
    double NemenyiAlevel = 0.05, bool useFDR = false)
{
    Vector<double> rankSums = FriedmanRankSums(a);
    int n = a.getSize(), k = rankSums.getSize();
    Vector<double> temp(k * (k - 1) / 2);
    for(int i = 1, index = 0; i < k; ++i) for(int j = 0; j < i; ++j)
        temp[index++] = NemenyiAllPairsPValueUnadjusted(rankSums[i],
            rankSums[j], n, k);
    if(useFDR) FDRAdjust(temp);
    else HolmAdjust(temp);
    Matrix<double> result = Matrix<double>::identity(k);
    for(int i = 1, index = 0; i < k; ++i) for(int j = 0; j < i; ++j)
        result(i, j) = result(j, i) = temp[index++];
    return result;
}
```

The control comparison isn't implemented here, and the user needs to do it manually using the unadjusted *p*-values.

Often only want to find the best option or the best subset—just compare the lowest-rank alternative with all others using the adjusted pairwise *p*-values. Somewhat more efficient but more complicated are multiple comparison with the best procedures (Westfall et al. 2011).

In the ideal case would get confidence intervals on individual average ranks. Unfortunately variance estimates are only available for the differences. Instead of comparing matched observations it may make sense to ignore the matching and compute the percentages of satisfactory performances. These can be analyzed as several independent samples to get confidence intervals because Bonferroni correction allows dependence. But what defines satisfactory performance is domain-specific.

21.24 Comparing Independent Samples

A typical nonparametric goal is to estimate confidence intervals for the differences of two quantiles at a time. For $k > 2$ a simple correct solution is to compute Bonferroni-adjusted confidence intervals at a/k and form $\text{diff}_{i,j} = (l_i - u_j, u_i - l_j)$. This needs no further multiplicity adjustment.

Even for $k = 2$ this is very inefficient, and want to form the difference directly. But even though the normal approximations of each are additive, don't have a way to retrieve the resulting difference quantiles. One heuristic solution is to assume that the confidence intervals are derived from a normal, calculate the implied standard error, and form the normal difference. E.g., for the median get:

```
pair<double, double> normal2SampleDiff(double mean1, double stel,
    double mean2, double ste2, double z)
{//difference of approximately normal-based confidence intervals
    NormalSummary n1(mean1, stel * stel), n2(mean2, ste2 * ste2),
    diff = n1 - n2;
    double ste = diff.stddev() * z;
    return make_pair(diff.mean - ste, diff.mean + ste);
}
pair<double, double> normalConfDiff(double mean1, pair<double, double> const<double, double> conf1,
    double mean2, pair<double, double> const<double, double> conf2, double z)
{//difference of approximately normal-based confidence intervals
    return normal2SampleDiff(mean1, (conf1.second - conf1.first)/2/z,
        mean2, (conf2.second - conf2.first)/2/z, z);
}
pair<double, double> median2SampleDiffConf(Vector<double> const& samples1,
    Vector<double> const& samples2, double z = 2)
{
    return normalConfDiff(median(samples1), quantileConf(samples1, 0.5, false,
        z), median(samples2), quantileConf(samples2, 0.5, false, z), z);
}
```

The approximation is reasonable even if both distributions are skewed in the same direction. It will lose accuracy only when one is skewed left, and the other one right. But asymptotically this is a correct test. A

more standard procedure is to estimate the variance of the median, and use it for a combined confidence interval (Wilcox 2016). But this is more difficult than estimating a confidence interval directly—e.g., the picked a matters. So the above procedure is simple and effective.

Perhaps the most efficient general difference estimator is difference between trimmed means. Though for asymmetric distributions get pseudo-medians, for interpretation the difference between any nearby location measures is as good as any other.

```
pair<double, double> trimmedMean2SampleDiffConf(Vector<double> const&
    samples1, Vector<double> const& samples2, double z = 2)
{
    return normal2SampleDiff(trimmedMean(samples1),
        trimmedMeanStandardError(samples1), trimmedMean(samples2),
        trimmedMeanStandardError(samples2), z);
}
```

For the difference of means and large n , the difference of the CLT normals is safe to use. For smaller n the t -distribution no longer applies directly because the difference isn't a t -distribution. As for medians, a simple approximation is that the individual sampling distributions are normal with the standard errors adjusted by $\frac{t_{\text{value}}}{z_{\text{value}}}$.

For these estimators the two-sample bootstrap works also.

21.25 Permutation Tests

Permutation (or **randomization**) tests are usually the most assumption-free where they apply, though computationally expensive. Suppose compare two groups for equality and have iid, unpaired observations. The null here is that the **compared distributions are equal**, whereas the usual tests assume equality of a particular statistic. This leads to **exchangeability**—e.g., given two treatments, under the null the observations from one group might as well have appeared in the other. Since a particular observation appeared where it did, this could be by chance or because the null is false. To test, pick a statistic, and check its value \forall assignment of observations to groups (Howell 2016). All possible assignments form a **permutation group** that defines the null. This abstract formulation is useful theoretically—see Lehmann & Romano (2005) for more details.

The statistic can be anything, and usually pick the mean or whatever makes most sense for the problem. An important computational trick, as for bootstrap, is sampling some number b of all possible assignments, maybe 10000 (to get a reasonable estimated p -value precision (Chihara & Hesterberg 2018 suggest larger numbers).

1. Identify exchangeability, and choose a statistic f
2. b times
3. Generate a random assignment (“permutation”) within the exchangeability
4. Calculate $f_i = f(\text{assignment})$
5. Count the number of $f_i \geq$ or \leq (or both for two-sided tests) than the original one
6. From the counts calculate the lower-bound p -value

The counts tell how extreme $f(\text{sample})$ is as a low or a high value. Because two-sided testing is equivalent to using one-sided testing with $a/2$, and don't know whether look for smaller or larger, compute both p -values, and double the smaller one. As a minor trick, add 1 to the counts to avoid:

- Deceptive 0 p -value
- Numerical issues for computing confidence intervals by numerical inversion (discussed later)

```
template<typename PERM_TESTER> double permutationTest(PERM_TESTER& p,
    int b = 10000, int side = 0, Random<>& rng = GlobalRNG())
{//"more extreme" means larger for 1-sided, side -1 for smaller, 1 for larger
    assert(b > 1 && abs(side) <= 1);
    double f = p();
    int nLeft = 1, nRight = 1;//start with 1 not 0
    for(int i = 0; i < b; ++i)
    {
        p.exchange(rng);
        double fr = p();
        nLeft += (f <= fr);
        nRight += (f >= fr);
    }
}
```

```

    double leftP = nLeft * 1.0/(b + 1), rightP = nRight * 1.0/(b + 1);
    return side == 0 ? min(1.0, 2 * min(leftP, rightP)) :
        side == -1 ? leftP : rightP;
}

```

Identifying exchangeability is sometimes tricky. Not everything is exchangeable—e.g., given patients and treatments, treatments are exchangeable, but patients aren't. In case of such paired observations, previously handled by the sign test, all assignments are generated by considering treatment swaps for a patient. Then, over all possible swaps calculate the distribution of differences in means, and check whether the difference on the sample is too large. This also works as is for a single distribution that is symmetric around 0—flip the sign of the data points to form a permutation group.

```

template<typename LOCATION_F> struct PairedTestLocationPermuter
{
    Vector<double> diffs;
    LOCATION_F f;
    PairedTestLocationPermuter(Vector<double> const& data) : diffs(data) {}
    PairedTestLocationPermuter(Vector<pair<double, double>> const& data)
    {
        for(int i = 0; i < data.getSize(); ++i)
            diffs.append(data[i].first - data[i].second);
    }
    double operator()() const{return f(diffs);}
    void exchange(Random<>& r)
    {
        for(int i = 0; i < diffs.getSize(); ++i)if(r.mod(2))diffs[i] *= -1;
    }
    void setShift(double shift) //must not be called after exchange
    {
        for(int i = 0; i < diffs.getSize(); ++i) diffs[i] += shift;
    }
};

```

For the median and the trimmed mean the permutation hypothesis is also that use a symmetric distribution around 0. Theoretically (Romano 1990) permutations tests for paired data that test a measure of location are:

- Exact when the difference distribution is symmetric
- Asymptotically exact when asymmetric but with finite variance

E.g., for the sample trimmed mean can use:

```

struct trimmer
{
    double operator()(Vector<double> observations) const
    {
        return trimmedMean(observations);
    }
};

```

For $k > 2$ alternatives the null is that any two are exchangeable, so can compare average ranks based on that instead of Nemenyi test, and correct for multiple testing. If the pairs come from the same difference distribution, don't need to convert to ranks because can form the average directly.

Permutation tests are exact (if try all assignments instead of random b) when the permutation group works as required—e.g., for location measures if the null is asymmetric, type I control becomes approximate and must be studied by simulation.

To form a confidence interval use test inversion: add a constant to every value of the sample to make it have the desired null θ . But such transformations work only with specific pivotal f . For the exponential search use the same seed for all tests. That the search is guaranteed to work isn't trivial—need the extreme tail p -values < the wanted a , so the latter can't be too small (roughly below $1/n$). This also means the starting value must have p -value > the wanted a . So the starting value is – the original statistic. If the search in a particular direction fails to find a root, assume that the statistic value is on the boundary. It helps if b is odd, because the computed p -value – 0.05 is unlikely to be exactly 0. The p -value as a function of the shift of data isn't continuous but discrete, and the exact 0 sometimes can't be selected by binary search as the solution because it's located at the sign-change.

```

template<typename PERM_TESTER> struct permConfHelper
{
    PERM_TESTER const& p;
    double a;
    int seed, b;
    permConfHelper(PERM_TESTER const& theP, double theA, int theSeed,
        int theB): p(theP), a(theA), seed(theSeed), b(theB) {}
}

```

```

double operator() (double shift) const
{
    Random<> rng(seed);
    PERM_TESTER p2 = p;
    p2.setShift(shift);
    return permutationTest(p2, b, 0, rng) - a;
}
};

template<typename PERM_TESTER> pair<double, double> permutationConf(
    PERM_TESTER& p, double a = 0.05, int b = 10000) //assume two-sided
{
    double stat = p();
    permConfHelper<PERM_TESTER> f(p, a, time(0), b);
    double left = exponentialSearch1Sided(f, -stat, -0.001).first,
        right = exponentialSearch1Sided(f, -stat).first;
    left = isfinite(left) ? left : -stat;
    right = isfinite(right) ? right : -stat;
    return make_pair(2 * stat + left, 2 * stat + right);
}

```

E.g., for location measures get:

```

template<typename LOCATION_F> pair<double, double> permutationLocationConf(
    Vector<double> const& data, double a = 0.05, int b = 10000)
{
    PairedTestLocationPermuter<LOCATION_F> p = {data};
    return permutationConf(p, a, b);
}

```

Distribution and Statistic	T				Normal			
	aL	aR	%L	ACL	aL	aR	%L	ACL
Normal Mean	0.025	0.025	0.034	0.231	0.029	0.030	-0.009	0.243
LogNormal Mean	0.002	0.113	-0.065	0.261	0.004	0.123	-0.104	0.268
2-Normal Mix Mean	0.023	0.024	0.031	0.224	0.028	0.028	-0.012	0.236
Average Ranks	1.667	3.000	1.667	4.000	1.333	3.333		
Bootstrap-t-capped	Permutation							
aL	aR	%L	ACL	aL	aR	%L	ACL	
0.021	0.022	0.071	0.223	0.025	0.025	0.034	0.232	
0.011	0.057	0.200	0.337	0.003	0.121	-0.117	0.250	
0.027	0.025	0.072	0.244	0.024	0.024	0.028	0.225	
1.667	4.000	3.000	2.667	1.667	2.000			

Figure 21.7: Performances of several confidence intervals for the mean with $n = 30$. The same settings as for the bootstrap simulations, except repeated 10000 times.

Distribution and Statistic	Nonp				Permutation				Bootstrap-t-capped			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Normal Median	0.020	0.018	0.082	0.211	0.066	0.065	-0.015	0.362	0.038	0.036	0.210	0.330
Levy Median	0.023	0.020	0.411	0.285	0.001	0.233	1.835	0.281	0.024	0.026	0.633	0.371
2-Normal Mix Median	0.022	0.020	0.274	0.261	0.016	0.006	0.470	0.215	0.017	0.016	0.184	0.214
Average Ranks	1.667	1.667	2.000	2.333	2.333	2.000	2.000	2.000	2.000	2.000	2.000	2.000

Figure 21.8: Performances of several confidence intervals for the median

Distribution and Statistic	Normal				Permutation				Bootstrap-t-capped			
	aL	aR	%L	ACL	aL	aR	%L	ACL	aL	aR	%L	ACL
Normal TrimmedMean	0.037	0.034	0.08	0.267	0.039	0.037	0.015	0.281	0.025	0.032	0.097	0.237
LogNormal TrimmedMean	0.015	0.062	-0.025	0.265	0.001	0.127	-0.026	0.249	0.017	0.028	0.149	0.246
Levy TrimmedMean	0.003	0.126	-0.136	0.286	0.000	0.258	-0.455	0.193	0.002	0.023	12.119	2.321
2-Normal Mix TrimmedMean	0.024	0.038	-0.021	0.242	0.013	0.024	0.025	0.193	0.018	0.032	0.058	0.237
Average Ranks	2.000	1.000	3.000	3.000	1.000	1.000	1.000	1.000	3.000	1.000		

Figure 21.9: Performances of several confidence intervals for the sample trimmed mean. The bootstrap isn't robust, while permutation testing apparently inherits robustness of whatever statistic it uses.

Based on the above permutation methods never win over the customized solution despite making fewer assumptions. In some cases, such as Levy sample median and all sample trimmed means, a is exceeded dra-

matically.

For correlation, the permutation group is formed by all permutations of the x and y values. Unfortunately for inverting it's unclear how to shift the data in response to a shift, especially because can violate range constraints.

Distribution and Statistic	Normal-Z				Bootstrap-t-capped			
	aL	aR	%L	ACL	aL	aR	%L	ACL
Normal Error Almost Line	0.012	0.006	0.267	0.203	0.027	0.016	0.353	0.273
Normal Error Cubic	0.002	0.000	1.083	0.087	0.019	0.046	0.180	0.281
Average Ranks	1.000	2.000	1.000	2.000	2.000	1.000	2.000	2.000

Figure 21.10: Performances of several confidence intervals for the Pearson correlation

Exchangeability is weaker than iid, but is enough. Beware that all properties are the same under the null, not just the mean. In a proper experiment with random assignments, no difference may be correct to assume as a null, but don't have a way to detect a difference in something other than the tested statistic. Proper randomization means, e.g., no biases in the data with regard to the test—i.e., the tested group may already be predisposed to a particular outcome.

For independent samples the situation is different—e.g., if compare two distributions with different variances but same means, don't have complete exchangeability. The situation is similar to the **Wilcoxon-Mann-Whitney test** (see Gibbons & Chakraborti 2020 for more information), which works with ranks but needs symmetry so that all possible rank assignments are equally likely. For two independent samples the mean difference is asymptotically exact if and only if the sample size ratio $\rightarrow 1$, or the finite variances are equal (Romano 1990). For the median or the trimmed mean differences get even less. So permutation test inversion is a bad option for confidence intervals on independent sample differences. Still, per Chihara & Hesterberg (2018) permutation tests give good results for small n where deviations from assumptions such as different variances can't be detected.

Permutations tests are definitely useful as a way to calculate the exact distributions for nonparametric statistics, such as the one for the Nemenyi test, particularly with ties, or for the corresponding confidence intervals. But for standard cases such as matched comparisons (and unmatched, though here get confidence intervals directly), it's best to apply nonparametric tests, which are more robust, though software StatXact is said to use permutation tests. For other cases, as for bootstrap, do problem-specific research first.

21.26 Comparing Many Alternatives on Multiple Domains

Because multiple domains are modeled well with matched samples, converting scores to ranks, and using Nemenyi test is a good general method. It assumes that all domains are equally valuable and combines performances using averaged ranks. But \exists other ways to do this. The main problem is that of any test—can have statistical significance without practical significance. So want an estimator of performance for each alternative. For statistical significance must assume the domains to be chosen at random for a set of practical domains.

It usually makes no sense to aggregate over all domains using functions such as the average. E.g., even if \forall domain and alternative, performance scores \sim normal, average weighs larger-scale domains more. Must transform domain-specific scores so that averaging them is more meaningful. Some generic transformations:

- Convert into ranks—what Nemenyi does, but don't get useful grades. E.g., given 10 alternatives with similar performances, their obvious “rank grades” $1 - \frac{\text{rank} - 1}{10}$ are 1, 0.9, 0.8, 0.7, etc.
- Map from $[\min, \max]$ to $[0, 1]$. This preserves information better than the ranks but runs into trouble when the performances are very similar. E.g., students with grades 99 and 98 would respectively get adjusted grades 100 and 0.
- For percentage scores and when larger values are better, map $[0, \max]$ into $[0, 1]$. This avoids magnifying differences between close scores.
- When min is better, map absolute scores into relative ones by dividing by the performance of a baseline alternative, which is usually the worst-performing method. E.g., this is useful for comparing runtimes.

To study significance for options other than ranks, can bootstrap where resample the domains. But should have maybe ≥ 10 domains for bootstrap to give accurate results.

Seem to have power loss when the scores themselves are aggregates with known distributions. But using this information is difficult because transformations like min-max are estimated from data. And must assume that the scores can be aggregated over the domains. So this only makes sense if have domain-spe-

cific data-independent scaling and weights for a weighted average.

A domain-aware grading option is to convert scores into binary for whether the performance is acceptable. Then can take averages directly and treat the problem as that of independent comparisons. In practice this is how do comparisons, and usually pick methods that have low chance of catastrophic performance, even if they are slightly worse on average than the rest. Unfortunately analysis of Friedman ranks is useless for this purpose, and only serves as extra information.

The average isn't the only way to aggregate, and all run into **Arrow's impossibility theorem** (Wikipedia 2014b): With ≥ 3 alternatives and their rankings by various voters, can't have all of:

- Absence of dictatorship—no single voter controls the result
- Unanimity—if all voters prefer A over B, so does the group
- **Independence of irrelevant alternatives (IIA)**—the group's preference of A vs B must depend only on voters' preferences of the two and not on the other alternatives

Most rank combination systems sacrifice IIA, which isn't necessarily a bad thing in statistics. Prefer both an alternative that performs better on more domains (i.e., with more voters) and one that doesn't perform very poorly on some domains. So adding a noncompetitive alternative can give a different best performer because the existing best is worse than that alternative on one domain. E.g., given two good students, one of whom got an F on a single exam, adding more C students will lower the rank of the unlucky student's performance on that test, perhaps enough to let the other win by average rank.

A special case of IAA is the **Condorcet criterion**—no other candidate may be preferred by the majority over the winner. E.g., in the 2000 U.S. election Gore would have won without Nader. A Condorcet winner doesn't exist under transitive preferences, but in practice this is rare. Similar issues apply to curved grade methods, because IIA can give advantage to alternatives doing superbly on tasks where the majority does well.

An interesting method is **approval voting**—a person marks every candidate as approved or not, and the winner is the one with the highest approval count (Wikipedia 2019d). In elections this is a convenient strategy, adopted by many municipalities:

- Voters aren't penalized for expressing their preferences because, e.g., it's safe to approve both Nader and Gore
- Arrow's theorem doesn't apply to the grading scheme, though it does to the ranking scheme, because a voter technically gets more than one vote despite the obvious fairness
- More-centrist alternatives are automatically preferred by the voters

This is similar to something such as voting for the mode and not for the median or the mean. In a continuum each "voter" effectively creates an ϵ -band around the desired choice and a maximum-coverage region is chosen. This is much like the highest-probability null value in statistical testing. Another way to look at this is disapproval voting. I.e., least liked alternatives will be eliminated. This may seem useful for selecting alternatives that are never too bad but is more about avoiding alternatives that are never unpopular with the majority. The penalty for an arbitrarily bad performance with a small group of voters is actually the same.

Can do tournament selection if the alternatives form some logical subgroups. Split the domains in half or so. Then consider subgroups of alternatives on the first half, and the winners and those not significantly worse on the second half. One advantage is that can do FDR multiplicity adjustment, and with large fraction of poor alternatives many will drop out in the first round.

Sometimes want to know only whether, given a current (presumably implemented and deployed) solution, picking another gives better enough result to switch. Also beware that poor alternatives may do well if the problem is too easy. For the comparison to be meaningful, need a baseline expert method to outperform a baseline poor method.

Another goal may be to find an alternative with a particular property (e.g., simple), whose performance isn't much worse than that of the best alternative or a set of alternatives. A general heuristic for this is the **one standard error rule**—pick an alternative with the desired property, whose performance \geq performance of the best model – standard error of its performance estimate. Or can pick an alternative whose performance isn't worse than the best by maybe 5% of the best performance but has a wanted property like simplicity. Can add further penalty to make it worth picking such alternative.

21.27 Working with Count Data

Often model counts using $\text{binomial}(p, n)$ distribution (p here not to be confused with p -value), which $\approx \text{normal}\left(p, \frac{p(1-p)}{n}\right)$ with little loss of accuracy if p isn't close to 0 or 1. Otherwise, and when p is esti-

mated, this is very inaccurate. E.g., to test the difference of p from 0 at certain a z-score, the null equality is rejected if $\bar{x} \geq \frac{1}{1+n/z^2}$, which is too often. E.g., after seeing n yes samples, $\Pr(\text{no}) = 0$ by the normal approximation.

When p is estimated, Wald interval works poorly. A better interval is obtained by inverting a normal test that uses the null variance $p(1-p)$ instead of $s^2 = \bar{x}(1-\bar{x})$. The sign test actually does this in a different context. The logic is that if tested exactly, would use the binomial likelihood with p , which has the null variance. This corresponds to a general construction in mathematical statistics of a **score test**. Some equation solving leads to **Wilson score interval** $\left(\frac{p+t/2-\Delta}{1+t}, \frac{p+t/2+\Delta}{1+t} \right)$, where $t = \frac{z^2}{n}$, and $\Delta = z \sqrt{\frac{\bar{x}(1-\bar{x})+t/4}{n}}$

(Wikipedia 2019c). The interval is asymmetric when $m \neq 0.5$ but symmetric about a Bayesian estimate, which for $z=2$ is equivalent to adding two true and two false observations.

```
pair<double, double> wilsonScoreInterval(double p, int n, double z = 2)
{
    assert(p >= 0 && p <= 1 && n > 0 && z >= 0);
    double temp = z * z/n, delta = z * sqrt((p * (1 - p) + temp/4)/n);
    return make_pair((p + temp/2 - delta)/(1 + temp),
                     (p + temp/2 + delta)/(1 + temp));
}
```

The interval doesn't give strict α control. But practical performance is so good that it's claimed to be better than the exact intervals based on inverting the binomial because it doesn't pick the largest available $\alpha \geq$ the requested one (Brown et al. 2001). For p -values this argument doesn't hold, but the score test is still useful with ties and computationally. Wilson score interval also allows tie handling from paired comparisons by allowing half-counts. It's the method of choice, e.g., for classification performance assessment (see the "Machine Learning—Classification" chapter).

Distribution and Statistic	Wilson				Normal			
	aL	aR	%L	ACL	aL	aR	%L	ACL
Bernoulli05 Mean	0.021	0.021	0.041	0.211	0.021	0.021	0.073	0.224
Bernoulli01 Mean	0.026	0.000	-0.059	0.062	0.008	0.184	-0.121	0.332
Bernoulli001 Mean	0.036	0.000	0.969	0.034	0.0000	0.740	-0.461	0.464
Average Ranks	3.000	1.667	3.000	3.667	1.333	4.000		
Hoef_Ori	Hoeffding							
	aL	aR	%L	ACL	aL	aR	%L	ACL
	0.003	0.003	0.378	0.100	0.003	0.003	0.487	0.108
	0.002	0.000	0.213	0.023	0.0000	0.000	0.490	0.006
	0.003	0.000	1.093	0.012	0.000	0.000	2.869	0.000
	1.667	3.000	1.667	1.000	4.000	1.333		

Figure 21.11: Performance comparisons for Bernoulli samples with different parameters. Wilson score doesn't lose accuracy even at 0.01 unlike the normal. Finite-sample methods give much longer intervals.

Note that can't combine intervals based on which one is shorter or any other criteria based on the data because this leads to multiple testing.

A solution to the 0 problem, which is a special case, is the "**rule of three**" (Wikipedia 2015a). If only yes events were observed, p must be small. $\Pr(n \text{ yes events}) = (1-p)^n$. 95% confidence needs $(1-p)^n \geq 0.05$. Solving this, that $\lg(0.05) \approx 3$, and that $\lg(1-p) \approx p$ for very small p , lead to $p \leq 3/n$. I.e., with 95% confidence $p \in [0, 3/n]$. Though the proportion of no events shrinks with n at a fixed confidence, can't eliminate their existence. In software development terms, testing doesn't prove absence of buggy use cases but reduces the chance of discovering one.

A similar issue is the **black swan problem**, i.e., find $\Pr(\exists \text{ a black swan after seeing } n \text{ white ones})$. Fundamentally, a black swan event isn't predictable because the data isn't iid (blank swans were eventually discovered in Australia and don't live on other continents). The warning of "black swan" logic is that a model might have unknown unknowns. One idea is that things are the way they are because they got this way, and a few extreme, unpredictable events tend to account for most of the "got". E.g., consider a sum of Cauchy samples—most of its value is usually attributed to several samples. Same for the stock market—most value change is due to few very good and very bad days and to few great companies among many average ones. In general have a both a fat-tail distribution and lack of iid. A general strategy to counter this is to avoid estimating probabilities—i.e., agnostically don't put all eggs in the same basket. This way get basic safety by

working directly in the strategy space.

21.28 Testing Distribution Differences

To test match of a sample to a discrete distribution use the **chi-squared test**. For a bounded distribution with n independent bins, approximately the statistic $\sum_{mean_i>0} \frac{(count_i - mean_i)^2}{mean_i}$ ~ chi-squared with n

degrees of freedom (Wikipedia 2018b). The approximation is considered accurate if all means > 5 . If have dependencies between the bins overlapping regions, the degrees of freedom will change. E.g., use $n - 1$ if the means sum to a constant.

The test needs to evaluate the chi-squared CDF, but the exact evaluation is slow and has to deal with many cases (Temme 1994), so evaluate approximately. Chi-squared \approx normal, and a transformation improves the approximation (Canal 2005): If $X \sim \text{ChiCDF}(n)$, $x = \left(\frac{X}{n}\right)^{1/6} - \frac{1}{2}\left(\frac{X}{n}\right)^{1/3} + \frac{1}{3}\left(\frac{X}{n}\right)^{1/2} \sim \text{normal}\left(\frac{x-\mu}{\sigma}\right)$, where μ and σ^2 are $\sum \frac{a_i}{x'}$, with the respective $a_i = \left\{ \frac{5}{6}, -\frac{1}{9}, -\frac{7}{648}, \frac{25}{2187} \right\}$ and $\left\{ 0, \frac{1}{18}, \frac{1}{162}, -\frac{37}{11664} \right\}$. This has the worst case $O(10^{-2})$ and typical $O(10^{-5})$ errors, becoming more accurate with larger n .

```
double evaluateChiSquaredCdf(double chi, int n)
{
    assert(chi >= 0 && n > 0);
    double m = 5.0/6 - 1.0/9/n - 7.0/648/n/n + 25.0/2187/n/n/n,
        q2 = 1.0/18/n + 1.0/162/n/n - 37.0/11664/n/n/n, temp = chi/n,
        x = pow(temp, 1.0/6) - pow(temp, 1.0/3)/2 + pow(temp, 1.0/2)/3;
    return approxNormalCDF((x - m)/sqrt(q2));
}

double chiSquaredP(Vector<int> const& counts,
    Vector<double> const& means, int degreesOfFreedomRemoved = 0)
{
    double chiStat = 0;
    for(int i = 0; i < counts.getSize(); ++i)
        //enforce 5 in each bin for good approximation
        assert(means[i] >= 5);
        chiStat += (counts[i] - means[i]) * (counts[i] - means[i])/means[i];
    }
    return evaluateChiSquaredCdf(chiStat,
        counts.getSize() - degreesOfFreedomRemoved);
}
```

The test also applies to $D > 1$. Only need to be able to define the bins, and calculate the expected counts.

21.29 Comparing Data to a Distribution

Using DKW can compare a sample to a fully-specified continuous distribution T using the empirical distribution F . First, compute the statistic $\max_x(|T(x) - F(x)|)$. The maximum occurs either at a data point or ϵ before the next one in sorted order. Because of continuity, $\max_x(|T(x) - F(x)|) = \max_x(|T(x_i) - F(x_i)|, |T(x_i) - F(x_{i-1})|)$. So sort each sample, and make a mergesort-like scan for event points.

```
template<typename CDF> double findMaxKDiff(Vector<double> x, CDF const& cdf)
{//helper to calculate max diff
    quickSort(x.getArray(), x.getSize());
    double level = 0, maxDiff = 0, del = 1.0/x.getSize();
    for(int i = 0; i < x.getSize(); ++i)
    {
        double cdfValue = cdf(x[i]);
        maxDiff = max(maxDiff, abs(cdfValue - level));
        level += del;
        while(i + 1 < x.getSize() && x[i] == x[i + 1])
        {
            level += del;
        }
    }
}
```

```

        ++i;
    }
    maxDiff = max(maxDiff, abs(cdfValue - level));
}
return maxDiff;
}
template<typename CDF> double DKWPValue(Vector<double> const& x,
                                         CDF const& cdf)
{//DKW invalid for p-value < 0.5
    double delta = findMaxKDiff(x, cdf);
    return min(0.5, 2 * exp(-2 * x.getSize() * delta * delta));
}

```

Asymptotically (Gibbons & Chakraborti 2020), for the adjusted statistic $d = \frac{\max_x(|T(x) - F(x)|)}{1/\sqrt{n}}$, p -value = $2 \sum_{1 \leq i \leq \infty} (-1)^{i-1} e^{-2i^2 d^2}$ (Gibbons & Chakraborti 2020), which leads to single-sample **Kolmogorov test**. The above **DKW test** is identical when using only the first term of the series, but is finite-sample. Continuity isn't needed for validity, but can lose some power if don't have it.

Knowing a fully-specified CDF is rare, except for special cases such as comparing randomly generated numbers to a uniform(0, 1). Estimating it from data and comparing to it isn't the same as knowing a distribution beforehand. But it's a good heuristic for large samples (>200 maybe). So ! \exists a good general extension of DKW test to matching a sample against a distribution family, except for ad hoc matching of a shape into a DKW band. In some cases, such as with the normal, can center and scale the data, which limits the estimation error.

Knowing a distribution approximately is useful domain knowledge in many cases. But the test isn't the right way to show it because it looks for an exact match. A distribution can be a good model for data from a certain domain, but it's unlikely to match in some decimal place, which a significance test will eventually detect. Perhaps the best way to test this is to pick a DKW tolerance and check if it's exceeded. But a better way seems to be parametric bootstrap—sample from both the data and the conjectured distribution, and see if the results for the wanted application differ enough. E.g., for the contaminated normal distribution, the Kolmogorov distance can be small to the standard normal, but the variance difference can be very large (Wilcox 2016).

A particular application is testing for normality. Here the most practically powerful test is **Shapiro-Wilks**, which is based on correlation between the data and quantiles of the normal distribution. The critical values are obtained by simulation. Use Shapiro-Wilks until n is large enough to exceed tabled values, and then switch to the DKW test. See Gibbons & Chakraborti (2020) for details if curious.

A commonly suggested strategy of testing for normality and using parametric or nonparametric tests depending on the result is flawed because the combined procedure has different power and type I error than the separate ones. For small n usually can't reject normality anyway, and for large nonparametric tests have enough power and are more robust. So never use distribution testing for procedure selection.

More generally, don't have methods to check arbitrary assumptions. At best can do some prediction analysis based on experiments with the chosen model, and conclude that violations are minor or the model has limited sensitivity to them, without necessarily knowing which is the case. Some exploratory analysis on a reserved fraction of the data may help to determine the degree of deviation, though don't actually get a quantitative measure (e.g., google "qq plot for normality"). In Bayesian logic actually any specific assumption is almost surely false exactly.

21.30 Comparing Distributions of Two Samples

Given two samples of size m and n , a simple conceptual solution is to estimate EDFs F_1 and F_2 for both samples, compute their confidence intervals to the true distributions using DKW, and check if they fail to intersect anywhere.

But direct comparison is more powerful. Compute **Kolmogorov-Smirnov statistic** $\max_x(|F_1(x) - F_2(x)|)$ by sorting each sample and making a mergesort-like scan for event points. Purely by chance, each EDF can be higher than the other. With a binomial distribution, the expected deviation is $O(1/\sqrt{n})$. For the KS statistic, it's $\sqrt{\frac{m+n}{mn}}$. This leads to the adjusted statistic $d = \max_x \left(\frac{|F_1(x) - F_2(x)|}{\sqrt{(m+n)/mn}} \right)$. Asymptotically (Gibbons &

Chakraborti 2020), its p -value = $2 \sum_{1 \leq i \leq \infty} (-1)^{i-1} e^{-2i^2 d^2}$. To be extra conservative, use only the first term of

the sequence ($\approx 95\%$ of the relative value), but even then the test isn't exact unless $n = m \geq 458$ (Wei & Dudley 2012).

```

double findMaxKSDiff(Vector<double> a, Vector<double> b)
//helper to calculate max diff
    quickSort(a.getArray(), a.getSize());
    quickSort(b.getArray(), b.getSize());
    double aLevel = 0, bLevel = 0, maxDiff = 0, delA = 1.0/a.getSize(),
    delB = 1.0/b.getSize();
    for(int i = 0, j = 0; i < a.getSize() || j < b.getSize();)
    {
        double x, nextX = numeric_limits<double>::infinity();
        bool useB = i >= a.getSize() || (j < b.getSize() && b[j] < a[i]);
        if(useB)
        {
            x = b[j++];
            bLevel += delB;
        }
        else
        {
            aLevel += delA;
            x = a[i++];
        }
        //handle equal values--process all before diff update
        if(i < a.getSize() || j < b.getSize())
        {
            useB = i >= a.getSize() || (j < b.getSize() && b[j] < a[i]);
            nextX = useB ? b[j] : a[i];
        }
        if(x != nextX) maxDiff = max(maxDiff, abs(aLevel - bLevel));
    }
    return maxDiff;
}
double KS2SamplePValue(Vector<double> const& a, Vector<double> const& b)
//calculate the adjustment first, then find p-value of d
    double stddev = sqrt(1.0 * (a.getSize() + b.getSize()) /
    (a.getSize() * b.getSize())),
    delta = findMaxKSDiff(a, b)/stddev;
    return 2 * exp(-2 * delta * delta);
}

```

To have finite-sample correctness can calculate DKW bound distance for each band using the same statistic, but this has much lower power due to smaller expected deviation of d .

21.31 Sensitivity Analysis

Given function $y=f(x)$, want to know effects of x on y . Typically would use finite differences (see the “Numerical Algorithms—Working with Functions” chapter) to compute derivatives, but this is problematic if f is noisy. And interpreting derivatives that are different in various regions is hard. One problem is caused by Simpson's paradox (see the “Machine Learning—Regression” chapter)—a variable omitted by model selection can be highly correlated to another variable that isn't, and the derivative of the model wouldn't reflect the former's impact. Also, unlike most of statistics, sensitivity analysis studies the estimator and not the estimate.

An alternative is global sensitivity analysis (Saltelli et al. 2008). A particular useful approach is to calculate **Sobol's sensitivity index**. \forall input variable x_i , $S_i = \frac{\text{Var}(E[y|x_i])}{\text{Var}(y)}$ (don't confuse S_i with sample standard deviation). To calculate the numerator, \forall value of x_i calculate its expectation with respect to all other variables, and then the variance of the expectations. S_i make sense only if $\text{Var}(y)$ is bounded—need a bounded range for x or a distribution with flat tails. Then have the decomposition $\text{Var}(y) = \sum \text{Var}(E[y|x_i]) + \sum \text{Var}(E[y|x_i, x_j]) + \dots$ + higher-order interactions. The first term represents the **main effects**.

A simple way to calculate S_i from n evaluations of f is to use histogram-like slicing— $\forall x_i$ sort the values, bin them into maybe \sqrt{n} bins, and calculate the expectation over each bin, and then the variance. Though this is consistent (from k -NN arguments; see the “Machine Learning—Classification” chapter), it introduces

a small bias. **Saltelli's algorithm** (Saltelli et al. 2008) is more efficient and gives better accuracy:

1. Given $2n$ D-dimensional samples of x , split them into "matrices" XA and XB of n examples each
2. $\forall 0 \leq j \leq D$ create $XC_j = XB$ but with j^{th} variable in all examples taken from XA
3. Evaluate f on rows of XA and XC to form vectors of size n YA and YC
4. $S_j = \frac{YAYC_j/n - \text{Ave}(YA)^2}{YAYA/n - \text{Ave}(YA)^2}$

Use bootstrap on YA and YC to calculate mixed multiplicity-adjusted confidence intervals. These are useful for checking the stability of the calculation (e.g., have instability if the $\text{Var}(y)$ is unbounded, etc).

```

Vector<double> findSobolIndicesHelper(Vector<double> const& ya,
    Vector<Vector<double>> const& yc)
{//calculate S from YA and YC
    int n = ya.getSize(), D = yc.getSize();
    Vector<double> result(D);
    IncrementalStatistics s;
    for(int i = 0; i < n; ++i) s.addValue(ya[i]);
    double f02 = s.getMean() * s.getMean(), tempa = dotProduct(ya, ya)/n;
    for(int j = 0; j < D; ++j)
        result[j] = max(0.0, (dotProduct(ya, yc[j])/n - f02)/(tempa - f02));
    return result;
}

template<typename FUNCTOR> pair<Vector<double>, Vector<pair<double, double>>>
findSobolIndicesSaltelli(Vector<pair<Vector<double>, double>> const&
    data, FUNCTOR const& f, int nBoots = 200, double a = 0.05)
{//calculate ya and yb
    int D = data[0].first.getSize(), n = data.getSize()/2;
    Vector<double> ya(n), yb(n), yaR(n);
    for(int i = 0; i < 2 * n; ++i)
        if(i < n) ya[i] = data[i].second;
        else yb[i - n] = data[i].second;
//calculate yc
    Vector<Vector<double>> yc(D, Vector<double>(n)), ycR = yc;
    for(int j = 0; j < D; ++j)
        for(int i = 0; i < n; ++i)
        {
            Vector<double> x = data[n + i].first;
            x[j] = data[i].first[j];
            yc[j][i] = f(x);
        }
//bootstrap to find standard deviations
    Vector<IncrementalStatistics> s(D);
    for(int k = 0; k < nBoots; ++k)
{//resample data rows
        for(int i = 0; i < n; ++i)
        {
            int index = GlobalRNG().mod(n);
            yaR[i] = ya[index];
            for(int j = 0; j < D; ++j) ycR[j][i] = yc[j][index];
        }
//evaluate
        Vector<double> indicesR = findSobolIndicesHelper(yaR, ycR);
        for(int j = 0; j < D; ++j) s[j].addValue(indicesR[j]);
    }
    Vector<double> indices = findSobolIndicesHelper(ya, yc);
    Vector<pair<double, double>> confs;
    double z = getMixedZ(a/D);
    for(int j = 0; j < D; ++j)
    {
        double delta = s[j].stdev() * z;
        confs.append(make_pair(indices[j] - delta, indices[j] + delta));
    }
    return make_pair(indices, confs);
}

```

21.32 Sobol Sequence

In many cases, such as calculating multidimensional integrals, the simulation usually happens on a unit hypercube, possibly extended by the inverse method to more general regions. Here further variance reduction is possible. The trick is using special nonrandom samples. Uniformly random unit hypercube samples tend to form small gaps and clusters and not cover the hypercube uniformly, unlike desired.

Want from the coverage small **star discrepancy** $D^*(n) = \max_X \left(\frac{\text{the number of samples in } X}{n} - \text{Vol}(X) \right)$, where X is the D -dimensional hyperrectangle with $0 \leq x_j \leq u_j \leq 1$ over all possible choices of u , (Wikipedia 2016a). Many **quasi-random sequences** are known that have the asymptotically optimal $D^*(n) = O\left(\frac{\lg(n)^D}{n}\right)$.

Their usefulness is shown by the **Koksma-Hlawka inequality**: Given function f with bounded **total variation** (Wikipedia 2016b) V in a hypercube, for n samples x_i , $|\text{Ave}(f(x_i)) - \int_{\text{hypercube}} f| \leq V(f) D(n)$. So have $O\left(\frac{\lg(n)^D}{n}\right)$ worst-case error, which, despite exceeding Monte Carlo's 95% confidence $O(1/\sqrt{n})$ for typical n and not too small D , is better at least for $D \leq 3$ and in practice for larger D .

For differentiable f , $V(f) = \int ||\nabla f||$; for nondifferentiable f can often take a limit. E.g., for $f(x) = 1$ if in unit circle and 0 otherwise, using a limiting thin disk of width h where have linear increase from 0 to 1, $V(f) = 2\pi$.

Sobol sequence has the theoretical performance and is fast to evaluate (Bratley & Fox 1988). A dimension precompute initialization data:

- Pick a not-yet-picked primitive binary polynomial $x^D + a_1x^{D-1} + \dots + a_{D-1}x + 1$ of smallest degree
 - Let precision of a double be B bits. $\forall d \in [0, D-1]$ and $j \in [0, B-1]$ compute $v_j = \frac{2^{j+1}}{2^{B-1-j}}$ for $j < D$, and $v_{j-D} \wedge v_{j-D}/2^D \wedge a_{D-1}v_{j-1} \wedge \dots \wedge a_1v_{j-D}$ otherwise
 - Set the count $k = 1$ and initial variate $x = 0$
 - To generate: $x_i = v_{i,c}$ where c is the position of the rightmost 1 bit of k

The implementation uses a fixed number of polynomials, found in Press et al. (2007) or online. That need $k < 2^B$ isn't an issue even for very long simulations. Values of v are binary fractions represented as integers with the normalizing factor 2^{-B} .

```

        value = v[vIndex(i, 1)];
        value ^= value/twoPower(SobolDegs[i]);
        for(int k = 1; k < SobolDegs[i]; ++k)
            if(Bits::get(SobolPolys[i], k - 1))
                value ^= v[vIndex(i, 1 + k)];
    }
    v[vIndex(i, j)] = value;
}
next();
}
void next()
{
    for(int i = 0, c = rightmost0Count(k++); i < x.getSize(); ++i)
        x[i] ^= v[vIndex(i, c)];
}
double getU01Value(int i) const {return x[i] * factor;}
double getUValue(int i, double a, double b) const
    {return a + (b - a) * getU01Value(i);}
};

```

E.g., for estimating π a particular simulation with Sobol sequence gave 3.14158992, with worst-case error $\leq 8.5e^{-5}$, assuming $D^*(n) = \frac{\lg(n)^D}{n}$. This beats the Monte Carlo estimate's accuracy (see the "Random Number Generation" chapter) by about a factor of 5.

Sobol sequence (like other quasi-random sequences) has limitations:

- f must be deterministic for Koksma-Hlawka to apply
- $V(f)$ is rarely known
- Koksma-Hlawka is useless beyond very small D
- \exists a random bound—the sequence is completely deterministic, and can design a function whose properties aren't discovered with reasonable n

Despite the lack of reasonable guarantees, for certain tasks like computing integrals, Sobol sequence gets very good accuracy (Sobol et al. 2011). It seems to have many good random number properties but isn't random by avoiding gaps and clusters of random sampling, which makes it biased. Perhaps the reason for avoiding the worst-case behavior is the same as that for pseudo-random numbers—the information from the sequence and the application domain tends to be independent.

Because Sobol sequence is fully deterministic, the only source of uncertainty is values of f at not evaluated x . A simple way to get a confidence bound is **randomized quasi Monte Carlo (RQMC)**—generate a uniformly random u in the hypercube, and merge it with every Sobol sample x_i using $x_{new,i} = (x_i + u) \% 1$. This doesn't change the discrepancy of the resulting sequence (Lemieux 2009). Using many such configurations the different sequences would sample f at different, essentially iid points. The individual Sobol batch averages should have low variances, so always using maybe 30 different u is enough for accurate CLT bounds.

Another simple extension is to use random sampling if go beyond the maximum supported dimension.

```

class ScrambledSobolHybrid
{
    int D;
    Vector<double> shifts;
    mutable Sobol s;
    Vector<pair<double, double>> box;
public:
    ScrambledSobolHybrid(Vector<pair<double, double>> const& theBox):
        D(theBox.getSize()), shifts(D), s(min(D, Sobol::maxD())), box(theBox)
    {
        for(int i = 0; i < D; ++i)
            shifts[i] = GlobalRNG().uniform(box[i].first, box[i].second);
    }
    Vector<double> operator()() const
    // first get Sobol variates for the supported dimensions
    Vector<double> next(D);
    for(int i = 0; i < min(D, Sobol::maxD()); ++i)
        next[i] = s.getUValue(i, box[i].first, box[i].second);
};

```

```

        s.next();
        //random for remaining dimensions;
        for(int i = min(D, Sobol::maxD()); i < D; ++i)
            next[i] = GlobalRNG().uniform(box[i].first, box[i].second);
        //scramble
        for(int i = 0; i < D; ++i)
        {
            next[i] += shifts[i] - box[i].first;
            if(next[i] > box[i].second)
                next[i] -= box[i].second - box[i].first;
        }
        return next; //generate first value
    }
}

```

Whether Sobol sequence is more accurate than random sampling is unknown, but in practice it usually is at least up to $D = 1000$ or so, which is where it has been tested in the literature. RQMC is usually the preferred method in practice. But pure Sobol sequence with no confidence interval usually gives a better estimate, for which can heuristically use the mathematically invalid iid interval.

Generation works with box regions, but with transformations can sample other regions. E.g., in machine learning it's often useful to sample geometric grids, e.g., from 10^{-3} to 10^3 or so. A log transformation turns this region into a hyperrectangle:

```

template<typename SAMPLER = ScrambledSobolHybrid> class GeometricBoxWrapper
{
    static Vector<pair<double, double>> transformGeometricBox
        (Vector<pair<double, double>> box)
    {
        for(int i = 0; i < box.getSize(); ++i)
        {
            assert(box[i].first > 0 && box[i].first < box[i].second);
            box[i].first = log(box[i].first);
            box[i].second = log(box[i].second);
        }
        return box;
    }
    SAMPLER s;
public:
    GeometricBoxWrapper(Vector<pair<double, double>> const& box) :
        s(transformGeometricBox(box)) {}
    Vector<double> operator()() const
    {
        Vector<double> result = s();
        for(int i = 0; i < result.getSize(); ++i) result[i] = exp(result[i]);
        return result;
    }
};

```

21.33 Design of Experiments—Main Ideas

Can't get useful conclusions from data obtained through a poorly planned study. Want to design an experiment to get maximum validity and reduce variance as much as possible. These are sometimes in conflict. E.g., to compare several algorithms, can use:

- Independent inputs—get a confidence interval for each algorithm, assuming iid input selection.
- Common inputs—reduce variance due to input selection, but can no longer give a generally valid confidence interval for each algorithm. But pairwise intervals for performance differences are valid and shorter than those for independent inputs.

Requirements of permutation tests are generally what for-comparison designs try to satisfy. Usually **factors** (variables in DOE language) are continuous, possibly in bounded ranges according to domain knowledge. General principles of experimental design:

- **Randomization**—sample the data randomly or at least allocate subjects to compared groups at random. This allows using permutation testing because all possible assignments become equally likely. This is the only way to make sure that random behaviors in the environment don't bias the

- results more than by chance. So set variables that can be controlled to random values.
- **Blocking**—for variables over which have direct control but which aren't being studied, randomization is applicable, but can reduce variance by comparing responses at fixed **levels** (variable values in DOE language). This is similar to pairing in tests like Nemenyi's and the common random numbers strategy. Want to prevent variance in responses due to level differences. So block what can be blocked and randomize what can't. Blocking levels should be scientifically meaningful—e.g., for comparing algorithms use the largest feasible instance sizes and smaller ones that matter for some uses cases. Might argue that for algorithm comparison input selection uses blocking and not randomization (though assume the latter for analysis).
 - **Replication**—to reduce bias due to particular level choices, use many different levels. E.g., compare algorithms on many different problems. This applies to both blocked and randomized variables. Replication ≠ repetition because use different level values. Replication is essentially equivalent to increasing the sample size in a smart way.

A classic experiment is **lady tasting tea**—the famous statistician Fisher gave a coworker, who claimed to be able to identify a certain tea quality by taste, four cups of that tea and four cups of another in random order. The chance of identifying a correct permutation by chance is small. This leads to a permutation test where every decision sequence is as likely as any other. But note that no power analysis was done here (Fisher was against the idea of an alternative hypotheses). Using four cups above only gave a good enough resolution for the p -value, but power analysis needs specifying an indifference region and an alternative that satisfies it. Also the test assumes random predictive ability, though a slightly-better-than-random one might be expected logically.

Designs for regression (see the “Machine Learning—Regression” chapter) are fundamentally different from for-comparison designs. As does most of the literature, focus on linear regression due to its usefulness in preliminary analysis. Its low estimation error allows creating a useful model from few samples, which is crucial when experiments are expensive. One particular aspect is the assumptions that study y given x and the relationship is linear. This means that don't need to explore the domain by sampling, e.g., in different areas. Want efficiency of estimation.

To find an optimal value of something, should skip regression and design, and use a numerical minimization technique such as Nelder-Mead or SPSA (see the “Numerical Optimization” chapter). But often want to get some insight into the behavior of a studied system—e.g., for a bread baking system possible factors are temperature, the amount of ventilation, flour type, etc.

In particular, for linear regression with k factors consider a 2^k **full factorial design**—use any 2 levels \forall variable and try all possibilities. This also allows easy solving— \forall variable have 2^{k-1} estimates of the corresponding linear regression coefficient, which are averaged. An interesting feature is that this makes the most efficient use of the 2^k runs for estimating a linear model under the normal iid error assumption. This follows from properties of the linear model—an optimal design for minimizing variance follows from **D-optimality**—maximize $\det(X^T X)$, where X is the matrix of factor levels values for the runs, scaled so that -1 is the first value, and 1 the second. Full factorial is far more efficient than the obvious **one-factor-at-a-time**.

With large k full factorial is impractical, and need only $k + 1$ examples for estimation. A simple work-around is computing an optimal design for a given number of runs using D-optimality (Goos & Jones 2012). This works for estimating linear models for low-variance systems and can accommodate constraints such as prior knowledge of infeasibility of some treatments in the factorial design. Can use the results to decide which factors are likely to be irrelevant (based on multiplicity-adjusted normal error hypothesis tests, see Goos & Jones 2012 for examples; though it seems better to use lasso regression for this—see the “Machine Learning—Regression” chapter) or for optimization. The literature on DOE is extensive; a practical problem is that very many designs have been proposed and analyzed, but only relatively recently the practical trend is to generate one automatically.

For subsequent optimization, pick the best value from the linear model experiment in a doubling trust region (see the “Numerical Optimization” chapter). Such surrogate models (see the “Machine Learning—Regression” chapter) tend to work better by allowing more powerful models. This may be more efficient than direct numerical optimization.

For many regression problems estimation resources aren't severely limited, and will try several models with a selection process. Here randomization seems to be a better choice than the blocking of factorial design because the determinant criteria doesn't apply to other models, and exploration is important for subtasks like estimation of performance. Want a design that works well for any model, and random is a good candidate. When evaluations are expensive, want to avoid bad designs, but rarely need the best one.

Designs based on low-discrepancy sequences are another good candidate here because they attempt to work well for all possible functions. Another possibility is **latin hypercube sampling**—break up the space

into regions and take a sample from each region. This is something between random and quasi-random sampling. LHS is probably the most nonrandom sampling that still has enough randomness to support much mathematical analysis. But it performs worse than Sobol sequence on integration problems (Kucherenko et al. 2015) and doesn't scale beyond small D (though could combine it with fractional factorial using the latter to pick the regions). It's also not computable online.

Observational studies are very different from designed ones due to lacking explicit randomization. E.g., in a comparison of two populations random assignment means that any significance difference can be attributed to the population membership because randomization keeps everything else balanced. But if look at data that occurred, don't have exchangeability, so the attribution is invalid. E.g., if a group of patients taking one drug gets better than the group taking another drug, it may be because the latter group are in a worse condition to begin with. The general fallacy is concluding that a subpopulation has a property that the general population doesn't—the way the subpopulation defined may predispose it to have such property.

Because a large fraction of data is observational and designed studies aren't practical in many cases, observational studies are scientifically useful if show that the allocation of samples wouldn't be materially different if it were random. This was done, e.g., for the classic study of a link between smoking and cancer. The statistic is the rate of cancer of smokers vs nonsmokers. Unlike in a designed study, don't know if the smokers with cancer will have developed it if they haven't been smoking; the other way for nonsmokers. The argument was that an individual's decision to smoke isn't related to any other potentially cancer-causing factors such as some genetic predisposition or a cultural preference. Also the sample sizes were large, external factors such as occurrence of mouth cancers made an impact on the surgeon general's conclusions.

A general improvement technique is to balance the allocation by some logical factors, so the all relevant categories of samples are represented. E.g., in medical studies would pair people of similar age, overall health, etc. As long as apparent biases are avoided, such studies are generally considered reasonably valid. Still, many cases are known where a follow-up experiment contradicts the results of the observational study, so need scientific care. Don't see the **counterfactual**—out of many possible outcomes, get to observe only specific ones and not the ones that haven't seen—e.g., how would your life have changed if you didn't go to college. Studying association between variables in an exploratory way is always justified. The idea is that only what happened can be observed, not what didn't happen, and detecting association only uses for former. See Imbens & Rubin (2015) for more. In some cases designs are **quasi-experimental**—can't use random sampling or assignment, but try to randomize and balance whatever can.

21.34 Markov Chain Monte Carlo

For distributions without known generators, feasible and sometimes the only available generation methods use the **(MCMC)** techniques. They produce slightly dependent and identically distributed samples, which is the next best thing when iid samples are unavailable.

Random walk Metropolis algorithm (RWM) requires PDF f of the distribution, specified up to a normalization constant:

1. Start from an initial sample x_0 such that $f(x_0) \neq 0$
2. Use a user-provided sampler p to generate a sample
3. $x_{\text{new}} = x + \text{the sample}$
4. Set $x \leftarrow x_{\text{new}}$ if $\text{uniform}(0, 1) \leq f(x_{\text{new}})/f(x)$

Theoretically, p must be such that the PDF > 0 everywhere and symmetric. E.g., p can be normal with mean 0 or fat-tailed symmetrized Levy. Then x_i form a Markov chain, whose eventual distribution is f . This is because the chain is **ergodic**, and it satisfies the **detailed balance** condition by construction—see Kroese et al. (2011) for the exact definitions of these and some theorems. The rate of convergence matters and depends on unknown constants (Efron & Hastie 2016). Can test convergence with Kolmogorov-Smirnov and is eventual because the initial x_0 may be very unlikely and on average lead to more rejections of the test than expected. So the rule of thumb is to discard some number, usually 1000, of the first x_i .

Though it's almost never an issue, technically the chain is never ergodic because numerical error causes p values to become ≈ 0 in the tails, particularly if p has thin tails. So in practice can get away with large-range uniform (which isn't > 0 outside the bounds). Theoretical guidance on picking p and its variance (or an equivalent parameter) is limited, but it should be such that the acceptance rate is far from both 0 and 1. To remove correlation between consecutive samples can discard some after the last used sample, but it's unclear how many.

A simple heuristic to solve these problems is to let the chain find the correct distribution using grid search (see the "Numerical Optimization" chapter). To generate a new sample, sample from a uniform dis-

tribution with very small variance, and double it to very large. Small variance proposals are mostly accepted but make no difference, and large-variance ones mostly rejected. So only use correct-magnitude variances, and take several steps to reduce correlation of the resulting samples. The number of calls to f is still $O(1)$ due to $O(1)$ difference in logs of variance bounds.

```
template<typename PDF> class GridRWM
{
    PDF f;
    double x, fx, aFrom, aTo;
    int from, to;
    double sampleHelper(double a)
    {
        double xNew = x + GlobalRNG().uniform(-a, a), fxNew = f(xNew);
        if(fx * GlobalRNG().uniform01() <= fxNew)
        {
            x = xNew;
            fx = fxNew;
        }
        return x;
    }
public:
    GridRWM(double x0 = 0, PDF const& theF = PDF(), int from = -10,
            int to = 20): x(x0), f(theF), fx(f(x)), aFrom(pow(2, from)),
            aTo(pow(2, to)) {}
    double sample()
    {
        for(double a = aFrom; a < aTo; a *= 2) sampleHelper(a);
        return x;
    }
};
```

For $D > 1$, generate a multidimensional uniform sample, but, because the chance of a rejection increases with D , an additional heuristic is to use the multiplicative factor $2^{1/D}$ for the grid. Due to only factor- D increase in runtime, this scales to large D .

```
template<typename PDF> class MultidimGridRWM
{
    PDF f;
    Vector<double> x;
    double fx, aFrom, aTo, factor;
    Vector<double> sampleHelper(double a)
    {
        Vector<double> xNew = x;
        for(int i = 0; i < xNew.getSize(); ++i)
            xNew[i] += GlobalRNG().uniform(-a, a);
        double fxNew = f(xNew);
        if(fx * GlobalRNG().uniform01() <= fxNew)
        {
            x = xNew;
            fx = fxNew;
        }
        return x;
    }
public:
    MultidimGridRWM(Vector<double> const& x0, PDF const& theF = PDF(),
                     int from = -10, int to = 20): x(x0), f(theF), fx(f(x)), aFrom(pow(2,
                     from)), aTo(pow(2, to)), factor(pow(2, 1.0/x.getSize())) {}
    Vector<double> sample()
    {
        for(double a = aFrom; a < aTo; a *= factor) sampleHelper(a);
        return x;
    }
};
```

For very large D grid search may not give independent enough samples, so it's best to test its outputs for any new distribution by checking whether sample-based estimates of some properties converge often

enough. Reducing the multiplication factor is one way to reduce correlation, but due to the curse of dimensionality this is unlikely to help without becoming inefficient.

21.35 Bayesian Methods

Many problems are perceived intuitively as well-defined but, when specified mathematically, have no best sensible solution. Such problems are **ill-posed**, i.e., lack some necessary information. In statistics usually get ill-posed problems when don't have enough data. E.g., consider estimating s^2 from a single example.

The main solution approach is assuming something that should hold naturally, but isn't part of the problem specification. Such assumptions can make a problem solvable but result in a somewhat different problem. But for many problems can't do inference without some assumptions, and it's better to make assumptions and produce a useful model than not be able to do anything at all.

Frequentist statistics means looking at the world based on some iid data and not knowing the reality. Bayesian logic allows modeling beliefs about the reality, taking point of view of a decision maker. Assuming belief \approx reality is a modeling assumption, like that a distribution is normal. So, e.g., in a frequentist confidence interval the bounds are random and θ fixed, but in **Bayesian credible interval** θ is random and bounds fixed. The former says that with high probability, the bounds are close to θ ; the latter that θ is close to the bounds.

The basic algorithm for Bayesian inference:

1. Pick an uninformative prior distribution for the data and a likelihood distribution for the parameters
2. Try to form the posterior distribution analytically using Bayes theorem
3. If formed, make inference from it analytically
4. Else make inference from samples simulated using MCMC

This only applies to probabilistic models and isn't much different from frequentist inference. Some points:

- The prior usually introduces a small bias but reduces variance. Bayesian methods are well-advised to use uninformative, **objective priors** for good science (i.e., deriving reproducible results). Using **subjective priors**, based on elicitation of domain knowledge is frowned upon in statistical inference because even with care and good intentions can't get rid of certain arbitrary inputs, lose reproducibility, and with strong enough prior knowledge don't need to perform the experiment to begin with—just use the prior as the posterior.
- The prior sometimes makes ill-posed problems well-posed or may allow solving problems that aren't tractable or are very clumsy analytically with frequentist methods. One example is **hierarchical models** (Gelman et al. 2013), where some inputs to the model are themselves a result of a sub-model. Though Bayesian logic is natural here, sometimes such models can be easily solved with frequentist logic by using the EM algorithm on a mixture of distributions (see the "Machine Learning—Clustering" chapter for EM with a mixture of Gaussians).
- The prior enables using Bayes theorem, which can substantially simplify inference in some cases—even though usually the prior is at best estimated from other information such as previous studies. E.g., for the OCBA derivation (discussed later in the chapter), working from Bayes theorem is much simpler.
- When the posterior is known analytically, both frequentist maximum likelihood and posterior inference tend to be consistent (Wasserman 2004). If don't know the posterior analytically, also lose computation efficiency.
- Bayesian methods need a probabilistic model and don't have equivalents of various nonparametric tests, though \exists nonparametric Bayesian methods.

See Efron & Hastie (2016) and Gelman et al. (2013) for more on Bayesian methods and Samaniego (2010) for a comparison of frequentist and Bayesian estimators. One particular challenge for the latter is high-dimensional spaces, where the prior effectively suppresses the data.

In particular, Bayesian methods often give good estimates from frequentist point of view. In Bayesian view they are automatically correct by being derived from certain logical axioms, but this is true only given correct prior and its weight. As pointed out by Neyman (1977), frequentism was discovered by the first crook who figured out to profit from loading a die for chance games. In such situation only repeated experimentation and no amount of belief can reveal favorable properties. So frequentist properties like consistency are important. Effectively Bayesian statistics is like maximum likelihood—a principle for deriving good estimators. Credible intervals can also be used as an algorithm for confidence intervals, with subsequent frequentist validation, but this is less popular than for estimates. So they appear to be useful only for

decision making outside of statistics. Any attempt to evaluate performance for a particular situation must be frequentist, though a Bayesian estimate might win.

Making decisions for an individual is very different from estimating population parameters because the posterior distribution is usually known reasonably well. Here Bayesian logic shines—e.g., if a medical test for an individual has a significant outcome for some condition, by Bayes's theorem it's very unlikely that the individual has the condition if it's very rare. Bayesian logic also answers the quintessential problem of why not try everything in life, including some potentially harmful or dangerous things to get an opinion on them—strong prior knowledge means that new “data” is unlikely to affect it by much. Also, certain events, such as who will win a particular presidential election, don't have associated probability distribution from a frequentist viewpoint because they occur only once, but Bayesian methods model beliefs, and this causes no conceptual flaw. But a particular estimation method (which might include polling strategies, etc.) can be evaluated in frequentist logic for many presidential elections.

In statistics estimation is usually only a part of the problem. The other part is assumptions and how to use the estimate. So with regard to estimation the difference is that frequentists look at other evidence after analysis and Bayesians before. E.g., in the stock market estimate the unknowns using frequentist methods, and use Bayesian logic to decide what to bet on. And many of these bets go through regression testing based on known investment history to show good frequentist properties.

21.36 Finding Best Alternative via Simulation

With $k = 2$, to decide if one alternative is statistically better, simulate each many times (at least until the normality assumptions becomes reasonable). If do this with a budget, and test at the end, compute k confidence intervals with multiplicity k .

But it may be more efficient to simulate until discover a winner with confidence, hopefully before run out of budget. This adds extra multiplicity due to potentially omitted future comparisons. A efficient way to compare is to do so after doubling the number of simulations. This increases multiplicity by only a factor $O(\lg(\text{budget}))$. The implementation assumes all multiplicity is put in a .

```
bool isNormal0BestBonf(Vector<NormalSummary> const& data, double aLevel)
// smallest is best with precision meanPrecision
{
    int k = data.getSize();
    assert(k > 1);
    double z = find2SidedConfZBonf(k, 1 - aLevel),
           upper = data[0].mean + z * data[0].stddev();
    for(int i = 1; i < k; ++i)
    {
        double lower = data[i].mean - z * data[i].stddev();
        if(lower <= upper) return false;
    }
    return true;
}
```

This naive approach wastes simulations by giving them to alternatives with small variances and non-competitive means. They need up to k times fewer simulations if use the same termination criteria, which gives enough room for improvement.

Bayesian logic if useful for deciding the next alternative to simulate, mostly because given prior beliefs, multiple testing and stopping rules are ignored. In particular, given best alternative with mean m_0 , deem it best if $\Pr(m_0 < m_1 \cap \dots \cap m_0 < m_{k-1})$ is high enough. Applying Bonferroni inequality, this $\geq 1 - \sum_{i>0} (1 - \Pr(m_0 < m_i))$. When $m_i \sim \text{normal}$, e.g., in case of simulation with many enough samples, deduce $\Pr(m_0 < m_i)$ from the normal difference between alternatives 0 and i , as for $k = 2$.

The **OCBA heuristic** based on this is simple and one of the most efficient. OCBA theorem (Chen & Lee 2010): If, as a result of initial simulations, $m_i \sim \text{normal}$, with variances s_i^2 , the number of additional simulations T needed to reach Bonferroni confidence p is minimized as $p \rightarrow 1$ when the relative allocation ratios

$$\text{are given by } R_i = \begin{cases} s_0 \sqrt{\frac{R_i^2}{\sum_{i>0} \frac{s_i^2}{s_i^2}}}, & i=0 \\ \frac{s_i^2}{(m_i - m_0)^2} & i>0 \end{cases}$$

So heuristically give the next simulation to the alternative with the highest R_i . E.g., if currently have $\text{normal}(1, 0.1)$, $\text{normal}(2, 0.2)$, $\text{normal}(3, 0.3)$, then $R_1 = 0.2$, $R_2 = 0.075$, and $R_0 \approx 0.15$, so choose alternative 1.

1. Simulate each alternative n_0 times to get the initial m_i and s_i
2. Until out of budget or have a winner with adjusted confidence
3. Simulate the alternative with the highest R_i

n_0 should be ≥ 30 to use the normal distribution and for the CLT to take reasonable effect. For applications where simulations are very expensive, use $n_0 \geq 5$ (Chen & Lee 2010). The runtime per iteration is $O(k)$.

```
template<typename MULTI_FUNCTION> struct OCBA
{
    MULTI_FUNCTION const& f;
    Vector<IncrementalStatistics> data;
    int nDone;
    OCBA(MULTI_FUNCTION const& theF = MULTI_FUNCTION(), int initialSims = 30):
        f(theF), data(theF.getSize())
    {
        int k = f.getSize();
        for(int i = 0; i < k; ++i)
            for(int j = 0; j < initialSims; ++j) data[i].addValue(f(i));
        nDone = k * initialSims;
    }
    pair<Vector<NormalSummary>, int> findBest()
    {
        int k = f.getSize();
        Vector<NormalSummary> s;
        for(int i = 0; i < k; ++i) s.append(data[i].getStandardErrorSummary());
        int bestI = 0, bestRatioI = -1;
        double bestMean = s[0].mean, ratioSum = 0, bestRatio;
        for(int i = 1; i < k; ++i)
            if(s[i].mean < bestMean) bestMean = s[bestI = i].mean;
        swap(s[0], s[bestI]);
        return make_pair(s, bestI);
    }
    void simulateNext()
    {
        pair<Vector<NormalSummary>, int> best = findBest();
        int k = f.getSize(), bestI = best.second, bestRatioI = -1;;
        Vector<NormalSummary> s = best.first;
        //compute the largest OCBA ratio
        double bestMean = s[0].mean, ratioSum = 0, bestRatio;
        for(int i = 1; i < k; ++i)
        {
            double meanDiff = s[i].mean - bestMean, ratio =
                s[i].variance / (meanDiff * meanDiff);
            ratioSum += ratio * ratio / s[i].variance;
            if(bestRatioI == -1 || ratio > bestRatio)
            {
                bestRatio = ratio;
                bestRatioI = i;
            }
        }
        double ratioBest = sqrt(ratioSum * s[0].variance);
        if(ratioBest > bestRatio) bestRatioI = bestI;
        else if(bestRatioI == bestI) bestRatioI = 0;
        //simulate the largest ratio alternative
        data[bestRatioI].addValue(f(bestRatioI));
        ++nDone;
    }
    int simulateTillBest(int simBudget = 100000, double aLevel = 0.05)
    {
        assert(nDone < simBudget);
        int k = f.getSize(), nTests = lgCeiling(simBudget) - lgFloor(nDone);
        while(nDone < simBudget)
        {

```

```

        simulateNext();
        if(isPowerOfTwo(nDone) || nDone == simBudget - 1)
        {
            Vector<NormalSummary> s = findBest().first;
            if(isNormal0BestBonf(s, aLevel/nTests)) break;
        }
    }
    return nTests;
}
}

```

E.g., given 6 alternatives, let the true performance of alternative i (starting from 1) be given by $\text{normal}(i, 10 - i)$. OCBA and the naive approach respectively took on average about 11000 ± 1000 and 29000 ± 2400 simulations (based on 10000 repetitions) to find the best with 0.95 confidence. From run to run, alternative 0 is always correctly selected, but the total number of simulations for both approaches varies substantially.

If several best alternatives have the same $E[x]$, OCBA will exceed the budget trying to distinguish between them. An interesting extension (not implemented here) is to introduce a mean indifference parameter ϵ , such that the user doesn't care if the picked alternative is worse than the best by ϵ . OCBA would use $m_0 - \epsilon$ instead of m_0 when computing R_i and compare for best accordingly. The number of needed simulations is unknown, but in the worst case, when all alternatives have the same $E[x]$ and large s_i^2 , should needn't-particularly-efficient $O(\sum s_i^2 \epsilon^{-2})$ simulations, based on how the Monte Carlo confidence is calculated.

The most useful aspect of OCBA seems to be allocation of the next simulation given a fixed budget. For values $\in [0, 1]$ UCB criteria (see the "Machine Learning—Other Tasks" chapter) is more popular, but it's unclear how it compares to OCBA. The comparison and the naive algorithm extend to finding several best alternatives, but OCBA doesn't.

21.37 Sample Size Calculations

Before doing a statistical experiment, usually need to specify its goals to justify the cost. In particular, need to know how many samples are needed for the wanted conclusions. This isn't crucial for computer experiments, where extra samples are essentially free, but is, e.g., for medical studies with human subjects.

A rule of thumb is that hypothesis test power should be about 80% (Ryan 2013). The problem is that even for parametric methods such as confidence intervals for the mean under normality assumptions, need to have prior estimates of the variance and decide the size of significant difference. Using such Bayesian thinking, many formulas have been developed for various procedures (Ryan 2013).

Instead of the formulas, a simple but computationally intensive process is using simulation. Setup a few what-if scenarios where expect to detect a certain difference, and use exponential search to find n , simulations with which approximately lead to 80% power or wanted confidence interval length. This is very general and applies even to nonparametric procedures. Still need to decide which distributions to simulate from—and it's usual for estimates to be off by even a factor of 5 (Ryan 2013). Usually normal is a good candidate, but may want to include some fat-tail ones such as Cauchy and asymmetric ones such as lognormal.

It's more efficient to do the calculation for a confidence interval if deal with a test. Use the confidence interval for its artificial statistic (e.g., the average rank in case of Nemenyi test). This way don't depend on the null value, removing all parameters from the simulation.

21.38 Time Series Analysis

Statistics generally studies iid data, but much of common data is dependent to a degree that can't be ignored. A particular type of dependence is temporal, i.e., when the data are ordered in time. E.g., consider end-of-day stock price for a company x_t as is evolves from x_0 after the initial public offering to the latest market price x_n .

There are several simple strategies to reduce the analysis to familiar methods:

1. Calculate the differences in the data $\Delta x_t = x_t - x_{t-1}$, and assume they are iid, enabling regular statistical inference. This is a useful model in many cases such as the stock market, which is known to react to mostly to new information. But it assumes that x_{t-1} is noise-free, which isn't the case because it's random. So a potentially better model is to take instead of x_{t-1} maybe an average of the last few x_i , just like the nearest neighbor regression.
2. Assume x_i are a function of time i , and use regression (see the "Machine Learning—Regression" chapter). While this will show a general trend, it won't show much else.
3. Ignore the time series structure and model x_i as a function of other information. While a good

approach, this is less effective if part of the behavior of x_i is dependent on the previous values to some degree.

The time series literature is almost exclusively focused on linear models of the form $x_{t+1} = \sum_{0 < i < p} \phi_i x_{t-i} + w_{t+1}$, where all noise terms $w \sim \text{normal}(0, \sigma)$. These a generalization of (1) where $\Delta x_t = w_{t+1}$ and is a function of the past p data points. This is called **autoregression (AR)** because the x_{t-i} are regressed on themselves. A good introduction is Hyndman & Athanasopoulos (2021), who use a slightly different notation.

While nonlinear models of these values are also possible, they haven't proven to be better. This might come as a surprise because, e.g., random forest regression is usually much better than linear regression, but the recursive structure of autoregression makes a big difference:

- Stability of the recursion is an issue. E.g., the Fibonacci series has AR form, and the values increase without bound. On the other hand, something like $x_{t+1} = 0.9x_t$ decays to a stable value. In general growth or decay is exponential and depends on the largest root in absolute value of the polynomial represented by the AR model (or the equivalent eigenvalue of the corresponding matrix—see the "Numerical Methods—Working with Functions" chapter). For nonlinear models don't have an explicit polynomial, but can generally use local linear approximations such as Taylor series to figure this out numerically.
- The recursive model isn't designed to have much predictive power—it's more to remove the dependence structure. A **dynamic regression model** with then use another, likely nonlinear model to model the noise from the AR model.

Estimation of the AR model and its generalizations is the main task in the literature. It might seem obvious to minimize the sum of squares of the residuals for a sample, where given current values of ϕ_i calculate residual_i = predicted $x_{t-i} - x_{t-i}$. Here can skip the first p values (**conditional sum of squares**) or assume they previous $x_i = 0$ (**unconditional sum of squares**). But a couple of issues with this approach:

- The time series process is infinite in the past—this is the only way for concepts like consistency of estimation to make sense. So the previous $x_{t-i} \neq 0$. While not directly a problem for AR estimation, it is for the more general ARMA (discussed later)
- The minimum of the least squares objective isn't necessarily the value giving the best estimate. Least squares minimization derives from the Gaussian likelihood maximization, and here the likelihood is different because of dependencies in x_i (discussed later). So prefer direct likelihood maximization, even though least squares is generally consistent.

A common generalization of AR is to also include a **moving average (MA)** component. Get $x_{t+1} = \sum_{0 < i < p} \phi_i x_{t-i} + \sum_{0 < i < q} \theta_i w_{t-i} + w_{t+1}$, leading to an ARMA(p, q) model. The noise terms are relative to the AR model, like for boosting (see the "Machine Learning—Regression" chapter). A pure MA process is also possible, in which case the noise terms are the part of the data not predicted by previous noise terms. MA processes have some plus and minuses:

1. Theoretically, a stable MA process can be converted to a stable MA process and vice versa. Also using both offers some advantages.
2. The noise terms are only observed in simulated data. They can't be observed in real data and have to be estimated. And estimation needs not p or q terms but the entire sample, making both estimation and prediction with a fit model computationally inefficient and clumsy.
3. Some sources advocate using a pure AR model over and ARMA or an MA model. It usually needs to have p larger than what $p + q$ would otherwise be but avoids (2).
4. A compromise used here is to use ARMA but limit history in prediction or residual calculation to 100 or so for large n . The justification is that for a stable process the effect of any particular noise term on future x_{t-i} is exponentially decaying, and 100 steps is usually enough to make it negligible. A more complicated and computationally expensive technique such as doubling could be more flexible but isn't considered further.

Because of the assumed stability of the process, assume also without loss of generality that is has 0 mean for simplicity. For fitting preprocess the data to subtract the sample mean to make this accurate.

Deriving the ARMA likelihood function is nontrivial and involves a lot of math and new concepts. The time series literature is not consistent in presentation due to several ways of doing this, with some historical and some based on newer concepts. A clean, modern presentation at the graduate level is Shumway & Stoffer (2000), on which all calculations here are based. Some ideas:

1. The exact likelihood is problematic to calculate because the data are dependent. So factor it as conditional likelihood, log of which = $\sum l(\text{predicted } x_{t-i} | \{x_{t-i}\})$, to condition on previous data points. This is simpler and more computationally efficient to work with.
2. Intuitively, a residual $\sim \text{normal}(0, \text{some std})$ by the Gaussian model (other distributions can be used also in principle, but this usually only gives extra complexity). This leads to the usual conditional

least squares regression. For an AR model the conditional likelihood leads directly to conditional least squares.

3. For an MA or ARMA process (2) doesn't quite work because of the dependence on more than p or q past observations which affect the noise terms. The very first, least recent data point in the sample has standard deviation $> \text{std}$ because the unobserved prior data points and noise terms affect it in addition to its own noise term. So the squared residual is adjusted by a factor of $1/r_{t-1}$, the calculation of which and the resulting likelihood are discussed later. These values are large for the first few least recent data points and converge to 1 for most recent ones. So the quality of both conditional and unconditional least squares depends on how fast this convergence happens. Naturally, for large samples the difference is likely negligible and for small ones not.
4. Because we use a likelihood-based information criteria such as BIC (the choice here, though many other sources use AIC and others), don't lose anything by working with likelihood minimization in all cases.

All ARMA code is in a class:

```
struct ARMA
{
    Vector<double> phi, theta;
    double std;
public:
    ARMA(): std(1){} //dummy constructor for fitting in future
    ARMA(Vector<double> const& thePhi, Vector<double> theTheta, double theStd):
        phi(thePhi), theta(theTheta), std(theStd){} //direct copy
};
```

A useful operation is converting a stable ARMA process to MA representation $x_{t+1} = \sum_{0 < i < \infty} \psi_i w_{t+1-i}$, which is necessarily trimmed at some logical point. This is done recursively by solving $\psi_i = \theta_i - \sum_{0 < k \leq i} \phi_k \psi_{i-k}$. Here $\phi_0 = 0$, as are ϕ and θ past p and q , respectively.

```
Vector<double> calculatePsi(int maxSize) const
{
    //first weight is 1, rest is given by relation
    Vector<double> psi(max<int>(maxSize,
        max<int>(phi.getSize(), theta.getSize() + 1)), 1);
    for(int i = 0; i < psi.getSize() - 1; ++i)
    {
        double sum = 0;
        for(int j = 0; j < min(i + 1, phi.getSize()); ++j)
            sum += phi[j] * psi[i - j];
        double thetai = i < theta.getSize() ? theta[i] : 0;
        psi[i + 1] = thetai + sum;
    }
    return psi;
}
```

The MA representation allows to calculate theoretical autocovariances. Instead of solving an equation system as suggested in Shumway & Stoffer (2000), use a simple, a good-enough numerical approximation based on the equation $\gamma(h) = \sigma^2 \sum_{0 \leq i \leq h} \psi_i \psi_{i+h}$.

```
Vector<double> gammaAll(int h) const
{
    //calculate using numerical approximation from psi
    Vector<double> gamma(h, 0), psi = calculatePsi(1000); //enough
    for(int i = 0; i < h; ++i) //order below sums smaller numbers first
        for(int j = psi.getSize() - 1 - i; j >= 0; --j)
            gamma[i] += psi[j] * psi[j + i];
    return gamma * (std * std);
}
```

Now get autocorrelations $\rho(h) = \gamma(h)/\gamma(0)$:

```
double rho(int h, Vector<double> const& gamma) const
{
    assert(h >= 0 && h < gamma.getSize());
    return gamma[h]/gamma[0];
}
```

A straightforward but programmatically complicated operation is prediction with a fit model. The unclear part is how much historical data to use with an MA model. The simple answer is to run through

whatever finite data is available to get a good estimate of the past q noise values. See Shumway & Stoffer (2000) for details.

Other than the natural next value prediction, want a confidence interval for it, and generalize to m future values. The approximate variance of the prediction is $\sigma^2 \sum_{0 \leq i \leq m} \psi_i \psi_{i+h}$. Logically this is because for the first prediction only have the variance of the noise, but for future prediction have extra variance due to some unknown past.

```

Vector<pair<double, double>> operator() //predictions and their variances
    Vector<double> const& xPast, int m) const
{ //based on truncated evaluation
    assert(m > 0);
    int p = phi.getSize(), q = theta.getSize();
    //the most recent value at index size - 1 in both
    //initialize with 0 unavailable past values
    Queue<double> xPastRolling(p), wPastRolling(q);
    for(int i = 0; i < p; ++i) xPastRolling.push(0);
    for(int i = 0; i < q; ++i) wPastRolling.push(0);
    //for prediction with last values
    double psiSum = 0;
    Vector<double> psi = calculatePsi(m);
    Vector<pair<double, double>> predictionsAndIntervals(m);
    //process relevant past data, then all future data
    for(int i = q > 0 ? 0 : max(0, xPast.getSize() - 1 - p);
        i < xPast.getSize() + m; ++i)
    { //both buildup and prediction
        double xPredicted = 0; //first evaluate prediction
        for(int j = 0; j < p; ++j)
            xPredicted += phi[p - 1 - j] * xPastRolling[j];
        for(int j = 0; j < q; ++j)
            xPredicted += theta[q - 1 - j] * wPastRolling[j];
        double xi = xPredicted; //prediction case
        if(i < xPast.getSize()) xi = xPast[i]; //past data case
        else
        { //prediction case
            int predI = i - xPast.getSize();
            psiSum += psi[predI] * psi[predI];
            predictionsAndIntervals[predI] =
                make_pair(xPredicted, std * std * psiSum);
        }
        double wPredicted = xi - xPredicted; //0 for prediction case
        //update queues
        if(p > 0)
        {
            xPastRolling.pop();
            xPastRolling.push(xi);
        }
        if(q > 0)
        {
            wPastRolling.pop();
            wPastRolling.push(wPredicted);
        }
    }
    return predictionsAndIntervals;
}
//xPast is in the usual order, least recent to most recent
double operator()(Vector<double> const& xPast) const
    (return operator()(xPast, 1)[0].first); //single truncated prediction

```

For fitting the model the residuals are called "innovations" in the literature. They follow from the prediction of the current observation based on any previous ones. For computational savings, limit the history size.

```

enum{HISTORY_LIMIT = 100}; //heuristic for computational savings, can
//also use doubling but more complicated

```

```

int desiredHistorySize() const
{ //P for AR models, and at least HISTORY_LIMIT for MA and ARMA models
    return max(phi.getSize() + theta.getSize(),
               theta.getSize() == 0 ? 0 : HISTORY_LIMIT);
}
Vector<double> calculateTruncatedInnovations(Vector<double> const& x) const
{
    Vector<double> innovations = x;
    for(int i = 1; i < x.getSize(); ++i)
    {
        int nPast = min(i, desiredHistorySize());
        Vector<double> xPast(nPast); //nPast values through i - 1
        for(int j = 0; j < nPast; ++j) xPast[j] = x[i - nPast + j];
        innovations[i] -= operator()(xPast);
    }
    return innovations;
}

```

The **partial autocorrelation function (PACF)** ϕ_{hh} is the correlation of residual h steps apart. The notation is overloaded to similar to that of the AR, but the distinction is usually clear. The calculation is defined by a linear equation system with a structured matrix, and is solved by a special algorithm for efficiency (see Shumway & Stoffer 2000 for details).

```

Matrix<double> calculatePACFHelper(int n, Vector<double> const& gamma) const
{ //Durbin-Levinson to solve matrix equation
    assert(n >= 0 && gamma.getSize() == n + 1);
    Matrix<double> PACF(n + 1, n + 1);
    PACF(0, 0) = 0;
    for(int i = 1; i < n + 1; ++i)
    {
        double sum1 = 0, sum2 = 0;
        for(int j = 1; j < i; ++j)
        {
            sum1 += PACF(i - 1, j) * rho(i - j, gamma);
            sum2 += PACF(i - 1, j) * rho(j, gamma);
        }
        PACF(i, i) = (rho(i, gamma) - sum1)/(1 - sum2);
        for(int j = 1; j < i; ++j) PACF(i, j) =
            PACF(i - 1, j) - PACF(i, i) * PACF(i - 1, i - j);
    }
    return PACF;
}
Matrix<double> calculatePACF(int n) const
{
    return calculatePACFHelper(n, gammaAll(n + 1));
}

```

The r values for the adjusting the likelihood are calculated recursively using $r_0 = \sum_{0 \leq i \leq \infty} \psi_i \psi_i$, and $r_i = r_{i-1} * (1 - \phi_{ii}^2)$.

```

double calculateR0() const
{ //infinite sum of squared psi, converges or diverges
    //1000 is typically more than enough, or can use doubling with
    //sum monitoring
    Vector<double> weights = calculatePsi(1000);
    double sum = 0; //loop order below matters for numerical stability
    for(int i = weights.getSize() - 1; i >= 0; --i)
        sum += weights[i] * weights[i];
    return sum; //for divergent this is huge or not finite
}
Vector<double> rValues(int n) const
{
    double r = calculateR0();
    Matrix<double> PACF = calculatePACF(min<int>(n, HISTORY_LIMIT));
    Vector<double> result;
    for(int i = 0; i < n; ++i)
    {

```

```

    if(i < HISTORY_LIMIT)
        //check for numerical error - likely process not causal
        if(r < 0) return Vector<double>(n,
            numeric_limits<double>::infinity());
        if(i > 0) r *= 1 - PACF(i, i) * PACF(i, i);
    }
    else r = 1;//limiting r value for a causal process
    result.append(r);
}
return result;
}

```

The log-likelihood is $\frac{-n}{2} \log(2\pi\sigma^2) - \sum \log(r_i) - \frac{S}{2\sigma^2}$, where $S = \sum \frac{\text{residual}_i^2}{r_i}$; r and S are functions of

the current values of ϕ and θ but not σ as long as $\sigma \neq 0$.

```

pair<double, double> LLHelper(Vector<double> const& x) const
{
    int n = x.getSize();
    double rSum = 0, SSum = 0;
    Vector<double> innovations = calculateTruncatedInnovations(x),
        r = rValues(n);
    if(!isfinite(r[0]) || log10(r[0]) > 3)//heuristic for inf
        //not a causal process if r0 diverged
        double inf = numeric_limits<double>::infinity();
        return make_pair(inf, inf);
    }
    for(int i = 0; i < n; ++i)
    {
        //r[i] = 1; uncomment to get unconditional least squares
        rSum += log(r[i]);
        SSum += innovations[i] * innovations[i]/r[i];
    }
    return make_pair(rSum, SSum);
}
double LL(Vector<double> const& x) const
{//for BIC
    pair<double, double> sums = LLHelper(x);
    //not a causal process if r0 diverged
    if(!isfinite(sums.first) || !isfinite(sums.second))
        return -numeric_limits<double>::infinity();
    int n = x.getSize();
    double temp = 2 * std * std;
    return -(log(temp * PI()) * n/2 + sums.first/2 + sums.second/temp);
}

```

This can be optimized directly, but a further improvement due to some math (Shumway & Stoffer 2000)

is to minimize the **concentrated likelihood** $\frac{-n}{2} \log(2\pi\sigma^2) - \sum \log(r_i) - \frac{S}{2\sigma^2}$. Then

$\sigma = S(\text{optimal params})/n$. Use local minimization (see the "Numerical Optimization" chapter).

```

double concentratedLL(Vector<double> const& x) const
{//for minimization
    int n = x.getSize();
    pair<double, double> sums = LLHelper(x);
    if(!isfinite(sums.first) || !isfinite(sums.second))
        return numeric_limits<double>::infinity();
    return log(sums.second/n) + sums.first/n;
}
static pair<Vector<double>, Vector<double>> unpackParams(
    Vector<double> const& params, int p)
{//params = phi(theta)std
    assert(p >= 0 && p <= params.getSize());
    Vector<double> thePhi(p), theTheta(params.getSize() - p);

```

```

    for(int i = 0; i < p; ++i) thePhi[i] = params[i];
    for(int i = p; i < params.getSize(); ++i)
        theTheta[i - p] = params[i];
    return make_pair(thePhi, theTheta);
}
struct LLFunctor
{
    Vector<double> const& x;
    int phiSize;
    double operator()(Vector<double> const& params) const
    { //std doesn't matter here as long as not 0
        pair<Vector<double>, Vector<double> > result =
            unpackParams(params, phiSize);
        ARMA a(result.first, result.second, 1);
        return a.concentratedLL(x);
    }
};

```

Because numerical minimization requires initial values, a simple approach is to set ϕ and θ to 0, corresponding to the pure noise model. A good, cheap initial estimator for the AR coefficients is based on using the empirical autocovariance as $\hat{y}(h) = \frac{1}{n} \sum x_{i+h}x_i$. From it can calculate empirical PACF.

```

Vector<double> gammaAllEmpirical(int n, Vector<double> const& x) const
{ //first n empirical autocovariances assuming mean 0
    assert(n > 0 && x.getSize() > 0);
    Vector<double> result(n);
    for(int i = 0; i < min(n, x.getSize()); ++i)
    {
        double sum = 0;
        for(int j = 0; j + i < x.getSize(); ++j) sum += x[j + i] * x[j];
        result[i] = sum/x.getSize();
    }
    return result;
}
Matrix<double> calculatePACFEmpirical(int n, Vector<double> const& x) const
{ return calculatePACFHelper(n, gammaAllEmpirical(n + 1, x)); }

```

AR process from PACF uses a linear uses $\phi_i = \phi_{p,i+1}$.

```

ARMA(int p, int q, Vector<double> const& x): phi(p, 0), theta(q, 0), std(1)
{ //fit the model; want more data than parameters
    assert(p >= 0 && q >= 0 && x.getSize() > p + q + 1);
    if(p + q > 0)
    {
        //uninformed initial solution - all 0
        Vector<double> x0(p + q, 0);
        LLFunctor ll = {x, p};
        //try Yule-Walker preliminary estimator for theta
        if(p > 0)
        {
            Vector<double> x1(p + q, 0);
            Matrix<double> pacfe = calculatePACFEmpirical(p, x);
            for(int i = 0; i < p; ++i) x1[i] = pacfe(p, i + 1);
            if(ll(x1) < ll(x0)) x0 = x1;
        }
        //maximum likelihood
        pair<Vector<double>, double> result =
            hybridLocalMinimize(x0, ll, 200); //don't need more
        pair<Vector<double>, Vector<double> > result2 =
            unpackParams(result.first, p);
        phi = result2.first;
        theta = result2.second;
        std = sqrt(LLHelper(x).second/x.getSize());
    }
}

```

```

    else
        {//noise model
            IncrementalStatistics s;
            for(int i = 0; i < x.getSize(); ++i) s.addValue(x[i]);
            std = s.stdev();
        }
    }
}

```

To choose p and q use the BIC criteria to search over all possible value up to some limits:

```

double BIC(Vector<double> const& x, int extraParams = 0) const
{
    int n = x.getSize(),
        k = phi.getSize() + theta.getSize() + 1 + extraParams; //1 for std
    assert(n >= 0 && extraParams >= 0);
    return k * log(n) - 2 * LL(x);
}

ARMA(Vector<double> const& x, int maxP = 5, int maxQ = 5): std(1)
{//select using smallest BIC based from 0 <= p, q <= maxOrder
    double bestBIC = numeric_limits<double>::infinity();
    for(int p = 0; p <= maxP; ++p)
        for(int q = 0; q <= maxQ; ++q)
            {//TODO - use prev model coeffs and 0 to init next model coeffs
                ARMA b(p, q, x);
                double bBIC = b.BIC(x);
                if(bBIC < bestBIC)
                    {//executes at least once unless NaN data
                        *this = b;
                        bestBIC = bBIC;
                    }
            }
}
}

```

The stability requirement is very restrictive on the universe of possible models. E.g., it doesn't allow to model trend or even a simple random walk. So a generalization of ARMA is **ARIMA**, with "I" for integrated, meaning the prediction need to reverse any difference operations applied to the data before fitting ARMA from it. Several computations must be adjusted to use the existing ARMA code:

1. A specified number d of differences are applied to the data recursively, which gives a ARIMA(p, d, q) model. Usually $d = 1$ is enough for practical data, though $d = 2$ sometimes gives better results. More is typically not even considered. E.g., with $d = 1$ the mean of the result is the linear trend.
2. ARIMA is also a good place to subtract the mean from the differenced data.
3. BIC with the number of parameters in ARMA increased by d still allow the same model selection because by the underlying MDL principle d is just another parameter that needs to be encoded.
4. The adjustment of predictions and their errors follows from the linearity of the normal model. I.e., for $d = 1$ just add the ARMA prediction and their variances to the last known predifference value.

An "ARI" model where $q = 0$ is likely to end up being an unstable AR process, so a natural question is why not fit an AR process to begin with using conditional least squares and avoid all the complexity. The answer is that estimation is much more accurate for a stable model due to the recursive nature of an AR process. And the process works for more general ARIMA as well. Also, a stable process doesn't change in the limit, which is important for properties such as consistency of estimation. A direct fit of a function to a data will just not have such properties, even though it's likely to be useful also.

```

struct ARIMA
{
    ARMA a;
    mutable double mean;
    int d;
    Vector<double> transformData(Vector<double>& x, bool computeMean) const
    {//difference
        Vector<double> xRemoved;
        for(int j = 0; j < d; ++j)
        {
            xRemoved.append(x.lastItem());
            for(int i = 0; i + 1 < x.getSize(); ++i) x[i] = x[i + 1] - x[i];
        }
    }
}

```

```

        x.removeLast();
    }
    if(computeMean)
    //estimate mean
    IncrementalStatistics s;
    for(int i = 0; i < x.getSize(); ++i) s.addValue(x[i]);
    mean = s.getMean();
}
//remove mean from data
for(int i = 0; i < x.getSize(); ++i) x[i] -= mean;
return xRemoved;
}

public:
ARIMA(int p, int theD, int q, Vector<double> x): d(theD)
{ //want more data than params
    assert(d >= 0 && p >= 0 && q >= 0 && x.getSize() > p + q + d + 3);
    transformData(x, true);
    //fit predictor
    a = ARMA(p, q, x);
}
ARIMA(Vector<double> const& x, int maxD = 1, int maxP = 5, int maxQ = 5)
{ //want more data than params
    assert(maxD >= 0 && maxP >= 0 && maxQ >= 0 &&
        x.getSize() > maxP + maxQ + maxD + 2);
    double bestBIC = numeric_limits<double>::infinity(), bestMean = 0;
    int bestD = 0;
    for(d = 0; d <= maxD; ++d)
        //BIC across d comparable because MDL compression still holds
        Vector<double> xCopy = x;
        transformData(xCopy, true);
        //fit predictor
        ARMA aNew = ARMA(xCopy, maxP, maxQ);
        double bic = aNew.BIC(xCopy, d + 1);
        if(bic < bestBIC)
        {
            a = aNew;
            bestBIC = bic;
            bestMean = mean;
            bestD = d;
        }
    }
    mean = bestMean;
    d = bestD;
}
Vector<pair<double, double> > operator()(Vector<double> x, int m) const
{//predictions and their variances
    assert(x.getSize() >= d + 1 && m > 0); //need at least 1 data point left
    Vector<double> xRemoved = transformData(x, false);
    Vector<pair<double, double> > predictions = a(x, m);
    //add back the mean
    for(int i = 0; i < m; ++i) predictions[i].first += mean;
    //integrate
    for(int i = 0; i < d; ++i)
        for(int j = 0; j < m; ++j)
        {
            predictions[j].first +=
                j == 0 ? xRemoved[d - 1 - i] : predictions[j - 1].first;
            predictions[j].second += //variances add up
                j == 0 ? 0 : predictions[j - 1].second;
        }
    return predictions;
}
double operator()(Vector<double> x) const return operator()(x, 1)[0].first;
}

```

```

double BIC(Vector<double> x) const
{
    transformData(x, false);
    return a.BIC(x, d + 1);
}
};

```

21.39 Using Statistics in Practice

Even for someone with substantial training, thoughtful use of statistics needs more than usual critical thinking and common sense—many statistical concepts aren't intuitive. Psychologically, it's easier to reason in terms of counts than probabilities (Gigerenzer 2011). Need care to not lie with statistics by accident, which happens often—see Hubert & Wainer (2012) for many examples. Imagine that get a data set, calculate the mean and its t -based confidence interval. But then find out that the samples are from a Cauchy distribution. Surprising as it may be, given just a sample without further context can do nothing with it (Huber 2011; Chatfield 1995).

Need to know the goal of the analysis:

- An existing study may already exist that answers the questions satisfactorily
- This may clarify the choices of procedure and statistic—such as mean vs median
- Get some idea of how many samples need to collect and what variables to measure

Need to get to know the data:

- The data, if already collected, might measure the wrong information or be obtained by an incompatible design. E.g., the following are common:
 - A standard data set is observational and not experimental.
 - Measurements may not be accurate enough—e.g., using time in seconds when want higher resolution.
 - Biases happen when collecting the data, e.g., by surveyors to streamline the process, or by respondents who don't want to be honest—might be able to anticipate some of these.
 - The data comes from a mix of several generation mechanisms and needs special aggregation or can't be aggregated at all—from the experience of Huber (2011) it's rare to get a large homogeneous sample.
 - The iid assumption may not be reasonable—here the independence is more important than the identical distribution because many real-world sequences don't have independence due to some learning or dynamic conditions. Also with tasks such as regression iid subsumes part of approximation error—e.g., if use a linear model for a nonlinear phenomenon with iid noise, in some areas the noise will be unnaturally larger than in others from the point of view of a linear model.
- Can get useful information about likely or unlikely distribution—but beware of data snooping
- Domain constraints can allow to detect impossible measurements and likely outliers

It's often a good idea to reserve a part of the data for exploration and another for subsequent validation of the final model. E.g., when try a questionable procedure on the first part, don't get cornered into snooping, and can try out another procedure as the final one of the second part. For very large data sets only automated analysis and preliminary checks are feasible, so should do whatever of the latter is feasible.

Finally, after the analysis is done, check the results against domain knowledge:

- Any statistics like confidence interval endpoints can't be outside allowable range values
- A computed model such as linear regression often doesn't have correct behavior in the limit by not matching the reality's stable state if any, which may be an issue

Computer scientists doing statistics must be aware of certain bad habits:

- Statisticians care about the meaning of the numbers and not the numbers. E.g., no point to show all possible digits of precision.
- The results of running algorithms are more important than the algorithms.
- Data is interesting in that it tells something useful about the parameters of the model and not just a boring artifact. E.g., if a single number looks too unusual, it's potentially an outlier, but will never notice if don't bother to look.

Statistics is vulnerable to lies because an interested party will try many methods and present the most favorable as the only tried one, committing multiple testing and data snooping. For publishing research in particular, the common requirement of $p \leq 0.05$ has lead to occasional data tweaking, and replicability is correlated to whether the authors respond to requests for data. Agencies such as FDA in the U.S. impose strict controls on the processes they manage to avoid bias.

Those who want to do statistical consulting should keep in mind the following (Adèr et al. 2008):

- A client's domain is likely to be unfamiliar and require learning of the basics to at least to able to present the results in their language
- Clients want a working solution, not a statistics lecture
- Often need a reasonably quick solution because don't have time to do proper research
- The communication aspect is probably the most important—in particular must not hesitate to criticize the client's ideas, or tell them that existing data isn't valid enough and will need expensive re-collection to bring to acceptable standards
- Many cases require complicated or imperfect experimental designs for data collection (not even considered in the book), particularly if the data comes from humans

In many applications of statistics, generic models do well. But for best results need to research specialized literature that uses domain-specific knowledge. E.g., significance of a fingerprint match strongly depends on prior knowledge about fingerprints, and "reasonable" domain-agnostic guesses are useless.

21.40 Analysis of Decisions

Given a decision problem, it's important to distinguish between **decision** and **outcome** (Howard & Abbas 2015). Either can be good/bad independently of the other. E.g., gambling your savings on a $\approx 50/50$ roulette bet and winning is a bad decision (assuming you are risk-averse like most people) with a good outcome.

Mathematically, best decisions maximize expected value assuming identity utility functions (see the "Optimization Algorithms" chapter). But expected values can be deceptive—e.g., a very rare event with a huge expected payoff seems attractive even though in practice it will never happen. Utilities can be hard to determine—something can be worth different amounts to different people at different times. E.g., when you buy a book, it's worth less after you read it. Also value may give way to risk, e.g., in investing give up some expected profit to reduce expected risk.

Almost all reasoning happens on Bayesian-like belief level that guides usually robust decisions, except without calculation of probabilities. Despite extensive developments in logic, statistics, game theory, economics, etc., the instinctual belief reasoning is still very applicable. Still, intuitive human judgment is known to be irrational when psychological effects take over, so, e.g., important decisions need more than just a good feeling. Most decisions tend to depend substantially only on few variables. But it's usually better to invest into getting more information than do more thinking with existing information.

Another instinctual concept is **regret**. People hate the feeling that a made decision missed a better opportunity. In most cases the decisions have no noticeable costs, so belief reasoning works well. But for major decisions such as career choice need much more care. A rational decision in possibly irrational belief world usually leads to no regrets, but only if the decision and the outcome are properly distinguished.

Consider allocation of investments for an individual. A rule of thumb is to do as the pension funds do (analogously to "do as the integers do" in object-oriented programming). i.e., invest in a diversified way, by picking not a single stock or bond but many (using an ETF). This reduces the risk from that of a single investment to that of the market. Even intuitively won't regret committing to any single investment. Another common decision is when to adapt to a particular new technology. It seems best to have a cautious policy—let others validate the novelty first. They may win a lottery, but the expected gain is usually better with waiting.

Decisions needn't be justifiable. It's tempting to be able to defend many decisions, particularly from scientific points of view, but in many cases must make a quick judgment call. As a result of a long sequence of judgment calls, things are the way they are because they got that way. E.g., most aspects of the current U.S. court system were determined hundreds of years ago, without any formal decision tools or with only minimal ones such as simple logic. But the idea of having jurors consider evidence and make the final decision based on their beliefs is heuristic, and with an important outcome.

In some cases the variables that want to measure can't be measured directly. The trick is to use indirect measurements, i.e., measure related quantities, and use the relationships to deduce an estimate. A famous example is the interview question about how much it costs to move Mount Fuji. Could use its height and radius to estimate the volume, stone density to estimate the weight, and advertised truck rates to estimate cost. Given a Bayesian prior on each of these, can even get a credible interval on the result. A more real-life task is to measure the value of good customer service to a company. Need to research known information such as perhaps sales rates to confirmed happy customers and the rest. The main difficulty is finding enough useful indirect information. See Hubbard (2014) for more.

An important model for sequential decisions is the **secretary problem**. Suppose you will interview 100 candidates for job one by one, have to give a yes/no answer to each, and can't change the decision. Mathe-

matically (Wikipedia 2016c), to maximize the expected quality of the hired person, the optimal strategy is to reject the first $1/e$ fraction, and hire the first that is better than any of them, or the last one if none. So for $n = 100$ the first 37 are rejected. This logic extends to many other situations, in particular time range by the limiting discretization. E.g., if want to buy a home within the next 3 years, during the last 2 buy the first that's better than any seen in the first year. But may prefer a more robust strategy and **satisfice**, i.e., not look for a better option if find a very good one.

Many entities make self-interested claims. The key aspect is **falsifiability**—most such claims can't be disproved because it's impossible or needs unreasonable resources. E.g., business "care" is unfalsifiable.

Some decisions have a time component, particularly which task to work on next. When swamped with tasks, a useful strategy is **Eisenhower diagram**—classify tasks into important/unimportant and due soon/due later (Wikipedia 2017b). Important and due soon get priority, but then give priority to important/due later over unimportant and due soon. As tempting as it may be to get the latter done, it's best to sacrifice them to make sure that nothing important is at the risk of being missed, and can work on the unimportant if have time later.

Much logic for decision making comes from economics, particularly about selecting products to buy among many choices (though the setup is general). The **monopolistic competition** model suggests that all items are slightly different because different manufacturers almost never make exactly the same choices. This tends to be more true in practice—even for the classic commodity soap, some are for babies, some have different scents, and some have certain minerals and other natural ingredients. So consider **features** of various items along with the price. Some matter and others don't, and don't want to overpay for the latter. Technically the correct approach is to form the **Pareto-optimal frontier** (see the "Optimization" chapter) of items that aren't worse than others in wanted features and price, and pick any of them. But usually want to save time and greedily look for best items and don't try to figure out which low-quality items are the cheapest.

The iid assumption isn't justifiable in many practical cases. A common one is when trying to determine an average product review when trying to compare before buying something. Here have an adversarial component in that effectively sample from a mixture distribution of fair ratings, ratings from interested parties, and rating from competitors. It's not hard to create 50 or so excellent ratings for an interested party.

Other decision strategies from economics include game theory models. A typical case that isn't modeled mathematically is a zero-sum game between very skilled professionals, such as fictional Dumbledore vs Grindelwald, to real-life chess, boxing, tennis, and professional negotiations. In any of these, any strong player can hold a reasonable game against any much better player, though most likely will eventually lose. But the loss will likely happen because of many small missteps and not a single major blunder. From a statistical point of view, for the weaker player to not allow "a shade more skillful" of the stronger player materialize into a win, want to add more variance to create uncertainty as to which expected value is larger. Likewise, the stronger player prefers to make safer, low-variance moves (I have confirmed this through many chess games against grandmasters; there is also a substantial psychological effect to this, such as fear for the weaker player and desire for certainty for the stronger one). But this is probably all there is for the application of statistical and Bayesian-like thinking for these kind of models. At best can create some probabilistic profile of human behavior with regard to the particular situation.

21.41 Implementation Notes

Among the estimators, the sample quantile is the hardest to implement due to the choices that must be made in solving ties. My implementation is the simplest and, I will claim, as good as any other.

Among confidence interval procedures, the bootstrap ones are by far the most complicated due to the choice of the procedure and any robustness logic required. I wasn't even able to settle on a single best procedure among the 2nd-order intervals, though it reasonably clear that the mixed interval is the method of choice by default.

The implementation of inverting permutation tests to form confidence intervals using common random numbers to get continuity is heuristic but apparently original.

Other procedures closely follow textbook descriptions or those from primary literature. A major difficulty was that need many sources for almost every subtopic.

21.42 Comments

The most formal approach to probability is using **measure theory**. Not all subsets of Ω can be assigned a probability, so restrict allowed events to a large collection such as all intervals. In practice this is an unnoticeable technicality, which most applications of probability theory don't need. Effectively all results remain

the same and are just stated more generally. One of the main differences is that every distribution is a mixture of continuous, discrete, and delta, and some mixtures are more complicated than others. But it's important to be aware of the theory, and need familiarity if want to understand many research papers or be able to study proper mathematical proofs. Most main ideas of statistics appeared before measure-theoretical probability was formalized, and have been restated formally, though in some cases with difficulties. A good study process would involve reading the following, in order:

1. Undergraduate statistics: Westfall & Henning (2013), Wasserman (2004), Kiefer (1987), and optionally Panaretos (2016)
2. Measure theory: Johnston (2015), Nelson (2015), and Katzourakis & Varvaruca (2018)
3. Measure-theoretic probability: Proschan & Shaw (2016), Florescu (2014), and Stoyanov (2013)
4. Measure-theoretic statistics: Shao(2003) (see also his lecture slides at <http://pages.cs.wisc.edu/~shao/>), Lehmann & Casella (1998), and DasGupta (2008)

Among very many other choices these I have found to be the most readable, but still currently ! \exists easy books on these, particularly on (4). Going further, the basic idea is that want an efficiency criteria that isn't hacked by **superefficient estimators** that have good behavior at the wanted point at the expense of neighboring points. So look at a small asymptotic neighborhood of size maybe $1/\sqrt{n}$, adjusted for a particular situation. Using this, good estimators come out better, as common sense says they should. For a readable introduction see the first chapters of Korostelev & Korosteleva (2011). Unfortunately this theory is usually presented at a much more abstract level, perhaps only useful to those who turn coffee into theorems. My own knowledge essentially ends here, so I won't recommend any other books.

The argument that Cramer-Rao lower-bounds other loss functions under some conditions is original, though somewhat obvious.

To aggregate risk, another way, which is more similar to numerical analysis with polynomials, is something like Bayes risk but using the squared loss. This puts more weight on large values. Many other extensions are possible but not explored in the literature.

The (two-sided) **empirical Bernstein inequality**, which applies in the same case as Hoeffding's, leads to the confidence interval $\mu \in \bar{x} \pm \frac{\sqrt{2(n-1)s \ln(3/p) + 3\ln(3/p)}}{n}$ (Audibert et al. 2009). But the $O(n^{-1})$ term

makes a big difference. E.g., from a simulation from low-variance Bernoulli(0.9), need $n \approx 600$ to win over Hoeffding's. Here even for $n = 10^6$ the latter is only $\approx 55\%$ longer, but for $n = 30$ it's factor ≈ 0.44 of the length. With a bounded-range variable and such large n , even the CLT is very accurate. So empirical Bernstein is useless practically (and theoretically due to being harder to work with in proofs).

Davies (2014) takes the idea of distance-based estimation to a new level and proposes **approximation intervals**—only values that pass DKW or similar distance-based hypotheses tests are to be considered as forming a valid interval. Unlike a confidence interval, this takes approximation error into account, so for models with larger approximation error, the computed interval may contain no values. Though this is conceptually useful, and potentially answers many questions about validity of regular confidence intervals under slightly wrong models, the difficulty is mostly computational:

- Approximation intervals must be computed numerically
- Don't have good distance tests for $D > 1$
- Using binary search with random data generation leads to holes in the intervals unless very cleverly use common random numbers

It seems that the essential useful idea behind this proposal is to look at predictive properties of the models and not confidence intervals, as in done in machine learning.

Prediction and tolerance intervals are useful for certain applications. E.g., when look up driving directions, usually get a range for how long the trip might take. This isn't a confidence interval on the average duration. When studying performance of randomized algorithms, such as hash tables, the difference between possible values and averages is apparent, and the uncertainty of the average is much smaller. A tolerance interval is easier to think about. E.g., for the driving can use 10th and 90th quantiles for the particular trip duration to give 80% prediction/tolerance interval. If know these exactly, then the intervals are the same. But if estimate the quantiles from recent driving activity, can use lower and upper one-sided confidence bounds respectively, and adjust for multiplicity. This would give an 80% tolerance interval with 95% confidence. A prediction interval would be different because it's supposed to be recomputed with every new data point and the 80% error would apply to the prequential error from online testing, which is more complicated to think about, less flexible, and often less useful despite doing away with a confidence level (Nelson 2011; see the "General Machine Learning" chapter for more on online testing). For specific ways of computing both types of intervals see Hahn et al. (2017) and Krishnamoorthy (2016).

The idea of using normal, Chebyshev, and exponential loss functions for confidence intervals is original.

Can multiply Chebyshev loss by $l = \frac{\sqrt{(1/a_{\text{target}})}}{\text{exact interval's length}}$ so that the target value is 1 for clearer interpretation, but this makes no difference to comparisons. Chebyshev loss follows another idea—that if a procedure misses a consistently, might as well run it with a **fractional a** , such as $\frac{1}{2}$ of the original or based on the estimate by how by it misses out. But don't have a well-justified way to select the fraction without domain knowledge. For the normal assumption, for which this is most useful, this is less likely to be accurate the tails, so, e.g., with half-level and 95% confidence the corresponding $z \approx 2.25$ may not be enough to make a difference.

For computing a confidence interval for Pearson correlation, further bias reduction in the asymptotic approximation can be useful. See DasGupta (2008) for more details.

The idea of bootstrap mixed Chebyshev interval, which generalizes the Wald interval even for nonbootstrap procedures, is original. Using Chebyshev inequality to cap bootstrap- t and define the binary search range for the calibrated interval search are also original. Can repeat calibration with an already calibrated interval recursively to improve order each time, but estimation error and computational resource increases rule this out.

Many sources present hypothesis tests to only accept/reject the null, without calculating p -values, by using significance tables that present allowable statistic values for particular confidence levels. p -values can be computed from these using interpolation (see the “Numerical Algorithms—Working with Functions” chapter), but usually a slightly different method can give a p -value directly. Confidence intervals derived from the tests are given directly by the tables though.

A somewhat different approach to hypotheses testing is **equivalence testing**. Want to show, e.g., that a generic drug \approx the brand drug in terms of treatment effect. The null is that they are different. Pick a domain-specific precision ϵ , and show that the alternative's performance is within $\theta \pm \epsilon$, where θ is the performance of the “gold standard”. The null is rejected if the calculated confidence interval on the performance $\in \theta \pm \epsilon$. From hypotheses testing view, it's a bit different because the null is the outside of $\theta \pm \epsilon$. **Noninferiority testing** is one-sided equivalence testing, and, for location statistics is equivalent to the usual 1-sided null testing by shifting the test statistic value by ϵ . This also enables reusing known tests for equivalence testing by doing two such one-sided tests at $a/2$ and returning their max p -value. But for nonparametric and nonlocation tests it's more complicated (Wellek 2010).

Meta-analysis is putting together results of several studies when can't directly combine the raw data. The combination is usually done using **normalized effect size**—e.g., z -score for normal-based tests. For nonparametric p -values, as a simple heuristic, can combine several independent studies by transforming to the equivalent z -scores, calculating the sum distribution (which is also normal), and converting its z -score to the p -value. See Borenstein et al. (2009) for an introduction to many methods. The main problem is publication bias—mostly studies with low p -values are published. So any combination is likely to be biased, and don't have a good way to correct for this. It seems most practical to select a single study based on sample size and how well it was done, and trust it. Publication bias is less of an issue for confidence intervals.

For hypotheses testing with the bootstrap can check if the null value is outside the computed interval. A domain-specific but more powerful alternative is to form the null distribution explicitly (Efron & Tibshirani 1993). But these methods effectively do what permutation tests do in a worse way and apply where the latter apply. Also this isn't automatic.

Another general computation-based technique is **empirical likelihood method** (Vexler et al. 2016). It requires statistics that are functions of weighted observations with weights $p_i \geq 0$ such that $\sum_i p_i = 1$.

$\prod_i p_i$ is **empirical likelihood**, based on the EDF. Given a null θ , maximize that likelihood such that $\text{statistic}(p, \text{data}) = \theta$. Without the null restriction, this is when $p_i = 1/n$, with the likelihood $= n^{-n}$. Then, as with parametric likelihood ratio tests, the log-likelihood ratio $\frac{\sum_i \log(p_i)}{-n \log(n)}$ \sim chi-square with some degrees of freedom as $n \rightarrow \infty$. This defines a hypotheses test, inverting which get a confidence interval. But according to some experiments (Wilcox 2016), the method seems to have limited applicability. Given recent research about its application to other problems (Vexler et al. 2016), need more research to justify general application. One apparent difficulty is the required numerical optimization—the constraint region is convex but the objective may not be, depending on the choice of the statistic. The null is rejected if the optimization fails, but this requires custom optimization methods for specific statistics because black-box methods are unreliable (see the “Numerical Optimization” chapter). So approximate optimization will give slightly or substantially wider intervals.

For permutation tests, because a single simulation produces a yes/no event, the estimated p -value has a

binomial distribution, so an interesting idea is using 95% confidence bounds instead of the average to give a more conservative *p*-value. But though the idea is appealing, the method has no precise justification.

Nemenyi test (sometimes not called so) is usually presented as a **post hoc test to Friedman test**, which tests whether all k alternatives are the same by computing sum of squared differences between rank sums and their expected values (Wikipedia 2015c; Gibbons & Chakraborti 2020). But the result isn't useful despite being more power than in the individual multiplicity-adjusted comparisons. The parametric test suite for this is **ANOVA**. Despite wide use of its many versions (the Friedman equivalent is **repeated measures**), it needs many assumptions and is only slightly more powerful than Friedman's when they are met. Likewise, **Tukey's HSD test** is an alternative to Nemenyi's. Many special tests have been developed for various parametric setups and have the advantage of making less drastic corrections than Bonferroni (Westfall et al. 2011).

For nonmatched samples, to get a Nemenyi equivalent, compute the ranks based on **Kruskal-Wallis test** (Gibbons & Chakraborti 2020). Both join all the samples and check the differences between average ranks of individual groups. But using trimmed means leads to meaningful confidence intervals and is just as robust.

Analogously to FDR, a **false coverage rate** gives multiple testing improvement for confidence intervals (Efron & Hastie 2016).

The use of grid search with RWM is also original. Another popular multivariate technique is the **Gibbs sampler**, but it needs some knowledge of the distribution and doesn't apply to all cases.

Several competitors to OCBA from simulation optimization literature are also **sequential methods**. Here do inference as data arrives online or in batches. One advantage of this comes, e.g., from using the first stage for FDR control and the second for FWER. For simulation stopping, use the first stage to estimate nuisance parameters such as the variance and the second for the main work. Online stopping leads to a more economical design in special cases. Sequential methods aren't popular, but see Mukhopadhyay & Silva (2008) and references therein for more information.

In manufacturing, the basic idea of **statistical quality control** is to monitor variance and intervene if it becomes high enough to cause defects. This needs specialized tools. A simple technique is a **control chart** that looks out for observations outside of 3 standard deviations from the mean, which corresponds to the outlier detection technique. A general idea is to define metrics, estimate the current performance, and use design of experiments to improve performance. The ideal goal is much of manufacturing is **six sigma**, which means very small allowance for the probability of failure. The idea is that if many independent component are put together, then the total probability of failure due to multiples testing must be small. This is achieved by reducing variance.

General techniques fail for **extreme value estimation** (Coles 2003). A typical example is estimating water dam height that is enough to prevent floods for several centuries with high probability, given measurements of several years of water levels. Simulation/resampling fails because an extreme future value will exceed anything observed. This is an ill-posed problem. A reasonable approach is to assume that water levels follow a normal or some other distribution, estimate its parameters, simulate several centuries of samples, and take the 99% or so. This is yet another example of that if a problem can be solved, it can be by some form of simulation.

Sampling methods are about picking samples to compute some quantity of interest. A typical example is estimating the number of infected trees in a large forest. It's infeasible to check every tree, so randomly pick several square-mile patches, count the infected trees in them to estimate the expected number of infected trees per square mile, and multiply that by the total number of squared miles to estimate the total. The only trick is accounting for finite population (square miles) to reduce variance (Thompson 2012). If have prior knowledge that some miles contain few trees and some many, can reduce variance by using **stratified sampling**—take more samples from the more populated miles, and weigh samples to account for this in calculations. This reduces variance and is similar to importance sampling. ∃ many ways to do this (Thompson 2012).

Exploratory data analysis (EDA) aims to present data in an intuitively perceivable way. The idea is that given a visual feel of the data, can effortlessly check assumptions, identify unusual patterns or outliers, etc. It's more a philosophy than a specific set of techniques. A number of visualization techniques such as the classic **box-and-whiskers plot** have been developed. Certain modern computation-intensive techniques such as clustering, principal component analysis, and density estimation can also be considered exploratory. EDA is meant for getting intuition about the data, unlike the traditional confirmatory analysis that focuses on getting statistically significant conclusions. In time-constrained corporate work environments, traditional experimental design has the advantage of allowing to easily present the results about the work, which gives in an important practical advantage.

EDA is often useful for solving ill-posed problems. Try several algorithms, each with different assumptions, and see how their output differs. Then use a confirmatory technique to see if what was observed is significant. But be careful—need different data to avoid snooping. A criticism of EDA is that it's manual and subjective. Also for data of a large dimension, a human can't visualize much, and algorithms do better. Try EDA only if known algorithms for the problem aren't good enough and human insight is expected to be useful.

A lot of “big data analytics” is counting events and using the counts to compute other statistics.

Saltelli's algorithm can also estimate sensitivities of interactions—when forming XC take all interacting variables from $X4$. But it's unclear which of the D^2 interactions to focus on. For large D , maybe use the main effects in an Apriori-like (see the “Machine Learning—Other Tasks” chapter) process to select $O(D)$ interactions. The algorithm extends to allow calculation of **total sensitivity indices**, which measure a variable's contribution due to all interactions with other variables, but these are less popular.

ARIMA is the most popular model for time series analysis, but many other models and extensions are possible:

- **Exponential smoothing models** are similar to but not entirely included by ARIMA(0, 1, 1) (Hyndman & Athanasopoulos 2021). They are weighted averages of recent observations and work well for a small number of training data (Chatfield 2001).
- Multidimensional models work for many correlated time series with a common correlation matrix and are a direct generalization of ARIMA.
- **State-space models** can be used to express all of the above and perform calculation using the **Kalman filter** algorithm.
- Nonlinear models go beyond the linear model of state-space models, but whether this works better in practice remains to be seen. A particular variation are **ARCH** and **GARCH** models, which allow nonconstant variance, though otherwise are linear.

21.43 Projects

- Research and implement methods for evaluating CDFs of other common distributions. Use these for testing distribution-match tests and random number generators. As a helper function, implement numerically safe evaluation of a binomial coefficient.
- If can't determine asymptotic variance analytically, can simulate with doubling size, and then perform regression on log scale to discover the rate of decrease. Experiment with it.
- Implement Kendall's correlation, and compare to Pearson and Spearman on simulated data.
- In case of a known mean for correlation an interesting idea is sign correlation—calculate the average number of matching direction of moves. Experiment with it.
- Bootstrap isn't robust when n is small, and excessive duplication caused f to have undefined behavior (such as correlation with all data being the same point). Change bootstrap confidence intervals to allow NaN values in such cases, and drop them.
- Implement bootstrap- t with mean-absolute-deviation pivot instead of the standard-deviation one. It should be more robust to large values—in fact even slight deviations from the normal favor this (at least for the contaminated normal model; Huber & Ronchetti 2009). Does it improve performance?
- Repeat the bootstrap simulations with smaller a , such as 0.01 and 0.001. Do the procedures still work as expected? Does the choice guidance change?
- Use multisample bootstrap to create a confidence interval for a difference of medians. Compare it with the simple difference of single-sample-median confidence intervals.
- Test the procedures for confidence intervals for location measures using a Laplace distribution. It should favor permutation test inversion due to being symmetric. Is that the case?
- Use bootstrap to study the efficiencies of the sample mean, median, and trimmed mean on several not-too-small real-life data sets by computing and comparing the standard deviations of the estimates on the resamples. Does the normality assumption offer an advantage?
- Test other 2nd-order bootstrap methods with large n .
- For the two-independent-sample mean difference which works better—the CLT, the bootstrap, or permutation testing? What about the median and the trimmed mean?
- Nemenyi test has IIA issues where adding or removing another alternative can sometimes affect the comparison result. Pairwise sign tests don't have this problem. Should this make a difference in practice? Do simulation to compare the two approaches.
- For comparing many independent samples, instead of calculating differences of k confidence inter-

vals it may be more statistically efficient to calculate intervals on the $\frac{k(k-1)}{2}$ differences. Do simulations for medians and the normal distribution to find out for which k this wins. Do the same for Cauchy. Does a matter?

- Do simulations for all presented confidence interval procedures to study their a attainment.
- Research continuity correction for the quantile and the Wilson score intervals. Does it improve accuracy for either?
- Do power analysis simulation for confidence intervals on the median for the normal distribution. What sample size is usually good enough for a not-too-short interval? The next step is to do this for two-sample intervals. Which simulation is easier?
- Implement generating extra primitive polynomials for the Sobol sequence. This needs researching number theory books.
- For the Sobol sampler implement general transformation functionality that allows, e.g., to sample a log-scale range as a special case.
- Apply sensitivity analysis to some classification models (see the “Machine Learning—Classification chapter). Does this help with variable selection?

21.44 References

- Adèr, H. J., Mellenbergh, G. J., & Hand, D. J. (2008). *Advising on Research Methods: A Consultant's Companion*. Johannes van Kessel Publishing (can order from <https://jvank.nl/ARMHome>).
- Andrews, D. W. (2000). Inconsistency of the bootstrap when a parameter is on the boundary of the parameter space. *Econometrica*, 68(2), 399–405.
- Anguita, D., Ghelardoni, L., Ghio, A., & Ridella, S. (2013). A survey of old and new results for the test error estimation of a classifier. *Journal of Artificial Intelligence and Soft Computing Research*, 3(4), 229–242.
- Audibert, J. Y., Munos, R., & Szepesvári, C. (2009). Exploration-exploitation tradeoff using variance estimates in multi-armed bandits. *Theoretical Computer Science*, 410(19), 1876–1902.
- Basu, A., Shioya, H., & Park, C. (2011). *Statistical Inference: The Minimum Distance Approach*. CRC.
- Beisbart, C., & Saam, N. J. (Eds.). (2018). *Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives*. Springer.
- Bickel, P. J., & Freedman, D. A. (1981). Some Asymptotic Theory for the Bootstrap. *The Annals of Statistics*, 1196–1217.
- Boos, D. D., & Stefanski, L. A. (2013). *Essential Statistical Inference*. Springer.
- Bratley, P., & Fox, B. L. (1988). Algorithm 659: implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software (TOMS)*, 14(1), 88–100.
- Brophy, A. L. (1987). Efficient estimation of probabilities in the t distribution. *Behavior Research Methods*, 19(5), 462–466.
- Brown, L. D., Cai, T. T., & DasGupta, A. (2001). Interval Estimation for a Binomial Proportion. *Statistical Science*, 16(2), 101–133.
- Canal, L. (2005). A normal approximation for the chi-square distribution. *Computational Statistics & Data Analysis*, 48(4), 803–808.
- Casella, G. (1996). The Ghosh-Pratt Identity. Technical report, Cornell University.
- Chatfield, C. (1995). *Problem Solving: A Statistician's Guide*. CRC.
- Chatfield, C. (2001). *TimeSeries Forecasting*. CRC.
- Chen, C. H., Lee L.H. (2010). *Stochastic Simulation Optimization: an Optimal Computing Budget Allocation*. World Scientific.
- Chihara, L. M., & Hesterberg, T. C. (2018). *Mathematical Statistics with Resampling and R*. Wiley.
- Coles, S. (2001). *An Introduction to Statistical Modeling of Extreme Values*. Springer.
- DasGupta, A. (2008). *Asymptotic Theory of Statistics and Probability*. Springer.
- Davies, P. L. (2014). *Data Analysis and Approximate Models: Model Choice, Location-Scale, Analysis of Variance, Nonparametric Regression and Image Analysis*. CRC.
- Davison, A. C., & Hinkley, D. V. (1997). *Bootstrap Methods and Their Application*. Cambridge University Press.
- Efron, B., & Hastie, T. (2016). *Computer Age Statistical Inference*. Cambridge.
- Efron, B., & Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. CRC.
- Florescu, I. (2014). *Probability and Stochastic Processes*. Wiley.
- Franses, P. H., Legerstee, R., & Paap, R. (2017). Estimating loss functions of experts. *Applied Economics*, 49(4), 386–396.
- Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian Data Analysis*.

- CRC.
- Gibbons, J. D., & Chakraborti, S. (2020). *Nonparametric Statistical Inference*. Springer.
- Goos, P., & Jones, B. (2011). *Optimal Design of Experiments: A Case Study Approach*. Wiley.
- Gigerenzer, G. (2011). What are natural frequencies?. *BMJ*, 343, d6386.
- Greenland, S., Senn, S. J., Rothman, K. J., Carlin, J. B., Poole, C., Goodman, S. N., & Altman, D. G. (2016). Statistical tests, P values, confidence intervals, and power: a guide to misinterpretations. *European Journal of Epidemiology*, 31(4), 337-350.
- Johnston, W. (2015). *The Lebesgue Integral for Undergraduates*. MAA.
- Hahn, G. J., Meeker, W. Q., & Escobar, L. A. (2017). *Statistical Intervals: A Guide for Practitioners*. Wiley.
- Hand, D. J. (2014). *The Improbability Principle: Why Coincidences, Miracles, and Rare Events Happen Every Day*. Scientific American.
- Hesterberg, T. (2014). What Teachers Should Know about the Bootstrap: Resampling in the Undergraduate Statistics Curriculum. *arXiv preprint arXiv:1411.5279*.
- Hjorth, J. U. (1994). *Computer Intensive Statistical Methods: Validation, Model Selection, and Bootstrap*. Routledge.
- Howard, R. A. & Abbas, A. E. (2015). *Foundations of Decision Analysis*. Pearson.
- Howell, D. C. (2016). Randomization/Permutation Tests. <https://www.uvm.edu/~dhowell/StatPages/>. Accessed October 16, 2016.
- Hubbard, D. W. (2014). *How to Measure Anything: Finding the Value of Intangibles in Business*. Wiley.
- Huber, P. J. (2011). *Data Analysis: What Can Be Learned from the Past 50 Years*. Wiley.
- Huber, P. J., & Ronchetti, E. M. (2009). *Robust Statistics*. Wiley.
- Hubert, L., & Wainer, H. (2012). *A Statistical Guide for the Ethically Perplexed*. CRC.
- Hyndman, R. J., & Athanasopoulos, G. (2021). *Forecasting: Principles and Practice*. OTexts.
- Imbens, G. W., & Rubin, D. B. (2015). *Causal Inference in Statistics, Social, and Biomedical Sciences*. Cambridge.
- Katzourakis, N., & Varvaruca, E. (2018). *An Illustrative Introduction to Modern Analysis*. CRC.
- Kiefer, J. C. (1987). *Introduction to Statistical Inference*. Springer.
- Korostelev, A. P., & Korosteleva, O. (2011). *Mathematical Statistics: Asymptotic Minimax Theory*. AMS.
- Krishnamoorthy, K. (2016). *Handbook of Statistical Distributions with Applications*. CRC.
- Kucherenko, S., Albrecht, D., & Saltelli, A. (2015). Exploring multi-dimensional spaces: a Comparison of Latin Hypercube and Quasi Monte Carlo Sampling Techniques. *arXiv preprint arXiv:1505.02350*.
- Lehmann, E. L., & Casella, G. (1998). *Theory of Point Estimation*. Springer.
- Lehmann, E. L., & Romano, J. P. (2005). *Testing Statistical Hypotheses*. Springer.
- Lemieux, C. (2009). *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer.
- Lunneborg, C. E. (2000). *Data Analysis by Resampling: Concepts and Applications*. Duxbury.
- Mukhopadhyay, N., & De Silva, B. M. (2008). *Sequential Methods and Their Applications*. Chapman and Hall/CRC.
- Nelson, G. S. (2015). *A User-friendly Introduction to Lebesgue Measure and Integration*. American Mathematical Society.
- Nelson, M. J. (2011). You might want a tolerance interval. <http://tinyurl.com/tol-interval>.
- Neyman, J. (1977). Frequentist probability and frequentist statistics. *Synthese*, 36(1), 97-131.
- Panaretos, V. M. (2016). *Statistics for Mathematicians*. Springer.
- Press, W.H. et al. (2007). *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge.
- Proschan, M. A., & Shaw, P. A. (2016). *Essentials of Probability Theory for Statisticians*. CRC.
- Rissanen, J. (2012). *Optimal Estimation of Parameters*. Cambridge University Press.
- Romano, J. P. (1990). On the behavior of randomization tests without a group invariance assumption. *Journal of the American Statistical Association*, 85(411), 686-692.
- Ryan, T. P. (2013). *Sample Size Determination and Power*. Wiley.
- Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., & Tarantola, S. (2008). *Global Sensitivity Analysis: The Primer*. Wiley.
- Samaniego, F. J. (2010). *A Comparison of the Bayesian and Frequentist Approaches to Estimation*. Springer.
- Santana, L. (2009). *Contributions to the m-out-of-n Bootstrap* (Dissertation). North-West University.
- Shumway, R. H., & Stoffer, D. S. (2000). *Time Series Analysis and Its Applications*. Springer.
- Self, S. G., & Liang, K. Y. (1987). Asymptotic properties of maximum likelihood estimators and likelihood ratio tests under nonstandard conditions. *Journal of the American Statistical Association*, 82(398), 605-610.
- Shao, J. (2003). *Mathematical Statistics*. Springer.
- Shao, J., & Tu, D. (1995). *The Jackknife and Bootstrap*. Springer.

- Sobol, I. M., Asotsky, D., Kreinin, A., & Kucherenko, S. (2011). Construction and Comparison of High-Dimensional Sobol' Generators. *Wilmott*, 2011(56), 64-79.
- StackExchange (2016a). How to investigate properties of the trimmed mean for a Cauchy variable?. <http://stats.stackexchange.com/questions/87354/how-to-investigate-properties-of-the-trimmed-mean-for-a-cauchy-variable>. Accessed October 16, 2016.
- StackExchange (2016b). Does the Hodges-Lehmann estimator perform better than trimmed/winsorized means?. <https://stats.stackexchange.com/questions/235392/does-the-hodges-lehmann-estimator-perform-better-than-trimmed-winsorized-means>. Accessed October 16, 2016.
- Stoyanov, J. M. (2013). *Counterexamples in Probability*. Dover.
- Temme, N. M. (1994). A set of algorithms for the incomplete gamma functions. *Probability in the Engineering and Informational Sciences*, 8(02), 291–307.
- Thompson, S. K. (2012). *Sampling*. Wiley.
- Vexler, A., Hutson, A. D., & Chen, X. (2016). *Statistical Testing Strategies in the Health Sciences*. CRC.
- Wasserman, L. (2004). *All of Statistics: A Concise Course in Statistical Inference*. Springer.
- Wei, F., & Dudley, R. M. (2012). Two-sample Dvoretzky–Kiefer–Wolfowitz inequalities. *Statistics & Probability Letters*, 82(3), 636–644.
- Wellek, S. (2010). *Testing Statistical Hypotheses of Equivalence and Noninferiority*. CRC.
- Westfall, P., & Henning, K. S. (2013). *Understanding Advanced Statistical Methods*. CRC.
- Westfall, P. H., Tobias, R. D., & Wolfinger, R. D. (2011). *Multiple Comparisons and Multiple Tests Using SAS*. SAS Institute.
- Wikipedia (2014a). Bootstrapping. [http://en.wikipedia.org/wiki/Bootstrapping_\(statistics\)](http://en.wikipedia.org/wiki/Bootstrapping_(statistics)). Accessed June 25, 2014.
- Wikipedia (2014b). Arrow's impossibility theorem. http://en.wikipedia.org/wiki/Arrow's_impossibility_theorem. Accessed November 23, 2014.
- Wikipedia (2014c). Error function. http://en.wikipedia.org/wiki/Error_function. Accessed June 25, 2014.
- Wikipedia (2015a). Rule of three. [https://en.wikipedia.org/wiki/Rule_of_three_\(statistics\)](https://en.wikipedia.org/wiki/Rule_of_three_(statistics)). Accessed December 31, 2015.
- Wikipedia (2015b). Kolmogorov-Smirnov test. https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test. Accessed November 18, 2015.
- Wikipedia (2015c). Friedman test. https://en.wikipedia.org/wiki/Friedman_test. Accessed November 18, 2015.
- Wikipedia (2016a). Low-discrepancy sequence. https://en.wikipedia.org/wiki/Low-discrepancy_sequence. Accessed September 15, 2016.
- Wikipedia (2016b). Total variation. https://en.wikipedia.org/wiki/Total_variation. Accessed September 15, 2016.
- Wikipedia (2016c). Secretary problem. http://en.wikipedia.org/wiki/Secretary_problem. Accessed September 2, 2016.
- Wikipedia (2016e). Efficiency (statistics). [https://en.wikipedia.org/wiki/Efficiency_\(statistics\)](https://en.wikipedia.org/wiki/Efficiency_(statistics)). Accessed October 16, 2016.
- Wikipedia (2017b). Time management. https://en.wikipedia.org/wiki/Time_management. Accessed July 23, 2017.
- Wikipedia (2018a). Student's *t*-distribution. https://en.wikipedia.org/wiki/Student%27s_t-distribution. Accessed June 30, 2018.
- Wikipedia (2018b). Pearson's chi-squared test. https://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test. Accessed October 6, 2018.
- Wikipedia (2019a). Beta-binomial distribution. https://en.wikipedia.org/wiki/Beta-binomial_distribution. Accessed April 12, 2019.
- Wikipedia (2019b). Fisher transformation. https://en.wikipedia.org/wiki/Fisher_transformation. Accessed June 15, 2019.
- Wikipedia (2019c). Binomial proportion confidence interval. https://en.wikipedia.org/wiki/Binomial_proportion_confidence_interval. Accessed July 20, 2019.
- Wikipedia (2019d). Approval voting. https://en.wikipedia.org/wiki/Approval_voting. Accessed October 19, 2019.
- Wikipedia (2021). Kendall rank correlation coefficient. https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient. Accessed October 10, 2021.
- Wilcox, R. R. (2016). *Introduction to Robust Estimation and Hypothesis Testing*, 4th ed. Academic Press.
- Yerukala, R., Boiroju, N. K., & Reddy, M. K. (2013). Approximations to the t-distribution. *International Journal of Statistica and Mathematica*, 8(1).

22 Numerical Algorithms—Introduction and Matrix Algebra

“The purpose of computing is insight and not numbers” – Richard Hamming

“God made the integers, all the rest is the work of man” – Leopold Kronecker

“If floating point errors disappeared, 90% of numerical analysis would remain as is” – Nick Trefethen

“The ever-decreasing cost of computer time biases users towards robustness rather than efficiency” – John Lambert

“It is much easier to teach numerical methods than implement robust numerical software” – Gander et al. (2014)

22.1 Introduction

The reader has ideally taken a class in basic numerical methods and is familiar with common numerical tasks such as calculating integrals using an undergraduate book with many examples such as Fausett (2003). To study numerical analysis need more prerequisites than the rest of the book. A minimal package is course in linear algebra, mathematical analysis, ordinary differential equations (unless skip the corresponding sections), and familiarity with complex numbers.

The goal of numerical analysis is to create numerical procedures that can solve mathematical problems reasonably well and be implemented on a computer as algorithms. It's worth over-emphasizing that numerical algorithms, like statistical and machine learning ones, aren't bulletproof black boxes, except for certain special cases. Even the most well-engineered implementations at best have well-documented and well-known limitations (sources of these will be discussed later in the chapter). In the ideal case, for a typical algorithm can justify every choice, with preference for simplicity, and specify what problems it can't solve, and potentially that it's unreasonable to try. An algorithm is only a reasonable procedure done by a computer—the automation doesn't guarantee quality. It's easy to implement a basic method and have it solve a few test problems perfectly, but creating a robust implementation is generally very difficult even for most basic algorithms.

This chapter first reviews general themes in numerical analysis—in particular correctness with respect to approximation and floating point errors. Then implementations of the most important matrix algorithms are discussed.

22.2 Floating Point Arithmetic

Hardware arithmetic has limited precision because exact arithmetic suffers from combinatorial explosion—e.g., need 64 bits to represent the product of two 32-bits numbers. But with floating point fix the precision, so in the above multiplication keep only the 32 most significant bits, and discard the rest by rounding. Floating point arithmetic has been standardized since 1985, with the most recent standard in 2019.

A number x of any precision type (single, double, etc.) is represented as $\pm(1+m)2^e$, where:

- m is the **mantissa**, normalized to $(0, 1]$
- e is the **exponent**

E.g., for a `double` of 64 bits, 1 bit goes to the **sign**, 52 to m (get 53 bits of precision with the implicit 1 \approx 16 decimal digits), and 11 to e . Can represent several types of numbers:

- **Normal**— m uses all of its allocated bits. Normals are what almost all computations are usually done with.
- **Subnormal**—when a normal number becomes too small (e is large and negative), to enable gradual change to 0, some bits of m are given to e . For a `double`, $|x| < 2^{-1022} \approx 1.8 \times 10^{-308}$ are subnormal. Subnormals exist only to allow some guarantees on operations with normals, such as accurate rounding in all cases.
- **Special constants**—the most important ones are 0 (which is represented by all bits = 0), -0 ($=0$ but apparently useful), $\pm\infty$, and several NaN values. These behave as expected—e.g., $1/0 = \infty$, and $0/0 =$

`Nan`, `NaN` and $\pm\infty$ make sure that all operations are well-defined.

The exact 0 is deceptive in that many nearby numbers can also be considered 0 when rounded. So algorithms shouldn't be looking for it. The exact floating point encoding details aren't important (see Overton 2001 if curious), and relying on representation details is asking for trouble. C++ and most other languages give portable access to some of the parameters, but only a few are used regularly. In particular, iterating over all floating point numbers from 0 to ∞ takes steps of 1 **ulp** (**unit in the last place**). For $x > 0$, $\text{ulp}(x) = (\text{the smallest floating point number} > x) - (\text{the largest floating point number} \leq x)$; this is a common definition that covers most cases. Then **machine epsilon** $\epsilon = \text{ulp}(1)$; i.e., ϵ is the smallest $x > 0$ such that $1 + x \neq 1$. `T = numeric_limits<double>` contains some important numbers:

Number	Accessor	Approximate Standard Value
Min x such that $1 + x \neq 1$	<code>T::epsilon()</code>	2.2E-16
Min normal x	<code>T::min()</code>	1.8E-308
Max normal x	<code>T::max()</code>	2.2E+308
Infinity	<code>T::infinity()</code>	∞
<code>NaN</code>	<code>T::quiet_NaN()</code>	N/A; anything = <code>NaN</code> is false

The corresponding negative numbers are $-x$. Use `isfinite` from `<cmath>` to check whether a value is ∞ or `NaN` (also can use `isnan` for `NaN`).

Ulps allow measuring errors. For a real x , let $\text{fl}(x) = x$ rounded to the appropriate floating point number, depending on the rounding mode:

- Nearest—using common rounding, i.e., $0.5 \rightarrow 1$, $0.49 \rightarrow 0$.
- Down—to $-\infty$, i.e., $0.5 \rightarrow 0$.
- Up—to $-\infty$, i.e., $0.49 \rightarrow 1$.

Nearest is the default mode because the error $\leq \text{ulp}(x)/2$. For down and up the error $\leq \text{ulp}(x)$, but these are useful for interval arithmetic (discussed in the comments section; another mode to 0 is never useful). An operation is **correctly rounded** when the result matches the best floating point number approximation given a rounding mode. Elementary floating point operations $\{+, -, \times, /\}, \sqrt{}$, and remainder of division are correctly rounded (Overton 2001; more are optionally correctly rounded in the latest standard).

Computation with floating point numbers happens in CPUs, and **floating point registers** have more precision (typically 80 bits). This preserves precision of intermediate results of sequences of operations if the computation stays in registers.

The real RAM model (see the “Background” chapter) is useful for reasoning about correctness of many numerical algorithms. Also sometimes make the **general position assumption** that have no equality in inputs. While this can be helpful analytically, must handle equality in code.

22.3 Errors from Using Floating Point Arithmetic

Even with correct rounding logical errors can happen:

- **Overflow**—e.g., $10!! = \infty$
- **Underflow**—e.g., $2^{-10000} = 0$
- **Relative underflow**—e.g., $1 + \epsilon/2 = 1$ (but will work as intermediate step in a register)

In the world of floating point arithmetic these are well-defined correct results. But in the world of idealized real arithmetic they can result in many surprises where a provably correct formula leads to a wrong result, and almost all algorithm theorems assume exact arithmetic. Inexactness is a necessary approximation and usually not a bad one, but need to recognize when it is. So ignore floating point error except when can't, but always need specialized analysis to check this.

The floating point standard gives predictability of results, but only in a special way. E.g., when f is deterministic, $f(x) = f(x)$ may be false if the result of the first call is stored in memory, of the second isn't, and the difference is due to rounding to different numbers of bits. But, assuming in both cases the intermediate steps of f are computed in the same memory/CPU register pattern in both cases, their difference in floating point arithmetic \leq that of rounding, but otherwise may not be. In general, a number in memory converted to register and back remains the same, but not the other way around.

The most problematic error comes from **round-off accumulation**—a sequence of operations may not be correctly rounded. E.g., try to calculate π by summing up a convergent series—eventually the terms become too small to influence the partial sum numerically (Higham 2002). Naive implementations of many algorithms fail despite correctness in the real RAM.

Standard laws of arithmetic such as commutativity and associativity don't hold, so compiler optimizations, such as rearranging expressions, become problematic. Also, certain structural decisions become difficult—e.g., is a matrix symmetric if some symmetric entries are different up to some small precision $> \epsilon$?—must make an arbitrary judgment.

Fortunately substantial round-off accumulation usually doesn't happen, and if it does can tweak the algorithm or use another one. A large part of numerical analysis is making sure that algorithm implementations that use floating point arithmetic give answers that are as close as possible to the real RAM ones. But have other types of errors—in fact numerical methods err for the same reasons as machine learning and statistical algorithms:

- **Approximation error**—solve a slightly different problem
- **Estimation error**—compute with limited precision, and can get inaccurate inputs
- **Optimization error**—don't compute enough

For reducing estimation error, a general technique is using a different mathematical formula for the same thing. This can mean using an entirely unrelated formula, but most commonly is about rewriting a particular expression to make it have less error. E.g., for roots of quadratic equations relative error can be high if $4ac < \epsilon b$ because essentially compute $b - \sqrt{b(b+\epsilon)}$, which leads to a bad subtraction. So compute the "+" root, and get the "-" root from it using another formula. Addition/subtraction preserve absolute precision and multiplication/division relative one. See Beebe (2017) for a most accurate procedure for solving quadratic equations.

Always need to look at a formula and make sure it's not bad in that, e.g., don't subtract equal quantities or divide by 0. Essentially forget about everything you know about real arithmetic and make no assumptions whatsoever. I.e., imagine a piece of generic C++ code where every possible operator is overloaded and can behave in unpredictable ways such as randomly throwing exceptions. See Acton (1996) for many examples. It also reminds that while certain algorithmic building blocks are well-known and readily available, numerical problem solving sometimes needs ingenuity beyond direct application of the building blocks. E.g., may need to replace cos with a several-term power series to avoid a bad subtraction.

A general pattern for minimizing precision loss of a sequence of calculations is to do the operations that lose more precision first if possible. Then subsequent operations that lose less will only be losing garbage bits.

Most algorithms need reasonably scaled inputs for precise answers. No clever technique can automatically determine the scale of the inputs because sufficiently small numbers are indistinguishable from 0. Here **relative precision** no longer matters but **absolute** one does. Since `T::min()` is much smaller than ϵ , and even a single operation can introduce absolute $O(\epsilon)$ error, an answer that is much less than $O(\epsilon)$ can't be trusted. Can do some heuristic scaling though.

Proofs of correct floating point computation are generally complicated even for simple programs due to many special cases. Automated proof assistants are slowly developing and have been used for some algorithms such as computation of elementary functions (Muller 2016).

Consider a seemingly simple task of deciding if $a = b$. Direct comparison is usually meaningless (but will work if a and b are represented exactly). A more reasonable alternative is to check if $|a - b| \leq \epsilon \max(|a|, |b|, \text{min normal})$. This is the best the standard can do, but usually want a more relaxed criteria:

- Use some $\epsilon \geq$ the unit round-off
- Use some absolute precision that is much greater than $\epsilon \times \text{min normal}$. Using ϵ is convenient for properly scaled inputs. Other sources such as Press et al. (2007) use a much smaller absolute precision such as ϵ^2 , but with proper scaling it seems unreasonable to go beyond $\epsilon/10^6$ or so, and must be very careful to not lose that extra precision later.

It's convenient to develop ϵ -based " $<$ " and express equality in terms of it. For efficiency and correct behavior with NaNs, first check if $a < b$.

```
bool isELess(double a, double b,
    double eRelAbs = numeric_limits<double>::epsilon())
{ return a < b && b - a >= eRelAbs * max(1.0, max(abs(a), abs(b))); }
bool isEEqual(double a, double b,
    double eRelAbs = numeric_limits<double>::epsilon())
{ return !isELess(a, b, eRelAbs) && !isELess(b, a, eRelAbs); }
```

The results should be correct for all possible values of a and b . Use a unit test that checks:

- NaN and $\pm\infty$. Remember that the comparison isn't transitive with NaN because $\text{NaN} \neq \text{NaN}$.
- Small numbers such as 0, ϵ , min normal, etc.

```
void testELessAuto()
```

```

double nan = numeric_limits<double>::quiet_NaN(),
        inf = numeric_limits<double>::infinity();
double es[4] = {numeric_limits<double>::epsilon(), highPrecEps,
                 defaultPrecEps, 0.1};
for(int i = 0; i < 4; ++i)
{
    double eps = es[i];
    //nan
    assert(isELess(nan, nan, nan) == false);
    assert(isELess(nan, nan, eps) == false);
    assert(isELess(nan, 1, eps) == false);
    assert(isELess(1, nan, eps) == false);
    assert(isELess(nan, inf, eps) == false);
    assert(isELess(inf, nan, eps) == false);
    //inf
    assert(isELess(inf, inf, eps) == false);
    assert(isELess(-inf, -inf, eps) == false);
    assert(isELess(-inf, inf, eps) == true);
    assert(isELess(-inf, 1, eps) == true);
    assert(isELess(-1, inf, eps) == true);
    //normal
    for(double x = numeric_limits<double>::min() * 10;
         x < numeric_limits<double>::max()/10; x *= 10)
    {
        double dx = eps * max(1.0, abs(x));
        assert(isELess(x, x + 2 * dx, eps) == true);
        assert(isELess(x, x + 0.5 * dx, eps) == false);
        assert(isELess(x - 2 * dx, x, eps) == true);
        assert(isELess(x - 0.5 * dx, x, eps) == false);
    }
}

```

Note from the unit test that this comparison isn't trying to squeeze out all possible standard-allowed floating point numbers between a and b —e.g., can use $\epsilon/2$ when the rounding mode is the default to nearest, etc., but this buys nothing in practice. Writing a 100% robust comparison is difficult, and some approaches compare ulps directly (see <http://floating-point-gui.de/errors/comparison/> and <https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/>), but it's far more intuitive to compare in terms of ϵ . Many algorithms use “ $1 + |x|$ ” over “ $\max(1, |x|)$ ” for simplicity and automatic NaN handling.

Testing for $x = 0$ depends on the problem. Using some default absolute precision as above makes sense in many but far from all cases. Also in financial trading something like price has infinite precision because $\text{price} \times \text{quantity}$ (ignoring some other things) determines the value of a trade. So even a small rounding error in price representation will cause a mistake with large enough quantity. This doesn't happen in practice but is something to keep in mind for financial programmers, especially with price-yield and related calculations.

22.4 Approximation Error

Even with exact arithmetic, a numerical algorithm may not work if:

- An iterative process it uses isn't guaranteed to converge
 - Want to compute some property of a black-box function f that is known only through evaluation at some number of points, unlike the complete analytic description

The former is handled case-by-case. The latter is a general problem with many tasks. In particular, \exists a reasonable answer to problems with bad properties, such as minimizing a function with a very narrow $-\infty$ spike. Here, a minimization algorithm's answer “10.0200 with absolute precision 0.001” is best possible when, e.g., $f(10.02) = -100000$, and $f(10.02001) = -\infty$. This is the nature of ill-posed problems in general. So algorithms usually make guarantees only for some restricted subset of functions.

Sampling f at a finite number of points still restricts the size of the set of all possible f to ∞ . If f isn't Lipschitz or doesn't have another suitable restriction that makes all these behave in the same way for any made conclusion, such a conclusion has no validity, and tasks based on working with black-box f such as

interpolation, integration, and differentiation (discussed in the “Numerical Methods—Working with Functions” chapter) have no solution. This restriction only applies to deterministic algorithms—e.g., Monte Carlo integration works for a discontinuous f (but still the error bounds are better if f has a bounded range).

A lot of numerical analysis is based on **Taylor's theorem** (Wikipedia 2017d): Given f with $k+1$ continuous derivatives $\in (a, x)$, $f(x) = \sum_{0 \leq i \leq k} \frac{f^{(i)}(a)}{i!} (x-a)^i + \frac{f^{(k+1)}(\xi)}{(k+1)!} (x-a)^{k+1}$ for some $\xi \in (a, x)$. A symmetric result applies to $x < a$. Often assume that $\|f^{(k+1)}\|_\infty$ or another expression is bounded by some constant M .

22.5 Stability and Condition Numbers

Think of a numerical algorithm as computing $A(x)$ in place of $y = f(x)$ for some f and x . With respect to floating point error accumulation, want to know the **forward error** $(A, x) = |y - A(x)|$. Unfortunately computing it requires knowing y , which is usually unavailable. An algorithm with $O(\epsilon)$ forward error is **forward stable**.

Usually work with **backward error** $(A, x) = \max |\Delta x = x_2 - x|$ such that $A(x) = f(x_2)$ in exact arithmetic. These definitions extend to more general input and output spaces by using proper norms. An algorithm with $O(\epsilon)$ backward error is **backward stable**, or just **stable** for simplicity. So:

- In exact arithmetic, backward error = 0
- In floating point arithmetic, have backward stability for correctly rounded operations

Correct rounding is backward and not forward error control. E.g., consider cancellation in subtraction of almost equal numbers—given the original exact arguments, of which the floating point representations are accurate to the last bit, the precision of the result will be low except for the rare case where rounding doesn't change arguments. A correctly rounded answer is only so for the rounded arguments and not the exact ones. Assuming that it's exact gives the definition of the backward error, but for the exact arguments correctly rounded lower bits are garbage. The closer the arguments are, the more precision will be lost.

The **modulus of continuity** of f is $w_f(a) = \max_{|\Delta x| \leq a} \|f(x + \Delta x) - f(x)\|$ (Powell 1981). This is typically defined in 1D but works with any norm. The **absolute condition number** of a problem is defined so that a certain backward error gives a bound on forward error—for absolute backward error $\leq a$,

$$\kappa_{\text{abs}}(a) = \max_{0 < |\Delta x| \leq a} \frac{w_f(a)}{\|\Delta x\|}.$$

Then $\|\Delta y\| \leq w_f(a) \|\Delta x\|$. The **relative condition number** looks at

relative errors—by rearrangement of the above $\frac{\|\Delta y\|}{\|y\|} \leq \kappa_{\text{rel}}(a) \frac{\|\Delta x\|}{\|x\|}$, where $\kappa_{\text{rel}}(a) = \kappa_{\text{abs}}(a) \frac{\|x\|}{\|y\|}$. The forward error is occasionally small even if the bounding variables aren't though. Then a finite condition number ensures that the problem is **well-posed**—i.e., the output depends continuously on the input.

f is **Lipschitz** on an (possibly multidimensional) interval if $\forall x_1, x_2 \exists$ a constant L such that $\|f(x_1) - f(x_2)\| < L \|x_1 - x_2\|$. I.e., for f that is sampled at some points can't have arbitrarily different behavior between the samples, as it could, e.g., given only continuity. If f is Lipschitz on the interval of interest with constant L , $\kappa_{\text{abs}} \leq L$, and $w_f(a) \leq La$. Also $|f'(x)| \leq L \leq \max |f'|$. The interval restriction is often important because, e.g., $f(x) = x^3$ changes much slower near 0.

Working with a is inconvenient, so these generalized definitions are only useful intuitively. Instead work with a linear approximation—for a differentiable f , based on the first few terms of Taylor series expansion,

$\kappa_{\text{abs}}(f, x) = f'(x) + O(\|\Delta x\|^2)$, and $\kappa_{\text{rel}}(f, x) = \frac{x}{f(x)} \kappa_{\text{abs}}(f, x)$ (Higham 2002). For small $\|\Delta x\|$ it's reasonable to ignore the 2nd-order term; otherwise need modulus of continuity for bounds. E.g., addition of two numbers b and c is a linear function from 2D to 1D. Let the norm be L_1 vector norm extended to the matrix norm (this is discussed later in the chapter); then $\|f'(b, c)\| = 1$ (Deuflhard & Hohmann 2003). Then

$\kappa_{\text{abs}} = 1$, and $\kappa_{\text{rel}} = \frac{|b| + |c|}{|b + c|}$. So if b and c have same magnitude and different sign, have a bad subtraction, as explained by κ_{rel} . See Datta (2010, p58) for a similar analysis with the ∞ norm. Large/small κ problems are called respectively **well-conditioned/ill-conditioned**.

With nonsmooth f such as $|x|$ or a broken line, can use the maximum-norm subgradient (see the “Numerical Optimization” chapter). Taylor series no longer works, so can't directly discard second-order terms, but still get a useful value. With arbitrary norms use Frechet derivative or the corresponding subgradient. For a jump discontinuity $\kappa = \infty$, but based on $w_f(a)$ in theory can bound forward error, which is likely to be bounded. In this sense linear condition analysis represents a worst case, which is very pessimistic for a small number of jump discontinuities. Can think of a local κ , which however is itself unstable if a jump discontinuity is nearby. In practice condition analysis is almost always used as a qualitative guidance, and I

have never seen a bound based on $w_f(a)$ despite its giving a worst-case and not asymptotic bound.

So if an algorithm is backward stable, and κ is $O(1)$ with respect to ϵ , the algorithm is forward stable. κ needs to be computed for a particular problem, and backward stability is a property of an algorithm. Theorem (Corliss & Fillion 2013): κ (either absolute or relative) is submultiplicative under function composition, i.e., $\kappa_f \leq \kappa_h \kappa_g$ for $f(x) = h(g(x))$.

Backward stability σ doesn't have such relationship because a sequence of operations can magnify the error of the result—e.g., for $g(x) = f(\lg(x))$ a small linear error in $\lg(x)$, that is backward from f , can be magnified in x . But have other useful relationships (Deuflhard & Hohmann 2003):

- $\sigma_f \kappa_f \leq (\sigma_h + \sigma_g \kappa_g) \kappa_h$
- If $\kappa_f = \kappa_h \kappa_g$, which is the case when f and g are scalar and differentiable, then $\sigma_f \leq \frac{\sigma_h}{\kappa_g} + \sigma_g$

Some algorithms that are correct in real RAM (and taught in math classes), such as **Gram-Schmidt orthogonalization**, are unstable. Some problems, such as derivative estimation have $\max_x(\kappa) = \infty$. But this doesn't mean a hopeless case—the worst-case behavior doesn't prevent average cases from giving satisfactory forward errors. In some rare cases \exists an exact input corresponding to the output (Corless & Fillion 2013), so backward analysis is impossible, but κ is still meaningful.

Stability of an algorithm means that if its inputs have uncertainty \geq the backward error, the computed answer will be as good as possible, even though the forward error can be large due to ill-conditioning—no other algorithm can do better unless more stable. So stable algorithms can be used without regret and further analysis of floating point effects, which is very useful because the latter is usually impractical. One main goal of numerical analysis is coming up with more stable algorithms. So ideally always:

- Use only stable algorithms—or as stable as reasonable
- Solve only well-conditioned problems as intermediate steps

Elementary arithmetic operations are stable, so lose stability only if a sequence of them solved an ill-conditioned subproblem.

22.6 Optimization Error

Computational difficulties are generally of two types:

- A direct algorithm that is correct in real RAM is available but too expensive to execute for some inputs. E.g., for many matrix algorithms $O(n^3)$ runtime doesn't allow $n > 10000$ or so. A general solution that works when available is to replace such algorithm with a cheaper approximate algorithm, which has happened a lot in matrix algebra.
- An iterative algorithm converges too slowly to run to full precision. Typically run only to whatever low precision is acceptable.

The first case is handled case-by-case, but the second is a major theme in numerical analysis. Iterative algorithms aren't exact in real RAM due to finitely many iterations, but can run them until any specific precision is satisfied.

Termination precision ϵ for an iterative algorithm can be:

- **Absolute**—the difference of some meaningful numbers relative to 0—meaningless for large numbers, but can bound the number of iterations to reach a particular precision if know the error decrease per iteration.
- **Cauchy**—the difference between two consecutive terms of a decreasing sequence. Usually stop when $|answer_{i+1} - answer_i| \leq \epsilon \times answer_i$ for some ϵ , and i isn't too small to avoid **premature termination**, where successive answers are too close by chance. Often use some condition-number-related property of the problem such as the $\epsilon \times \infty$ -norm of the input to make sure that don't compute beyond the expected round-off. The ∞ -norm is appealing from the worst-case point of view, but sometimes use the 2-norm for convenience. How the ∞ -norm is defined matters from mathematical view—the maximum looks at all values, and the limit of the L_p norm for $p \rightarrow \infty$ ignores sets of measure 0.
- **Relative**—unlike for the first two, don't know the answer to compute relative to it, so usually look at expected rounding errors. I.e., carry out the process until going further would only get accuracy beyond $\epsilon_{\text{machine}}$ —e.g., this is common in matrix algorithms.

These and any others aren't full-proof because the exact answer is usually unknown, so, e.g., can converge to a wrong value.

For many algorithms the relative error is much more difficult to control than the absolute error. E.g., for binary search equation solving in an interval, shrink the interval with respect to the absolute original

length, and not the significant digits in the current subdivision. This is optimization error. But in practice relative error control almost always works well because algorithms improve absolute error until relative error is good enough, though this is infeasible in the worst case. Users of an API must always understand what error control is used and what it means.

An algorithm's **rate of convergence** quantifies precision gain per iteration. It can be expressed in several ways:

- The intuitive, error $\leq O(\text{some function of the number of iterations } n)$. This defines the **order of a method**, i.e., by analogy to a polynomial. But don't say " $O(n)$ means linear convergence" because the latter is reserved for something else in numerical analysis.
- The iterative-specific, $\text{error}_{i+1} = O(\text{some function of error}_i)$, i.e., the number of extra digits gained per iteration. For **linear convergence** the error shrinks by a constant factor in every iteration, leading to convergence exponential in n . A good example of this is binary search, where the length of the remaining interval is the error. **Quadratic convergence** is super-exponential in n by additionally squaring the error.

Can also think of these as **arithmetic** and **geometric convergence**, respectively. A lot of numerical analysis is about coming up with approximations and processes that result in asymptotically faster convergence. In the latter case it's usually worth it, though a common advice is that linear convergence is enough. In the former other factors usually matter more, and usually strive for quadratic or cubic order.

An interesting decision is picking the explicit termination precision for a given problem, very generally given by finding $f(x)$ for the appropriate functional f . Some choices:

- Do as elementary functions do (from the "do as the integers do" in C++), i.e., compute $f(x)$ to full precision, and correctly round the result. This is generally too much to ask for even if have stability, so suffice for as much precision as possible. But note that elementary function routines don't take limits on the number of evaluations or precision from the user, and go for correct rounding or at least as precise computation as feasible.
- Only go up to the needed precision with minimal computation, usually measured in terms of function evaluations. This can lead to substantial savings when only need few digits of precision for applications such as machine learning.

The former is more of a challenge because computing to as many digits as possible can be of substantial interest, and excessively saving on function evaluations and other resources is often silly. Also usually smooth functions are cheap to evaluate, and expensive-to-evaluate ones, such as simulation output, aren't even Lipschitz, so can't hope for much accuracy. To get the best of both approaches, use algorithms that allow computation to full precision and a smaller termination precision as needed, though in some cases different algorithms may do better at getting low precision. In distant future will probably switch from double to quadruple precision, etc., and algorithms should ideally not be bothered by this in a sense that maximum-precision computations using them would still be feasible.

So algorithms that need a termination precision typically use as default the more appropriate of:

```
double defaultEps = sqrt(numeric_limits<double>::epsilon());
double highPrecEps = 100 * numeric_limits<double>::epsilon();
```

The 100 is somewhat arbitrarily, and 1000 or a similar number will do just as well to account for accumulated rounding errors.

22.7 Other Common Themes

Again, like statistical and machine learning algorithms, numerical algorithms work usually but not always. Various libraries make it seem that the problems for which implementations exist are solved completely by them, but that's far from the truth:

- No code can solve problems that have no solutions and nearby problems that are ill-conditioned. At best good libraries do well on typical problems—e.g., see Nash (2014) for a review and occasional criticism of many R tools for optimization.
- The black box you get may not be the black box you want because certain things are impossible and need extra information such as termination precision to be even well-defined.
- Open-source replicas of proprietary numerical computing software, such as Octave or MATLAB, may not implement the same algorithms for the same tasks—they respectively use the latest open source and proprietary libraries.

Compared to a basic algorithm in a book, at best a library additionally offers:

- Algorithm selection
- Implementations that take care of corner cases

- Extensive test suites, including use cases “at the edge of the machine”

E.g., take a look at the documentation and the source code of the GNU scientific library (Free Software Foundation 2017). It reimplements many algorithms from high-quality classical libraries such as QUADPACK, has useful tests for common use cases, and references papers from which implementation details were taken. But it’s unclear how good robustness logic, machine-edge behavior, and algorithm selection are relative to known good proprietary algorithms such as those in MATLAB.

Analytical preprocessing is essential when black-box methods don’t work well as is. See Bornemann et al. (2004) and Bornemann (2016) for a collection of problems. In general it needn’t succeed, and many problems don’t have good solutions with current (and probably future) methods. The main task of numerical research is to extend the set of problems solvable by black-box or semi-analytic methods. Other than needing strong mathematical knowledge, a practical problem with analytical preprocessing is that modern problems aren’t toy problems and may be too complicated and large-scale for human analysis.

Good numerical analysis consists of studying all of:

- Approximation and optimization errors of the algorithm
- Conditioning of the problem
- Stability properties of the algorithm

Further evidence against universal effectiveness of black boxes comes from complexity theories of exact arithmetic. There are some negative results—e.g., deterministic integration without further assumptions isn’t solvable (a finite number of function calls leads to finite **information**; Traub & Werschulz 1998). So pay attention to convergence and performance theorems of various algorithms—if the conditions under which they hold aren’t satisfied, at best get heuristic algorithms.

For convergent and stable algorithms it seems safe to assume 10-digit precision or so when computing with double precision, with high likelihood that more digits are correct, as long as the problem isn’t ill-conditioned. A general rule of thumb (Bornemann et al. 2004) is that when several unrelated algorithms give the same answer to a certain precision, than that is the correct answer to that precision. Using several algorithms may be worth investing into if the cost of accepting inaccurate answers is high.

A general technique to check for stability and lack of coding mistakes is to run the algorithm with randomly perturbed data, with perturbations of same magnitude as data accuracy, and check the distribution of the outputs. Given some idea of by how much to perturb, this is fully automatic—in case of multivariate outputs can check norms against the unperturbed solution.

22.8 Developing Robust Numerical Software

Given that full-proof black-box solutions are impossible, want software to be maximally robust. Numerical algorithms have different levels of engineering (Cody 1974):

1. Textbook code or pseudocode—often exclude essential details such as convergence and termination conditions, data structures used, and secondary arguments to known sub-algorithms.
2. Algorithms in various publications with all main details filled in. Programs in this book generally have minimal level-2 cleverness, only enough for basic robustness and efficiency. Same for level 3, where only the easy checks are made.
3. Algorithms in various highly used libraries—these are generally more sophisticated than (2) by taking care of language-specific and some language-independent issues such as checking inputs in all cases, etc.

Often create a **metaalgorithm**, i.e., a level-1 algorithm beefed up by level-2 logic that consists of other level-1 algorithms. But metalogic is usually much more heuristic than the main algorithm, and the problems associated with approximation, estimation, and optimization errors apply to it as well.

A common good practice in software engineering is coming up with a **specification** for the algorithm that consists of at least pre- and post-conditions. But for numerical algorithms such specifications are impossible, except for a few basic ones, due to the issues brought up by various types of errors. Usually in a good API the documentation mentions the main algorithm used for the problem, with the expectation that the user will be able to determine roughly how robust the code is, but this only shows level-1 details. An expectation is that an algorithm gives a good estimate of the answer with a good estimate of the error of that answer, unless the requested precision is satisfied. Perfect error estimates are impossible in general (Traub & Werschulz 1998). As with statistical inference, need to apply scientific judgment to decide how much to trust the conclusions, and in most cases only extensive testing on the problems of interest will form that trust.

While most numerical algorithms are standard and may have standard straightforward implementations, robustness improvements rarely are, and many implementations end up making their own choices. In

this book an attempt was made to explain the logic behind any choices. Eventually, published literature should address robustness logic of many algorithms. The best research solution is to study available well-tested implementations. See Miller (1984) for a discussion of robustness properties of several algorithms, and Rice (1992) for examples of testing library implementation for robustness (the latter sometimes recommends what seems to be excessive robustness logic).

To qualify an algorithm for inclusion in a high-quality library, want to reduce the hope-and-pray factor as much as possible, so as much as reasonable ask that:

- The chosen basic algorithm has good theoretical properties—i.e., converges, quickly, and is stable.
- Termination is guaranteed if all subalgorithms terminate—bad inputs lead to error messages, not crashes.
- Code defensively—e.g., handle unexpected ∞ , NaN, and possible out-of-range values in inputs and **information queries** (function or derivative calls in the terminology of information complexity). Must expect any function call or even a simple operation to give an exceptional answer (i.e., think of a C++ program where every operator is overloaded and can throw and exception due to things like running out of memory), and not only because of floating point arithmetic.
- The range of problems it can solve is maximally extended. It's often hard to identify the set of floating point inputs for which a good enough solution is returned. Tricks like overflow/underflow avoidance enlarge this set but still don't help identify it. E.g., it may be worth checking for overflow in x^2 but not in $2x$. Though not doing so technically doesn't work as well as possible for all possible inputs, it's faster and simpler for almost all useful inputs.
- It detects bad inputs—particularly when the solution doesn't exist or the inputs don't satisfy the required conditions. Some algorithms also estimate condition numbers. But remember that checking structural preconditions is ill-conditioned, so, e.g., only check if a matrix symmetric to low precision. A general pattern is to do the main computation anyway if not too expensive, and handle the NaN result if get it—this is safer than doing input checks and can be done in addition to them. Whatever checks can be done reasonably should be done.
- It provides an error estimate—in most cases just a conservative heuristic that is a reasonable guess. In general it's impossible to estimate the error reliably in all cases. E.g., any adaptive convergence criteria for tasks such as integration (see in the “Numerical Algorithms—Working with Functions” chapter) can be fooled. So can't guarantee that a method that failed will report failure, large error, or small error. Need some care here too—e.g., answers ∞ and NaN can't come with a finite precision. Also consider checking whether the answer satisfies some conditions, such as range checks, where appropriate.
- The API is well-designed—have minimum parameters, and those that can have reasonable defaults. In general choose between robustness and convenience/efficiency because a catch-all interface can't do both. E.g., if a subalgorithm has a parameter with a good default value, should it be included in the interface of the caller? This is more of a problem for metaalgorithms—different methods for the same problem make different assumptions and so often need different inputs. Here, for the inputs that differ, ask the user for the superset or assume default values of the subalgorithms? It may seem that having two functions, one that wants a superset and another that calls the first with the defaults will always work, but computing the superset may be inefficient or clumsy, so may need several functions for completeness, but this is against simplicity. E.g., one differential equation solver may need the Jacobian of the equations, and others don't—Shampine & Reichelt (1997) chose to compute the Jacobian numerically where needed, avoiding asking the user.
- Avoid excessive inefficiency. At least should be thoughtful about the termination criteria and algorithm selection. Might also need to cut corners on some of the above such as settling for a much cheaper but less accurate error estimate.
- It has been extensively tested—need an extensive, well-thought-out test suite that tries to cover as many distinct cases as possible, even floating point numbers “at the edge of the machine”. A good test case is where the basic algorithm gives the exact answer in theory, but should include some problems where the answer is close to the worst-case. For single-number inputs a good technique is to try all `float` numbers, and see if get any surprises. For larger inputs can try randomly generated words. It's generally hard to define acceptable accuracy for a test case to pass. One option is to use a low single-word precision, and expect everything to pass. A good test technique is to verify various algebraic identities from answers to somewhat different problems. E.g., to test an integration algorithm check that $\int_{[a,b]} f + \int_{[b,c]} f = \int_{[a,c]} f$ up to the returned error estimates. Otherwise may need arbitrary-precision arithmetic to verify answers. Also want to be able to decide automatically

if the answer on a test case is accurate enough, but need to commit to a lower accuracy to not be affected by tests cases that lead to much less accuracy than typical problems.

In designing maximally robust algorithms it may help to understand how they would be worked out by hand—this may sometimes reveal better way to automate certain tasks. But none of this makes an algorithm fully robust. Even the best library generality has no chance if:

- The problem is very ill-conditioned—hard to detect even if try.
- The problem scale is unexpected—this influences convergence criteria based on norms and rounding errors of smaller variables.
- The main f in the problem isn't smooth enough or has near-singularities or other special behavior that causes numerical difficulty—usually impossible to detect—e.g., f may have a singularity between two machine numbers which isn't noticeable at either of them (Ueberhuber 1997b).

Numerical methods generally work well for certain problems in theory and for a more extended class of problems in practice. But, based on game theory, an implementer still has an optimal randomized strategy for how to implement a method—it needn't be known, but with considerable work can usually determine what is worth doing and what isn't. Typically the decision to stop developing further is justified when:

- The current implementation is well-tested and performs well.
- Reasonable published literature research has been done to not miss any important implementation improvements. Look for algorithm selection, robustness improvements, and test use cases, i.e., the main economic value added by libraries. Good test case collections are particularly hard to find, and tend to be discussed in some of the most heavily cited papers in the domain-specific area; though one usually unfulfilled desiderata is that want every problem to be justified. Multi-problem test sets currently don't seem to be available, except maybe in open-source libraries. This usually needs to be revisited every few years, and new results need retesting, so algorithm libraries typically don't incorporate latest research when nearing a release.

The user can also do experimental evaluation when solving a problem of interest. E.g., given inputs x can do **sensitivity analysis** by viewing the performed computation as some function f and calculating $f(x)$ using finite differences (see in the “Numerical Algorithms—Working with Functions” chapter)—if see a big surprise then it's a sign of problems. Can also try random perturbations of x of the same magnitude as the finite difference default, and see how the result changes. Either the derivative (or subgradient—see the “Numerical Optimization” chapter) exists, or have jumps and ill-conditioning. Iterating through machine numbers by 1 ulp can also be useful, as would random perturbations by $O(\text{machine } \epsilon)$.

It's up to the user to decide when to accept a particular implementation, and this is often not easy but better than having no solution at all due to needing 100% correctness. The same logic applies to statistical and machine learning algorithms. Comparing algorithms is also difficult. Reliable algorithms can run as is, and unreliable ones need an expert babysitter.

Numerical methods, and many other algorithmic fields such as machine learning, cryptography, optimization, data compression are getting to the unavoidable librarization. It becomes silly to use one's own code instead of the available libraries. Only occasionally need to recreate them in a new language or special hardware. Researchers are working on improving the libraries and the theory. Education moves from numerical methods to scientific computation, where discuss how to use a library and perhaps some very basic algorithms for general appreciation, particularly to understand the limitations of various black boxes. E.g., condition analysis is always relevant for any implementation, while stability analysis depends on the exact algorithm. But somebody has to write books for researchers about the actual numerical methods, even though the implementations discussed are between those in the library use books and those in the libraries. Though not enough details are given in such cases, this is the best possible case (in some sense stability for books!).

Validation of numerical software is beyond the scope of numerical analysis, but one should check that the model being solved makes sense. A general pattern is that continual improvements in algorithms and hardware enable solving more complex models, which reduces the need to use artificial simplifications in models.

22.9 Matrix Algebra

Most algorithms are specifications of linear algebra methods. A common matrix is the identity, which for 2×2 is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

An array and a mapping from 2D coordinates to its indices represent a matrix. As in linear algebra, assume a vector to be a column vector. The complexities are for square matrices with n rows. Though the

item type is arbitrary for generality, all stability analysis will be for `double`.

```
template<typename ITEM> struct Matrix: public ArithmeticType<Matrix<ITEM>>
{
    int rows, columns;
    int index(int row, int column) const
    {
        assert(row >= 0 && row < rows && column >= 0 && column < columns);
        return row + column * rows;
    }
    Vector<ITEM> items;
public:
    ITEM& operator()(int row, int column) {return items[index(row, column)];}
    ITEM const& operator()(int row, int column) const
        {return items[index(row, column)];}
    Matrix(int theRows, int theColumns): rows(theRows), columns(theColumns),
        items(rows * columns) {}
};
```

Multiplication by a scalar is $O(n)$. E.g., $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times 2 = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$.

```
Matrix operator*(ITEM const& scalar)
{
    items *= scalar;
    return *this;
}
friend Matrix operator*(ITEM const& scalar, Matrix const& a)
{
    Matrix result(a);
    return result *= scalar;
}
friend Matrix operator*(Matrix const& a, ITEM const& scalar)
    {return scalar * a;}
```

Addition and subtraction are $O(n)$. E.g., $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 0 & -2 \\ -3 & -3 \end{bmatrix}$.

```
Matrix& operator+=(Matrix const& rhs)
{
    assert(rows == rhs.rows && columns == rhs.columns);
    items += rhs.items;
    return *this;
}
Matrix& operator-=(Matrix const& rhs)
{
    assert(rows == rhs.rows && columns == rhs.columns);
    items -= rhs.items;
    return *this;
}
```

So far stability properties are straightforward. Both addition and scalar multiplication act component-wise and inherit the stability of the underlying arithmetic. But for more complex operations it gets more complicated. Higham (2002) is the most comprehensive reference for stability analysis, and going forward, any computation error bound not explicitly cited comes from there. Also see Trangenstein (2018a) and Trangenstein (2018b) for an original treatment. As notation for component-wise bounds, for vector x let $|x|$ denote a vector or matrix with x_i replaced by $|x_i|$ (don't confuse with the determinant notation). Also

let $\gamma_n = \frac{n\epsilon_{\text{machine}}}{1+n\epsilon_{\text{machine}}}$, and $f_l(\text{exact quantity})$ refer to the floating-point computation of that quantity by some

algorithm. Then for inner product have stability: $f_l(xy) = (x + \Delta x)y = x(y + \Delta y)$, where $\Delta x \leq \gamma_n|x|$ and $\Delta y \leq \gamma_n|y|$. A forward bound is also available: $|xy - f_l(xy)| \leq \gamma_n|x||y|$. A bad cancellation can occur, e.g., when $x = [1, -1 + \epsilon]$. But in some cases, such as computing a sum of squares, have guaranteed small forward error. For an outer product have forward bound $x \otimes y - f_l(x \otimes y) = \Delta$, where $|\Delta| \leq \epsilon_{\text{machine}}|x \otimes y|$, but a backward bound doesn't exist. Can get a norm-based bound from a more general component-wise bound by changing the " $|$ " to a norm.

Multiplication is $O(n^2)$. E.g., $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$. Calculate it using row-by-column multiplication.

Left multiplication by a vector is a special case of this. Right multiplication is equivalent to the left one using the transpose.

```
Matrix& operator*=(Matrix const& rhs)
{
    assert(columns == rhs.rows);
    Matrix result(rows, rhs.columns);
    for(int i = 0; i < rows; ++i) //row
        for(int j = 0; j < rhs.columns; ++j) //by column
    {
        ITEM sum(0);
        for(int k = 0; k < rhs.rows; ++k)
            sum += (*this)(i, k) * rhs(k, j);
        result(i, j) += sum;
    }
    return *this = result;
}
Vector<ITEM> operator*(Vector<ITEM> const& v) const
{
    assert(columns == v.getSize());
    Vector<ITEM> result(rows);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < columns; ++j)
            result[i] += (*this)(i, j) * v[j];
    return result;
}
friend Vector<ITEM> operator*(Vector<ITEM> const& v, Matrix const& m)
{return m.transpose() * v;}
```

For matrix-vector multiplication have stability: $f(Ax) = (A + \Delta A)x$, where $|\Delta A| \leq \gamma_n |A|$. Also get a forward bound: $|Ax - f(Ax)| \leq \gamma_n |A| \|x\|$. As outer product, matrix multiplication isn't backward stable overall—it is only if consider a particular column. But have a forward bound: $|AB - f(AB)| \leq \gamma_n |A| |B|$.

Identity creation and transpose are $O(n)$, both perfectly stable. E.g., $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$.

```
static Matrix identity(int n)
{
    Matrix result(n, n);
    for(int i = 0; i < n; ++i) result(i, i) = ITEM(1);
    return result;
}
Matrix transpose() const
{
    Matrix result(columns, rows);
    for(int i = 0; i < rows; ++i)
        for(int j = 0; j < columns; ++j) result(j, i) = (*this)(i, j);
    return result;
}
```

It's also useful to work with submatrices (the assign function is a member function, the read one isn't):

```
template<typename ITEM, typename MATRIX> Matrix<ITEM> submatrix(
    MATRIX const& A, int r1, int r2, int c1, int c2)
{
    assert(r1 >= 0 && r2 < A.getRows() && c1 >= 0 && c2 < A.getColumns() &&
           r1 <= r2 && c1 <= c2);
    Matrix<ITEM> result(r2 - r1 + 1, c2 - c1 + 1);
    for(int r = r1; r <= r2; ++r)
        for(int c = c1; c <= c2; ++c) result(r - r1, c - c1) = A(r, c);
    return result;
}
```

Can convert various sparse and implicit matrices (discussed later in the chapter) to dense:

```
template<typename ITEM, typename MATRIX> Matrix<ITEM> toDense(MATRIX const& A)
```

```

return submatrix<ITEM>(A, 0, A.getRows() - 1, 0, A.getColumns() - 1);
void assignSubmatrix(Matrix const& sub, int r1, int c1)
{
    assert(r1 >= 0 && c1 >= 0 && r1 + sub.getRows() <= rows &&
           c1 + sub.getColumns() <= columns);
    for(int r = r1; r - r1 < sub.getRows(); ++r)
        for(int c = c1; c - c1 < sub.getColumns(); ++c)
            (*this)(r, c) = sub(r - r1, c - c1);
}

```

22.10 Matrix Norms

A norm must satisfy metric properties such as the triangle inequality. A common construction is to reduce to the corresponding vector norm using $\|A\| = \max_{\|x\|>0} \frac{\|Ax\|}{\|x\|}$. This works for any vector norm. My implementation of a vector Euclidean norm (see the “Fundamental Data Structures” chapter) is naive but reasonable because it causes no problem with realistic inputs. A basic improvement for avoiding overflow is to factor out the largest item. See Beebe (2017) for some others.

The resulting ∞ -norm is easy to compute: $\|A\|_\infty = \max_r \sum_c |A[r, c]|$; for vector x , $\|x\|_\infty = \max_i |x_i|$. For complex numbers $|x_i| = \text{radius}(x_i)$ (Ford 2014). The 2-norm can be computed much more expensively using the SVD (discussed later in the chapter).

```

template<typename X> double normInf(Vector<X> const& x)
{//works for complex vector too
    int xInf = 0;
    for(int i = 0; i < x.getSize(); ++i)
    {
        double ax = abs(x[i]);
        if(isnan(ax)) return ax; //check for NaN before max
        xInf = max(xInf, ax);
    }
    return xInf;
}

double normInf(Matrix<double> const& A)
{
    double m = A.getRows();
    Vector<double> rowSums(m);
    for(int r = 0; r < m; ++r)
        for(int c = 0; c < A.getColumns(); ++c) rowSums[r] += abs(A(r, c));
    return valMax(rowSums.getArray(), m);
}

```

Another common norm that is easy to calculate is **Frobenius norm**, defined by $\|A\|_F = \sqrt{\sum_{r,c} A[r, c]^2}$. In some special cases it's easier to work with analytically but otherwise rarely used.

```

double normFrobenius(Matrix<double> const& A)
{
    double sum = 0;
    for(int r = 0; r < A.getRows(); ++r)
        for(int c = 0; c < A.getColumns(); ++c) sum += A(r, c) * A(r, c);
    return sqrt(sum);
}

```

Use of ∞ -norm for functions and discrete objects such as vectors and matrices is very different—in the latter case any norm is within a dimension-dependent constant factor of any other, while in the former case this doesn't hold (Süli & Mayers 2003). E.g., L_p norms are defined by integration which isn't sensitive to f 's behavior on sets of measure 0, but the ∞ -norm is affected. In practice this is almost never a problem though, so the latter is preferred in many cases due to the worst-case meaning and usually straightforward calculation.

22.11 LUP Decomposition

Can break up a square matrix A such that $LU = A$; L is a lower-triangular matrix with 0 entries on and above the diagonal, and U is upper-triangular with 0 entries below the diagonal. This follows Gaussian elimi-

ination with pivoting from a linear algebra class.

Transposing a row in an augmented matrix doesn't change the matrix equation solution. P is a permutation matrix obtained by transposing rows but represented as an array. Permute to pick pivots with the largest absolute value to avoid 0 pivots and improve numerical accuracy, i.e., compute $LU = PA$. For implementation a single matrix packs L and U . Gaussian elimination with partial pivoting computes this in $O(n^3)$ time (Cormen et al. 2009). In linear algebra classes, this method is taught for solving matrix equations. For a matrix augmented with a right-hand side:

1. \forall column
2. Among the rows \leq at-diagonal row pick the one with the largest absolute value as pivot
3. Swap the pivot's row with the diagonal row if not the same
4. \forall row $<$ at-diagonal row
5. Subtract the multiple of the at-diagonal row needed to make the column's value of the row 0

Get an upper-triangular matrix U and some right-hand side, making the equation is easy to solve. E.g.,

omitting the augmentation, for a 3×3 matrix get
$$\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \end{bmatrix} \rightarrow \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \end{bmatrix}$$
. For LUP effectively

augment with an identity-matrix right-hand side L , and apply the updates to it. Apart from a separate array to store the permutation, put both U and L to the same matrix to save space.

For singular A get pivot = 0, but continue to the next step anyway (Golub & Van Loan 2012). This still computes a correct decomposition, though due to the singularity it's unlikely to be useful. The singularity determination needn't be accurate—it can miss near-singularities—probably the most stable method is to use SVD to compute the rank of A (discussed later in the chapter).

```
template<typename ITEM = double> struct LUP
{
    Matrix<ITEM> d;
    Vector<int> permutation;
    bool isSingular;
    LUP(Matrix<ITEM> const& a) : d(a), isSingular(false)
    {
        assert(d.rows == d.columns); //first create identity ?
        for(int i = 0; i < d.rows; ++i) permutation.append(i);
        for(int i = 0; i < d.rows; ++i)
        {
            ITEM p = 0;
            int entering = -1;
            for(int j = i; j < d.rows; ++j)
                if(abs(d(i, j)) > p)
                {
                    p = abs(d(i, j));
                    entering = i;
                }
            if(entering == -1)
            {
                isSingular = true;
                continue;
            }
            swap(permutation[i], permutation[entering]);
            for(int j = 0; j < d.rows; ++j) swap(d(i, j), d(entering, j));
            for(int j = i + 1; j < d.rows; ++j)
            {
                d(j, i) /= d(i, i);
                for(int k = i + 1; k < d.rows; ++k)
                    d(j, k) -= d(j, i) * d(i, k);
            }
        }
    }
};
```

Partial pivoting is commonly taught as "pivoting" in linear algebra classes. Also interesting are no pivoting and **complete pivoting** (see Golub & van Loan 2012 for details). A basic stability bound for all options:

$\|L\|U\|_{\infty} \leq (1 + 2(n^2 - n)\rho_n) \|AP\|_{\infty}$, where ρ_n is the **growth factor**. A well-documented statistical phenomenon is that in practice both partial and complete pivoting tend to have low ρ_n , even though for the latter $\rho_n \leq O(\sqrt{nn^{1/4}\log(n)})$ but for the former $\rho_n \leq 2^{n-1}$. So all major implementations use partial and not complete pivoting for efficiency.

The determinant = \prod diagonal entries of U . Its sign changes when the permutation maps an even entry to an odd one or vice versa. Mathematically the determinant is important, but numerically not at all. E.g., if A is scaled by a constant c , its determinant is scaled by c^n , so checking if the determinant < some ϵ is pointless. When need the determinant for mathematical reasons, usually it's more useful to work with $\log(\text{determinant})$ to avoid underflow/overflow:

```
double logAbsDet() const // -inf if 0
{
    double result = 0;
    for(int i = 0; i < d.rows; ++i) result += log(abs(d(i, i)));
    return result;
}
int signDet() const
{
    int sign = 1;
    for(int i = 0; i < d.rows; ++i)
    {
        if(d(i, i) < 0) sign *= -1;
        if(permuation[i] % 2 != i % 2) sign *= -1;
    }
    return sign;
}
```

See Higham (2002, p279) for a stability discussion.

Back-substitution solves $Ux = b$:

```
Vector<double> backsubstitution(Matrix<double> const& U, Vector<double> b)
{// overwrite b with solution of Ux = b
    int n = b.getSize();
    assert(U.getRows() == n && U.getColumns() == n);
    for(int i = n - 1; i >= 0; --i)
    {
        for(int j = i + 1; j < n; ++j) b[i] -= b[j] * U(i, j);
        b[i] /= U(i, i);
    }
    return b;
}
```

The function applies to all decompositions that can solve equations, such as QR (discussed later in the chapter). Have stability for solving with nonsingular triangular system $T = U$ or $T = L$: $(T + \Delta T)f(x) = b$, where $|\Delta T| \leq \gamma_n |T|$. Also get a component-wise forward error: $\frac{\|x - f(x)\|_{\infty}}{\|x\|_{\infty}} \leq \frac{\text{cond}(T, x)\gamma_n}{1 - \text{cond}(T)\gamma_n}$, where $\text{cond}(A, x) = \frac{\|A\|A^{-1}\|x\|_{\infty}}{\|x\|_{\infty}}$, and $\text{cond}(A) = \|A\|A^{-1}\|_{\infty}$. These component-wise condition numbers aren't the same as the norm-based $\kappa(A)$.

Forward substitution solves $Ax = b$ in $O(n^2)$ time by solving $Ly = Pb$ and then $Ux = y$:

```
Vector<ITEM> solve(Vector<ITEM> const& b) const
{
    Vector<ITEM> y(d.rows);
    for(int i = 0; i < d.rows; ++i)
    {
        y[i] = b[permuation[i]];
        for(int j = 0; j < i; ++j) y[i] -= y[j] * d(i, j);
    }
    return backsubstitution(d, y);
}
```

For complete equation solving have a stability bound: $(AP + \Delta AP)f(x) = b$, where $|\Delta AP| \leq \gamma_{3n} \|f(L)\| \|f(U)\|$. Conditioning analysis that describes the change in solution with a change in A

and b leads to $\frac{\|\Delta x\|}{\|x\|} \leq \left(\frac{\kappa(A)}{1 - \kappa(A)\|\Delta A\|/\|A\|} \right) \left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|} \right)$ when A is nonsingular, $b \neq 0$, and $\|\Delta A\| < \frac{1}{\|A^{-1}\|}$ (Datta 2010, p65). For component-wise analysis see Datta (2010, p143) and Higham (2002, p123).

Augmenting the matrix with the identity of the same size and solving computes A^{-1} in $O(n^3)$ time. \forall identity column the computation solves a matrix-vector equation (this also applies to other decompositions):

```
template<typename DECOMPOSITION> Matrix<double> inverse(DECOMPOSITION const& d,
    int n)
{
    Vector<double> identityRow(n, 0);
    Matrix<double> result(n, n);
    for(int i = 0; i < n; ++i)
    {
        identityRow[i] = 1;
        Vector<double> column = d.solve(identityRow);
        identityRow[i] = 0;
        for(int j = 0; j < n; ++j) result(j, i) = column[j];
    }
    return result;
}
```

Avoid using the inverse computationally due to unnecessary loss of precision—it's best to work with LUP directly. See Ueberhuber (1997b) and Datta (2010, p137) for some tricks for avoiding the inverse. When need to compute it, have a stability bound: $|A f(A^{-1}) - I| \leq \gamma_{O(n)} |f(L)| |f(U)| |f(A^{-1})|$.

After solving $Ax = b$, can calculate the **residual** $r = Ax - b$; r is the observed backward error—i.e., not the worse case. LUP equation solving is stable except for very unlikely worst cases, so $\|r\| = O(\epsilon_{\text{machine}})$ \forall norm if A isn't singular. In particular have $\frac{\|\Delta x\|}{\|x\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}$ Datta (2010, p140).

What is usually wanted is an error estimate for x , i.e., the forward error. If the true solution is x^* , $x - x^* = A^{-1}r$, so $\|x - x^*\| \leq \|A^{-1}\| \|r\|$, which is a good estimate for absolute error. Relative error = $\frac{\|x - x^*\|}{\|x^*\|}$, and from $\|A\| \|x^*\| \geq \|b\|$ and backward error get $\frac{\|x - x^*\|}{\|x^*\|} \leq \kappa(A) \frac{\|r\|}{\|b\|}$, which is a good estimate for relative error; $\kappa(A) = \|A\| \|A^{-1}\|$ is the (relative) **condition number** of A .

Computing $\|A^{-1}\|_\infty$ for $\kappa(A)$ estimates has no shortcut. The standard alternative is to estimate $\kappa(A)$ by other means. See Higham (2002) for some algorithms, but they are complicated and not discussed further. SVD can compute $\|A^{-1}\|_2$ accurately (discussed later in the chapter), so usually use $\kappa_2(A)$ when need an error estimate:

```
pair<double, double> estimateMatrixEquationError2Norm(Matrix<double> const& A,
    Vector<double> const& b, Vector<double> const& x)
//abs, rel
SVD svd(A);
double nr = norm(A * x - b);
return make_pair(nr * svd.norm2Inv(), nr * svd.condition2() / norm(b));
}
```

This is an expensive estimate though, with $O(n^3)$ runtime and a much larger constant than for LUP (see estimates for SVD in Golub & Van Loan 2012), so use it only when needed.

22.12 Cholesky Decomposition

If A is symmetric and positive definite, can compute a lower-triangular matrix L such that $A = LL^T$. E.g., this is useful for multidimensional covariance matrices.

Compute L column-by-column:

1. $\forall c \text{ and } r \geq c$
2. $s = A(c, c) - \sum_{0 \leq k < c} L(r, k)L(c, k)$
3. $L(c, c) = \sqrt{s}$
4. $L(r, c) = \frac{s}{L(c, c)}$

```
template<typename ITEM> struct Cholesky
{
```

```

Matrix<ITEM> l;
bool failed;
Cholesky(Matrix<ITEM> const& a): l(a.rows, a.columns), failed(false)
{//a must be symmetric and positive definite
    for(int c = 0; c < l.columns; ++c)
        for(int r = c; r < l.rows; ++r)
    {
        ITEM sum = a(r, c);
        for(int k = 0; k < c; ++k) sum -= l(r, k) * l(c, k);
        if(r == c)
        {
            if(sum <= 0){failed = true; return;}
            l(c, c) = sqrt(sum);
        }
        else l(r, c) = sum/l(c, c);
    }
}
};

```

Need $O(n^3)$ time. Have stability: $f(L)f(L)^T = A + \Delta A$, where $|\Delta A| \leq \gamma_{n+1} |f(L)| |f(L)^T|$.

Cholesky decomposition can be used, among other things, for calculating determinants and solving equations:

- $\det(A) = \det(L)^2$
- To solve $Ax = b$ solve $L^T x = \text{solve}(Lx = b)$. For the latter use back-substitution.

```

double logDet() const //-inf if 0
{
    assert(!failed);
    double result = 0;
    for(int i = 0; i < l.rows; ++i) result += log(abs(l(i, i)));
    return 2 * result;
}
Vector<ITEM> solve(Vector<ITEM> b) const
{
    int n = b.getSize();
    assert(l.getRows() == n && l.getColumns() == n);
    assert(!failed);
    for(int i = 0; i < n; ++i)
    {
        for(int j = 0; j < i; ++j) b[i] -= b[j] * l(i, j);
        b[i] /= l(i, i);
    }
    return backsubstitution(l.transpose(), b);
}

```

For equation solving have stability bound: $(A + \Delta A)f(x) = b$, where $|\Delta A| \leq \gamma_{3n+1} |f(L)| |f(L)^T|$.

22.13 Band Matrices

In many cases the matrix is symmetric and 0 except for several diagonals close to the main one, e.g.,

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 5 \\ 0 & 6 & 7 \end{bmatrix}$$
, so represent it more efficiently. A common case is a **tridiagonal matrix**. In the implementation

only some functions are present (not the full set of general matrix functions).

```

template<int D = 3, typename ITEM = double> struct BandMatrix
//below main diagonal d < 0
    Vector<Vector<ITEM>> diagonals;
    int diagIndex(int d) const {return D/2 + d;} //"0" is the main diagonal
    int rowIndex(int r, int d) const {return r + min(0, d);}
    int diag(int r, int c) const {return c - r;} //storage index
    int getRows() const {return diagonals[diagIndex(0)].getSize();}
    int getColumns() const {return getRows();}
    void setupBands(int n)

```

```

//initialize the banded parts
    assert(n > 0 && D % 2 == 1);
    for(int d = -D/2; d <= D/2; ++d)
        diagonals[diagIndex(d)] = Vector<ITEM>(n - abs(d));
}
BandMatrix(int n): diagonals(D) {setupBands(n); }
BandMatrix(Matrix<ITEM> const& A): diagonals(D)
{//copy the banded part only
    int n = A.getRows();
    setupBands(n); //this must be called first
    assert(n == A.getColumns());
    for(int r = 0; r < n; ++r)
        for(int c = max(0, r - D/2); c <= min(n - 1, r + D/2); ++c)
            (*this)(r, c) = A(r, c);
}
ITEM operator()(int r, int c) const
{
    int d = diag(r, c);
    assert(r >= 0 && r < getRows() && c >= 0 && c < getColumns());
    return abs(d) <= D/2 ? diagonals[diagIndex(d)][rowIndex(r, d)] : 0;
}
ITEM& operator()(int r, int c)
{
    int d = diag(r, c);
    assert(r >= 0 && r < getRows() && c >= 0 && c < getColumns() &&
           abs(d) <= D/2);
    return diagonals[diagIndex(d)][rowIndex(r, d)];
}
static BandMatrix identity(int n)
{
    BandMatrix result(n);
    result.diagonals[result.diagIndex(0)] = Vector<ITEM>(n, 1);
    return result;
}
BandMatrix& operator+=(BandMatrix const& A)
{
    assert(getRows() == A.getRows());
    diagonals += A.diagonals;
    return *this;
}
BandMatrix& operator*=(ITEM const& a)
{
    diagonals *= a;
    return *this;
}
friend BandMatrix operator*(ITEM const& scalar, BandMatrix const& A)
{
    BandMatrix result(A);
    return result *= scalar;
}
BandMatrix operator-() const
{
    BandMatrix result(*this);
    return result *= -1;
}
BandMatrix& operator-=(BandMatrix const& A){return *this += -A;}
void assignSubmatrix(Matrix<ITEM> const& sub, int r1, int c1)
{//assign only the tridiagonal part
    assert(r1 >= 0 && c1 >= 0 && r1 + sub.getRows() <= getRows() &&
           c1 + sub.getColumns() <= getColumns());
    for(int r = r1; r - r1 < sub.getRows(); ++r)
        for(int c = c1; c - c1 < sub.getColumns(); ++c)
            if(abs(diag(r, c)) <= D/2) (*this)(r, c) = sub(r - r1, c - c1);
}

```

```

    }
};

template<typename ITEM> using TridiagonalMatrix = BandMatrix<3, ITEM>;

```

22.14 Solving Tridiagonal Matrix Equations

These are common in special algorithms such as calculation of cubic spline interpolants and some methods for solving partial differential equations (see the next chapter for both).

A simple approach is Gaussian elimination without pivoting (often called **Gauss-Thomson algorithm**), where take the diagonal entry as the pivot and use its row to eliminate those below.

```

Vector<double> solveTridiag(TridiagonalMatrix<double> const& A,
    Vector<double> r)
{ // U = above diag, D = diag, L = below diag; overwrite r with solution
    Vector<double> U = A.diagonals[2], D = A.diagonals[1];
    Vector<double> const& L = A.diagonals[0];
    int n = r.getSize();
    assert(n > 1 && D.getSize() == n);
    // forward elimination
    U[0] /= D[0];
    r[0] /= D[0];
    for(int i = 1; i < n; ++i)
    {
        double denom = D[i] - L[i - 1] * U[i - 1];
        if(i < n - 1) U[i] /= denom; // allow 0 denom; user handles inf or NaN
        r[i] = (r[i] - L[i - 1] * r[i - 1]) / denom;
    } // back substitution
    for(int i = n - 2; i >= 0; --i) r[i] -= U[i] * r[i + 1];
    return r;
}

```

The runtime is $O(n)$. Pivoting would break the tridiagonal structure, but without it the algorithm needs **diagonal dominance** for stability (every diagonal entry $\geq \sum|\text{other entries in its row}|$). This avoids cancellation of diagonal elements and subsequent divisions by 0. The user must check that this holds before applying the method.

Elimination without pivoting easily extends to more general band matrices, though performance will depend on the number of diagonals, and diagonal dominance becomes less likely. Solving diagonally dominant systems with Gaussian elimination with no pivoting is stable because for the implicitly computed LU decomposition $\|L\|U\|_{\infty} \leq (2n-1)\|A\|_{\infty}$.

22.15 Orthogonal Transformations

LUP decomposition uses elementary row operations to transform matrices. They correspond to multiplications by certain matrices, a sequence of which leads to a useful calculation. Orthogonal matrix multiplication also leads to useful transformations. Q is **orthogonal** if $QQ^T = I$. For an orthogonal matrix the condition number = 1 (Trefethen & Bau 1997), and a sequence of multiplications has condition number = 1. Multiplication by an orthogonal matrix is stable (Datta 2010, p40). Algorithms based on sequences orthogonal matrices are generally stable, though can't directly infer this from the above.

A **Householder reduction** (also called **reflection** or **transformation**) is a multiplication by an orthogo-

nal matrix such that in the result below the specified entry all entries = 0. E.g., $\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \end{bmatrix} \times H = \begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \end{bmatrix}$.

Can also apply it to submatrices. Specifically, let x be the vector of A where wish to eliminate all entries except the first. Then (Higham 2002) **Householder matrix** $H = I - \left(\frac{2}{\|v\|}\right)v \otimes v$, where $v = x$ except

$v[0] = x[0] + \text{sign}(x[0])\|x\|$. Here $x_0 = 0$ causes no problems—the sign function is not undefined at 0 but a subgradient (see the “Numerical Optimization” chapter) of $f(x) = |x|$. This formula is as stable as the one in Golub & Van Loan (2012), but much simpler. For convenience have a special case for x of size 2.

For efficiency don't form H explicitly, so for x of size n need $O(n^2)$ time for matrix multiplications:

- $HA = \left(I - \left(\frac{2}{\|v\|} \right) v \otimes v \right) A = A - \left(\frac{2}{\|v\|} \right) v \otimes v A$
- $AH = A - A \left(\frac{2}{\|v\|} \right) v \otimes v$

```

double sign(double x){return x > 0 ? 1 : -1;}
Vector<double> HouseholderReduction(Vector<double> x)
{
    x[0] += sign(x[0]) * norm(x);
    double normX = norm(x);
    if(normX > 0) x *= 1/normX;
    return x;
}
Vector<double> HouseholderReduction2(double a, double b)
{
    Vector<double> x;
    x.append(a);
    x.append(b);
    return HouseholderReduction(x);
}
template<typename MATRIX> void HouseholderLeftMult(MATRIX& M,
    Vector<double> const& h, int r, int c = 0, int c2 = -1)
{
    if(c2 == -1) c2 = M.getColumns() - 1;
    assert(r + h.getSize() <= M.getRows());
    Matrix<double> sub = submatrix<double>(M, r, r + h.getSize() - 1, c, c2);
    sub -= outerProduct(h * 2, h * sub);
    M.assignSubmatrix(sub, r, c);
}
template<typename MATRIX> void HouseholderRightMult(MATRIX& M,
    Vector<double> const& h, int c, int r = 0, int r2 = -1)
{
    if(r2 == -1) r2 = M.getRows() - 1;
    Matrix<double> sub = submatrix<double>(M, r, r2, c, c + h.getSize() - 1);
    sub -= outerProduct(sub * h, h * 2);
    M.assignSubmatrix(sub, r, c);
}

```

In general operations with outer products are more efficient when done implicitly. Matrix-vector multiplication is another such case. Let $B = A + u \otimes v$. Then $Bx = Ax + u(vx)$, and $xB = xA + (xu)v$:

```

Vector<double> outerProductMultLeft(Vector<double> const& u, //u column, v row
    Vector<double> const& v, Vector<double> const& x)
    {return u * dotProduct(v, x);}
Vector<double> outerProductMultRight(Vector<double> const& u, //u column, v row
    Vector<double> const& v, Vector<double> const& x)
    {return v * dotProduct(x, u);}

```

Multiplication of A by a sequence of Householder matrices implicitly is stable (see Higham 2002, p359).

22.16 QR Decomposition

Like LUP, creating $A = Q^T R$, where R is upper-triangular and Q orthogonal, allows many subsequent operations to be done more efficiently, particularly calculation of eigenvalues.

The general-case algorithm goes through column-by-column, using Householder reductions H_i to zero out the entries below the diagonal, like Gaussian elimination. Then $R = HA$, and $Q = HI$, where $H = \prod_{n-1 \geq i \geq 0} H_i$. This takes $O(n^3)$ time.

A matrix is in **Hessenberg form** when it's 0 below the first subdiagonal—e.g.,

$$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \end{bmatrix}. A$$

sequence of n orthogonal transformations from top to bottom eliminates the subdiagonal in $O(n^2)$ time.

```
struct QRDecomposition
```

```

{
    Matrix<double> Q, R;
    void doHessenbergQR()
    {
        for(int c = 0; c < R.getRows() - 1; ++c)
        {
            Vector<double> x = HouseholderReduction2(R(c, c), R(c + 1, c));
            HouseholderLeftMult(R, x, c, c);
            HouseholderLeftMult(Q, x, c);
        }
    }
    QRDecomposition(Matrix<double> const& A, bool isHessenberg = false):
        Q(Matrix<double>::identity(A.getRows())), R(A)
    {
        int n = R.getRows();
        assert(R.getColumns() == n);
        if(isHessenberg) doHessenbergQR(); //Hessenberg already
        else //regular case
            for(int c = 0; c < n - 1; ++c)
            {
                Vector<double> x(n - c);
                for(int j = c; j < n; ++j) x[j - c] = R(j, c);
                x = HouseholderReduction(x);
                HouseholderLeftMult(R, x, c, c);
                HouseholderLeftMult(Q, x, c);
            }
    }
};
}

```

QR decomposition is stable, as is solving equations with it (see Higham 2002, p360–361).

Solving equations and calculating determinants is almost the same as for LUP. But QR doesn't know the sign of the determinant because $\det(Q) = 1$ or -1 , and $\det(QQ^T) = 1$.

```

Vector<double> solve(Vector<double> const & b) const
{ //solve Rx = Qb to solve Ax = b
    assert(Q.getRows() == b.getSize());
    return backsubstitution(R, Q * b);
}
double logAbsDet() const // -inf if 0
{
    double result = 0;
    for(int i = 0; i < R.rows; ++i) result += log(abs(R(i, i)));
    return result;
}

```

QR allows $O(n^2)$ rank-1 updates, i.e., adding an outer product. Let $A = Q^T R$ (Golub & Van Loan 2012 use Q and not Q^T). Then $A + u \otimes v = Q^T (R + w \otimes v)$ for $w = Qu$. If apply a sequence of orthogonal transformation matrices Q_w , get $(QQ_w)^T (Q_w R + Q_w w \otimes v)$. Let Q_w consist of a sequence that eliminates the last nonzero element of w one-by-one. After this has been done, only the first element of $Q_w w$ is non-zero, so $Q_w w \otimes v$ is 0 except for the first row. And the matrix $Q_w R$ is in Hessenberg form due to the nature of the updates, so their sum can be efficiently decomposed. Q needn't start as an identity for this to work.

```

void rank1Update(Vector<double> const& u, Vector<double> const& v)
{ //first calculate Q_w R
    int n = u.getSize();
    assert(v.getSize() == n && Q.getRows() == n);
    Vector<double> w = Q * u;
    for(int r = n - 1; r > 0; --r)
        { //find orthogonal transformation to eliminate w[r]
            Vector<double> x = HouseholderReduction2(w[r - 1], w[r]);
            Matrix<double> W(2, 1); //apply it to temp submatrix
            W(0, 0) = w[r - 1];
            W(1, 0) = w[r];
            HouseholderLeftMult(W, x, 0);
        }
}

```

```

w[r - 1] = W(0, 0); //copy back the result
w[r] = W(1, 0);
//apply to R also
HouseholderLeftMult(R, x, r - 1, r - 1);
HouseholderLeftMult(Q, x, r - 1);
}
for(int c = 0; c < n; ++c) R(0, c) += w[0] * v[c];
doHessenbergQR(); //process Hessenberg form of R
}

```

22.17 Symmetric Matrix Eigenvalues and Eigenvectors

For $n \times n$ matrix A (Wikipedia 2017b):

- An **eigenvalue** λ is such that $\det(A - \lambda I) = 0$.
- An **eigenvector** v corresponding to λ is such that $Av = \lambda v$.
- Eigenvalues and eigenvectors of real symmetric A are real; otherwise they may be complex.
- Let V be the matrix whose columns are n linearly independent eigenvectors corresponding to the eigenvalues that make up the diagonal of the diagonal matrix Λ , if they exist. Then $A = V\Lambda V^{-1}$ is the **spectral decomposition** of A .

Because eigenvalues are the roots of a polynomial, \exists a finite-time algorithm to calculate them for $n > 4$. So the algorithms must be iterative. Usually use **shifted QR iteration** (Golub & Van Loan 2012):

1. Convert A to Hessenberg form (for efficiency)
2. Until convergence
3. Deflate A (remove converged bottom-right submatrix if any; explained later)
4. Pick shift μ that causes quick deflation
5. $QR = A - \mu I$
6. $A = RQ$

It computes $D = QAQ^T$ for $Q = \prod Q_i$ being the accumulation of the orthogonal transformations (many sources use the equivalent $D = Q^T AQ$). Different cases result in different types of D . For real symmetric A :

- $\Lambda = D$
- $V = Q^T$, and $V^{-1} = Q$
- Using **Wilkinson shifts** (discussed later) guarantees convergence (Trefethen & Bau 1997)
- The Hessenberg form is tridiagonal because of symmetry (this saves a constant factor in efficiency but isn't used here for code reuse; for the optimized conversion see Golub & Van Loan 2012)

The first step is conversion to Hessenberg form. Any square A can be converted with $O(n^3)$ operations, resulting in $H = QAQ^{-1}$ (for the Q so far). See Golub & Van Loan (2012) for a justification of the algorithm.

```

pair<Matrix<double>, Matrix<double>> toHessenbergForm(Matrix<double> A)
{
    int n = A.getRows();
    assert(A.getColumns() == n);
    Matrix<double> Q = Matrix<double>::identity(n);
    for(int c = 0; c < n - 2; ++c)
    {
        Vector<double> x(n - (c + 1));
        for(int j = c + 1; j < n; ++j) x[j - (c + 1)] = A(j, c);
        x = HouseholderReduction(x);
        HouseholderLeftMult(A, c + 1, c, x);
        HouseholderLeftMult(Q, c + 1, c, x);
        HouseholderRightMult(A, 0, c + 1, x);
    }
    return make_pair(A, Q);
}

```

The Wilkinson shift for the current iteration = the smallest eigenvalue of 2×2 bottom-right submatrix of A . Use a stable formula (Golub & Van Loan 2012):

```

template<typename MATRIX> double wilkinsonShift(MATRIX const& A, int n)
{//use stable formula
    double d = (A(n - 2, n - 2) - A(n - 1, n - 1))/2,
           temp = A(n - 1, n - 2) * A(n - 1, n - 2);
    return A(n - 1, n - 1) - temp/(d + sign(d) * sqrt(d * d + temp));
}

```

Iteration details:

- Use the same orthogonal transformation process as for Hessenberg QR. But store the transformations to compute RQ .
- Set any sub- = super-diagonal entry to 0 if $|A[c+1,c+1]| \leq \epsilon \times (|A[c,c]| + |A[c+1,c+1]|)$; ϵ is user-provided. This is an example of relative precision that is based on floating point limits—it's also justified by the Gershgorin circle theorem (discussed later in the chapter).
- Deflation leaves out any bottom-right submatrix of A where the nondiagonal entries = 0. In the code reduce n . But (optionally) operate on full Q to accumulate it correctly—when want eigenvectors.
- For safety check symmetry of the input to single precision.

```

template<typename MATRIX> Vector<double> QREigenTridiagonal(MATRIX A,
    Matrix<double>* Q = 0, int maxIter = 1000, double prec = highPrecEps)
{
    int n = A.getRows();
    assert(A.getColumns() == n);
    while(maxIter--)
    {
        int lastNZ = 0;
        for(int c = 0; c < n - 1; ++c) if(abs(A(c + 1, c)) <= prec *
            (abs(A(c, c)) + abs(A(c + 1, c + 1)))) {
            A(c + 1, c) = 0;
            A(c, c + 1) = 0;
        }
        else lastNZ = c + 1;
        //deflate
        n = lastNZ + 1;
        if(n < 2) break;
        //shift
        double shift = wilkinsonShift(A, n);
        A -= shift * MATRIX::identity(A.getRows());
        //QR = A
        Vector<Vector<double>> reductions(n - 1);
        for(int c = 0; c < n - 1; ++c)
        {
            reductions[c] = HouseholderReduction2(A(c, c), A(c + 1, c));
            HouseholderLeftMult(A, reductions[c], c, c, min(n - 1, c + 2));
            if(Q) HouseholderLeftMult(*Q, reductions[c], c);
        }
        //R = PQ
        for(int c = 0; c < n - 1; ++c)
            HouseholderRightMult(A, reductions[c], c, max(0, c - 1), c + 1);
        //unshift
        A += shift * MATRIX::identity(A.getRows());
    }
    Vector<double> eig(A.getRows());
    for(int d = 0; d < A.getRows(); ++d) eig[d] = A(d, d);
    return eig;
}
bool isESymmetric(Matrix<double> const& A, double eRelAbs = defaultPrecEps)
{
    int n = A.getRows();
    if(n != A.getColumns()) return false;
    for(int r = 0; r < n; ++r)
    {
        if(!isfinite(A(r, r))) return false;
        for(int c = r + 1; c < n; ++c) if(!isfinite(A(r, c)) ||
            !isEqual(A(r, c), A(c, r), eRelAbs)) return false;
    }
    return true;
}
pair<Vector<double>, Matrix<double>> QREigenSymmetric(Matrix<double> A,

```

```

int maxIter = 1000, double prec = highPrecEps)
{//rows of Q are eigenvectors
    assert(isESymmetric(A));
    pair<Matrix<double>, Matrix<double>> TQ = toHessenbergForm(A);
    Vector<double> eigve = QREigenTridiagonal(TQ.first, &TQ.second, maxIter,
                                                prec);
    return make_pair(eigve, TQ.second);
}

```

The convergence is quadratic in the number of iterations in the worst case but typically cubic (Trefethen & Bau 1997).

Conditioning of eigenvalues and eigenvectors is more complicated than for linear systems. See Datta (2010) and Watkins (2007) for details. The main conclusions about perturbation of A by ΔA :

- All eigenvalues of symmetric matrices are perfectly conditioned
- Many eigenvalues of nonsymmetric matrices may be ill-conditioned
- In both cases the eigenvectors may be very ill-conditioned depending on how close the eigenvalues are

Trefethen & Embree (2005) argue that ill-conditioned eigenpairs are scientifically useless and present an alternative.

22.18 Singular Value Decomposition

Given a $m \times n$ matrix A , without loss of generality with $m \geq n$, $A = U\Sigma V^T$, where (Wikipedia 2017c):

- U is orthogonal $m \times m$
- Σ is $m \times n$ such that the $n \times n$ top submatrix = nonnegative diagonal, and the $(m-n) \times n$ bottom submatrix = 0; the former are the **singular values** of A
- V is orthogonal $n \times n$

In some sources U and V are transposed, and go by what the algorithm returns. If $n > m$, compute SVD of A^T to get $A = U^T \Sigma V^T$. Think of SVD as breaking up A into a sum of outer products, i.e., $A = \sum \sigma_i u_i \otimes v_i$. E.g., this interpretation allows using SVD for compression where only use some number of the largest σ_i .

A simple algorithm, derived from computing eigenvalues of $A^T A$ using a real symmetric algorithm, leads to $\Sigma^2 = Q(A^T A)Q^T$. But forming $A^T A$ explicitly leads to numerical instability due to potential excessive roundoff accumulation. The **Golub-Kahan algorithm** maintains $A^T A$ implicitly:

1. Convert A to bidiagonal form, where $A = I + \text{upper sub-diagonal}$
2. Do shifted QR iteration implicitly on $A^T A$ until convergence

For convenience all functions and the result are put in a class:

```

struct SVD
{
    Vector<double> svds;
    Matrix<double> U, V;
    SVD(Matrix<double> A, int maxIter = 100, double prec = highPrecEps):
        svds(A.getColumns()), U(Matrix<double>::identity(A.getRows())),
        V(Matrix<double>::identity(A.getColumns()))
    {//swap U and V if needed
        bool needSwap = A.getColumns() > A.getRows();
        if(needSwap)
        {
            svds = Vector<double>(A.getRows());
            A = A.transpose();
            swap(U, V);
        }
        toBidiagonalForm(A); //convert to bidiagonal form
        int n = A.getColumns();
        //work with square submatrix if needed
        if(n < A.getRows()) A = submatrix<double>(A, 0, n - 1, 0, n - 1);
        SVDBidiagonal(A, maxIter, prec);
        if(needSwap)//swap back U and V if needed
        {
            Matrix<double> temp = V.transpose();
            V = U.transpose();
            U = temp;
        }
    }
}

```

```

    }
}
};
```

For explanation of the $O(n^3)$ bidiagonalization see Golub & Van Loan (2012).

```

void toBidiagonalForm(Matrix<double>& A)
{
    int m = A.getRows(), n = A.getColumns();
    assert(m >= n);
    for(int c = 0; c < n; ++c)
    {
        Vector<double> x(m - c);
        for(int j = c; j < m; ++j) x[j - c] = A(j, c);
        x = HouseholderReduction(x);
        HouseholderLeftMult(A, x, c, c);
        HouseholderLeftMult(U, x, c);
        if(c < n - 2)
        {
            x = Vector<double>(n - (c + 1));
            for(int j = c + 1; j < n; ++j) x[j - (c + 1)] = A(c, j);
            x = HouseholderReduction(x);
            HouseholderRightMult(A, x, c + 1, c);
            HouseholderRightMult(V, x, c + 1);
        }
    }
}
```

The symmetric QR iteration needs some changes:

- The shift is computed from $A^T A$, so need to form its 2×2 bottom-right submatrix, which doesn't lead to instability (for the formula see Golub & Van Loan 2012)
- As is, the algorithm can return negative singular values (beware—this isn't mentioned in Golub & Van Loan 2012), but can move them into V (StackOverflow 2017):
 - $\Sigma V = \Sigma I V$ where I = identity but with $I(i,i) = -1$
 - To find IV multiply $V(i, *)$ by -1 .
- Applying orthogonal transformations to A in certain way gives both U and V without further work (see Golub & Van Loan 2012 for the explanation)

```

void SVDBidiagonal(Matrix<double>& A, int maxIter = 100,
                    double prec = highPrecEps)
{
    assert(A.getColumns() == A.getRows());
    int n = A.getRows();
    while(maxIter--)
    {
        int lastNZ = 0;
        for(int c = 0; c < n - 1; ++c) if(isEEqual(A(c, c), 0) ||
            abs(A(c, c + 1)) <= prec *
                (abs(A(c, c)) + abs(A(c + 1, c + 1)))) A(c, c + 1) = 0;
        else lastNZ = c + 1;
        //deflate
        n = lastNZ + 1;
        if(n < 2) break;
        Matrix<double> T22 =
            submatrix<double>(A, n - 2, n - 1, n - 2, n - 1);
        double dm = T22(0, 0), dn = T22(1, 1), fm = T22(0, 1),
        fmm1 = (n >= 3 ? A(n - 3, n - 2) : 0);
        T22(0, 0) = dm * dm + fmm1 * fmm1;
        T22(0, 1) = dm * fm;
        T22(1, 0) = dm * fm;
        T22(1, 1) = dn * dn + fm * fm;
        double shift = wilkinsonShift(T22, 2),
        t00 = A(0, 0) * A(0, 0), t01 = A(0, 0) * A(0, 1),
        y = t00 - shift, z = t01;
        for(int c = 0; c < n - 1; ++c)
```

```

    }
    Vector<double> x = HouseholderReduction2(y, z);
    HouseholderRightMult(A, x, c);
    HouseholderRightMult(V, x, c);
    y = A(c, c);
    z = A(c + 1, c);
    x = HouseholderReduction2(y, z);
    HouseholderLeftMult(A, x, c);
    HouseholderLeftMult(U, x, c);
    if(c < n - 2)
    {
        y = A(c, c + 1);
        z = A(c, c + 2);
    }
}
n = A.getRows();
for(int d = 0; d < n; ++d)
//ensure non-negative svds by moving minuses into V
if(A(d, d) < 0)
{
    A(d, d) *= -1;
    for(int r = 0; r < n; ++r) V(r, d) *= -1;
}
svds[d] = A(d, d);
}
}

```

Because the SVD calculation is equivalent to the symmetric eigenvalue calculation of $A^T A$, just more stably implemented, it converges with the same rate. See Golub & Van Loan (2012), Trangenstein (2018b), and Stewart and Sun (1990) for conditioning/perturbation analysis of SVD. The last one in particular proves many theorems that are only stated in other sources.

SVD has many applications. In particular (Trefethen & Bau 1997):

- $\text{Rank}(A)$ = the number of singular values > 0 ; use some absolute ϵ
- $\|A\|_2$ = the maximum singular value
- $\|A^{-1}\|_2$ = the minimum singular value
- $\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2$

```

int rank(double precFor0 = highPrecEps) const
{
    int non0 = 0;
    for(int i = 0; i < svds.getSize(); ++i) non0 += (svds[i] >= precFor0);
    return non0;
}
double norm2() const {return valMax(svds.getArray(), svds.getSize());}
double norm2Inv() const {return 1/valMin(svds.getArray(), svds.getSize());}
double condition2() const {return norm2() * norm2Inv();}

```

22.19 Asymmetric Matrix Eigenvalues and Eigenvectors

The shifted QR iteration works for asymmetric matrices too, but with modifications and without guaranteed convergence. The modifications:

- The iteration converges to a quasi-triangular matrix, with 2×2 blocks on the diagonal that represent complex conjugate eigenvalues. These are solved analytically using the trace-determinant formula, as for Wilkinson shift.
- No single real shift can force a complex block to deflate, so do two successive real shifts efficiently using the **implicit shift algorithm**; see Golub & Van Loan (2012) and Ford (2014) for more details. Watkins (2011) explains it differently with a shortcut. A key trick is that a double shift is equivalent to two complex conjugate shifts a_1 and a_2 . In particular use the 2×2 bottom-right corner of H to compute these in the hope that they are good estimates for the eigenvalues which are the ideal shifts to deflate the last row/column. The calculation uses $\text{trace} = a_1 + a_2$ and $\text{determinant} = a_1 a_2$, which are both real.

- Q , if accumulated, no longer directly gives the eigenvectors. Can solve for them from Q (and in some cases more efficiently; Golub & Van Loan 2012), but it's much simpler to use **inverse iteration** on H and its Q with the computed eigenvalues (discussed later).

```
pair<Vector<complex<double>>, Matrix<double>> QREigen(Matrix<double> A,
    int maxIter = 1000, double prec = highPrecEps)
{
    pair<Matrix<double>, Matrix<double>> TQ = toHessenbergForm(A);
    Vector<complex<double>> eigs = QREigenHessenberg(TQ.first, maxIter, prec);
    return make_pair(eigs, findEigenvectors(TQ.first, TQ.second, eigs));
}
```

Some counterexamples are known where converge fails (Ford 2014), such as permutation matrices, the

simplest of which is $\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$ with eigenvalues $\{1, -0.5 \pm \sqrt{3}i/2\}$, all of magnitude 1 (roots of $x^3 - 1$). The

implementation actually converges on this example in 38 iterations because the minor roundoff errors in orthogonal transformations break the symmetry. Theorem (Watkins & Elsner 1991; see also Watkins 2007): If all eigenvalues are distinct, the QR iteration uses Rayleigh shifts, and it converges, then the rate is quadratic. This is a good moral support bound—Wilkinson shift is used, and the convergence isn't guaranteed. An experimental observation is that do 1–2 iterations per eigenvalue (Golub & Van Loan 2012). Stability is proven in Aurentz et al. (2018), who also describe recent improved algorithms for many special cases. Datta (2010) has excellent numerical examples (with minor algorithmic modifications) for this and many other algorithms presented here.

A heuristic fix strategy is **exceptional shifts**—i.e., after 10 iterations without deflation, use a special shift to break symmetries (Day 1996). Several deterministic shift strategies are considered in that paper, but none converges in all cases. So use random shifts. **Gershgorin circle theorem** (Ueberhuber 1997b): H has an eigenvalue with radius in a complex disk with center $A[n-1, n-1]$ and radius $|A[n-1, n-2]|$. So can randomly sample an eigenvalue $a+ib$ from this disk, and compute trace = $2a$ and determinant = a^2+b^2 . This cuts the number of iterations on the above matrix to 15–17 in my tests.

```
pair<double, double> traceDet2x2(Matrix<double> const& A, int d)
{
    double trace = (A(d, d) + A(d + 1, d + 1)), det =
        A(d, d) * A(d + 1, d + 1) - A(d, d + 1) * A(d + 1, d);
    return make_pair(trace, det);
}

Vector<complex<double>> QREigenHessenberg(Matrix<double> A,
    int maxIter = 10000, double prec = highPrecEps)
{
    assert(A.getColumns() == A.getRows());
    int n = A.getRows(), lastDeflateIter = maxIter;
    while(maxIter--)
    {
        //process converged entries
        for(int c = 0; c < n - 1; ++c) if(abs(A(c + 1, c)) <= prec *
            (abs(A(c, c)) + abs(A(c + 1, c + 1)))) A(c + 1, c) = 0;
        //deflate
        while(n >= 3)
        {
            if(A(n - 1, n - 2) == 0) --n;
            else if(A(n - 2, n - 3) == 0) n -= 2;
            else break;
            lastDeflateIter = maxIter;
        }
        if(n < 3) break;
        pair<double, double> td = traceDet2x2(A, n - 2);
        if(lastDeflateIter - maxIter > 10)
        {
            //unlikely exceptional shift
            pair<double, double> ab = GlobalRNG().pointInUnitCircle();
            double r = abs(A(n - 1, n - 2));
            ab.first = A(n - 1, n - 1) + ab.first * r;
            ab.second *= r;
            td.first = 2 * ab.first;
        }
    }
}
```

```

        td.second = ab.first * ab.first + ab.second * ab.second;
        lastDeflateIter = maxIter;
    }//calculate Householder vector
    Vector<double> xyz(3);
    xyz[0] = A(0, 0) * A(0, 0) + A(0, 1) * A(1, 0) - td.first * A(0, 0) +
        td.second;
    xyz[1] = A(1, 0) * (A(0, 0) + A(1, 1) - td.first);
    xyz[2] = A(1, 0) * A(2, 1);
    for(int c = 0; c < n - 1; ++c)
    {//process it
        xyz = HouseholderReduction(xyz);
        HouseholderLeftMult(A, xyz, c, max(0, c - 1), n - 1);
        HouseholderRightMult(A, xyz, c, 0, min(c + 3, n - 1));
        if(c + 2 == n) break;
        xyz[0] = A(c + 1, c);//update it for next iteration
        xyz[1] = A(c + 2, c);
        if(c + 3 < n) xyz[2] = A(c + 3, c);
        else xyz.removeLast();
    }
}//extract eigenvalues from blocks
n = A.getRows();
double NaN = numeric_limits<double>::quiet_NaN();
Vector<complex<double>> eigs(n, complex<double>(NaN, NaN));
for(int c = 0; c < n; ++c)
{
    if(c == 0 || A(c, c - 1) == 0)
    {//found block
        if(c + 1 >= n || A(c + 1, c) == 0)//1 x 1
            eigs[c] = complex<double>(A(c, c), 0);
        else if(c + 2 >= n || A(c + 2, c + 1) == 0)//2 x 2
        {
            pair<double, double> td = traceDet2x2(A, c);
            td.first /= 2;
            double temp = td.first * td.first - td.second,
                temp2 = sqrt(abs(temp));
            if(temp > 0)
            {//stable formula for e = trace05 +- temp2
                double eig1 = td.first + sign(td.first) * temp2;
                eigs[c] = complex<double>(eig1, 0);
                eigs[c + 1] = complex<double>(td.second/eig1, 0);
            }
            else
            {
                eigs[c] = complex<double>(td.first, td.second);
                eigs[c + 1] = complex<double>(td.first, -td.second);
            }
            ++c;
        }
        else c += 3;//didn't converge - not isolated
    }
}
return eigs;
}

```

Random shifts should converge because an exact eigenvalue is generated eventually, but this hasn't been proven (and the next question is the convergence rate).

Inverse iteration (Golub & Van Loan 2012) allows computing accurate eigenvectors from accurate approximations to eigenvalues. Given matrix A with an eigenvalue λ , the corresponding eigenvector v is defined by $(A - \lambda I)v = 0$. Inverse iteration assumes approximate eigenvalues $a \approx \lambda$. Then get $(A - \lambda I)v = (\lambda - a)v$, and $x = (A - \lambda I)^{-1}v$, where x is a constant $\times v$. Because eigenvectors are normalized, this gives a fixed point iteration (discussed in the "Numerical Methods—Working with Functions" chapter) where initial v is provided by the user, and then update $v = \frac{x}{\|x\|}$. Some implementation details (Ipsen 1997):

- A randomly chosen initial vector leads to a good solution in one iteration with high probability, and among the deterministic choices using the all-1 vector is safe, though both fail in rare cases.
- For a repeated eigenvalue, the attraction eigenvector depends on the initial vector. So use a random initial vector for initialization to allow finding distinct eigenvectors for duplicate eigenvalues.
- 5 is good limit on the number of iterations.

When A is in Hessenberg form $Q^T HQ$, an iteration becomes more efficient because instead work with H to compute $v(H, a)$ using Hessenberg QR to solve the equation in $O(n^2)$ time. Then $v(A, a) = Q^T v(H, a)$.

1. Pick arbitrary nonzero starting eigenvector v ; use random here

2. $v = ||v||$
3. Until $||Hv||_\infty \leq \epsilon ||H||_\infty$
4. Solve $(H - aI)x = v$

$$5. v = \frac{x}{||x||}$$

6. $Q^T v$ is the eigenvector of A corresponding to a

During (4) $H - \lambda I$ can be exactly singular— x can end up with ∞ (due to rounding error/0) or NaN (due to 0/0) components in case of λ with multiplicity > 1 (Ford 2014). Handle this by setting them to 0—though invalid for solving equations, this strategy works for inverse iteration.

The mathematical process is well-defined even if $a = \lambda$, just don't have an inverse concept but can still pick a solution of the equation as v . When that $a \neq \lambda$, the solution will be pulled slightly to other eigenvectors, and this pull $\rightarrow 0$ as $a \rightarrow \lambda$.

Inverse iteration can find complex eigenvectors from complex eigenvalues (Ford 2014), but for simplicity the implementation returns NaN eigenvectors.

```
Vector<double> findEigenvector(Matrix<double> H, double eig,
    int maxIter = 5, double prec = highPrecEps)
{
    // inverse iteration
    int n = H.getRows();
    double HInf = normInf(H);
    H -= eig * Matrix<double>::identity(n);
    QRDecomposition qr(H, true);
    Vector<double> x = GlobalRNG().randomUnitVector(n);
    while(maxIter--)
    {
        x = qr.solve(x);
        for(int i = 0; i < n; ++i) if(!isfinite(x[i])) x[i] = 0;
        x *= 1/norm(x);
        if(normInf(H * x) <= prec * HInf) break;
    }
    return x;
}

Matrix<double> findEigenvectors(Matrix<double> const& H,
    Matrix<double> const& Q, Vector<complex<double>> const& eigs)
{
    int n = eigs.getSize();
    Matrix<double> result(n, n);
    for(int i = 0; i < n; ++i)
    {
        if(eigs[i].imag() == 0)
        {
            Vector<double> eigve = findEigenvector(H, eigs[i].real());
            for(int r = 0; r < n; ++r) result(r, i) = eigve[r];
        }
        else for(int r = 0; r < n; ++r)
            result(r, i) = numeric_limits<double>::quiet_NaN();
    }
    return Q.transpose() * result;
}
```

Some convergence results (Ipsen 1997):

- For normal (symmetric) matrices, the residual $r = (A - aI)v$ converges to the best attainable value, and the iterates belong to the eigenspace associated with the closest eigenvalue.
- For nonnormal diagonalizable matrices, the situation is very different—the residuals usually

increase after the first iterations, and iterating until convergence or to reduce bad effect of the initial starting vector makes sense only if the eigenvector matrix V is well-conditioned. Iterates approach the subspace of the closest eigenvector even with residual increase, but have no particular convergences guarantees.

- With finite precision the ill-conditioning due to the near-singularity of $(A - \alpha I)$ doesn't cause problems—any ill-conditioning in the calculation will be due to the problem and not the algorithm (Trefethen & Bau 1997).

But don't expect inverse iteration to succeed in all cases—unlike a symmetric matrix, a nonsymmetric one can be **defective**, i.e., not have an eigen-decomposition (Trefethen & Bau 1997). Can also have a case where A has a repeated eigenvalue but distinct eigenvectors, such as for $A = \alpha I$. As mentioned, this case is handled by the random initial vector with high probability as long as the generation doesn't lead to a repetition (or near-repetition).

22.20 Sparse Matrices

A is sparse when it has enough zeros to take advantage of usually $O(n)$ nonzero entries, sometimes denoted by the **number of nonzeros = nnz**. Several useful ways to represent a sparse matrix—as a collection of:

- Entries—a list of entries—simple for storage but doesn't support efficient operations.
- Entries—a hash table—allows efficient representation when $nnz = O(1)$ and supports some operations such as random access, sparse matrix \times dense vector (using random order iteration), but not in-order iteration by columns or rows. Use it as an intermediate representation for sparse matrix \times sparse matrix.
- Column vectors—an array of sorted sparse vectors—the choice here. It allows efficient in-order iteration by column, random read access using binary search, and element insertions aren't too expensive.
- Column vectors—compressed column storage—same as (3) but everything is represented compactly with three arrays like a compressed graph (see the "Graph Algorithms" chapter). Compared to (3), this saves memory by a constant factor but is more clumsy to program, and element insertions are much more expensive. But if A has $o(n)$ elements, it's much more memory-efficient because (3) needs $O(n)$ storage for columns.
- Column vectors—list of unsorted sparse vectors—similar to (3) but tries to be more efficient for some operations (Duff et al. 2017).
- Row vectors—any equivalent of (3–5)—essentially the same properties, but sparse matrix \times dense vector operation used in iterative methods benefits from in-order row iteration. Other methods are still efficient for this operation, and probably due to the traditional consideration of vectors as row vectors this is less popular.

A further consideration for (3–6) is how to store the vectors. A simple option is a vector of pairs; another is a vector of values and a vector of indices. The second is more memory-efficient, but the first is simpler to work with and more cache-efficient.

A typical pattern for a sparse matrix algorithm is to convert the main representation into a more convenient one, do the work, and convert back. One of the simplest conversions is to take a transpose of (3–6), which along with the original representation gives in-order or unsorted iteration by both row and column. Can do it implicitly for some operations to save memory.

Another pattern is to allow 0 elements that are explicitly set, though can do basic checks against this. Specialized algorithms could use absolute ϵ to convert to 0, but this is rarely done.

Most of the implementation of (3) is straightforward, though sparse vector addition and dot product need some care to make sure that go through both sequentially (and are good interview questions). Insertion is worst-case $O(n)$ but amortized $O(nnz/n)$.

```
template<typename ITEM = double>
class SparseMatrix: public ArithmeticType<SparseMatrix<ITEM>>
{
    int rows;
    typedef pair<int, ITEM> Item;
    typedef Vector<Item> SparseVector;
    Vector<SparseVector> itemColumns;
    int findPosition(int r, int c) const
    { // return index of the given element or -1 if it doesn't exist
        assert(0 <= r && r < rows && 0 <= c && c < getColumns());
        SparseVector const& column = itemColumns[c];
```

```

    return binarySearch(column.getArray(), 0, column.getSize() - 1,
        Item(r, 0), PairFirstComparator<int, ITEM>());
}
public:
SparseMatrix(int theRows, int theColumns): rows(theRows),
    itemColumns(theColumns){}
int getRows() const return rows;
int getColumns() const return itemColumns.getSize();
ITEM operator()(int r, int c) const
//absent entries are 0
    int position = findPosition(r, c);
    return position == -1 ? 0 : itemColumns[c][position].second;
}
void set(int r, int c, ITEM const& item)
{//first try to find and update
    SparseVector& column = itemColumns[c];
    int position = findPosition(r, c);
    if(position != -1) column[position].second = item;
    else if(item != 0)//if can't need to do vector insertion by shifting
    //down the rest of the column
        Item temp(r, item);
        column.append(temp);
        position = column.getSize() - 1;
        for(;position > 0 && column[position - 1].first > r; --position)
            column[position] = column[position - 1];
        column[position] = temp;
    }
}
static SparseVector addSparseVectors(SparseVector const& a,
    SparseVector const& b)
{//take elements from both, adding where both exist
    SparseVector result;
    for(int aj = 0, bj = 0; aj < a.getSize() || bj < b.getSize();)
    {
        bool considerBoth = aj < a.getSize() && bj < b.getSize();
        int j = aj < a.getSize() ? a[aj].first : b[bj].first;
        ITEM item = aj < a.getSize() ? a[aj++].second : b[bj++].second;
        if(considerBoth)//made init with a, now consider b
            if(j == b[bj].first) item += b[bj++].second;
            else if(j > b[bj].first)
            {
                j = b[bj].first;
                --aj;//undo aj increment
                item = b[bj++].second;
            }
        if(item != 0) result.append(Item(j, item));//just in case
    }
    return result;
}
SparseMatrix& operator+=(SparseMatrix const& rhs)
{//add column-by-column
    assert(rows == rhs.rows && getColumns() == rhs.getColumns());
    SparseMatrix result(rows, getColumns());
    for(int c = 0; c < getColumns(); ++c) result.itemColumns[c] =
        addSparseVectors(itemColumns[c], rhs.itemColumns[c]);
    return *this = result;
}
SparseMatrix& operator*=(ITEM a)
{
    for(int c = 0; c < getColumns(); ++c)
        for(int j = 0; j < itemColumns[c].getSize(); ++j)
            itemColumns[c][j].second *= a;
    return *this;
}

```

```

}

friend SparseMatrix operator*(SparseMatrix const& A, ITEM a)
{
    SparseMatrix result(A);
    return result *= a;
}

SparseMatrix operator-() const
{
    SparseMatrix result(*this);
    return result *= -1;
}

SparseMatrix& operator-=(SparseMatrix const& rhs)
{
    return *this += -rhs;
}

static SparseMatrix identity(int n)
{
    SparseMatrix result(n, n);
    for(int c = 0; c < n; ++c) result.itemColumns[c].append(Item(c, 1));
    return result;
}

static ITEM dotSparseVectors(SparseVector const& a,
                               SparseVector const& b)
//add to sum when both present
{
    ITEM result = 0;
    for(int aj = 0, bj = 0; aj < a.getSize() && bj < b.getSize();)
        if(a[aj].first == b[bj].first)
            result += a[aj++].second * b[bj++].second;
        else if(a[aj].first < b[bj].first) ++aj;
        else ++bj;
    return result;
}

static Vector<ITEM> sparseToDense(SparseVector const& sv, int n)
//need n because don't know sparse tail
{
    assert(sv.getSize() == 0 || sv[sv.getSize() - 1].first < n);
    Vector<ITEM> v(n);
    for(int i = 0; i < sv.getSize(); ++i) v[sv[i].first] = sv[i].second;
    return v;
}

static SparseVector denseToSparse(Vector<ITEM> const& v)
{
    SparseVector sv;
    for(int i = 0; i < v.getSize(); ++i)
        if(v[i] != 0) sv.append(Item(i, v[i]));
    return sv;
}

friend SparseVector operator*(SparseVector const& v, SparseMatrix const& A)
{
    assert(v.getSize() == 0 || v.lastItem().first < A.getRows());
    SparseVector result;
    for(int c = 0; c < A.getColumns(); ++c)
        //add one row at a time
        ITEM rc = dotSparseVectors(v, A.itemColumns[c]);
        if(rc != 0) result.append(Item(c, rc));
    }
    return result;
}

friend SparseVector operator*(SparseMatrix const& A, SparseVector const& v)
{
    return v * A.transpose();
}

friend Vector<ITEM> operator*(Vector<ITEM> const& v, SparseMatrix const& A)
{
    return sparseToDense(denseToSparse(v) * A, A.rows);
}

friend Vector<ITEM> operator*(SparseMatrix const& b, Vector<ITEM> const& v)
{
    return sparseToDense(b * denseToSparse(v), b.getColumns());
}

friend double normInf(SparseMatrix const& A)
//first calculate transpose for better iteration

```

```

SparseMatrix AT = A.transpose();
double maxRowSum = 0;
for(int r = 0; r < A.getRows(); ++r)
{
    double rSum = 0;
    for(int cj = 0; cj < AT.itemColumns[r].getSize(); ++cj)
        rSum += abs(AT.itemColumns[r][cj].second);
    maxRowSum = max(maxRowSum, rSum);
}
return maxRowSum;
};
}

```

To calculate the transpose append the first column of the original matrix as the first row to every transpose column that has an entry. Because the columns are processed in order, the transpose entries are also created in order.

```

SparseMatrix transpose() const
{
    SparseMatrix result(getColumns(), rows);
    for(int c = 0; c < getColumns(); ++c)
        for(int j = 0; j < itemColumns[c].getSize(); ++j)
            result.itemColumns[itemColumns[c][j].first].append(
                Item(c, itemColumns[c][j].second));
    return result;
}

```

Some matrix-vector products use the transpose explicitly, but can do this implicitly without needing to store the transpose—but omitted for simplicity.

```

friend SparseVector operator*(SparseVector const& v, SparseMatrix const& A)
{
    assert(v.getSize() == 0 || v.lastItem().first < A.getRows());
    SparseVector result;
    for(int c = 0; c < A.getColumns(); ++c)
        //add one row at a time
        ITEM rc = dotSparseVectors(v, A.itemColumns[c]);
        if(rc != 0) result.append(Item(c, rc));
    }
    return result;
}
friend SparseVector operator*(SparseMatrix const& A, SparseVector const& v)
    {return v * A.transpose();}
friend Vector<ITEM> operator*(Vector<ITEM> const& v, SparseMatrix const& A)
    {return sparseToDense(denseToSparse(v) * A, A.rows);}
friend Vector<ITEM> operator*(SparseMatrix const& b, Vector<ITEM> const& v)
    {return sparseToDense(b * denseToSparse(v), b.getColumns());}

```

For matrix multiplication, the usual row-by-column dot product needs $O(n^2)$ dot products, most of which will be 0. But one of the arrangements of the dense multiplication triple loop corresponds to a sum of outer products: $AB = \sum_k A[* , k] \otimes B[k , *]$. Each outer product is usually sparse because $O(1)$ vector \times $O(1)$ vector = $O(1)$ matrix. So with enough sparsity need $O(n)$ time. The column vector representation is inefficient for $O(1)$ outer product matrices, so use the hash table representation:

1. Transpose B to get efficient row access
2. Accumulate all outer product matrices into a hash table
3. Iterate over the hash table to convert it into the main representation
4. Sort each column

```

SparseMatrix& operator*=(SparseMatrix const& rhs)
//O(n^2) * space factor(1 to n)
assert(getColumns() == rhs.rows);
SparseMatrix result(rows, rhs.getColumns()), bT = rhs.transpose();
//compute sum of outer product sums
typedef typename Key2DBuilder<>::WORD_TYPE W;
LinearProbingHashTable<W, ITEM> outerSums;
Key2DBuilder<> kb(max(result.rows, result.getColumns())),

```

```

        result.rows >= result.getColumns());
    for(int k = 0; k < rhs.rows; ++k)
        for(int aj = 0; aj < itemColumns[k].getSize(); ++aj)
            for(int btj = 0; btj < bT.itemColumns[k].getSize(); ++btj)
            {
                int r = itemColumns[k][aj].first,
                    c = bT.itemColumns[k][btj].first;
                W key = kb.to1D(r, c);
                ITEM* rcSum = outerSums.find(key), rcValue =
                    itemColumns[k][aj].second *
                    bT.itemColumns[k][btj].second;
                if(rcSum) *rcSum = rcValue;
                else outerSums.insert(key, rcValue);
            }
        //convert outer sum hash table into final data structure
        for(typename LinearProbingHashTable<W, ITEM>::Iterator iter =
            outerSums.begin(); iter != outerSums.end(); ++iter)
        { //n must be the larger of r, c to make sense!
            pair<unsigned int, unsigned int> rc = kb.to2D(iter->key);
            result.itemColumns[rc.second].append(Item(rc.first, iter->value));
        } //sort each column to fix order
        for(int c = 0; c < result.getColumns(); ++c) quickSort(
            result.itemColumns[c].getArray(), 0,
            result.itemColumns[c].getSize() - 1,
            PairFirstComparator<int, ITEM>());
    return *this = result;
}

```

22.21 Iterative Methods for Sparse Matrices

Iterative methods work as is with dense matrices too but make sense mostly for sparse matrices. The most useful ones run in a **Krylov subspace**, i.e., for matrix A the set of vectors $Ax, A(Ax)$, etc. Think of common dense matrix algorithms such as multiplication as doing n iterations with $O(n^2)$ work each. Iterative methods in the best case need:

- $< n$ iterations for enough accuracy
- $O(n)$ of work per iteration if A is very sparse

So might even have $O(n)$ total work.

The iterative method of choice for solving equations with symmetric and positive definite (SPSD) matrices is the **conjugate gradient algorithm** (see Ford 2014 for a description). The nonlinear optimization method by the same name is derived from it (see the “Numerical Optimization” chapter). In exact arithmetic it will terminate after n iterations due to a certain orthogonality of iterates, but with finite precision much orthogonality can be lost. So use n as the limit on the number of iterations, but deem convergence when reach a specific 2-norm precision.

$Ax=b$ implies that $A^T Ax=A^T b$. Because $A^T A$ is SPSD, conjugate gradient applies as is to both general equations and least-squares problems. This modification is called **CGNR** (“NR” for “normal equations”). The rate of convergence for the SPSD case is $O\left(1 + \frac{2}{\sqrt{\kappa(A)}}\right)$, where κ is the condition number (Trefethen & Bau 1997). For non-SPSD, don't have the $\sqrt{\cdot}$ because effectively square A .

A **preconditioner** is any matrix P that is “similar” to A such that ideally $P^{-1}A \approx I$. An agnostic choice is $P = I$, but a good choice can greatly reduce the number of iterations when solving $P^{-1}Ax=P^{-1}b$ instead because of better conditioning of $P^{-1}A$ (Ford 2014). A cheap-to-invert P is **Jacobi matrix** = $\text{diag}(A)$. It's almost always better than not using preconditioning; at least it improves scale (Ueberhuber 1997b). For CGNR compute P based on $A^T A$, and because P is approximate to begin with, the explicit product formation doesn't create problems.

```

template<typename MATRIX> MATRIX findJacobiPreconditioner(MATRIX A,
    bool isSPSD = true)
{
    if(!isSPSD) A = A.transpose() * A;
    int n = A.getRows();
    MATRIX diagInv(n, n);

```

```

    for(int r = 0; r < n; ++r) diagInv.set(r, r, 1/A(r, r));
    return diagInv;
}

template<typename MATRIX, typename PRECONDITIONER> pair<Vector<double>, double>
conjugateGradientSolve(MATRIX const& A, Vector<double> b,
PRECONDITIONER const& pInv, bool isSPSD = true,
Vector<double> x = Vector<double>(), double eFactor = highPrecEps)
{
    MATRIX AT(1, 1); //dummy for SPSD case
    int n = A.getRows(), maxIter = n;
    if(x.getSize() == 0) x = Vector<double>(n, 0);
    assert(x.getSize() == n && b.getSize() == A.getColumns());
    if(!isSPSD)
    {
        AT = A.transpose();
        b = AT * b;
    }
    Vector<double> temp = A * x, r = b - (isSPSD ? temp : AT * temp),
    z = pInv * r, p = z;
    while(maxIter-- > 0 && norm(r) > eFactor * (1 + norm(b)))
    {
        Vector<double> ap = A * p;
        if(!isSPSD) ap = AT * ap;
        double rz = dotProduct(r, z), a = rz/dotProduct(p, ap);
        if(!isfinite(a)) break;
        x += p * a;
        r -= ap * a;
        z = pInv * r;
        p = z + p * (dotProduct(r, z)/rz);
    }
    return make_pair(x, norm(r));
}

```

Domain-specific preconditioners usually do better than generic ones.

22.22 Iterative Methods for Eigenvalues

Finding eigenvalues of a sparse matrix is more complicated (Ford 2014; Trefethen & Bau 1997). It's hopeless to ask for all eigenvectors because this would take too much space, so need to settle for less. Typically get some eigenvectors and eigenvalues, and iterative methods return those that are of interest in some applications but not others.

For the symmetric case **Lanczos iteration** (Wikipedia 2017e) allows getting all eigenpairs or some approximate ones. Applied to a symmetric matrix, it produces a tridiagonal matrix like Hessenberg form conversion, but:

- It computes one column of matrix V at a time at the cost of one matrix-vector product
- Can stop after $m < n$ iterations to get a matrix whose eigenvalues approximate some of those of A

The implementation returns a 5-diagonal matrix so that the result can be passed as is to symmetric QR iteration which needs the extra diagonals as workspace. The algorithm produces V and T , where ideally $V^T TV \approx A$ if m is large enough or $= n$.

```

template<typename MATRIX> BandMatrix<5> LanczosEigReduce(MATRIX const& A,
    Vector<Vector<double>*>* vs = 0, int m = -1,
    Vector<double> v = Vector<double>())
{
    int n = A.getRows();
    if(m == -1) m = n;
    if(v.getSize() == 0) v = GlobalRNG().randomUnitVector(n);
    assert(A.getColumns() == n && v.getSize() == n);
    BandMatrix<5> result(m);
    double b = 1;
    Vector<double> prevV;
    for(int i = 0; i < m; ++i)
    {

```

```

if(vs) vs->append(v);
Vector<double> w = A * v;
double a = dotProduct(w, v);
result(i, i) = a;
w -= v * a;
if(i > 0)
{
    w -= prevV * b;
    result(i, i - 1) = result(i - 1, i) = b;
}
b = norm(w);
if(b < numeric_limits<double>::epsilon())
//can't continue so return what have
    BandMatrix<5> result2(i + 1);
    for(int j = 0; j <= i; ++j)
    {
        result2(j, j) = result(j, j);
        if(j > 0)
            result2(j, j - 1) = result2(j - 1, j) = result(j, j - 1);
    }
    return result2;
}
prevV = v;
v = w * (1/b);
}
return result;
}

```

Apply the tridiagonal QR iteration to the result to get an eigendecomposition:

```

template<typename MATRIX> pair<Vector<double>, Matrix<double> >
LanczosEigenSymmetric(MATRIX const& A, int m = -1, int maxIter = 1000,
double prec = highPrecEps)
{//rows of Q are eigenvectors
int n = A.getRows();
if(m == -1) m = n;
Vector<Vector<double>> vs;
BandMatrix<5> T = LanczosEigReduce(A, &vs, m);
Matrix<double> QT(vs.getSize(), n);
for(int c = 0; c < vs.getSize(); ++c)
    for(int r = 0; r < n; ++r) QT(c, r) = vs[c][r];
Vector<double> eigve = QREigenTridiagonal(T, &QT, maxIter, prec);
return make_pair(eigve, QT);
}

```

If A has $O(n)$ entries, the runtime is $O(n)$. Unfortunately Lanczos is unstable in that later v_i get far from orthogonal to the earlier ones. As presented, the implementation is useful for getting maybe 10 most extreme approximate eigenpairs of a very large matrix. They can be very approximate, but this is still useful, e.g., for estimating the condition number. See the comments for a better method.

For asymmetric matrices have **Arnoldi iteration**, but it computes the Hessenberg matrix one column at a time, so must stop before memory runs out, and get only some eigenvalues. This isn't discussed further, but the method in the comments applies also.

22.23 Introduction to Interval Arithmetic

An interesting addition to floating point arithmetic is **interval arithmetic**. Some main ideas (Tucker 2011):

- The true result of a single operation is between the rounded down and rounded up values
- This allows simple bounds for the standard arithmetic operations
- C++ has a thread-safe API to control rounding mode, so can implement an interval type
- Can express many numerical algorithms using this type

There is also an interval arithmetic standard IEEE 1788-2015. Hopefully it will eventually remove the main use obstacle—the lack of standard library support for elementary functions.

Interval arithmetic only addresses the floating point estimation error—still have approximation and optimization errors. For nonnumerical calculations, such as the CCW test in computational geometry (see

the “Computational Geometry” chapter), it’s very useful because can compute in interval arithmetic and switch to exact rational arithmetic only when the interval bounds don’t guarantee correctness.

Tucker (2011) presents a basic implementation of an interval arithmetic class. A cleaner and more portable implementation is given below. Only the multiplication operation is complicated, so see Tucker (2011) for a proof of correctness of all cases. Also see Moore et al. (2009) and references therein.

```
//beware below, though part of C99 standard, not supported by some compilers
//such as clang
#pragma STDC FENV_ACCESS ON
template<typename ITEM = double>
class IntervalNumber: public ArithmeticType<IntervalNumber<ITEM>>
{//all operations must restore nearest rounding for other calculations
    ITEM left, right;
public:
    IntervalNumber(ITEM rounded)
    {//epsilon addition exact; change this to ulp?
        std::fesetround(FE_DOWNWARD);
        left = rounded * (1 + numeric_limits<ITEM>::epsilon());
        std::fesetround(FE_UPWARD);
        right = rounded * (1 - numeric_limits<ITEM>::epsilon());
        std::fesetround(FE_TONEAREST);
    }
    IntervalNumber(long long numerator, long long denominator)
    {
        std::fesetround(FE_DOWNWARD);
        left = numerator/denominator;
        std::fesetround(FE_UPWARD);
        right = numerator/denominator;
        std::fesetround(FE_TONEAREST);
    }
    bool isfinite() const{return std::isfinite(left) && std::isfinite(right);}
    bool isnan() const{return std::isnan(left) && std::isnan(right);}
    IntervalNumber& operator+=(IntervalNumber const& rhs)
    {
        std::fesetround(FE_DOWNWARD);
        left += rhs.left;
        std::fesetround(FE_UPWARD);
        right += rhs.right;
        std::fesetround(FE_TONEAREST);
        return *this;
    }
    IntervalNumber operator-() const//minus exact in floating representation
    {
        return IntervalNumber(-right, -left);
    }
    IntervalNumber& operator-=(IntervalNumber const& rhs)
    {
        return *this += -rhs;
    }
    IntervalNumber& operator*=(IntervalNumber const& rhs)
    {
        std::fesetround(FE_DOWNWARD);
        ITEM temp1 = min(left * rhs.left, left * rhs.right);
        ITEM temp2 = min(right * rhs.left, right * rhs.right);
        ITEM newLeft = min(temp1, temp2);
        std::fesetround(FE_UPWARD);
        temp1 = max(left * rhs.left, left * rhs.right);
        temp2 = max(right * rhs.left, right * rhs.right);
        ITEM newRight = max(temp1, temp2);
        std::fesetround(FE_TONEAREST);
        left = newLeft;
        right = newRight;
        return *this;
    }
    IntervalNumber& operator*=(long long a)
    {//exact constant multiplication
        std::fesetround(FE_DOWNWARD);
    }
}
```

```

    left *= a;
    std::fesetround(FE_UPWARD);
    right *= a;
    std::fesetround(FE_TONEAREST);
    return *this;
}

bool contains(ITEM a) const {return left <= a && a <= right;}
IntervalNumber& operator/=(IntervalNumber rhs)
{
    if(rhs.contains(ITEM(0)))
    {
        rhs.left = -numeric_limits<ITEM>::infinity();
        rhs.right = numeric_limits<ITEM>::infinity();
    }
    else
    {
        std::fesetround(FE_DOWNWARD);
        rhs.left = ITEM(1)/rhs.right;
        std::fesetround(FE_UPWARD);
        rhs.right = ITEM(1)/rhs.left;
    }
    return (*this) *= rhs;
}
};

```

The next step is to implement the algorithms presented here and in the next chapter, but this is far from trivial, and many special algorithms have been developed for interval arithmetic. Some issues:

- Different equivalent math formulas give intervals of different width. Finding the best syntax tree is practically impossible in general, so usually implement a natural sequence of operations unless know problem-specific research.
- A function of an interval is generally a multi-interval. E.g., think about equation solving with $f(x)=\sqrt{x}$. A quick solution is to take a convex hull interval of the multi-intervals, but this is often wasteful, particularly for multidimensional diagonal sets that are small relative to the containing box.

So interval arithmetic is useful for some numerical analysis applications but not in general.

22.24 Implementation Notes

The epsilon-minded comparisons and constants for default precision are original. They occur in many algorithms.

This chapter presents the most useful matrix algorithms, even though they are only a small selection of everything available. Golub & Van Loan (2012) present almost everything, and the main difficulty was cutting down many others. I was able to do that by basing everything on LUP and QR decompositions and not presenting specialized algorithms for structured matrices with better constant factors.

22.25 Comments

IEEE-754-2019 also standardizes floating point arithmetic in base 10. This is needed in financial calculations that must give exact results—e.g., $1/3$ isn't its binary approximation. Some laws, such as for tax calculations, specify decimal calculation and rounding rules, which base-10 arithmetic allows to implement exactly. Overton (2001) is the best introduction floating point arithmetic, and Muller et al. (2018) and the standard give more details. Perhaps the most important practical change is that formal proofs are becoming more feasible—e.g., see Boldo & Melquiond (2017) for details on the **gappa** tool.

Condition numbers are usually defined by a vague “bound on change magnitude” and thereafter assumed to be derivatives of the true solution F with respect to the inputs x . My modulus-of-continuity-based definition tries to be more general.

An interesting idea is to have **machine-independent** precision checks (Gander et al. 2014) in a sense that eventually there are no machine numbers between the compared quantity (absolute or relative) and the candidate answer. Many algorithms can be expressed without any mention of ϵ or using tricky comparison tests. Pros:

- Squeeze out maximum precision

- The implementation tends to be a bit simpler in terms of convergence criteria

Cons:

- The resulting precision can be illusory—rounding errors are still there. E.g., consider finding the root of $f(x) = 1 - e^{-x}$. The answer is 0, but it's silly to get it down to some number \approx the smallest normal because $f(x) \approx 0$ for $x \approx \epsilon$ due to the subtraction.
- The process can be very inefficient. For relative comparisons CPU registers use more precision bits, and the process can try to get them all right and round down to store the answer in memory. For comparisons against 0 this is much worse because the calculations will continue until reaching the smallest normal and possibly to subnormals.

Matrix multiplication seems straightforward, but is heavily optimized in popular libraries. Golub & Van Loan (2012) discuss a number of issues. On a parallel computer expressing the matrix product as a sum of outer products tends to win because the latter use **BLAS-2 operations**, which can be optimized in assembly and specialized to the hardware. Another direction is using fast matrix algorithms, such as the one due to Strassen, which beat the regular algorithm for $n > 100$ by a small margin. But have no component-wise stability, and the norm-based bounds are worse by unknown constant factors (Higham 2002).

Solutions of equations using LUP or QR are further improved by **iterative refinement**:

- Theoretically get component-wise stability (Higham 2002)
- Practically get reduced sensitivity to the initial scale

See Golub & Van Loan (2012) for details. The cost is $O(n^2)$, so it's feasible even for repeated equation solving with the same factorization. But need to store the original matrix, which may make the process not worth it.

A **Givens rotation** has the same effect as a Householder reduction on x of size 2, but it's a bit more efficient to compute (Golub & Van Loan 2012). For several elements Householder is faster. Because computing a rotation is faster than applying it, using a Givens rotation is an unnecessary optimization.

For calculating eigenvalues of a real symmetric matrix, several competitive methods have been established (Ford 2014). Compared to QR iteration, some are faster and some more accurate, but not by much. Only in case of very tiny eigenvalues that are close to the used ϵ , it may pay off to use somewhat slower **Jacobi algorithm**, which can be implemented to be machine-independent (for more details see Ford 2014, Golub & Van Loan 2012, and Gander et al. 2014).

Using random values drawn from Gershgorin circles is a new idea—it doesn't seem to have been discovered yet despite the obvious convergence guarantee.

Direct methods for sparse matrices try to preserve sparsity of intermediate results. E.g., for Gaussian elimination, if the first column and the row of the chosen pivot are dense, sparsity will be lost. Many different heuristic methods have been proposed to deal with this (Duff et al. 2017), in particular:

- Local methods such as the **Markowitz strategy**—pick a pivot using which minimizes the immediate fill-in, but to preserve stability it must not be much smaller than the usual pivot
- Global band-reduction methods such as the reverse **Cuthill-McKee algorithm**—interpret the matrix as a graph, and permute the vertices in a certain greedy way

For something like Cholesky factorization, which works on SPSD matrices and doesn't need pivoting, reverse Cuthill-McKee reordering preserves SPSD by only permuting the variables on the diagonal. Then do Cholesky factorization as usual, though operate on a sparse matrix, and use special algorithms (Duff et al. 2017; Davis 2006). It's unclear how the result competes with conjugate gradient with generic preconditioning. In the worst case the former can increase memory use by much, and the latter can take many iterations.

Iterative methods for solving equations for various cases are discussed in much more detail in Golub & Van Loan (2012) and references therein. The issue is that CGNR can converge too slowly and other algorithms try to fix this, but also have to give up something:

- **GMRES**—the main algorithm for general matrices, but it can run out of memory before convergence unless use a restarting strategy.
- Many other methods try to keep low memory use of CGNR and converge faster but run into stability issues. It's unclear which one of them is the best.

Automatic preconditioners also get more complicated. E.g., for CG one option is **incomplete Cholesky factorization**—i.e., calculate Cholesky factorization of a sparse matrix but don't store entries where the matrix doesn't have them. This also needs code to solve the resulting sparse matrix-vector equations, but performs well.

Can fix orthogonality problems in Lanczos with **complete reorthogonalization** (Ford 2014) at the cost of factor $O(m)$ more work. This is feasible only for small m —where rounding errors are unlikely to accumulate enough to cause problems. Commonly used software is ARPACK, and it's based on Arnoldi/Lanczos with complete reorthogonalization and a technique called **implicit restarting**. Assume that want k smallest

eigenpairs. Then:

- 1. Do Lanczos until have $2k$ or so.**
- 2. Until converged:**
- 3. Sort by eigenvalue**
- 4. Remove the largest k**
- 5. Restart the iteration from the smallest k**

For details, particularly about how to restart, see Bai et al. (2000). Watkins (2007) has some theory. It's unclear how many extra ones need to compute and under what conditions the process is guaranteed to converge, but it usually does and is the best available solution. Still, finding several smallest eigenvalues of a large sparse matrix reliably and efficiently is an open problem. Theoretically also have issues, particularly with eigenvectors—see Kuczyński & Woźniakowski (1992).

Numerical linear algebra is a very extensive topic, and all the books cited make good further reading. In addition to those already mentioned, see also Hogben (2013) for an extensive collection of surveys for theoretical and numerical topics in linear algebra. Björck (2014) is also of interest. For iterative solving of linear systems, Saad (2003) is the most comprehensive. For iterative calculation of eigenvalues and analysis of the algorithms, Saad (2011) is useful.

22.26 Projects

- Create a string constructor for interval arithmetic. Also allow construction from a wide interval, not a single-number one.
- For tridiagonal systems, does diagonal dominance need to be bounded away by some ϵ for solution stability? Research.
- As implemented, Jacobi preconditioner for CG isn't robust to zeros on the diagonal. Implement some strategies for handling this. A simple approach is to use weighted average with the average of all diagonal elements, but that can still be 0, so maybe average in 1 in the latter.
- Implement implicitly restarted Lanczos.
- For an interval number allow construction for a integer fraction.

22.27 References

- Acton, F. S. (1996). *Real Computing Made Real: Preventing Errors in Scientific and Engineering Calculations*. Dover.
- Aurentz, J. L., Mach, T., Robol, L., Vandebril, R., & Watkins, D. S. (2018). *Core-Chasing Algorithms for the Eigenvalue Problem*. SIAM.
- Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., & van der Vorst, H. (Eds.). (2000). *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM.
- Beebe, N. H. F. (2017). *The Mathematical-Function Computation Handbook*. Springer.
- Björck, Å. (2014). *Numerical Methods in Matrix Computations*. Springer.
- Boldo, S., & Melquiond, G. (2017). *Computer Arithmetic and Formal Proofs: Verifying Floating-point Algorithms with the Coq System*. Elsevier.
- Bornemann, F. (2016). The SIAM 100-digit challenge: a decade later. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 118(2), 87-123.
- Bornemann, F., Laurie, D., Wagon, S., & Waldvogel, J. (2004). *The SIAM 100-digit Challenge: a Study in High-accuracy Numerical Computing*. SIAM.
- Cody, W. J. (1974). The construction of numerical subroutine libraries. *SIAM Review*, 16(1), 36-46.
- Corless, R. M., & Fillion, N. (2013). *A Graduate Introduction to Numerical Methods*. Springer.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Datta, B. N. (2010). *Numerical Linear Algebra and Applications*. SIAM.
- Davis, T. A. (2006). *Direct Methods for Sparse Linear Systems*. SIAM.
- Day, D. (1996). How the QR algorithm fails to converge and how to fix it.
- Deuflhard, P., & Hohmann, A. (2003). *Numerical Analysis in Modern Scientific Computing: An Introduction*. Springer.
- Duff, I. S., Erisman, A. M., & Reid, J. K. (2017). *Direct Methods for Sparse Matrices*. Oxford University Press.
- Fausett, L. V. (2003). *Numerical Methods: Algorithms and Applications*. Pearson.
- Ford, W. (2014). *Numerical Linear Algebra with Applications: Using MATLAB*. Academic Press.
- Free Software Foundation (2017). *GNU Scientific Library 2.4*.
<https://www.gnu.org/software/gsl/doc/html/index.html>.

- Gander, W., Gander, M. J., & Kwok, F. (2014). *Scientific Computing - An Introduction using Maple and MATLAB*. Springer.
- Golub, G. H., & Van Loan, C. F. (2012). *Matrix Computations*. JHU Press.
- Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM.
- Hogben, L. (2013). *Handbook of Linear Algebra*. CRC.
- Ipsen, I. C. (1997). Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2), 254-291.
- Kuczyński, J., & Woźniakowski, H. (1992). Estimating the largest eigenvalue by the power and Lanczos algorithms with a random start. *SIAM journal on matrix analysis and applications*, 13(4), 1094-1122.
- Miller, W. (1984). *The Engineering of Numerical Software*. Prentice-Hall.
- Muller, J. M. (2016). *Elementary Functions*. Birkhäuser.
- Muller, J. M., Brunie, N., de Dinechin, F., Jeannerod, C. P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., and Torres, S. (2018). *Handbook of Floating-Point Arithmetic*. Springer.
- Overton, M. L. (2001). *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM.
- Moore, R. E., Kearfott, R. B., & Cloud, M. J. (2009). *Introduction to Interval Analysis*. SIAM.
- Nash, J. C. (2014). *Nonlinear Parameter Optimization Using R Tools*. Wiley.
- Powell, M. J. D. (1981). *Approximation Theory and Methods*. Cambridge University Press.
- Rice, J. R. (1992). *Numerical Methods, Software, and Analysis*. Academic Press.
- Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. SIAM.
- Saad, Y. (2011). *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. SIAM.
- Shampine, L. F., & Reichelt, M. W. (1997). The MATLAB ode suite. *SIAM Journal on Scientific Computing*, 18(1), 1-22.
- StackOverflow (2017). Convention on non-negative singular values?.
<https://math.stackexchange.com/questions/170746/convention-on-non-negative-singular-values>. Accessed March 5, 2017.
- Stewart, G. W., & Sun, J. G. (1990). *Matrix Perturbation Theory*. Academic Press.
- Süli, E., & Mayers, D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge University Press.
- Trangenstein, J. A. (2018b). *Scientific Computing: Vol. II - Eigenvalues and Optimization*. Springer.
- Traub, J. F., & Werschulz, A. G. (1998). *Complexity and Information*. Cambridge University Press.
- Trefethen, L. N., & Bau III, D. (1997). *Numerical Linear Algebra*. SIAM.
- Trefethen, L. N., & Embree, M. (2005). *Spectra and Pseudospectra: The Behavior of Nonnormal Matrices and Operators*. Princeton University Press.
- Tucker, W. (2011). *Validated Numerics: A Short Introduction to Rigorous Computations*. Princeton University Press.
- Ueberhuber, C. W. (1997a). *Numerical Computation 1: Methods, Software, and Analysis*. Springer.
- Ueberhuber, C. W. (1997b). *Numerical Computation 2: Methods, Software, and Analysis*. Springer.
- Watkins, D. S. (2007). *The Matrix Eigenvalue Problem: GR and Krylov subspace methods*. SIAM.
- Watkins, D. S. (2011). Francis's algorithm. *The American Mathematical Monthly*, 118(5), 387-403.
- Watkins, D. S., & Elsner, L. (1991). Convergence of algorithms of decomposition type for the eigenvalue problem. *Linear Algebra and Its Applications*, 143, 19-47.
- Wikipedia (2017b). Eigendecomposition of a matrix.
https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix. Accessed March 5, 2017.
- Wikipedia (2017c). Singular value decomposition.
https://en.wikipedia.org/wiki/Singular_value_decomposition. Accessed March 5, 2017.
- Wikipedia (2017d). Taylor's theorem. https://en.wikipedia.org/wiki/Taylor's_theorem. Accessed May 6, 2017.
- Wikipedia (2017e). Lanczos algorithm. https://en.wikipedia.org/wiki/Lanczos_algorithm. Accessed August 13, 2017.

23 Numerical Algorithms—Working with Functions

“For most $f(x)$ convergence theory is only useful for moral support” – John Boyd

“Ask not whether SOLVE can fail; rather ask, “When will it succeed?”” – William Kahan

23.1 Introduction

This chapter continues where the previous one left off. The focus is on numerical methods for functions. In particular, discuss algorithms for interpolation, differentiation, integration, solving equation systems, and ODEs.

23.2 Fast Fourier Transform

Given n complex numbers x_i , the **discrete Fourier transform (DFT)** computes $y_k = \sum x_i e^{-2\pi j k i / n}$ (i here is the complex number) for $0 \leq k < n$ (Wikipedia 2017a). This calculation is an important part of many algorithms, including those that don't deal with complex numbers. Computing all y_k using the above formulas takes $O(n^2)$ time.

The **fast Fourier transform (FFT)** does it in $O(n \lg(n))$. The simplest case is when n is a power of two. The numbers $e^{-2\pi j k i / n}$ are **roots of unity**, which are equally spaced on the unit circle in the complex plane. Some higher-frequency points overlap with some lower-frequency ones. In particular, some of the products $x_i e^{-2\pi j k i / n}$ appear in various y_k several times. So FFT computes common products only once. See Cormen et al. (2009) for further discussion of the algorithm; contrary to the standard DFT definition, their implementation uses positive-power roots of unity (some other sources such as Press et al. 2007 also use it).

```
complex<double> unityRootHelper(int j, int n)
    {return exp((j * PI() / n) * complex<double>(0, 1));}
Vector<complex<double>> FFTPower2(Vector<complex<double>> const&x)
{
    int n = x.getSize(), b = lgFloor(n);
    assert(isPowerOfTwo(n));
    typedef complex<double> C;
    Vector<C> result(n);
    for(unsigned int i = 0; i < n; ++i) result[reverseBits(i, b)] = x[i];
    for(int s = 1; s <= b; ++s)
    {
        int m = twoPower(s);
        C wm = unityRootHelper(-2, m);
        for(int k = 0; k < n; k += m)
        {
            C w(1, 0);
            for(int j = 0; j < m/2; ++j, w *= wm)
            {
                C t = w * result[k + j + m/2], u = result[k + j];
                result[k + j] = u + t;
                result[k + j + m/2] = u - t;
            }
        }
    }
    return result;
}
```

The relative forward error of power-of-two FFT is $O(\epsilon_{\text{machine}} \lg(n))$ in the 2-norm, assuming computation of roots of unity to near $\epsilon_{\text{machine}}$ (Higham 2002).

The **inverse DFT** is defined by $y_k = \frac{1}{n} \sum x_i e^{2\pi j k i / n}$ (Wikipedia 2017a). Can compute it by reduction to regular DFT, computed by FFT. Aside from the factor $1/n$, the “ $-$ ” in the exponential works with the same roots of unity, just in different order for $j \neq 0$. The order for $j > 0$ is actually reversed, and can either reverse the

output or the input of the FFT of x (Lyons 2015). The former allows passing input by constant reference and is used here.

```
Vector<complex<double>> IFFTHelper(Vector<complex<double>> fftx)
{
    int n = fftx.getSize();
    fftx.reverse(1, n - 1);
    return fftx * (1.0/n);
}

Vector<complex<double>> inverseFFTPower2(
    Vector<complex<double>> const& x)
{
    assert(isPowerOfTwo(x.getSize()));
    return IFFTHelper(FFTPower2(x));
}
```

When $n \neq$ a power of two, reduce to a suitable power-of-two case. For some applications can use the FFT created by appending 0's to the data to make it a power of two. **Bluestein's algorithm** (Wikipedia 2017b) works in the general case. Given two discrete sequences a and b of size n such that $b_{-l} = b_{n-l}$, their **convolution** = y , where $y_k = \sum a_j b_{k-j}$. Also, convolution(a, b) = $\text{FFT}^{-1}(\text{pairwise product}(\text{FFT}(a), \text{FFT}(b)))$, and convolutions aren't affected by 0-padding.

```
Vector<complex<double>> convolutionPower2(Vector<complex<double>> const& a,
                                             Vector<complex<double>> const& b)
{
    int n = a.getSize();
    assert(n == b.getSize() && isPowerOfTwo(n));
    Vector<complex<double>> fa = FFTPower2(a), fb = FFTPower2(b);
    for(int i = 0; i < n; ++i) fa[i] *= fb[i];
    return inverseFFTPower2(fa);
}
```

Reduce FFT to a convolution of a power of two. The DFT exponential is a function of $jk = (j^2 + k^2 - (k - j)^2)/2$. Let $r(j, n) = e^{\pi j/n}$. Then $y_k = r(-k^2, n) \sum a_i b_{k-i} r(-j^2, n)$, and $b_{k-i} = r(-(k - i)^2, n)$. Here a and b are functions, but need them to be arrays such that:

- They are padded to length m that is a power of two
- $b_{-l} = b_{n-l}$ for $l = k - j$

So choose $m = \text{next-power-of-two}(2n - 1)$. Then $A_j = \begin{cases} a_j, & \text{if } j < n \\ 0, & \text{otherwise} \end{cases}$, and $B_l = B_{m-l} = \begin{cases} b_l, & \text{if } l < n \\ 0, & \text{otherwise} \end{cases}$.

```
Vector<complex<double>> FFTGeneral(Vector<complex<double>> const& x)
{//Bluestein's algorithm
    int n = x.getSize(), m = nextPowerOfTwo(2 * n - 1);
    if(isPowerOfTwo(n)) return FFTPower2(x);
    Vector<complex<double>> a(m), b(m); //C-padded by default constructor
    for(int j = 0; j < n; ++j)
    {
        a[j] = x[j] * unityRootHelper(-j * j, n);
        b[j] = unityRootHelper(j * j, n); //could precompute b its fft
        if(j > 0) b[m - j] = b[j];
    }
    Vector<complex<double>> ab = convolutionPower2(a, b);
    while(ab.getSize() > n) ab.removeLast();
    for(int k = 0; k < n; ++k) ab[k] *= unityRootHelper(-k * k, n);
    return ab;
}

Vector<complex<double>> IFFTGeneral(Vector<complex<double>> const& x)
{
    return IFFTHelper(FFTGeneral(x));
}
```

A good introduction to some of the mathematics is Smith (2007).

In many cases the input is real, and this allows some savings. In particular, can pack two real sequences x and y in a single complex sequence z , and from its FFT derive their FFTs in $O(n)$ time. The algorithm is (Kopriva 2009):

1. Form z using $z_i = x_i + iy_i$
2. Let $z_f = \text{FFT}(z)$

3. Form x_f using $x_f[j] = (z_f[j] + \text{conjugate}(z_f[(n - j) \% n]))/2$
4. Form y_f using $y_f[j] = (z_f[j] - \text{conjugate}(z_f[(n - j) \% n]))i/2$
5. Return x_f and y_f

```
pair<Vector<complex<double>>, Vector<complex<double>> > FFTReal2Seq(
    Vector<double> const& x, Vector<double> const& y)
{
    int n = x.getSize();
    assert(n == y.getSize());
    typedef complex<double> C;
    typedef Vector<C> VC;
    VC z(n);
    for(int i = 0; i < n; ++i) z[i] = C(x[i], y[i]);
    VC zf = FFTGeneral(z);
    pair<VC, VC> result;
    for(int i = 0; i < n; ++i)
    {
        C temp = conj(zf[(n - i) \% n]);
        result.first.append(0.5 * (zf[i] + temp));
        result.second.append(0.5 * (zf[i] - temp) * C(0, -1));
    }
    return result;
}
```

A further enhancement is using this to calculate FFT of a single real sequence of even n . The reduction is a single step of recursive power-of-two FFT (Cormen et al. 2009; again, beware they use positive exponents):

```
Vector<complex<double>> FFTRealEven(Vector<double> const& x)
{
    int n = x.getSize(), n2 = n/2;
    assert(n \% 2 == 0);
    typedef complex<double> C;
    typedef Vector<C> VC;
    Vector<double> xOdd(n2), xEven(n2);
    for(int i = 0; i < n; ++i) (i \% 2 ? xOdd[i/2] : xEven[i/2]) = x[i];
    pair<VC, VC> xSplitF = FFTReal2Seq(xEven, xOdd);
    VC xF(n);
    C wn = unityRootHelper(-2, n), wi(1, 0);
    for(int i = 0; i < n2; ++i, wi *= wn)
    {
        xF[i] = xSplitF.first[i] + wi * xSplitF.second[i];
        xF[n2 + i] = xSplitF.first[i] - wi * xSplitF.second[i];
    }
    return xF;
}
```

Many other useful transforms reduce to the calculation of FFT. A particular case is the **discrete cosine transform (DCT) of type I**, given by $DCTI(x)_k = \sum_{0 \leq j \leq n} x_j \cos(jk\pi/n)$; the double prime notation means the first and the last terms are halved (Dahlquist & Björck 2008). E.g., it's used for calculating Chebyshev polynomial interpolation (discussed later in the chapter). Compute it by reduction to FFT (Dahlquist & Björck 2008):

1. Let $n = \text{size of the input} - 1$
2. Make new array y of size $2n$
3. Set $y_j = x_j$ for $0 \leq j \leq n$ and $y_{2n-j} = x_j$ for $1 \leq j \leq n$
4. Return the real part of first $n + 1$ elements of $\text{FFT}(y)/2$

For efficiency use real FFT of an even-length sequence.

```
Vector<double> DCTI(Vector<double> const& x)
{
    int n = x.getSize() - 1;
    assert(n > 0);
    Vector<double> y(2 * n), result(n + 1);
    for(int i = 0; i <= n; ++i) y[i] = x[i];
    for(int i = 1; i < n; ++i) y[2 * n - i] = x[i];
```

```

Vector<complex<double>> yf = FFTRealEven(y);
for(int i = 0; i <= n; ++i) result[i] = yf[i].real()/2;
return result;
}

```

For type I DCT, the inverse = $DCT \times 2/n$ (Wikipedia 2017e).

```

Vector<double> IDCTI(Vector<double> const& x)
    {return DCTI(x) * (2.0/(x.getSize() - 1));}

```

23.3 Interpolation—General Ideas

Want to approximate f by another function that:

- Has useful analytical properties
- Gives a good enough approximation

Usually consider only a finite interval $[a, b]$. A computationally efficient approximation method is interpolation, where use values at the evaluated points of f (often called **nodes**) to evaluate others. From information complexity, given any set of nodes, can define another function that matches f at the node points, but elsewhere behaves arbitrarily. To prevent problems it's enough for f to be Lipschitz, which forces all possible f to behave similarly.

To reduce approximation error, want an interpolant to:

- Converge when f is Lipschitz
- Converge with the order that matches the number of continuous derivatives of f
- Not have large oscillations between the nodes even if when f isn't Lipschitz

The nodes can be selected by the interpolation method or predetermined. Polynomials are commonly used as interpolants:

- They have many known analytical properties
- The basic operations are fast
- The approximation quality is well-understood

Weierstrass theorem: If f is continuous on $[a, b]$, can represent it arbitrarily well by a polynomial. So wonder if can interpolate using polynomials of increasing the degree by doubling it until convergence. This doesn't work for only continuous f —an algorithm that works with black-box f needs to have some fixed node selection strategy, i.e., independent of f , which isn't assumed by the theorem. **Erdos-Vertesi theorem:** \forall fixed node set strategy \exists continuous f such that $\limsup_{n \rightarrow \infty} (\text{interpolant}(f, \text{node set}(n))) = \infty$ almost everywhere (Erdos & Vertesi 1980; “almost” because exact at nodes, and “sup” because some n better than others for various x).

So Weierstrass theorem doesn't cheat the Lipschitz requirement, which prevents such worst cases. The overall error $\rightarrow 0$, even though the Lebesgue constant grows slowly with n (discussed later in the chapter). Intuitively, can have problems when for nearby nodes the f values are arbitrarily different, and Lipschitz condition prevents large slopes. If f has enough continuous derivatives, can determine some error bounds.

Lagrange interpolation error theorem (LIE) (Ascher & Greif 2011; Suli & Mayers 2003): Given a set of $k + 1$ points x_i in sorted order, f with $k + 1$ continuous derivatives $\in [x_0, x_k]$, and p = the polynomial of degree k obtained from matching f at $\{x_i\}$, $|p(x) - f(x)| \leq LIE(x, \{x_i\})$, $\|f^{(k+1)}\|_\infty = \frac{\|f^{(k+1)}\|_\infty \prod(x - x_i)}{(k+1)!}$. Assume the

norms and results hold on $[x_0, x_k]$, though various theorem don't mention this explicitly—e.g., “ $\|f^{(k+1)}\|_\infty$ ” is on $[x_0, x_k]$, and may be larger outside, which doesn't matter.

Let $\frac{\|f^{(k+1)}\|_\infty}{(k+1)!} \leq$ some constant M . Then $\|p - f\|_\infty \leq M(x_k - x_0)^{k+1}$, so for $(x_k - x_0) \rightarrow 0$, as is often the case

for numerical methods that iteratively interpolate on shrinking intervals, have convergence of order $k + 1$. In practice can't find a bound on a distant derivative, so the formula only shows convergence and its approximate rate.

Accuracy bounds depend on the assumptions. For LIE the big assumption is that f has enough continuous derivatives. E.g., the derivative (more accurately subgradient) of the absolute value function is a step function, and its own derivative is a delta function, so using even using a quadratic polynomial on any domain containing 0 seems problematic.

For more general bounds need some tools from approximation theory:

- Theorem (Powell 1981): If approximant X is a linear projection, $\|f - X(f)\| \leq (1 + \|X\|)E(f)$, where $E(f)$ is the error of the best interpolant from the same normed space as X . In particular, for polynomials, $\|f - p\|_\infty \leq (1 + \Lambda(x_i))E_n(f)$, where Λ is the **Lebesgue constant** of the nodes. For

arbitrarily nodes, $\Lambda(\{x_i\})$ is a function of the interpolation (Trefethen 2019). The inequality derives from the logic that assumes a function g that is close to both f and p and deduce bounds based on its properties (Prenter 1975 makes this connection clear in his proof). Commonly $g =$ the best approximation to f by a polynomial of degree n .

- Let $h_{\min} = \min_i(x_{i+1} - x_i)$ and $h_{\max} = \max_i(x_{i+1} - x_i)$. Then $\Lambda(\{x_i\}) \leq \left(\frac{2h_{\max}}{h_{\min}}\right)^n$ (Prenter 1975). For bounds it's an unknown constant that doesn't depend on scaling from another interval to $[-1, 1]$.
- Jackson theorem** (Powell 1981): Let $E_n(f) = \|f - g\|_\infty$ on $[-1, 1]$. If $f^{(k)}$ is continuous, $E_n(f) \leq \frac{(n-k)!(\pi/2)^k}{n!} \|f^{(k)}\|_\infty$.
- Jackson theorem for Lipschitz f (Powell 1981): $E_n(f) \leq \frac{L\pi}{2(n+1)}$.
- For $(x_k - x_0) \rightarrow 0$, scale f to $[-1, 1]$, which scales $f^{(k)}$ by $\left(\frac{x_k - x_0}{2}\right)^k$. Intuitively, when f -values remain the same but expand to $[-1, 1]$, f changes faster. For consistency with forward difference derivative $f' = \frac{f(x+h) - f(x)}{h}$, h increases while f -values don't. The Lipschitz constant L also scales by $\frac{x_k - x_0}{2}$.

Putting these together, for f with k derivatives for interpolation with n points on $[-1, 1]$, $|p(x) - f(x)| \leq 1 + \left(1 + \left(\frac{2h_{\max}}{h_{\min}}\right)^n\right) \frac{(n-k)!(\pi/2)^k}{n!} \|f^{(k)}\|_\infty$. Prenter (1975) gives slightly different constants due to using different bounds for $E_n(f)$. Because the scaling is the only variable factor, for a fixed n and $(x_k - x_0) \rightarrow 0$ have:

- Convergence for Lipschitz f
- Order- k convergence for f with k continuous derivatives, with hard-to-determine constants $\left(\frac{2h_{\max}}{h_{\min}}\right)^n$ is minimized for equal-spaced nodes, which seem to be a natural candidate for interpolation, but this is deceptive:
 - For equal-spaced nodes asymptotically $\Lambda(\{x_i\}) \approx \frac{2^{n+1}}{\text{enlog}(n)}$, which is among the worst possible configurations (Trefethen 2019).
 - For **Chebyshev extrema nodes** (discussed later in the chapter) in $[-1, 1]$, $\Lambda(\{x_i\}) \leq \frac{2}{\pi} \log(n+1) + 1$, which is asymptotically optimal because \forall fixed node set $\Lambda(\{x_i\}) \geq O(\lg(n))$ (Trefethen 2019).

Λ serves as a condition number of interpolation—Theorem (Dahlquist & Bjorck 2008): $\|p\|_\infty \leq \Lambda(\{x_i\}) \|f\|_\infty$.

Because of fast-growing Λ , interpolation at equal-spaced nodes beyond some small degree (usually < 5) is a bad idea. In fact, have divergence for beyond small n because the LIE can increase with degree if $\|f^{(k-1)}\|_\infty$ grows very quickly with k . A common example where this happens is **Runge's function** $f(x) = \frac{1}{1+25x^2}$, with $[a, b] = [-5, 5]$. So don't have guaranteed converge for Lipschitz f as $n \rightarrow \infty$.

For Chebyshev nodes it follows from the above that for Lipschitz f as $n \rightarrow \infty$ have convergence with error $O(\lg(n)/n)$ in any fixed interval. It may appear problematic that $\lg(n) \rightarrow \infty$ as $n \rightarrow \infty$, but this makes no difference because if the best approximation error $\rightarrow 0$, some multiple of it is still ≈ 0 .

23.4 Polynomial Interpolation from Existing Data

Equal-spaced nodes polynomials of small degree are useful. Unfortunately the familiar monomial basis representation has a high condition number (Trefethen 2019), so try to avoid calculating the monomial coefficients explicitly. In particular, can create and evaluate an interpolation polynomial based on degree + 1 points implicitly.

The most stable way to do this is using the **barycentric interpolation formula** (Trefethen 2019):
 $p(x) = \frac{\sum y_i w_i / (x - x_i)}{\sum w_i / (x - x_i)}$, where $w_i = \frac{1}{\prod_{j \neq i} (x_j - x_i)}$. $x = x_i$ leads to $p(x) = y_i$, and checking for division by 0 is enough for stability for $x \approx x_i$.

```

class BarycentricInterpolation
{
    Vector<pair<double, double>> xy;
    Vector<double> w;
public:
    BarycentricInterpolation(Vector<pair<double, double>> const & thexy)
    {//O(n^2)
        for (int i = 0; i < thexy.getSize(); ++i)
            addPoint(thexy[i].first, thexy[i].second);
    }
    void addPoint (double x, double y)
    {
        double wProduct = 1;
        for (int i = 0; i < xy.getSize(); ++i)
        {
            wProduct *= (x - xy[i].first);
            w[i] /= xy[i].first - x;//must update previous wi
            assert(isInfinite(w[i]));//signal repeated point or overflow
        }
        w.append(1/wProduct);
        assert(isInfinite(w.lastItem()));
        xy.append(make_pair(x, y));
    }
    double operator() (double x) const
    {
        assert(isInfinite(x));
        double numSum = 0, denomSum = 0;
        for (int i = 0; i < xy.getSize(); ++i)
        {
            double factorI = w[i]/(x - xy[i].first);
            if (!isInfinite(factorI)) return xy[i].second;//inf if x is in xy
            numSum += factorI * xy[i].second;
            denomSum += factorI;
        }
        return numSum/denomSum;
    }
};
```

For polynomials of a large degree can have overflow or underflow, but this general formula is only useful for small-degree polynomials where this isn't a problem.

Can remove points by reversing the adding:

```

void removePoint (int i)
{
    int n = xy.getSize();
    assert(i >= 0 && i < n);
    for (int j = 0; j < n; ++j)
        if (j != i) w[j] *= (xy[j].first - xy[i].first);
    for (int j = i + 1; j < n; ++j)
        {//make generic vector remove func as nonmember?
            xy[j - 1] = xy[j];
            xy.removeLast();
            w[j - 1] = w[j];
            w.removeLast();
        }
}
```

Can compute derivatives directly (Kopriva 2009):

- For the general case, $p'(x) = \frac{\sum t_i w_i / (x - x_i)}{\sum w_i / (x - x_i)}$ for $t_i = \frac{p(x) - y_i}{x - x_i}$.

- At a node i , $p'(x_i) = \frac{\sum_{j \neq i} (y_j - y_i) w_j / (x_i - x_j)}{w_i}$. This follows from the previous formula for $x \rightarrow x_i$
because $p'(x_i) \rightarrow t_i = \frac{\sum (y_j - y_i) w_j / (x - x_j)}{(x - x_i) / \sum w_j / (x - x_j)} \rightarrow \frac{\sum_{j \neq i} (y_j - y_i) w_j / (x_i - x_j)}{w_i}$.

From my experiments the general formula is unstable, reaching only about single-precision accuracy, probably due to forward-difference-like cancellation. A different approach is suggested by Berrut & Trefethen (2004), but they don't present an explicit formula.

The ability to evaluate $p'(x)$ is enough for stable and efficient evaluation—use these values instead of $p(x)$ in the barycentric formula to interpolate—the nodes and the weights stay the same. Though can evaluate these points one-by-one, it's better to use a differentiation matrix D whose entries are defined by direct correspondence to the formulas (Kopriva 2009):

- For $j \neq i$, $D[i, j] = \frac{w_j / (x_i - x_j)}{w_i}$
- $D[i, i] = -\sum_{j \neq i} D[i, j]$

```
Matrix<double> diffMatrix() const
{
    // duplicate points impossible here due to weight filtering
    int n = xy.getSize();
    assert(n > 1); // need at least two points for this to make sense
    Matrix<double> diff(n, n);
    for(int r = 0; r < n; ++r) for(int c = 0; c < n; ++c) if(r != c)
    {
        diff(r, c) = w[c]/w[r]/(xy[r].first - xy[c].first);
        diff(r, r) -= diff(r, c);
    }
    return diff;
}
Vector<double> getY() const
{
    int n = xy.getSize();
    Vector<double> y(n);
    for(int i = 0; i < n; ++i) y[i] = xy[i].second;
    return y;
}
BarycentricInterpolation overwriteY(Vector<double> const& y) const
{
    int n = y.getSize();
    assert(xy.getSize() == n);
    BarycentricInterpolation result = *this;
    for(int i = 0; i < n; ++i) result.xy[i].second = y[i];
    return result;
}
BarycentricInterpolation deriver() const
{
    return overwriteY(diffMatrix() * getY());
}
```

Derivative estimates from interpolation polynomials also have error estimates. Theorem (Prenter 1975):

$$\|f^{(j)} - p^{(j)}\|_\infty \leq \|f^{(n+1)}\|_\infty \frac{n! h_{\max}^{n+1-j}}{(j-1)!(n+1-j)!}.$$

This theorem, but with the error expression in terms of $(b-a)^{n-j}$, is rare among numerical analysis texts, and seem to have been first established in Kranzer (1963); Isaacson & Keller (1994) follows the same argument.

When f doesn't have $n + 1$ continuous derivatives, published bounds ought to exist but I haven't been able to find any. But interpolation followed by derivation is also a linear operator, so can extend the approximation-based theorem for function values on $[-1, 1]$ using $f^{(j)}$ as the function to get

$$\|p^{(j)} - f^{(j)}\| \leq (1 + \|p^{(j)}\|) \frac{(n-k)!(\pi/2)^k}{n!} \|f^{(k-j)}\|_\infty.$$

No longer have the Lebesgue constant bound, but can apply **Markov brothers' inequality**: for $[-1, 1]$ $\|p'\| \leq n^2 \|p\|_\infty$, so all derivatives are bounded (Wikipedia 2017e). So $\|p^{(j)}\| \leq (n^2)^j \left(\frac{2h_{\max}}{h_{\min}}\right)^n$. For the below theorem only need that this be a constant with respect to the change of interval from $[-1, 1]$ to $[a, b]$, which it is because the ratio of h values is scale-free.

Proved the following theorem: If p interpolates f at $\{x_i\}$, and f has k continuous derivatives $\in [a = x_0, b = x_n]$ for $n \geq k$, then for $i < k$ $p^{(i)}$ approximates $f^{(i)}$ with convergence of order $k - i$ for $(b - a) \rightarrow 0$. So intuitively, order = min(formula degree, number of derivatives) – derivative number.

For already given or random points the Lebesgue constant depends on the barycentric weights. Though low-degree polynomials can be useful in this case, they are rarely used due to potential arbitrary swings between the nodes. E.g., already for a quadratic in x_0, x_1, x_2 , where $x_0 \approx x_1$ but $f(x_0)$ is far from $f(x_1)$, the quadratic extremum will swing far to accommodate if f isn't Lipschitz.

23.5 Chebyshev Polynomials

For polynomial interpolation don't need equally spaced points. If minimize the LIE part due to node placement, get the roots of **Chebyshev polynomials**. These polynomials are defined on $[-1, 1]$ recursively by

$$T_{k+1}(x) = \begin{cases} 1, & \text{if } k=0 \\ x, & \text{if } k=1 \\ 2xT_k(x) - T_{k-1}(x) \end{cases} \quad (\text{Trefethen 2019}) \quad \text{and can be scaled to other ranges. Also}$$

$T_k(x) = \cos(k \cos^{-1}(x))$. Some properties:

- The roots of $T_n(x)$ are $\cos((j - \frac{1}{2})\pi/n)$ for $1 \leq j \leq k$
- The extrema of $T_n(x)$ are $\cos(j\pi/n)$ for $0 \leq j \leq k$

Interpolation at either leads to approximation $f(x) = \sum_{0 \leq k \leq n} c_k T_k(x)$, which is very accurate with many enough terms:

- The Lebesgue constant bound means that Chebyshev interpolation won't be too bad even if f isn't Lipschitz. Even for $n = 10^6$ the maximum error is only an order of magnitude worse than that of the optimal polynomial of the same degree. So Chebyshev interpolation works reasonably even if have discontinuities. Indeed, for $f =$ the unit step function, have **Gibbs phenomenon** where the error near the discontinuity \geq a constant (Trefethen 2019), but the interpolant is great everywhere else. So using Chebyshev polynomials of bounded degree doesn't lead to arbitrarily oscillations.
- It may seem that allocating more points to near the endpoints is wasteful, which it may be for other interpolation methods, but for polynomials asymptotically it's the best option.
- Looking at **bounded variation** (see the "Computational Statistics" chapter) gives better but less intuitive bounds than the formulas based on bounded derivatives (Trefethen 2019).

Interpolation is preferred to direct fitting of Chebyshev polynomials to f using their orthogonality properties (see Muller 2016 for some examples), which is much less efficient and accurate.

Many sources use the roots (e.g., Press et al. 2007; presumably because with endpoint singularities this will still work, though how well is questionable). But the extrema allow simpler application of FFT for efficient computation of c_k and to reuse previously evaluated points with doubling. Both have the same approximation properties (Trefethen 2019).

In particular (Mason & Handcomb 2002), $c_k = \frac{2}{n} \sum_{0 \leq j \leq n} f(y_j) T_k(y_j)$, where $y_j = \cos(j\pi/k)$. Then $f(x) = \sum c_k T_k(x)$, which is implemented by halving the first and last c_k when they are computed. Recalling the definition of DCT of type I, with some algebra $c = \frac{2}{n} \text{DCT}(f(y))$ (Mason & Handcomb 2002). So the computation is $O(n \lg(n))$, though it helps with constant factors when n is a power of two.

Based on the FFT error estimates, after interpolation can remove all higher-term coefficients $< \lg(n) \epsilon_{\text{machine}} \|c\|_\infty$, with the ∞ -norm being more appropriate for individual coefficients and giving a constant safety factor.

Incidentally, deem convergence of interpolation when the two highest-order coefficients go below $\epsilon_{\text{machine}}$ (Boyd 2014), which here is actually the FFT rounding error per above (the estimate code is a member function of Chebyshev functor). Look at two and not one because, e.g., for odd functions every other coefficient is 0. Though this criteria isn't full-proof, it's robust in practice because worst-case inputs such as those with natural sequences of 0 coefficients don't occur.

```
class ChebFunction
{
    Vector<double> ci;
    bool converged;
    double ciAbsE() const { return numeric_limits<double>::epsilon() *
        lgCeiling(ci.getSize()) * (1 + normInf(ci)); }
    void trim()
```

```

//remove if ci too small
    int oldN = ci.getSize();
    double cutoff = ciAbsE();
    while(ci.getSize() > 1 && abs(ci.lastItem()) < cutoff)
        ci.removeLast();
    if(oldN - ci.getSize() > 1) converged = true;
}
//f values must be sorted values evals at cos(jPI/n) for 0 <= j <= n
void ChebFunctionHelper(Vector<double> const& fValues)
//DCTI does most work
{
    ci = DCTI(fValues) * (2.0/(fValues.getSize() - 1));
    ci[0] /= 2; //half first and last
    ci.lastItem() /= 2;
    converged = false;
    trim();
}
public:
    ChebFunction(Vector<double> const& fValues, int n)
    {ChebFunctionHelper(fValues);}
    template<typename FUNCTION> ChebFunction(FUNCTION const& f, int n)
    {
        assert(n > 0);
        Vector<double> fValues(n + 1);
        for(int i = 0; i <= n; ++i)
            fValues[i] = f(cos(PI() * i/n));
        ChebFunctionHelper(fValues);
    }
    bool hasConverged() const{return converged;}
};

```

Can even base a heuristic error estimate on this convergence test:

```
double error() const{return converged ? ciAbsE() : abs(ci.lastItem());}
```

It seems to be reasonably accurate if the interpolation converged, or tried with not-too-small n . Beware that if perform operations like formal integration or differentiation to get a new Chebyshev functor, this estimate loses meaning.

Can evaluate a Chebyshev polynomial efficiently using **Clenshaw's algorithm** (Press et al. 2007; Mason & Handscomb 2002), which is a generalization of Horner's method for regular polynomials:

1. $d_{n+2} = 0, d_{n+1} = 0$
2. For $i = n$ to 0, $d_i = (i == 0 ? 1 : 2)xd_{i+1} - d_{i-2} + c_i$
3. Return d_0

```

double operator()(double x) const
{
    assert(x >= -1 && x <= 1);
    double d = 0, dd = 0;
    for(int i = ci.getSize() - 1; i >= 0; --i)
    {
        double temp = d;
        d = (i == 0 ? 1 : 2) * x * d - dd + ci[i];
        dd = temp;
    }
    return d;
}

```

In general, if need to do some task with polynomials, need to figure out how to do it in Chebyshev basis. Can integrate the representation in Chebyshev base directly: $\int \sum c_k T_k(x) dx = \sum_{k=0}^{n+1} C_k T_k(x)$, where for $k > 0$ $C_k = \frac{c_{k-1} - c_{k+1}}{2k}$, implicitly $c_{n+1} = c_{n+2} = 0$, and C_0 is determined from the constant of integration (Mason & Handcomb 2002). Because $T_0(x) = 1$, $\frac{C_0}{2} = F(-1) - \sum_{k=1}^{n+1} C_k T_k(-1)$.

```

ChebFunction integral(double FMI = 0)
//special case for 0 polynomial

```

```

    if(ci.getSize() == 1 && ci.lastItem() == 0) return *this;
    Vector<double> result;
    result.append(FM1);
    for(int i = 1; i - 1 < ci.getSize(); ++i)
        result.append(((i - 1 > 0 ? 1 : 2) * ci[i - 1] -
                      (i + 1 > ci.getSize() - 1 ? 0 : ci[i + 1]))/2/i);
    ChebFunction cf(*this);
    cf.ci = result;
    cf.ci[0] -= cf(-1);
    return cf;
}

```

This allows evaluating $\int_{[-1,x]} f(x)$, but usually want only to evaluate for $x = 1$. In this case use a direct formula: $\int_{-1 \leq x \leq 1} \sum_{0 \leq k \leq n} c_k T_k(x) = 2 \sum_{k \text{ even}, 0 \leq k \leq n} \frac{c_k}{1-k^2}$ (Trefethen 2019). Chebyshev interpolation followed by this formula leads to **Clenshaw-Curtis quadrature**, which is one of the top algorithms for integration. Having interpolation error $|f - p| \leq \epsilon$ means the integration error is also bounded: $|\int f - \int p| \leq \int |f - p| \leq (x-a)\epsilon$, so can base a heuristic error estimate on this.

```

pair<double, double> integrate() const
{
    double result = 0;
    for(int i = 0; i < ci.getSize(); i += 2)
        result += 2 * ci[i]/(1 - i * i);
    return make_pair(result, 2 * error());
}

```

From my tests, this estimate is a bit pessimistic when the interpolation converges, which is good, and a bit too pessimistic when it doesn't, which is acceptable. The doubling error (discussed later in the chapter) performs similarly but is a bit too optimistic in all cases. Because it also needs more code, it's not considered further. A more complicated estimate is recommended in Mason & Handcomb (2002).

Theoretically have order- k convergence for f with k continuous derivatives. This estimate is pessimistic but useful. For a slightly better bound with simple constant factors see de Villiers (2012).

For calculating derivatives have similar logic, obtained from reversing the integral process: $c_{k-1} = c_{k+1} + 2kC_k$, where implicitly $c_{n+1} = c_{n+2} = 0$ (Mason & Handcomb 2002).

```

ChebFunction derivative() const
{
    int n = ci.getSize() - 1;
    ChebFunction result(*this);
    if(n == 0) result.ci[0] = 0;
    else
    {
        result.ci = Vector<double>(n, 0);
        for(int i = n; i > 0; --i) result.ci[i - 1] =
            (i + 1 > n - 1 ? 0 : result.ci[i + 1]) + 2 * i * ci[i];
        result.ci[0] /= 2;
    }
    return result;
}

```

Unlike for integration, the accuracy of the derivative doesn't follow from the accuracy of the interpolant. E.g., for a noisy function the derivative doesn't exist even for minor noise. Need strong correlation of errors.

To find all roots of f , create a **colleague matrix** from polynomial coefficients whose eigenvalues are the roots (Trefethen 2019):

1. Let n be the largest such that $c_n \neq 0$
2. Form $n \times n$ matrix C such that all entries are 0 except:
3. $C[0, 1] = 1$
4. For $1 \leq r \leq n - 1$ $C[r, r - 1] = C[r, r + 1] = 0.5$
5. For $0 \leq c < n$ add $-c_c/(2c_n)$ to $C[n - 1, c]$

The real roots of the polynomial sum correspond to the roots of f (though not the complex ones).

```

Vector<double> findAllRealRoots(double complexAbsE = highPrecEps) const
{
    int n = ci.getSize() - 1;

```

```

if(n == 0) return Vector<double>(1, 0); //all 0 case
else if(n == 1) return Vector<double>(); //no roots constant poly
//setup colleague matrix
Matrix<double> colleague(n, n);
colleague(0, 1) = 1;
for(int r = 1; r < n; ++r)
{
    colleague(r, r - 1) = 0.5;
    if(r + 1 < n) colleague(r, r + 1) = 0.5;
    if(r == n - 1) for(int c = 0; c < n; ++c)
        colleague(r, c) -= ci[c]/(2 * ci[n]);
} //solve + only keep real roots
Vector<complex<double>> croots = QREigenHessenberg(colleague);
Vector<double> result; //remove complex and extrapolated roots
for(int i = 0; i < croots.getSize(); ++i) if(abs(croots[i].imag()) <
    complexAbsE && -1 <= croots[i].real() && croots[i].real() <= 1)
    result.append(croots[i].real());
return result;
}

```

Boyd (2014) recommends to balance the matrix (see Golub & Van Loan 2012), but that's not done here. In my tests with $f(x) = \sin(x)$, on various ranges the roots are generally not very accurate, and some artificial ones show up, but these are easily handled with secant polishing (discussed later in the chapter).

To decide how many points to use to get sufficient accuracy, double until converged:

```

template<typename FUNCTION>
Vector<double> reuseChebEvalPoints(FUNCTION const& f, Vector<double> const& fx)
{
    int n = 2 * (fx.getSize() - 1);
    assert(isPowerOfTwo(n));
    Vector<double> result;
    for(int i = 0; i <= n; ++i)
        result.append(i % 2 ? f(cos(PI() * i/n)) : fx[i/2]);
    return result;
}
template<typename FUNCTION> ChebFunction adaptiveChebEstimate(
    FUNCTION const& f, int maxEvals = 10000, int minEvals = 17)
{
    int n = minEvals - 1;
    assert(minEvals <= maxEvals && isPowerOfTwo(n));
    Vector<double> fx(n + 1);
    for(int i = 0; i <= n; ++i) fx[i] = f(cos(PI() * i/n));
    ChebFunction che(fx, n);
    while(maxEvals >= fx.getSize() + n && !che.hasConverged())
    {
        fx = reuseChebEvalPoints(f, fx);
        che = ChebFunction(fx, n *= 2);
    }
    return che;
}

```

Allow scaling to an arbitrary range $[a, b]$. Then can interpolate f as though it's defined on $[-1, 1]$, and evaluate the interpolant's value and other properties:

- Interpolation—use values $f(x)$, where $x = u$ scaled from $[-1, 1]$ to $[a, b]$
- Evaluation—return $\text{interpolant}(u)$ for $u = x$ scaled from $[a, b]$ to $[-1, 1]$
- Integration—multiply the result by $(b - a)/2$
- Derivative—divide the result by $(b - a)/2$ (intuitively, it must be consistent with its finite difference estimate)
- Roots—rescale each from $[-1, 1]$ to $[a, b]$

```

static double xToU(double x, double a, double b)
{
    assert(a <= x && x <= b);
    double u = (2 * x - a - b) / (b - a);
    return u;
}

```

```

    }
    static double uToX(double u, double a, double b)
    {
        assert(-1 <= u && u <= 1);
        return ((b - a) * u + a + b)/2;
    }
template<typename FUNCTION> class ScaledFunctionM11
{//to allow [-1, 1] functions from any range
    FUNCTION f;
    double a, b;
public:
    ScaledFunctionM11(double theA, double theB, FUNCTION const& theF =
                      FUNCTION()): f(theF), a(theA), b(theB) {assert(a < b);}
    double operator()(double u) const{return f(ChebFunction::uToX(u, a, b));}
};

struct ScaledChebAB
{//to eval Cheb at any range
    ChebFunction f;
    double a, b;
public:
    template<typename FUNCTION> ScaledChebAB(FUNCTION const& theF, int n,
                                              double theA, double theB): a(theA), b(theB),
                                              f(ScaledFunctionM11<FUNCTION>(theA, theB, theF), n) {assert(a < b);}
    ScaledChebAB(Vector<double> const& fValues, double theA, double theB):
        f(fValues, 0), a(theA), b(theB) {assert(a < b);}
    ScaledChebAB(ChebFunction const& theF, double theA, double theB):
        f(theF), a(theA), b(theB) {assert(a < b);}
    double operator()(double x) const{return f(ChebFunction::xToU(x, a, b));}
    pair<double, double> integrate() const
    {
        pair<double, double> result = f.integrate();
        result.first *= (b - a)/2;
        result.second *= (b - a)/2;
        return result;
    }
    double evalDeriv(double x) const
    {
        return 2/(b - a) * f.derivative()(ChebFunction::xToU(x, a, b));
    }
    Vector<double> findAllRealRoots() const
    {//default params good enough
        Vector<double> roots = f.findAllRealRoots();
        for(int i = 0; i < roots.getSize(); ++i)
            roots[i] = ChebFunction::uToX(roots[i], a, b);
        return roots;
    }
};
}

```

23.6 Piecewise Interpolation

Linear interpolation creates a line between the two closest points enclosing the input x and calculates the y value as if it's on that line.

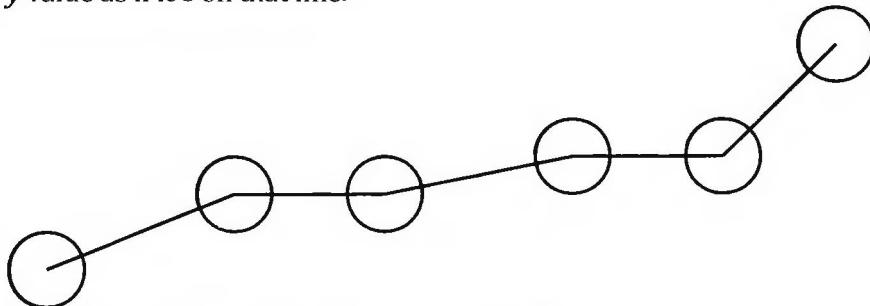


Figure 23.1: A set of points connected by lines

If f is Lipschitz, linear interpolation converges as $\max \Delta x \rightarrow 0$ because $|f - L| \leq O(\max \Delta x)$. By LIE, with enough derivatives the error in each piece is $O(\Delta x^2)$. Despite lack of smoothness, linear interpolation is

often the first method to use for given data points because the interpolated function values are always inside the range of the endpoints of the used segment, even if f isn't continuous.

The implementation handles points dynamically using a dynamic sorted sequence. To interpolate at x , find its predecessor and successor, and use their values. This takes $O(\lg(n))$ time.

Because of NaNs and a possibility of comparing a value in register with values in the data structure and then rounding before storage, can have a case where equal-when-rounded values are stored. So technically a treap is undefined with key = double, and need special care. The relation " $<$ " isn't preserved by rounding (" \leq " is).

So look for a range, and check values using some ϵ ; this also prevents duplicate points, which is often useful. The inclusive predecessor operation that uses " \leq " gives the correct result (same for inclusive successor), in a sense that the returned range is guaranteed to contain x as long as it's inside the endpoints and \neq NaN.

```
template<typename DATA> class PiecewiseData
{
    typedef Treap<double, DATA> POINTS;
    mutable POINTS values;
    double eRelAbs;

public:
    int getSize() const { return values.getSize(); }
    PiecewiseData(double theERelAbs = numeric_limits<double>::epsilon()):
        eRelAbs(theERelAbs) {}

    typedef typename POINTS::NodeType NODE;
    pair<NODE*, NODE*> findPiece(double x) const
    {
        assert(!isnan(x));
        NODE* left = values.inclusivePredecessor(x), *right = 0;
        if(!left) right = values.findMin();
        else
        {
            typename POINTS::Iterator i(left);
            ++i;
            right = i != values.end() ? &*i : 0;
        }
        return make_pair(left, right);
    }

    NODE* eFind(double x) const
    {
        assert(!isnan(x));
        pair<NODE*, NODE*> piece = findPiece(x);
        if(piece.first && isEEqual(x, piece.first->key, eRelAbs))
            return piece.first;
        if(piece.second && isEEqual(x, piece.second->key, eRelAbs))
            return piece.second;
        return 0;
    }

    NODE* findMin() const
    {
        assert(values.getSize() > 0);
        return values.findMin();
    }

    NODE* findMax() const
    {
        assert(values.getSize() > 0);
        return values.findMax();
    }

    bool isInERange(double x) const
    {
        return getSize() > 1 && findMin()->key <= x && x <= findMax()->key;
    }

    void insert(double x, DATA const& y)
    {
        NODE* node = eFind(x);
        if(node) node->value = y;
        else values.insert(x, y);
    }
};
```

```

    }
    void eRemove(double x)
    {
        NODE* node = eFind(x);
        if(node) values.removeFound(node);
    }
    Vector<pair<double, DATA>> getPieces() const
    {
        Vector<pair<double, DATA>> result;
        for(typename POINTS::Iterator iter = values.begin();
            iter != values.end(); ++iter)
            result.append(make_pair(iter->key, iter->value));
        return result;
    }
};


```

Implement linear interpolation using the above:

```

class DynamicLinearInterpolation
{
    PiecewiseData<double> pd;
public:
    DynamicLinearInterpolation() {}
    DynamicLinearInterpolation(Vector<pair<double, double>> const& xy)
    //filter points to ensure eps > distance, use machine eps
        assert(isSorted(xy.getArray(), 0, xy.getSize() - 1,
            PairFirstComparator<double, double>()));
        for(int i = 0, lastGood = 0; i < xy.getSize(); ++i)
            if(i == 0 || isELess(xy[lastGood].first, xy[i].first))
            {
                insert(xy[i].first, xy[i].second);
                lastGood = i;
            }
    }
    double operator()(double x) const
    {
        if(!pd.isInERange(x)) return numeric_limits<double>::quiet_NaN();
        typedef typename PiecewiseData<double>::NODE NODE;
        pair<NODE*, NODE*> segment = pd.findPiece(x);
        assert(segment.first && segment.second); //sanity check
        double ly = segment.first->value, lx = segment.first->key;
        return ly + (segment.second->value - ly) * (x - lx) /
            (segment.second->key - lx);
    }
    void eRemove(double x){pd.eRemove(x);}
    bool eContains(double x){return pd.eFind(x) != 0;}
    void insert(double x, double y){pd.insert(x, y);}
};


```

The remove operation isn't common but useful if the evaluated points get outdated.

The next step is higher-degree piecewise polynomials. For f with enough derivatives they reduce approximation error. To use piecewise polynomials of degree d , split $n=dm+1$ points into joined segments of d points each, with the convention that the left data point of a segment contains the polynomial. The setup is $O(nd)$, and evaluation $O(\lg(n/d) + d)$.

Because Chebyshev polynomials are more stable, use them as basis for piecewise interpolation.

```

template<typename INTERPOLANT> class GenericPiecewiseInterpolation
{
    PiecewiseData<INTERPOLANT> pd;
public:
    GenericPiecewiseInterpolation()
        PiecewiseData<INTERPOLANT> const& thePd): pd(thePd){}
    double operator()(double x) const
    {
        if(!pd.isInERange(x)) return numeric_limits<double>::quiet_NaN();
    }
};


```

```

typename PiecewiseData<INTERPOLANT>::NODE* segment =
    pd.findPiece(x).first;
assert(segment); //sanity check
return segment->value(x);
}

Vector<pair<double, double>, INTERPOLANT> > getPieces() const
{
    Vector<pair<double, INTERPOLANT>> pieces = pd.getPieces();
    assert(pieces.getSize() > 1); //1 real, 1 dummy
    Vector<pair<double, double>, INTERPOLANT> > result;
    for(int i = 0; i < pieces.getSize(); ++i)
    {
        if(result.getSize() > 0) //set right boundary of prev piece
            result.lastItem().first.second = pieces[i].first;
        result.append(make_pair(make_pair(pieces[i].first, 0),
            pieces[i].second));
    }
    result.removeLast(); //last piece is dummy
    return result;
}

GenericPiecewiseInterpolation<INTERPOLANT> deriver() const
{
    Vector<pair<double, INTERPOLANT>> pieces = pd.getPieces();
    PiecewiseData<INTERPOLANT> result;
    for(int i = 0; i < pieces.getSize(); ++i) //last one dummy
        result.insert(pieces[i].first, i < pieces.getSize() - 1 ?
            pieces[i].second.deriver() : pieces[i].second);
    return GenericPiecewiseInterpolation<INTERPOLANT>(result);
}

double integrate() const
{
    double sum = 0;
    Vector<pair<double, double>, INTERPOLANT> > pieces = getPlaces();
    for(int i = 0; i < pieces.getSize(); ++i)
        sum += pieces[i].second.integrate();
    return sum;
}
};

```

It's hard to decide how many points to use, so can use a global adaptive strategy, as for adaptive integration (discussed later in the chapter). For an error estimate on an interval, use the maximum error among new points that will be used to split the interval. The idea is simple, though the code is a bit tricky.

```

template<typename ADAPTIVE_INTERVAL> struct AdaptiveIntervalComparator
{
    double deltaLength;
    bool operator()(ADAPTIVE_INTERVAL const& lhs,
        ADAPTIVE_INTERVAL const& rhs) const
    {
        return (lhs.length() > deltaLength || rhs.length() > deltaLength) ?
            lhs.length() > rhs.length() : lhs.error() > rhs.error();
    }
};

template<typename INTERPOLATION_INTERVAL, typename FUNCTION>
pair<GenericPiecewiseInterpolation<
    typename INTERPOLATION_INTERVAL::INTERPOLANT>,
    double> interpolateAdaptiveHeap(FUNCTION const& f, double a, double b,
    double param, double eRelAbs = highPrecEps, int maxEvals = 1000000,
    int minEvals = -1)

{
    typedef INTERPOLATION_INTERVAL II;
    typedef typename II::INTERPOLANT I;
    if(minEvals == -1) minEvals = sqrt(maxEvals);
    assert(a < b && maxEvals >= minEvals && minEvals >= II::initEvals(param));
}

```

```

INTERPOLATION_INTERVAL i0(f, a, b, param);
double scale = i0.scaleEstimate();
AdaptiveIntevalComparator<II> ic = ((b - a)/minEvals};
Heap<II, AdaptiveIntevalComparator<II>> h(ic);
h.insert(i0);
for int usedEvals = II::initEvals(param);
    usedEvals < minEvals || (
        usedEvals + II::splitEvals(param) <= maxEvals &&
        !isEEqual(scale, scale + h.getMin().error(), eRelAbs));
    usedEvals += II::splitEvals(param)
{
    II next = h.deleteMin();
    Vector<II> division = next.split(f);
    for (int i = 0; i < division.getSize(); ++i) h.insert(division[i]);
}//process heap intervals
PiecewiseData<I> pd;
double error = h.getMin().error(),
    right = -numeric_limits<double>::infinity();
while (!h.isEmpty())
{
    II next = h.deleteMin();
    Vector<pair<double, I>> interpolants = next.getInterpolants();
    for (int i = 0; i < interpolants.getSize(); ++i)
        pd.insert(interpolants[i].first, interpolants[i].second);
}//need dummy right endpoint interpolator
pd.insert(b, i0.getInterpolants().lastItem().second);
return make_pair(GenericPiecewiseInterpolation<I>(pd), error);
}

```

May also want to guard against too small subintervals, though in my tests this wasn't needed.

For Chebyshev pieces, if doubling doesn't lead to deeming convergence when reach some threshold k , such as 64 used here, recursively split a given interval in two, and repeat, though without being able to reuse points. When f isn't Lipschitz, or the convergence of doubling is only linear, this has many advantages:

- The Lebesgue constant is bounded
- The calculations are more efficient—need $O(k \lg(n/k))$ time per evaluation
- On some pieces f may be Lipschitz or allow high-order convergence, and these won't be affected by other pieces. So the L_2 error should be better.

But, as with regular piecewise polynomials, derivatives aren't continuous. Unlike a recursive implementation, global heap subdivision offers better control.

```

class IntervalCheb
{
    ScaledChebAB cf;
    pair<double, double> ab;
    int maxEvals;
public:
    typedef ScaledChebAB INTERPOLANT;
    template<typename FUNCTION> IntervalCheb(FUNCTION const& f, double a,
        double b, int theMaxEvals = 64): ab(a, b), cf(f, theMaxEvals, a, b),
        maxEvals(theMaxEvals){}
    Vector<pair<double, ScaledChebAB>> getInterpolants() const
    {
        return Vector<pair<double, ScaledChebAB>>(1, make_pair(ab.first, cf));
    }
    double scaleEstimate() const {return 1;}
    double length() const {return 0;}
    double error() const //larger first
        {return cf.f.hasConverged() ? 0 : ab.second - ab.first;}
    template<typename FUNCTION>
    Vector<IntervalCheb> split(FUNCTION const& f) const
    {
        Vector<IntervalCheb> result;
        double middle = (ab.first + ab.second)/2;
    }
}

```

```

        result.append(IntervalCheb(f, ab.first, middle, maxEvals));
        result.append(IntervalCheb(f, middle, ab.second, maxEvals));
        return result;
    }
    static int initEvals(int maxEvals) {return maxEvals;}
    static int splitEvals(int maxEvals) {return 2 * maxEvals;}
} ;

```

23.7 Splines—For Existing Data

For piecewise polynomials a problem is that the derivatives aren't continuous at the nodes, and in certain applications need smoothness of first and maybe 2nd derivative. Also it's inconvenient to fit anything but linear piecewise to existing data. **Cubic splines** ensure continuous derivatives by enforcing for spline s continuity of s' and s'' at the interpolation points. Let $M_i = s''(x_i)$. Then for $x \in [x_{i-1}, x_i]$

$$s''(x) = \frac{M_{i-1}(x_i - x) + M_i(x - x_{i-1})}{h_i}, \quad \text{where} \quad h_i = x_i - x_{i-1}. \quad \text{Integrating twice,}$$

$$s_{i-1}(x) = \frac{M_{i-1}(x_i - x) + M_i(x - x_{i-1})}{6h_i} + b_{i-1}(x - x_{i-1}) + c_{i-1}. \quad \text{Matching segment end point values leads to}$$

$$b_{i-1} = f_{i-1} - M_{i-1} \frac{h_i^2}{6}, \text{ and } c_{i-1} = \Delta f_i - (M_i - M_{i-1}) \frac{h_i}{6}, \text{ where } \Delta f_i = \frac{f_i - f_{i-1}}{h_i}.$$

Enforcing continuity of f' leads to a system of equations $\mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} = d_i$, for $1 \leq i \leq n-2$, where $\mu_i = \frac{h_i}{h_i + h_{i+1}}$, $\lambda_i = \frac{h_{i+1}}{h_i + h_{i+1}}$, and $d_i = \frac{6(\Delta f_{i+1} - \Delta f_i)}{h_i + h_{i+1}}$. Need boundary conditions of the form $2M_0 + \lambda_0 M_1 = d_0$ for $i=0$ and $\mu_{n-1} M_{n-2} + 2M_{n-1} = d_{n-1}$ for $i=n-1$.

The simplest alternative is the **natural cubic spline** that becomes a line outside the data range (Fausett 2003), given by $2M_0 = 2M_{n-1} = 0$. Its accuracy in a compact interval inside the endpoints is $O(h^4)$, assuming enough continuous derivatives, and $h \rightarrow 0$ (Dahlquist & Björck 2008). But the limiting accuracy near the endpoints is only $O(h^2)$, so it's better to use a **not-a-knot spline** which maintains the $O(h^4)$ accuracy inside the endpoints (Beatson & Chacko 1990; Dahlquist & Björck 2008). The accuracy of not-a-knot and some other splines is hard to analyze, with some main results not known until Beatson (1986).

The idea of not-a-knot is to enforce continuity of $s^{(3)}$ at x_1 and x_{n-2} , i.e., the first two and last two segments become the same polynomial by “dropping” these points and only using them for the boundary conditions. This doesn't change h at the endpoints for the bounds because s still passes through those points.

Differentiating $s''(x)$, get $s^{(3)}(x_1) = \frac{M_1 - M_0}{h_1} = \frac{M_2 - M_1}{h_2}$, which leads to

$2h_2 M_0 - 2(h_1 + h_2) M_1 + 2h_1 M_2 = 0$. When plug in the values of μ_1 and λ_1 into the non-boundary equations, get $h_1 M_0 + 2(h_1 + h_2) M_1 + h_2 M_2 = d_1(h_1 + h_2)$. This allows to eliminate M_1 , so that in the not-a-knot system

get the boundary condition $2M_0 + 2M_2 \frac{2h_1 + h_2}{2h_2 + h_1} = 2d_1 \frac{h_1 + h_2}{2h_2 + h_1}$. Though M_1 is missing, x_1 is used to compute h_1, h_2 , and d_1 .

Analogously, $s^{(3)}(x_{n-2}) = \frac{M_{n-2} - M_{n-3}}{h_{n-2}} = \frac{M_{n-1} - M_{n-2}}{h_{n-1}}$, which leads to

$2h_{n-1} M_{n-3} - 2(h_{n-2} + h_{n-1}) M_{n-2} + 2h_{n-2} M_{n-1} = 0$. When plug in the values of μ_{n-2} and λ_{n-2} into the non-boundary equations, get $h_{n-2} M_{n-3} + 2(h_{n-2} + h_{n-1}) M_{n-2} + h_{n-1} M_{n-1} = d_{n-2}(h_{n-2} + h_{n-1})$. After eliminating M_{n-2}

get $2M_{n-3} \frac{2h_{n-1} + h_{n-2}}{2h_{n-2} + h_{n-1}} + 2M_{n-1} = 2d_{n-2} \frac{h_{n-2} + h_{n-1}}{2h_{n-2} + h_{n-1}}$.

The resulting tridiagonal system isn't diagonally dominant due to the first and last rows, but the Gauss-Thomas algorithm still works. The worst case for the first boundary equation happens when $\frac{h_2}{h_1} \rightarrow 0$, leading to $2M_0 + 4M_2 = R_0$. Then, because $h_1 + h_2 \rightarrow h_1$, the limiting 2nd (nonboundary) equation of the system without M_1 becomes $M_0 \frac{h_1}{h_1 + h_3} + 2M_2 + M_3 \frac{h_3}{h_1 + h_3} = R_1$. So for $\frac{h_3}{h_1} \rightarrow 0$, noting that $\frac{h_1}{h_1 + h_3} \approx 1 - \frac{h_3}{h_1}$, after

elimination of M_0 by subtracting the boundary equation $\times \frac{1}{2} \left(1 - \frac{h_3}{h_1}\right)$, the second equation $\approx 2 \frac{h_3}{h_1} M_2 + \frac{h_3}{h_1} M_3 = R_1 - \left(1 - \frac{h_3}{h_1}\right) \frac{R_0}{2}$, which is diagonally dominant.

By symmetry, if the elimination process was done in reverse, the one-before-the-last equation would remain diagonally dominant. So when the regular elimination, which keeps all but the last equation diagonally dominant, gets to the last equation, the one-before-the-last equation will retain enough diagonal dominance (and usually gain more), so there will be no cancellation. For this process to work, need at least one nonboundary equation, which is the case if $n = 5$ —the solution consists of two splines, which is the minimum given that one spline is impossible with not-a-knot.

```

class NotAKnotCubicSplineInterpolation
{//back M is a, back d is R
    struct Data{double a, b, c;};
    PiecewiseData<Data> pd;
public:
    NotAKnotCubicSplineInterpolation(Vector<pair<double, double> > xy,
        double eRelAbs = numeric_limits<double>::epsilon()): pd(eRelAbs)
    {//filter points to ensure eps x distance
        int n = xy.getSize(), skip = 0;
        assert(isSorted(xy.getArray(), 0, n - 1,
            PairFirstComparator<double, double>()));
        for(int i = 1; i + skip < n; ++i)
        {
            if (!isELess(xy[i - 1].first, xy[i + skip].first, eRelAbs)) ++skip;
            if(i + skip < n) xy[i] = xy[i + skip];
        }
        while(skip--) xy.removeLast();
        n = xy.getSize();
        assert(n > 3);//need 4 points to fit a cubic
//special logic for endpoints
        double h1 = xy[1].first - xy[0].first, h2 = xy[2].first - xy[1].first,
            t1 = (xy[1].second - xy[0].second)/h1,
            t2 = (xy[2].second - xy[1].second)/h2,
            hnm2 = xy[n - 2].first - xy[n - 3].first,
            hnm1 = xy[n - 1].first - xy[n - 2].first,
            tnm2 = (xy[n - 2].second - xy[n - 3].second)/hnm2,
            tnm1 = (xy[n - 1].second - xy[n - 2].second)/hnm1;
//take out points 1 and n - 2
        for(int i = 1; i < n - 3; ++i) xy[i] = xy[i + 1];
        xy[n - 3] = xy[n - 1];
        xy.removeLast();
        xy.removeLast();
        n = xy.getSize();
//setup and solve tridiagonal system
        TridiagonalMatrix<double> T(
            2 * TridiagonalMatrix<double>::identity(n));
        Vector<double> R(n);
//boundary conditions
        double D0Factor = 2/(2 * h2 + h1), DnmlFActor = 2/(2 * hnm2 + hnm1);
        T(0, 1) = (2 * h1 + h2) * D0Factor;
        R[0] = 6 * (t2 - t1) * D0Factor;
        T(n - 1, n - 2) = (2 * hnm1 + hnm2) * DnmlFActor;
        R[n - 1] = 6 * (tnm1 - tnm2) * DnmlFActor;
        for(int i = 1; i < n - 1; ++i)
        {
            double hk = xy[i].first - xy[i - 1].first,
                tk = (xy[i].second - xy[i - 1].second)/hk,
                hkpl = xy[i + 1].first - xy[i].first, hSum = hk + hkpl,
                tkpl = (xy[i + 1].second - xy[i].second)/hkpl;
            R[i] = 6 * (tkpl - tk)/hSum;
        }
    }
}

```

```

        T(i, i + 1) = hkp1/hSum;
        T(i, i - 1) = hk/hSum;
    }
    Vector<double> a = solveTridiag(T, R);
    //compute b and c
    for(int i = 1; i < n + 1; ++i)
    {
        double bi = 0, ci = 0;
        if(i < n)
        {
            double hi = xy[i].first - xy[i - 1].first;
            bi = (xy[i].second - xy[i - 1].second)/hi -
                (a[i] - a[i - 1]) * hi/6;
            ci = xy[i - 1].second - a[i - 1] * hi * hi/6;
        }
        Data datai = {a[i - 1], bi, ci};
        pd.insert(xy[i - 1].first, datai);
    }
}
double operator()(double x, int deriv = 0) const
{
    assert(deriv >= 0 && deriv <= 2); //support 2 continuous derivatives
    if(!pd.isInERange(x)) return numeric_limits<double>::quiet_NaN();
    typedef typename PiecewiseData<Data>::NODE NODE;
    pair<NODE*, NODE*> segment = pd.findPiece(x);
    assert(segment.first && segment.second); //sanity check
    double dxl = x - segment.first->key, dxr = segment.second->key - x,
           aim1 = segment.first->value.a, ai = segment.second->value.a,
           bi = segment.first->value.b, ci = segment.first->value.c,
           hi = dxr + dxl;
    if(deriv == 2) return (aim1 * dxr + ai * dxl)/hi;
    if(deriv == 1) return (-aim1 * dxr * dxr + ai * dxl * dxl)/(hi * 2) +
        bi;
    return (aim1 * dxr * dxr * dxr + ai * dxl * dxl * dxl)/(hi * 6) +
        bi * dxl + ci;
}
;
}

```

Why use cubic splines and not quadratic or higher-order ones? Several reasons for their being a golden middle:

- The computation is $O(n)$.
- With two continuous derivatives have a mechanical interpretation—both velocity and acceleration are continuous so don't have abrupt motion.
- Quadratic splines have only one continuous derivative, which isn't enough for some applications.
- A high degree of smoothness may be needed only in rare application-specific cases. But the derivation becomes tedious, the resulting matrix is no longer tridiagonal (but in some cases few-diagonal), and for stability reasons the computation is generally handled using **B-splines**, which aren't discussed here (but see the comments section).

It's hard to compare errors of piecewise cubic and cubic splines. Both have $O(h^4)$ error with sufficiently differentiable f , and my simple experiments on several functions show that either can be better than the other by a small constant factor. The **complete cubic spline**, which assumes knowledge of f at the endpoints, has better constant factors in the error compared to a piecewise cubic, and not-a-knot approximates it with tiny error (Dahlquist & Björck 2008), so it appears to be slightly better. Beatson (1986) gives error bounds for when f has few continuous derivatives, but the constants are hard to compare with the piecewise ones. But because piecewise polynomials have known simple analytical properties, use them in most cases.

For chosen points splines don't make sense. Picking n isn't easy, so doubling is a possible strategy:

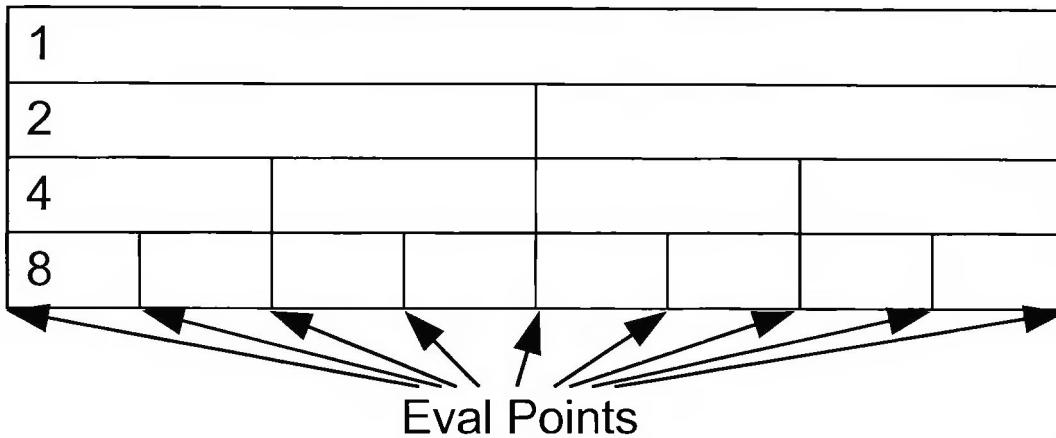


Figure 23.2: Reuse of evaluation points with doubling the number of intervals

The issue is more that other methods do a much better job at choosing the wanted points to get the wanted accuracy adaptively.

For splines the Lebesgue constant is constant for equispaced points (Ueberhuber 1997a). Spline interpolation will beat linear interpolation for already given data and a smooth f , but the use case is rare, particularly because interpolation is rarely the final task.

23.8 Comparison of Interpolation Methods

For piecewise polynomials of a fixed degree, as discussed previously, the Lebesgue constant is bounded by a constant that depends on the lengths of the pieces. In particular, it's also bounded by a constant for equispaced points.

But nothing helps at a discontinuity point. No approximants can have 0 error there—they only converge at nearby points. Considering the goal of interpolation simplifies decisions. Some typical use cases:

1. Want to evaluate $f(x)$ much more efficiently than by direct evaluation. If f is an elementary function such as \cos or anything else that is continuous with many derivatives, Chebyshev interpolation of some fixed degree, picked by experimentation, will work. But approximating elementary functions is a field on its own with many useful techniques, discussed later in the chapter. Usually this can't be done for a black-box f , and for a particular f need to consider its properties carefully. Needless to say, don't try to optimize a library function such as e^x —will lose any robustness.
2. Same as (1) but f is a result of an expensive experiment. The true output function is highly likely to not be continuous. Linear interpolation is the default option, but can use piecewise quadratics or cubics if expect some smoothness. If expect noise, regression is more approximate (see the "Machine Learning—Regression" chapter).
3. Want to interpolate from a table of values—use linear interpolation due to limited accuracy of table values. But instead in most cases special library functions can compute the wanted value reasonably cheaply and accurately.
4. Model some task such as integration, where analytical properties of polynomial pieces are useful. Interpolation in this case will use task-dependent heuristics such as error estimates and adaptive strategies. E.g., all interpolants do poorly on the step function with respect to the error in the maximum norm and the number of function evaluations, but integrators do well in both.

Function	Cheb Adapt		Cheb64 Adaptive		DynLin	
	Err	Evals	Err	Evals	Err	Evals
Step	0.00	4097	-15.65	6175	-0.13	1000
Abs	-3.84	4097	-15.65	1105	-2.96	1000
Lin	-15.65	17	-15.65	1105	-15.65	1000
Square	-15.65	17	-15.65	1105	-4.22	1000
Cube	-15.65	17	-15.65	1105	-4.11	1000
Quad	-15.65	17	-15.65	1105	-3.74	1000
Exp	-14.77	17	-15.17	1105	-4.52	1000
SqrtAbs	-1.94	4097	-14.59	12155	-1.86	1000
Runge	-13.40	1025	-14.44	1105	-2.47	1000
Log	-15.30	33	-14.68	1105	-5.37	1000
XSinXM1	-1.60	4097	-5.32	1015625	-1.23	1000
Sin	-15.65	17	-14.87	1105	-4.74	1000
AbsIntegral	-7.65	4097	-15.65	1105	-4.60	1000
Tanh	-15.00	33	-15.15	1105	-4.84	1000
NormalPDF	-14.83	125	-14.76	1105	-3.19	1000
DeltaPDF	-2.54	4097	-15.65	10595	-1.33	1000
F575	0.83	4097	-14.67	6825	0.00	1000
Average Ranks	1.7	1.6	1.2	2.9	2.8	1.5

Figure 23.3: Experiments on some f . Errors are given in terms of decimal digits of relative-absolute error with respect to the known answer. Same for estimated errors, where applicable. To estimate maximum error used 10^6 uniformly random points in a function's range (specified in the test file). In all cases all interpolants do poorly on f with jumps such as the step function, so such cases won't be discussed further.

Some rough general patterns:

- For adaptive interpolation:
 - When f has many continuous derivatives, doubling Chebyshev interpolation wins by far, often estimating f to machine precision with <100 evaluations (though it took ≈ 1000 for Runge's function). But if f has nondifferentiability, it exhausts the evaluation limit and appears to give very roughly $O(1/n)$ error for n evaluations.
 - Using piecewise Chebyshev of maximum degree 64 uses ≈ 1000 evaluations in all cases (due to safety limit in the adaptive procedure), but always goes to $\epsilon_{\text{machine}}$, except for functions like $x\sin(1/x)$ with a deliberate continuum of issues.
- For 1000 random preselected evaluation points:
 - Linear interpolation does well, with very roughly error $O(1/n)$ for nondifferentiable f and $O(1/n^2)$ for differentiable f , as predicted by theory.
 - Splines have slightly larger errors in the former cases, but almost reach $\epsilon_{\text{machine}}$ in the latter case.

So how to do interpolation depends on the final goal. Some general conclusions:

- Use Chebyshev polynomials for everything for analytic f .
- For intermediate result interpolation, piecewise polynomials are the easiest to work with analytically.
- For graphing a data set linear interpolation is the default option in MATLAB because other options like cubic splines can lead to oscillations (Moler 2013). The “ugliness” of it is just a deception to the human eye.

Splines aren't generally useful, but win for working with existing data in some cases:

- Graphing data where linear interpolation is too coarse
- Estimating first and second derivatives where must ensure continuity of estimates

23.9 Integration

The definite integral of f over a range = the volume of the range \times the average f value in the range. Want to find it with only the ability to evaluate f at any point in the range. This assumes a well-defined integral, i.e., no infinities, NaNs, infinite ranges, etc., though can handle many of these by analytical preprocessing. E.g., consider $\int_{0 \leq x \leq 1} e^x$. Analytically the answer is e , but want it numerically.

For 1D, the simplest method is the **trapezoid rule**—linearly interpolate the function at equally spaced intervals, and integrate the interpolation. So use $n + 1$ sample points to get $\int_{a \leq x \leq b} f(x) dx \approx$

$\sum_{0 \leq i < n} (x_{i+1} - x_i) \frac{f(x_{i+1}) + f(x_i)}{2}$. From information complexity, this is worst-case optimal for Lipschitz f (Traub & Werschulz 1998).

The trapezoid rule is easy to use for already evaluated f values, in particular random values. The range of integration is assumed to be given by the smallest and the largest x_i . Duplicate values don't cause problems. The only trick is estimating the error, for which use the **doubling error** = [the integral using all the values – the integral using half of the values]. This estimate is an example of Cauchy convergence and can differ from the actual error by orders of magnitude, but usually not by much.

```
pair<double, double> integrateFromData(Vector<pair<double, double>> > xyPairs)
{
    assert(xyPairs.getSize() >= 3);
    quickSort(xyPairs.getArray(), 0, xyPairs.getSize() - 1,
        PairFirstComparator<double, double>());
    double result[2] = {0, 0}, last = 0;
    for(int j = 2; j >= 1; --j)
        for(int i = 0; i + j < xyPairs.getSize(); i += j) result[j - 1] +=
            last = (xyPairs[i + j].first - xyPairs[i].first) *
            (xyPairs[i + j].second + xyPairs[i].second)/2;
    if(xyPairs.getSize() % 2 == 0) result[1] += last;
    return make_pair(result[0], abs(result[0] - result[1]));
}
```

Can do the same with piecewise polynomials of a fixed order, but this is clumsy. E.g., the barycentric form doesn't allow integration (see the comments). Integration from data is already questionable as potentially the wrong thing to do, but use of trapezoid rule seems to be a good option.

The doubling error is a general estimate that applies to any algorithm. It's an enhancement to the general doubling until "happy" strategy. But "happy" is a Boolean, so it makes sense to stop doubling when the error is small because reach diminishing returns. As an error estimate, it's perhaps justified by that using double the number of information points supposedly leads to an entirely different algorithm, and the difference in their results is some indication of by how much one of them is wrong.

The trapezoid rule is convergent if $\max_i (x_{i+1} - x_i) \rightarrow 0$, and f is Lipschitz with constant C . Assuming same-length intervals for simplicity, the worst case error in such interval is $O(\Delta x^2)$, given by a triangle shape with peak height bounded $C\Delta x/2$ and area $C\Delta x^2/2$, so the total error is $O(\Delta x)$. This logic applies to all rules of the type $\sum \Delta x f_{\text{ave}}$, such as the basic calculus Riemann sums, Simpson, etc., where f_{ave} is estimated by a weighted average of several evaluations inside the Δx interval.

For f with 2 continuous derivatives and equal-length intervals, $\left| \int_{a \leq x \leq b} f(x) - \Delta x \sum'_{0 \leq i < n} f(x_i) \right| \leq \|f''\| \Delta x^2 \frac{b-a}{12}$ (Ascher & Greif 2011; Süli & Mayers 2003), i.e., the convergence order is $O(n^{-2})$.

Chebyshev interpolation leads to efficient and accurate **Clenshaw-Curtis quadrature**. To apply it, first transform the integration range to $[-1, 1]$. No precision target is given, only the limit on the number of evaluations because try to get $\epsilon_{\text{machine}}$ precision.

```
template<typename FUNCTION> pair<double, double> integrateCC(FUNCTION const& f,
    double a, double b, int maxEvals = 5000, int minEvals = 17)
{
    ScaledChebAB c(adaptiveChebEstimate(ScaledFunctionM11<FUNCTION>(a, b, f),
        maxEvals, minEvals), a, b);
    return c.integrate();
}
```

For analytic f Clenshaw-Curtis converges exponentially fast in n ; otherwise convergence is polynomial in n and exponential in the number of continuous derivatives (Trefethen 2019).

Simpson's rule uses piecewise quadratic interpolation where each segment is evaluated at the left (0), the middle (1), and in the right (2). Assuming one segment for the whole interval, $\int_{a \leq x \leq b} f(x) \approx (b-a) \frac{f_0 + 4f_1 + f_2}{6}$ (Press et al. 2007). Extending this to many segments gives $O(f^{(4)} \Delta x^4)$ error;

see Ascher & Greif (2011) or Süli & Mayers (2003) for the exact expression. In many cases, such as for the trapezoid and Simpson rules, the doubling error has a known analytic expression when f has enough continuous derivatives. E.g., for Simpson halving gives error = the difference/15 (Ascher & Greif 2011). But relying on such estimates isn't robust, whereas the justification for agnostic doubling difference is.

Adaptive integration allocates more f evaluations to subregions of higher variation. The heap strategy

picks next the interval with greatest potential for error reduction (Gonnet 2009), like the A* state space search and nearest neighbor algorithms of multidimensional trees. In a way it's the most efficient way to reduce uncertainty about $[a, b]$ given the current evaluation points. Adaptive integration also gives a more robust error estimate than the doubling strategy.

For convergence need $\max \Delta x \rightarrow 0$, so $> O(1)$ evaluations must be given to a particular interval, and $\sqrt{\max \text{evals}}$ seems to be a good heuristic choice. So intervals with larger errors are preferred, except for large enough intervals use preference by size to avoid deceptive cases where a large interval reports a small error by chance. Designing such a comparator is a bit tricky—e.g., using relative interval sizes and not a pre-determined minimum length leads to a “ $<$ ” operator that isn't transitive (priority design with several variables is difficult because can't easily combine them into a single priority).

```
template<typename INTEGRATION_INTERVAL, typename FUNCTION> pair<double, double>
integrateAdaptiveHeap(FUNCTION const& f, double a, double b, double
eRelAbs = highPrecEps, int maxEvals = 1000000, int minEvals = -1)
{
    typedef INTEGRATION_INTERVAL II;
    if(minEvals == -1) minEvals = sqrt(maxEvals);
    assert(a < b && maxEvals >= minEvals && minEvals >= II::initEvals());
    II i0(f, a, b);
    double result = i0.integrate(), totalError = i0.error();
    AdaptiveIntervalComparator<II> ic = {(b - a)/minEvals};
    Heap<II, AdaptiveIntervalComparator<II> > h(ic);
    h.insert(i0);
    for(int usedEvals = II::initEvals();
        usedEvals < minEvals || (usedEvals + II::splitEvals() <= maxEvals &&
        !isEEqual(result, result + totalError, eRelAbs));
        usedEvals += II::splitEvals())
    {
        II next = h.deleteMin();
        Vector<II> division = next.split(f);
        result -= next.integrate();
        totalError -= next.error();
        for(int i = 0; i < division.getSize(); ++i)
        {
            h.insert(division[i]);
            result += division[i].integrate();
            totalError += division[i].error();
        }
    }
    return make_pair(result, totalError);
}
```

Using Simpson intervals works well (implementation not presented), but prefer higher-order formulas such as the **Gauss-Lobatto-Kronrod rule** (Gander et al. 2014). It's based on polynomial interpolation with orthogonal polynomials on $[-1, 1]$, but instead of Chebyshev polynomials uses derivatives of Legendre polynomials. They have some important properties:

- They are exact for polynomials of high degree for a given number of evaluation nodes
- The evaluation nodes include the endpoints -1 and 1 , need which to reuse the evaluations within a subdivision
- **Kronrod extension** allows constructing a higher-order formula while reusing previous evaluations. In particular, from a 4-point rule can create a 7-point rule, and divide the interval in 6 pieces if need to increase depth.

See Gander et al. (2014) for details. The implementation below is much simpler in a sense that the error measure = the 7-point Kronrod integral – the less accurate 4-point Gauss-Lobatto.

```
class IntervalGaussLobattoKronrod
{
    double a, b, y[7];
    void generateX(double x[7]) const
    {
        x[0] = -1; x[1] = -sqrt(2.0/3); x[2] = -1/sqrt(5); x[3] = 0;
        x[4] = -x[2]; x[5] = -x[1]; x[6] = -x[0];
    }
}
```

```

template<typename FUNCTION> void initHelper(FUNCTION const& f, double fa,
    double fb)
{
    assert(a < b);
    y[0] = fa;
    y[6] = fb;
    double x[7];
    generateX(x);
    ScaledFunctionM11<FUNCTION> fM11(a, b, f);
    for(int i = 1; i < 6; ++i) y[i] = fM11(x[i]);
}
double integrateGL() const
{
    double w[7] = {1.0/6, 0, 5.0/6, 0};
    w[4] = w[2]; w[5] = w[1]; w[6] = w[0];
    double result = 0;
    for(int i = 0; i < 7; ++i) result += w[i] * y[i];
    return result * (b - a)/2;
}
template<typename FUNCTION> IntervalGaussLobattoKronrod(FUNCTION const& f,
    double theA, double theB, double fa, double fb): a(theA), b(theB)
    {initHelper(f, fa, fb);}
public:
    template<typename FUNCTION> IntervalGaussLobattoKronrod(FUNCTION const& f,
        double theA, double theB): a(theA), b(theB)
        {initHelper(f, f(a), f(b));}
    double integrate() const
    {
        double w[7] = {11.0/210, 72.0/245, 125.0/294, 16.0/35};
        w[4] = w[2]; w[5] = w[1]; w[6] = w[0];
        double result = 0;
        for(int i = 0; i < 7; ++i) result += w[i] * y[i];
        return result * (b - a)/2;//need to scale back
    }
    double length() const{return b - a;}
    double error() const{return abs(integrate() - integrateGL());}
    template<typename FUNCTION>
    Vector<IntervalGaussLobattoKronrod> split(FUNCTION const& f) const
    {
        Vector<IntervalGaussLobattoKronrod> result;
        double x[7];
        generateX(x);
        for(int i = 0; i < 7; ++i) x[i] = a + (x[i] - -1) * (b - a)/2;
        for(int i = 0; i < 6; ++i) result.append(
            IntervalGaussLobattoKronrod(f, x[i], x[i + 1], y[i], y[i + 1]));
        return result;
    }
    static int initEvals(){return 7;}
    static int splitEvals(){return 30;}
};

```

The n -point **Gauss-Lobatto rule** is exact for polynomials of degree $2n - 1$ (Ascher & Greif 2011; Süli & Mayers 2003), so with $n = 4$ integrate exactly polynomials of degree 7. The 7-point Kronrod extension integrates exactly polynomials of degree 10 (Dahlquist & Björck 2008). Instead using Clenshaw-Curtis with 4 and 8 points needs breaking into 7 intervals, which is worse than into 6. With point reuse can't have both adaptivity and high order.

Given that Clenshaw-Curtis converges to $\epsilon_{\text{machine}}$ by about 1000 evaluations for very smooth f , and often even by 100, combine it with adaptive integration for nonsmooth f (adaptivity is extremely efficient with discontinuities) or very smooth f joined by a discontinuity. Give $\min(\max(\text{evals}/2, 1000))$ evaluations to it and the rest, if necessary, to an adaptive integrator, which appears to be a good heuristic choice. Adaptive integration can't reuse the Clenshaw-Curtis evaluations as implemented, but this is a small loss for simplicity.

```
template<typename FUNCTION> pair<double, double> integrateHybrid(
```

```

FUNCTION const< f, double a, double b, double eRelAbs = highPrecEps,
int maxEvals = 1000000, int minEvals = -1)

{
    int CCEvals = min(maxEvals/2, 1000);
    pair<double, double> resultCC = integrateCC(f, a, b, CCEvals);
    if(isEEqual(resultCC.first, resultCC.first + resultCC.second, eRelAbs))
        return resultCC;
    pair<double, double> resultGLK = integrateAdaptiveHeap<
        IntervalGaussLobattoKronrod>(f, a, b, eRelAbs, maxEvals - CCEvals,
        minEvals);
    return resultCC.second < resultGLK.second ? resultCC : resultGLK;
}

```

From information complexity, it's impossible to integrate any deterministic f exactly (but no problem if f is Lipschitz) (Corless & Fillion 2013). To construct a problematic function g , run any integration algorithm on $f = 0$, and record the evaluation points x_i . Then define $g(x) = c \prod (x - x_i)$ for constant c . The algorithm will say that $\int g = 0$, while changing c makes the integral take on any value. But to defeat this with adaptive partitioning it's enough to do the very first split randomly because subsequent regular split points are unpredictable thereafter (assume f won't be able to remember this point). This isn't implemented here because adversarial f don't occur in practice.

None of the error estimates are reliable. The Clenshaw-Curtis one is a heuristic, and the Monte Carlo integration bound (discussed in the next section) comes with < 100% confidence. For black-box f Monte Carlo is almost always the least accurate, is by far the theoretically safest because f may not be Lipschitz. It's usually best to know what type of f is being integrated and what algorithms work on it. In particular, Clenshaw-Curtis seems to be the algorithm of choice for typical analytic f . But no algorithm can accurately integrate f with a concentrated spike because the integral = ∞ .

Function	CC			CCDoubling			AS			AGLK			Hybr			MC			TrapData		
	Err	Est	Evals	Err	Est	Evals	Err	Est	Evals	Err	Est	Evals	Err	Est	Evals	Err	Est	Evals	Err	Est	Evals
Step	-3.4	-3.6	4097	-3.4	-3.4	4097	-13.7	-13.7	4229	-14.4	-14.4	17407	-14.4	-14.0	17920	-3.6	-3.2	10000000	-7.8	-6.7	10000000
Abs	-7.0	-6.7	4097	-7.0	-6.5	4097	-13.7	-13.7	4361	-15.3	-15.7	1027	-15.3	-15.7	1540	-3.3	-3.4	10000000	-14.4	-14.8	10000000
Lin	-15.7	-15.1	17	-15.7	-15.7	33	-13.7	-13.7	4361	-15.7	-15.7	1027	-15.7	-15.1	17	-3.9	-3.1	10000000	-15.7	-15.7	10000000
Square	-15.7	-14.9	17	-15.7	-15.7	33	-13.7	-13.7	4353	-15.3	-15.7	1027	-15.3	-14.9	17	-3.5	-3.4	10000000	-13.1	-13.1	10000000
Cube	-15.7	-14.8	17	-15.7	-15.7	33	-13.7	-13.7	4345	-15.7	-15.7	1027	-15.7	-14.8	17	-3.9	-3.3	10000000	-15.6	-15.5	10000000
Quad	-15.7	-14.7	17	-15.7	-15.7	33	-13.7	-13.7	8329	-15.7	-15.7	1027	-15.7	-14.7	17	-4.0	-3.5	10000000	-12.8	-12.8	10000000
Exp	-15.7	-14.9	17	-15.7	-15.7	33	-13.9	-13.8	4353	-15.4	-15.7	1027	-15.7	-14.5	17	-4.2	-3.5	10000000	-13.7	-13.7	10000000
SqrtAbs	-5.2	-5.1	4097	-5.2	-4.9	4097	-14.4	-13.7	15189	-14.8	-13.7	18487	-14.8	-13.7	19000	-5.2	-3.7	10000000	-10.3	-11.3	10000000
Runge	-15.1	-13.6	1025	-15.0	-15.0	1025	-15.0	-13.7	25345	-15.2	-13.7	17467	-15.2	-13.7	17980	-3.2	-3.0	10000000	-13.8	-13.3	10000000
Log	-15.7	-14.8	33	-15.7	-15.7	33	-13.7	-13.7	4341	-15.5	-15.7	1027	-15.7	-14.8	33	-4.1	-3.9	10000000	-14.6	-14.6	10000000
XSinXM1	-4.3	-5.6	4097	-4.3	-5.4	4097	-10.3	-11.3	999997	-11.8	-11.0	999997	-11.8	-11.0	999490	-4.0	-3.3	10000000	-9.2	-8.3	10000000
Sin	-15.7	-14.5	17	-15.7	-15.7	33	-13.9	-13.7	4365	-15.7	-15.7	1027	-15.7	-14.5	17	-3.3	-3.2	10000000	-15.7	-15.7	10000000
AbsIntegral	-15.7	-13.4	4097	-15.7	-15.7	33	-13.7	-13.7	4345	-15.4	-15.7	1027	-15.7	-15.7	1540	-4.5	-3.5	10000000	-15.7	-15.7	10000000
Tanh	-15.7	-14.4	33	-15.7	-15.7	33	-15.7	-13.7	4733	-15.7	-14.4	1027	-15.7	-14.1	33	-3.4	-3.2	10000000	-15.7	-15.7	10000000
NormalPDF	-15.7	-13.5	129	-15.7	-15.7	257	-15.1	-13.7	19677	-15.2	-13.7	4657	-15.2	-13.7	4786	-3.6	-2.9	10000000	-15.4	-14.7	10000000
DeltaPDF	-4.4	-3.9	4097	-4.4	-3.7	4097	-14.0	-13.7	4541	-15.7	-15.7	17437	-15.7	-15.7	17950	-3.0	-2.5	10000000	-11.5	-11.0	10000000
F575	-1.4	1.3	4097	-1.4	-1.7	4097	-14.8	-13.7	130457	-14.9	-13.7	38767	-14.9	-13.7	39280	-2.2	-0.9	10000000	-2.0	-1.4	10000000
Average Ranks	2.8	4.5	1.0	2.9	2.6	1.9	4.4	4.5	4.6	1.7	1.4	3.5	1.2	2.5	2.5	6.7	6.9	6.0	3.8	3.6	6

Figure 23.4: Performance comparisons on some f . The hybrid method wins. Doubling Chebyshev loses to generic interpolation. Adaptive Simpson isn't as good as adaptive GLK. Monte Carlo loses for 1D.

A simple way to parallelize or handle memory limits of adaptive methods with large evaluation budget n is to break up $[a, b]$ into k equal pieces and integrate each separately. Each piece gets budget n/k . Error tolerance is harder to distribute, but the errors from adding the integrals will cancel out, so given original tolerance ϵ , using $\min\left(\frac{\epsilon}{\sqrt{k}}, \epsilon_{\text{machine}}\right)$ makes sense. For an error estimate use the conservative \sum estimated piece errors. This or a slight modification is the strategy of less effective recursive integrators that don't use a heap, and ODE solvers use it out of necessity (discussed later in the chapter), but it works well here.

A simple heuristic for singularities is setting them to 0. This is often effective but tends to work well only with adaptive integrators that can adapt to the discontinuity between the 0 and its neighbors.

```

template< typename FUNCTION> struct SingularityWrapper
{
    FUNCTION f;
    mutable int sCount;
    SingularityWrapper(FUNCTION const& theF = FUNCTION()): f(theF), sCount(0) {}

```

```

double operator() (double x) const
{
    double y = f(x);
    if(isinfinite(y)) return y;
    else
    {
        ++sCount;
        return 0;
    }
}

```

From my tests with this heuristic Chebyshev integration fails while adaptive methods (including the hybrid strategy) work well.

Another easy extension is to $\int_{a \leq x < \infty} f$: find min b such that for $x > b$ $f(x)$ is too small in some relative sense, and integrate over $[a, b]$. The “too small” is usually $\int_{b \leq x < \infty} f \leq \epsilon \int_{a \leq x \leq b} f$ for some ϵ ; instead of ∞ use $2b$ per doubling strategy. Computationally, starting from some initial b (picking which needs analytical knowledge), at each new step double b , and add on $\int_{b \leq x \leq 2b} f$ until the next change is too small.

Many such semi-analytic techniques are discussed by Kythe & Schäferkotter (2004) with many examples; they also have some test sets with known solutions.

Integration is well-conditioned absolutely but not relatively (Corless & Fillion 2013; Heath 2018). Even Monte Carlo computes an answer in an absolute uncertainty range. When the answer ≈ 0 , at best can compute more absolute decimal places, and don't gain more relative precision when the answer is still 0. Some cases where this is a problem are called **oscillatory integrals** and are usually handled using analytical pre-processing; see Corless & Fillion (2013) and Kythe & Schäferkotter (2004) for some techniques. Some of the algorithms are recent (Gao & Iserles 2016).

Integration isn't well-conditioned with respect to changes in the range. If most of the integral value is concentrated near a boundary, can have ill-conditioning with respect to the boundary change (Ueberhuber 1997b; Heath 2018).

23.10 Multidimensional Integration

A simple way to evaluate a multidimensional integral is to reduce it to 1D integrals using **Fubini's theorem** (Wikipedia 2017c): Let X, Y be the integration domains that satisfy that using $|f|$ instead leads to a finite answer and some minor technical conditions (i.e., they are σ -finite measure spaces). If $\int_{X \times Y} f$ exists and $< \infty$, then $\int_{X \times Y} f = \int_X (\int_Y f) = \int_Y (\int_X f)$.

This allows evaluating multidimensional integrals recursively using any 1D integration rule. It's hard to partition the number of allowed evaluations better than equally, so use Clenshaw-Curtis for 1D integrator due to its needing few evaluations for high accuracy, which works well with the curse of dimensionality of large D .

Getting a good error estimate is a bit tricky. Can do analysis in exact arithmetic based on backward analysis of the recursion. Assume that:

- Recursive 1D integration over variable i makes average error e_i
- The base case 1D integration is from a_0 to b_0

Then the total error = $\frac{\text{Vol}(X \times Y) \sum e_i}{\prod_{0 \leq i \leq 1} (b_i - a_i)}$. If additionally:

- All e_i are of similar magnitude
- Any e_i is much less than any hypercube side

Then the total error $\approx \frac{\text{Vol}(X \times Y) e_0}{b_0 - a_0}$. So errors at lower levels of recursions matter more. This is easy to

compute and gives a good heuristic estimate. Some implementation tricks:

- Keep average error for each recursion level
- As a base case, use estimated f evaluation error under correct rounding
- The loop index i goes in the reverse direction for convenience

```

struct Cheb1DIntegrator
//ensure getting error estimate
template <typename FUNCTION> pair<double, double> operator() (
    FUNCTION const $\&$  f, double a, double b, int maxEvals)

```

```

    {return integrateCC(f, a, b, maxEvals, min(17, (maxEvals - 1)/2 + 1));}
};

template<typename FUNCTION, typename INTEGRATOR1D = Cheb1DIntegrator>
class RecursiveIntegralFunction
{
    FUNCTION f;
    mutable Vector<double> xBound;
    typedef Vector<pair<double, double>> BOX;
    BOX box;
    mutable Vector<pair<double, long long>>* errors; //copy-proof
    int maxEvalsPerDim; //default for low D power of 2 + 1
    pair<double, double> integrateHelper() const
    {
        INTEGRATOR1D i1D;
        int i = xBound.getSize();
        return i1D(*this, box[i].first, box[i].second, maxEvalsPerDim);
    }
public:
    RecursiveIntegralFunction(BOX const& theBox, int theMaxEvalsPerDim = 33,
        FUNCTION const& theF = FUNCTION()): f(theF), box(theBox), errors(0),
        maxEvalsPerDim(theMaxEvalsPerDim){}
    pair<double, double> integrate() const //the main function
    {
        errors = new Vector<pair<double, long long>>(box.getSize());
        pair<double, double> result = integrateHelper();
        double error = 0;
        for(int i = box.getSize() - 1; i >= 0; --i) error = (box[i].second -
            box[i].first) * (error + (*errors)[i].first/(*errors)[i].second);
        result.second += error;
        delete errors;
        return result;
    }
    double operator()(double x) const //called by INTEGRATOR1D
    {
        xBound.append(x);
        bool recurse = xBound.getSize() < box.getSize();
        pair<double, double> result = recurse ? integrateHelper() :
            make_pair(f(xBound), numeric_limits<double>::epsilon());
        if(!recurse) result.second *= max(1.0, abs(result.first));
        xBound.removeLast();
        int i = xBound.getSize();
        (*errors)[i].first += result.second;
        ++(*errors)[i].second;
        return result.first;
    }
};

```

This code is useful for smooth f for up to $D < 8$ or so. As a black box it can't handle nonrectangular domains because, e.g., setting $f = 0$ outside the wanted custom domain inside a dummy rectangle would cause discontinuities and prevent fast convergence of 1D integration for very smooth f . But if the caller provides some implicit boundary solver, then given currently bound variables can determine bounds on the current variable. This works for arbitrary convex regions, and nonconvex ones have to be broken into convex ones by the caller (not implemented here).

Analytical error bounds are hard get because don't have LIE bound for $D > 1$, so rely on other approaches. This isn't explored further, but for iterated integration in exact arithmetic if every 1D integral is done exactly \forall combination of bound variables, by induction that the result is also exact.

For $D > 1$ deterministic methods have $O(1/n^{1/D})$ convergence, and the integration range needn't be simple. By the CLT, **Monte Carlo integration** has $O(1/\sqrt{n})$ convergence. It also can handle nonrectangular domains using accept-reject sampling. Maintain incremental statistics on the integral to show the error. To calculate the integral given a membership tester:

1. Find a bounding hyperrectangle, and calculate its volume v
2. Generate n random points uniformly inside the hyperrectangle

3. Let $s = \sum$ function values of points \in range

4. Return sv/n

Using $f = 1$ computes the range's volume. For m points \in range, the average function value $= s/m$, and the volume $= vm/n$. The error estimate $= 2$ standard errors.

```
double boxVolume(Vector<pair<double, double>> const& box)
{
    double result = 1;
    for(int i = 0; i < box.getSize(); ++i)
        result *= box[i].second - box[i].first;
    return result;
}

struct InsideTrue{
    bool operator()(Vector<double> const& dummy) const{return true;};
}template<typename TEST, typename FUNCTION> pair<double, double>
MonteCarloIntegrate(Vector<pair<double, double>> const& box, int n,
TEST const& isInside = TEST(), FUNCTION const& f = FUNCTION())
{
    IncrementalStatistics s;
    for(int i = 0; i < n; ++i)
    {
        Vector<double> point(box.getSize());
        for(int j = 0; j < box.getSize(); ++j)
            point[j] = GlobalRNG().uniform(box[j].first, box[j].second);
        if(isInside(point)) s.addValue(f(point));
    }
    double regionVolume = boxVolume(box) * s.n/n;
    return make_pair(regionVolume * s.getMean(), regionVolume * s.error95());
}
```

Though the error of Monte Carlo integration is typically quantified in terms of the CLT, if $|f(x)|$ is bounded by a constant, Hoeffding inequality leads to finite-sample bounds. Otherwise, though variance is constant in n , it may grow quickly with D so that the algorithm isn't useful even for $n \approx 10^9$, though such cases are rare in practice.

Another problem is that accept-reject based on an arbitrarily shape doesn't scale with D . The basic problem is that $\frac{\text{Vol(hypersphere)}}{\text{Vol(the enclosing hypercube)}} \rightarrow 0$ as $D \rightarrow \infty$. Need custom generators for specific domains.

Using hypercube samples from the Sobol sequence (see the “Computational Statistics” chapter for description and analysis, which carries over perfectly to integration because compute an average) usually leads to a much smaller error. The 95% error estimate is heuristic due to nonrandom sampling, but works well (Hahn 2005).

```
template<typename TEST, typename FUNCTION> pair<double, double> SobolIntegrate(
    Vector<pair<double, double>> const& box, int n,
    TEST const& isInside = TEST(), FUNCTION const& f = FUNCTION())
{
    IncrementalStatistics s;
    ScrambledSobolHybrid so(box);
    for(int i = 0; i < n; ++i)
    {
        Vector<double> point = so();
        if(isInside(point)) s.addValue(f(point));
    }
    double regionVolume = boxVolume(box) * s.n/n;
    return make_pair(regionVolume * s.getMean(), regionVolume * s.error95());
}
```

23.11 Function Evaluation

The hardware can do only a fixed subset of operations such as $+$, $-$, \times , and $/$. The rest must be implemented in terms of the above in software or hardware. Special functions such as \sin and \cos are usually evaluated by (Muller 2016):

1. Partitioning the domain into suitable regions

2. Creating \forall region a specific approximation model, usually based on minimax polynomials (see the comments section), equation solving, or special methods
3. To evaluate $f(x)$, find its region, and use the region's model

For elementary functions must pay attention to every bit because for a library implementation it's generally unacceptable to get ≥ 2 bits wrong. Automated proof assistants, which mostly work for straight-line (i.e., no loops) programs, are particularly useful to check this. The evaluation of performance generally depends on x , but is effectively $O(1)$ with a large constant. The algorithms are very specialized, and the majority are unrelated to those of general numerical analysis. E.g., usually derive the approximations in high-precision arithmetic, and make some adjustments for standard precision. Might break up x into exponent and mantissa, and work on each separately. See Muller (2016) and Beebe (2017) for more details. Use good libraries instead of creating own implementations.

Evaluating **elementary functions** such as \cos by library routines generally gives the correctly rounded answer to full precision. For general f , this is impossible due to the **table maker's dilemma**—deciding which way to round may need evaluation to an arbitrary precision. So in some cases only guarantee **faithful rounding**—i.e., to one of the two nearby representations of the exact result. Don't forget that the calculation can be ill-conditioned—faithful rounding only controls backward error.

Some user-defined functions are evaluated by simulations that run for hours. Evaluating a function of D variables takes $O(D + F(D))$ time for some F .

23.12 Estimating Derivatives

Assuming f is differentiable, it's usually easy to calculate f' analytically from its syntax tree representation and impossible if f is a black box. So want to estimate f' from evaluations of f . f' must be well-conditioned

for this to work. E.g., $f(x) = \begin{cases} 0, & \text{if } x=0 \\ x^2 \sin\left(\frac{1}{x}\right), & \text{otherwise} \end{cases}$ is differentiable everywhere but not continuous at 0 and with large oscillations nearby. Effectively the linear condition number $f'(x)$ controls the sensitivity of $f'(x)$.

Can derive the main formulas from Taylor series (Ascher & Greif 2011): Given some small $h > 0$, assuming f has enough continuous derivatives, have:

- **Forward difference:** $f'(x) = \frac{f(x+h) - f(x)}{h} \pm h \max_{x \leq \xi \leq x+h} \frac{|f''|}{2}$
- **Backward difference:** $f'(x) = \frac{f(x) - f(x-h)}{h} \pm h \max_{x-h \leq \xi \leq x} \frac{|f''|}{2}$
- **Central difference:** $f'(x) = \frac{f(x+h) - f(x-h)}{2h} \pm h^2 \max_{x-h \leq \xi \leq x+h} \frac{|f'''|}{6}$
- **Five-point stencil:** $f'(x) = \frac{f(x-2h) + 8f(x+h) - 8f(x-h) - f(x+2h)}{12h} \pm h^4 \max_{x-h \leq \xi \leq x+h} \frac{|f^{(5)}|}{30}$

Can also derive these from polynomial interpolation on a set of h -separated points by calculating the interpolation polynomial, differentiating it analytically, and evaluating the result. With given, unequal-spaced points use barycentric interpolation.

Small h lead to estimation error in all formulas due to subtraction cancellations. Assume that $f(x)$ and all $f(x+ih)$ terms have the same error. In particular, let the backward error $= |x|\epsilon$. Then the forward error $\leq f'(x)|x|\epsilon$. When $h = |x|a$, the cancellation error $\leq \frac{c|f'(x)|\epsilon}{a}$, where c is a small constant—so think of a as defining the fraction of bits by which the estimate is wrong.

Want to minimize both the approximation and estimation error bounds, which happens when they are equal. Folding some constants into c , for the formula where the approximation error depends on k^{th} derivative get $a^*(k) = \epsilon_{\text{factor}}^{1/k}$, where $\epsilon_{\text{factor}} = \frac{c|f'(x)|\epsilon}{|x|^{k-1}||f^{(k)}||}$. For polynomial f the x cancel out, and $\epsilon_{\text{factor}} \approx \frac{ce}{d^{k-1}}$, where d is the degree. In the absence of other knowledge, and in view of that ϵ can be much larger than $\epsilon_{\text{machine}}$, it's reasonable to assume that $\epsilon_{\text{factor}} \approx \epsilon_{\text{machine}}$. Want absolute tolerance for small x , so use $h = \max(1, |x|)\epsilon_{\text{machine}}^{1/k}$.

Backward difference has the same properties as the more natural forward difference, and five-point stencil isn't generally better than central difference (discussed later). This also shows that it's pointless to find better approximation error based on something like local Chebyshev interpolation in $[-h, h]$. So forward and central are the main approaches:

```

template<typename FUNCTION> double estimateDerivativeFD(FUNCTION const& f,
    double x, double fx, double fEFactor = numeric_limits<double>::epsilon())
{
    double h = sqrt(fEFactor) * max(1.0, abs(x));
    return (f(x + h) - fx)/h;
}

template<typename FUNCTION> double estimateDerivativeCD(FUNCTION const& f,
    double x, double fEFactor = numeric_limits<double>::epsilon())
{
    double h = pow(fEFactor, 1.0/3) * max(1.0, abs(x));
    return (f(x + h) - f(x - h))/(2 * h);
}

```

Some things that can go wrong:

- The approximation error is much larger than expected, particularly when f doesn't have k derivatives, and h is large
- The estimation error is much larger than expected, and h is too small
- f isn't differentiable, and at best get garbage, e.g., for $f(x) = |x|$ calculation near $x = 0$ will produce a subgradient $\in [-1, 1]$ depending on the formula, x , and h
- The methods attempt to control absolute and not relative error. So near $x = 0$ may have high relative error, in particular $\text{sign}(f'(x))$ will be ill-conditioned. This could be important in some applications like the secant method for solving equations (discussed later in the chapter).

A particular example of failure is $\sin(x)$ for large x . Due to range reduction, higher-order bits of x will be lost, and only the lower-order bits determine the result, so the absolute backward error is large. Equivalently, the relative linear condition number is large due to large x .

Sometimes need care when taking derivatives near domain boundaries beyond which f is undefined. Here may want to use a hybrid algorithm that first tries central difference and switches to forward or backward if get NaNs, but this isn't done here for simplicity.

Can also use global estimators based on interpolation. They have much smaller estimation errors due to effectively using much larger h , and control approximation error by specialized convergence criteria. In particular, consider Chebyshev and piecewise Chebyshev polynomials.

Function	FD		CD		FPS		Cheb Doubling		Cheb64 Adaptive	
	Err	Evals	Err	Evals	Err	Evals	Err	Evals	Err	Evals
Step	-15.7	2	4.9	2	2.9	4	3.2	4097	-15.7	6175
Abs	-15.7	2	-0.2	2	0.0	4	0.0	4097	-15.4	1105
Lin	-9.9	2	-11.3	2	-12.9	4	-15.7	17	-15.4	1105
Square	-7.7	2	-10.9	2	-12.6	4	-15.7	17	-14.9	1105
Cube	-7.6	2	-10.4	2	-12.9	4	-15.3	17	-14.6	1105
Quad	-7.1	2	-9.8	2	-12.4	4	-15.1	17	-14.2	1105
Exp	-7.8	2	-10.7	2	-12.6	4	-13.1	17	-12.9	1105
Log	-7.8	2	-10.5	2	-12.4	4	-11.9	33	-11.9	1105
XSinXM1	4.0	2	2.5	2	1.6	4	1.8	4097	5.2	1015625
Sin	-7.9	2	-10.8	2	-12.7	4	-13.9	17	-12.0	1105
AbsIntegral	-8.0	2	-5.6	2	-3.6	4	-3.8	4097	-15.1	1105
Tanh	-8.0	2	-10.8	2	-12.6	4	-12.3	33	-12.0	1105
NormalPDF	-8.3	2	-11.0	2	-12.7	4	-12.2	129	-12.2	1105
DeltaPDF	-15.7	2	-11.1	2	-4.88	4	0.0	4097	-12.7	10335
Average Ranks	0.0	1.0	3.8	1.0	2.8	3.0	2.2	4.1	2.3	4.9

Figure 23.5: Some performance comparisons of differentiation methods. Used the same algorithm as for interpolation to estimate errors.

From some experimental and theoretical considerations, central difference seems to be the best method for general use:

- It uses only 2 function evaluations
- The approximation error with the default h isn't too large for f with < 3 derivatives
- When f isn't calculated to full precision, h isn't too small
- The cubic root gives some protection against misestimating c_{factor} by few orders of magnitude

Among the global methods, piecewise Chebyshev interpolation is the winner in general despite not guaranteeing continuous derivatives, though for known analytic f small-degree Chebyshev polynomials will be best.

For given points its best to use barycentric interpolation of degree to up 5 probably based on the closest points to the query point, as long as don't extrapolate (for which the barycentric formula is unstable).

Derivative estimation via finite differences is infinitely ill-conditioned (Corless & Fillion 2013), making it an ill-posed problem, so no such algorithm can estimate derivatives with high precision in the worst case. It's simply impossible to pick the right h —e.g., when f is computed with error $\approx h$ and not the assumed full precision, all accuracy is lost.

To estimate the gradient of a D -variable function, estimate all the partial derivatives. This takes $O(D^2 + DF(D))$ time. Evaluating the analytical gradient takes $O(D + G(D))$ time, and $G(D) = O(D)$ if \forall variable the partial derivative depends on $O(1)$ other variables. The central difference formula uses $2D$ evaluations.

To allow code reuse, it's convenient to create a scaled direction function that can be passed to central difference or any other algorithm as is. A simple way to scale in a unit direction u is to make the scale match the gradient h_i values when u is an axis direction and use the L_2 norm to interpolate these in other directions.

```
double findDirectionScale(Vector<double> const& x, Vector<double> u)
{
    for(int i = 0; i < x.getSize(); ++i) u[i] *= max(1.0, abs(x[i]));
    return norm(u);
}
```

The trick for creating a scaled direction function is to start it at step value $s = s_{\text{scaled}}$ and not 0.

```
template<typename FUNCTION> class ScaledDirectionFunction
{
    FUNCTION f;
    Vector<double> x, d;
    double scale;
public:
    ScaledDirectionFunction(FUNCTION const& theF, Vector<double> const& theX,
                           Vector<double> const& theD): f(theF), x(theX), d(theD),
                           scale(findDirectionScale(x, d * (1/norm(d)))) {assert(norm(d) > 0);}
    double getS0() const{return scale;} // returns x[i] if x is axis vector
    double operator()(double s) const{return f(x + d * (s - getS0()));}
};
```

Then to calculate gradient need to arrange axis vectors:

```
template<typename FUNCTION> Vector<double> estimateGradientCD(
    Vector<double> const& x, FUNCTION const& f,
    double fFFactor = numeric_limits<double>::epsilon())
{
    int D = x.getSize();
    Vector<double> result(D), d(D);
    for(int i = 0; i < D; ++i)
    {
        d[i] = 1;
        ScaledDirectionFunction<FUNCTION> df(f, x, d);
        result[i] = estimateDerivativeCD(df, df.getS0(), fFFactor);
        d[i] = 0;
    }
    return result;
}
```

Can estimate the Jacobian in the same way, but, because it's a vector-valued function, a wrapper isn't used because it won't be reusable.

```
template<typename FUNCTION> Matrix<double> estimateJacobianCD(FUNCTION const&
    f, Vector<double> x, double fFFactor = numeric_limits<double>::epsilon())
{
    int n = x.getSize();
    Matrix<double> J(n, n);
    Vector<double> dx(n);
    double temp = pow(fFFactor, 1.0/3);
    for(int c = 0; c < n; ++c)
    {
        double xc = x[c], h = max(1.0, abs(xc)) * temp;
```

```

x[c] += h;
Vector<double> df = f(x);
x[c] = xc - h;
df -= f(x);
x[c] = xc;
for(int r = 0; r < n; ++r) J(r, c) = df[r]/(2 * h);
}
return J;
}

```

Estimating directional derivatives in a unit direction u is even simpler. The implementation calculates $\nabla f d$, where d might not be of unit norm.

```

template<typename FUNCTION> double estimateDirectionalDerivativeCD(
    Vector<double> const& x, FUNCTION const& f, Vector<double> const& d,
    double fEFactor = numeric_limits<double>::epsilon())
{//estimates grad 'd' if d not unit
    ScaledDirectionFunction<FUNCTION> df(f, x, d);
    return estimateDerivativeCD(df, df.getS0(), fEFactor);
}

```

Often an argument is made for using forward difference to halve function evaluations, but even when low accuracy is OK, it's much more robust to use a more accurate method (i.e., you don't want a less accurate answer faster). I tried both for solving systems of nonlinear equations (discussed later in the chapter), and though forward difference leads to fewer evaluations on many f , for many others using central difference Jacobians improves accuracy and reduces the total number of f evaluations.

A more complicated task is estimating higher-order derivatives. For the 2nd can use a combination of several Taylor series around x . This leads to the **2nd-order central difference formula**

$$f''(x) = \frac{f(x+h) - f(x) + f(x-h)}{h^2} \pm h^2 \max_{x-h \leq \xi \leq x+h} \frac{|f^{(4)}(\xi)|}{12}$$

(Ascher & Greif 2011). Among the bigger problems is that the cancellation error is now $O(1/h^2)$. Optimal $h = \max(1, |x|) \epsilon_{\text{machine}}^{1/4}$.

```

template<typename FUNCTION> double estimate2ndDerivativeCD(FUNCTION const& f,
    double x, double fx = numeric_limits<double>::quiet_NaN(),
    double fEFactor = numeric_limits<double>::epsilon())
{
    if(!isfinite(fx)) fx = f(x);
    double h = pow(fEFactor, 1.0/4) * max(1.0, abs(x));
    return (f(x + h) - 2 * fx + f(x - h)) / (h * h);
}

```

To estimate the Hessian can use the relationship that $\text{Hessian} = \text{Jacobian}(\nabla f)^T$ (Wikipedia 2017g). Because of numerical errors this may lose symmetry, so recover by averaging symmetric entries (Nodecal & Wright 2006).

```

template<typename GRADIENT> Matrix<double> estimateHessianFromGradientCD(
    Vector<double> x, GRADIENT const& g,
    double fEFactor = numeric_limits<double>::epsilon())
{
    Matrix<double> HT = estimateJacobianCD(g, x, fEFactor);
    return 0.5 * (HT + HT.transpose()); //ensure symmetry
}

```

With numerical gradients use $\epsilon_{\text{factor}} = \epsilon_{\text{machine}}^{2/3}$, which assumes their calculation by central difference. But here the method will do more evaluations than necessary, compared to a direct finite difference of 2nd order (see Press et al. 2007 for some formulas). Still, get a simple generic solution that works in both cases.

23.13 Solving Nonlinear Equations and Systems

In exact arithmetic, an equation or a system of equations in the most general form is given by some function, and want to find x^* such that $f(x^*) = 0$. With limited-precision arithmetic, must settle for less to have verifiable roots:

- For D -dimensional f and x , want to find x^* and ϵ with minimal $\|\epsilon\|_\infty$ such that $\forall i \in [0, D-1] f_i(x^*) < 0$, and $f_i(x^* + \epsilon) > 0$. Then have a root \in box $[x^*, x^* + \epsilon]$, but this is practical only for $D = 1$.
- Find an **ϵ -root** such that $\|f(x^*)\|_\infty < \epsilon$. This is backward error that doesn't guarantee that x^* is near a root, but is useful in practice if $\epsilon <$ some small tolerance.

- Given an exact root x^* , measure the error in the answer x by $\|x - x^*\|_\infty$.

Assume that f -values and x have the same D . Solving linear equations makes sense only for square matrices—same for nonlinear systems because over- or under-parametrized systems most likely have respectively no or too many solutions. In such cases, as with linear equations, usually solve a least-squares minimization problem instead.

Indeed, it may seem that the simplest solution is to reduce to optimization by solving $\min \|f(x)\|_2$ using optimization methods in the next chapter. If the solution is 0, it's a root. But conversion to minimization can create artificial local minima that don't correspond to roots, so usually only try this as a last resort when specialized methods fail.

For a black-box f any of the following can happen:

- No solutions
- One solution
- Finitely many solutions
- Infinitely many solutions in a bounded or an unbounded region
- Some number of continuous regions of solutions

So in the worst case, even if a problem has a solution, it can be arbitrarily ill-conditioned because even a minor deviation from a similar problem can make a difference between having a solution and not.

In particular, in 1D with one root have ill-conditioning if f' is very small near the root x^* . This is because root-finding effectively computes f^{-1} , and $(f^{-1})' = \frac{1}{f'}$. Intuitively, x near x^* have nearly the same f -value and can be reasonably selected as the answer—e.g., consider $f(x) = x^{10}$. For $D > 1$ derivatives are given by the $J(f(x))$, where J is the Jacobian, and having a small entry in column c row r means that finding $x^*_c[r]$ is ill-conditioned.

In 1D, if have an interval where f -values at the endpoints have different sign, and f is continuous, the **bisection method** is robust. While the current interval < the previous one, the algorithm replaces *right/left* with *middle* if $f(middle)$ and $f(left)$ have different/same signs.

```
bool haveDifferentSign(double a, double b) {return (a < 0) != (b < 0);}
template<typename FUNCTION> pair<double, double> solveFor0(FUNCTION const& f,
    double xLeft, double xRight, double relAbsXPrecision = highPrecEps)
{
    double yLeft = f(xLeft), xMiddle = xLeft;
    assert(xRight > xLeft && haveDifferentSign(yLeft, f(xRight)));
    while(isELess(xLeft, xRight, relAbsXPrecision))
        //below formula more robust than simple average
        xMiddle = xLeft + (xRight - xLeft)/2;
        double yMiddle = f(xMiddle);
        if(haveDifferentSign(yLeft, yMiddle)) xRight = xMiddle;
        else
        {
            xLeft = xMiddle;
            yLeft = yMiddle;
        }
    //best guess and worst-case error
    return make_pair(xLeft + (xRight - xLeft)/2, xRight - xLeft);
}
```

The convergence is linear, $O(\lg(1/\text{absolute precision}))$, and the algorithm terminates in all cases. But for stochastic f the starting endpoints must have different signs with extremely high probability to avoid assertion failure in a Las Vegas sense. For continuous f a root exists by the mean value theorem. Without continuity it can converge to a value that isn't a root—e.g., to 0 for $f(x) = 1/x$ with starting interval = $[-1, 1]$.

Don't be greedy with accuracy—the error of evaluating f is the limit, and asking for more wastes evaluations and gives an unrealistic error estimate based on the found interval which needn't even have the true root due to working with noise. The implementation is very robust with all valid input because, e.g., $xLeft + xRight$ can overflow, but guard against this with a better formula.

A simple method that works without change in any D is **fixed point iteration**:

1. Form $g(x) = f(x) + x$
2. Until convergence or exceeding evaluation budget
3. $x = g(x)$
4. Return x as root of f if converged

It's only useful conceptually because the convergence condition is very restrictive (Süli & Mayers 2003):

need $\|J(x)\|_\infty < 1$ on the entire domain of action. E.g., adding a constant such as 100 to f can break the method.

Newton's method leads to a better iteration. In 1D, the idea is to go down in the $-f'(x)$ direction exactly until $f = 0$, i.e., take a step $\Delta x = \frac{-f(x)}{f'(x)}$. Then repeat until convergence, i.e., step again to correct when over- or under-stepping.

For $D > 1$ the only difference is that use the $J(x)$, which must be nonsingular. Then $\Delta x = -J(x)^{-1}f(x)$. The justification is also similar—if f is differentiable, by multivariate Taylor series nearby values are given by a linear model, i.e., $f(x + \Delta x) = f(x) + J(x)\Delta x + O(\|\Delta x\|^2)$, which solve for 0. So get the algorithm (simplified):

1. Start with a user-provided guess $x = x_0$
2. Until convergence
3. $x += -J(x)^{-1}f(x)$

It converges quadratically as long as $J(\text{root})$ isn't singular and some other conditions are satisfied (Süli & Mayers 2003). But have issues:

- When $J(x)$ is singular (e.g., in 1D $f(x) = x^3 + 1$ (with root $x = -1$) at $x = 0$ where $f'(x) = 0$), the method breaks down.
- The caller must supply J , which is unavailable for black-box f and inconvenient when know f .
- Need $O(D^3)$ per iteration to solve for Δx , which doesn't scale.
- In 1D, when a root is multiple only have linear convergence (Dahlquist & Björck 2008). This is related to ill-conditioning— $f(\text{multiple root}) = 0$.

Essentially Newton's method is useful only for specific problems, where it's proven to converge, and the formulas are simplified for the function and its derivative.

In 1D, if f is differentiable, then unless $f'(x) = 0$, f is locally linear, and $-sf'(x)$ is a **descent direction**, i.e., for small enough $s \leq 1$, $\Delta x = -sf'(x)$ gives decrease in f -value. For $D > 1$ and single f , $-\nabla(x)$ is a descent direction, but it's not unique and $-d$ such that $d\nabla(x) < 0$ also gives descent. For equation solving want decrease to 0, which is usually impossible in one step even with Newton's method.

For multidimensional f each component may want to take x in a different direction. But if $J(x)$ isn't singular, the Newton direction is descent in a sense that $\|f\|$ decreases in any norm: given small enough s , $f(x + \Delta x) \approx f(x) + J(x)\Delta x = f(x) + J(x)(-sJ(x)^{-1}f(x)) = (1-s)f(x)$. For computational efficiency, accept first s that gives a fraction of this decrease, which is called **backtracking** (or **damping**). Start with $s = 1$, and halve s until in the chosen norm (here ∞) $\|f(x + s\Delta x)\| \leq (1-as)\|f(x)\|$ for some small a (10^{-4} is typical). If the final s is bounded above 0, which must be the case if J is far from singular on the solution domain, have **sufficient descent**, i.e., a fixed fraction of descent per step. Then get uniform convergence to 0 in the norm used by backtracking.

```
double normInf(double x) { return abs(x); }
template<typename FUNCTION, typename X> bool equationBacktrack(
    FUNCTION const& f, X& x, X& fx, int& maxEvals, X const& dx, double xEps)
{
    bool failed = true;
    for(double s = 1;
        maxEvals > 0 && normInf(dx) * s > xEps * (1 + normInf(x)); s /= 2)
    {
        X fNew = f(x + dx * s);
        --maxEvals;
        if(normInf(fNew) <= (1 - 0.0001 * s) * normInf(fx))
        {
            failed = false;
            x += dx * s;
            fx = fNew;
            break;
        }
    }
    return failed;
}
```

Practical algorithms remove the need for derivatives by approximating them. In 1D **secant method** is Newton's method except estimate f' by finite difference based on the last two evaluations. For the first step use central difference. For pure secant method without backtracking have a local convergence theorem

(Dahlquist & Björck 2008; they also derive the constant): If in some neighborhood of root x^* f has two continuous derivatives, $f'(x^*) \neq 0$, and x_0 and x_{-1} are close enough to x^* , converge with order $(1 + \sqrt{5})/2$. Also see Trangenstein (2018a) for a detailed analysis of conditioning and convergence of many 1D equation solution methods. Trangenstein (2018b) does the same for $D > 1$.

For $D > 1$, a simple extension is to estimate J using finite differences at every step. But want to be more efficient and estimate using existing function evaluations. A particular generalization is **Broyden's method**. The idea is that after an update J must be such that $J\Delta x = \Delta f$. Among such J pick the one minimizing ΔJ in Frobenius norm, which has some nice properties here (see Dennis & Schnabel 1996). This leads to rank-1 update $J += u \otimes v$, where $u = \frac{\Delta f - J\Delta x}{\|\Delta x\|}$, and $v = \frac{\Delta x}{\|\Delta x\|}$. Estimate the initial J using finite differences, and update it using QR decomposition.

When J is singular (or near-singular), the problem is giving no descent direction that works for all function components. A reasonable heuristic, that applies to both Broyden and secant, is to take a random step drawn from the normal(0, the size of the last successful full step) distribution. This is a minor tweak to protect against singularities, and isn't standard in the literature.

```
template<typename X> struct BroydenSecant
{//for secant
    static int getD(double dummy){return 1;}
    static double generateUnitStep(double dummy, double infNormSize)
        {return GlobalRNG().normal(0, infNormSize);}
    class InverseOperator
    {
        double b;
    public:
        template<typename FUNCTION> InverseOperator(FUNCTION const& f,
            double x): b(estimateDerivativeCD(f, x)){}
        void addUpdate(double df, double dx){b = df/dx;}
        double operator*(double fx)const{return fx/b;}
    };
};

template<> struct BroydenSecant<Vector<double> >
{//for Broyden
    typedef Vector<double> X;
    static int getD(X const& x){return x.getSize();}
    static X generateUnitStep(X const& x, double infNormSize)
    {
        X dx(getD(x));
        for(int j = 0; j < dx.getSize(); ++j)
            dx[j] = GlobalRNG().normal(0, infNormSize);
        return dx;
    }
    class InverseOperator
    {
        QRDecomposition qr;
    public:
        template<typename FUNCTION> InverseOperator(FUNCTION const& f,
            X const& x): qr(estimateJacobianCD(f, x)){}
        void addUpdate(X const& df, X dx)
        {
            double ndx2 = norm(dx);
            dx *= (1/ndx2);
            qr.rank1Update(df * (1/ndx2) - qr * dx, dx);
        }
        X operator*(X const& fx)const{return qr.solve(fx);}
    };
};
```

The main algorithm is the same for both Broyden and secant, and the implementation introduces some further tricks:

- The current J estimate may be inaccurate enough to not give descent. So when backtracking fails, reinitialize it using finite differences.
- If reinitializing fails to give descent, try the random steps. If a random step gives descent, reinitialize

again in the hope that it's no longer singular.

A reasonable way to estimate the error of the result seems to be using the size of the last successful full step, i.e., before backtracking is applied. This estimate is somewhat conservative, particularly when a large step lands into near-0 exactly, but seems robust against too small successful steps and running out of evaluations, etc.

```

template<typename FUNCTION, typename X> bool equationTryRandomStep(
    FUNCTION const& f, X& x, X& fx, double stepNorm)
{
    X dx = BroydenSecant<X>::generateUnitStep(x, stepNorm), fNew = f(x + dx);
    bool improved = normInf(fNew) < normInf(fx);
    if(improved)
    {
        x += dx;
        fx = fNew;
    }
    return !improved;
}

template<typename FUNCTION, typename X> pair<X, double> solveBroyden(
    FUNCTION const& f, X const& x0, double xEps = highPrecEps,
    int maxEvals = 1000)
{
    int D = BroydenSecant<X>::getD(x0), failCount = 0;
    X x = x0, fx = f(x);
    assert(D == BroydenSecant<X>::getD(fx) && maxEvals >= 2 * D + 1);
    typedef typename BroydenSecant<X>::InverseOperator BIO;
    BIO B(f, x);
    maxEvals -= 2 * D + 1;
    double lastGoodNorm = 1, xError = numeric_limits<double>::infinity();
    if(!isfinite(normInf(fx))) return make_pair(x, xError);
    while(maxEvals > 0)
    {
        if(failCount > 1)
            //after 2nd fail try random step
            --maxEvals;
        if(!equationTryRandomStep(f, x, fx, lastGoodNorm))
        {
            assert(normInf(f(x)) <= normInf(f(x0)));
            if(maxEvals >= 2 * D + 1) //need enough evals for next step
            {
                B = BIO(f, x);
                maxEvals -= 2 * D;
                failCount = 0; //back to normal
            } //else keep making random steps
            continue;
        }
    }
    X dx = B * -fx, oldFx = fx, oldX = x;
    double ndx = normInf(dx);
    if(!isfinite(ndx))
        //probably singular, either after bad update or reestimation
        ++failCount;
    continue;
}
if(ndx < xEps * (1 + normInf(x))) break; //full step too small
if(!equationBacktrack(f, x, fx, maxEvals, dx, xEps))
{
    assert(normInf(f(x)) <= normInf(f(x0)));
    xError = lastGoodNorm = ndx; //last successful full step
    failCount = 0;
}
else ++failCount;
if(failCount == 1) //after first fail reestimate

```

```

    if(maxEvals >= 2 * D + 1) //need enough evals for next step
    {
        B = BIO(f, x);
        maxEvals -= 2 * D;
    }
    else ++failCount; //if cant do steps
    else B.addUpdate(fx - oldFx, x - oldX);
}
return make_pair(x, xError);
}//for type safety such such int w/o use wrapper
template<typename FUNCTION> pair<double, double> solveSecant(FUNCTION const&
f, double const& x0, double xEps = highPrecEps, int maxEvals = 1000)
{return solveBroyden(f, x0, xEps, maxEvals);}

```

In exact arithmetic the basic Broyden's method (without backtracking, reestimation, or random steps) converges superlinearly under some conditions (Nocedal & Wright 2006), but unlike for the 1D secant method the convergence rate is unknown (Dennis & Schnabel 1996). Some further aspects of Broyden's method are surveyed in (Martinez 2001).

For large D the $O(D^3)$ runtime for the first iteration and the $O(D^2)$ space aren't scalable. So want a limited-memory implementation, like L-BFGS for optimization (see the "Numerical Optimization" chapter). To make the Broyden formula computationally friendly, apply Sherman-Morrison formula to $B = J^{-1}$ to get rank-1 update with $B_{i+1} = B_i + u_i \otimes v_i$, where $u_i = \frac{\Delta x_i - B_i \Delta f_i}{\Delta x_i B_i \Delta f_i}$, and $v_i = \Delta x_i B_i$ (Dennis & Schnabel 1996). Can

choose $B_0 = I$. This doesn't have the $O(D^3)$ runtime for the first iteration, so you may wonder why the QR implementation is used at all. The problem is that using $B_0 = I$ is a bad choice when use backtracking because the resulting direction needn't be descent; in fact any permutation of I is as likely as any other to be a better approximation.

A limited solution is to use some form of preconditioning—use m iterations for $m < D$, in each calculating a forward differences of a new random component of x . For $m < D$ is this almost equivalent to estimating J , except that the "prior" $B_0 = I$ is hardly agnostic and influences the result. This seems to work for a limited-memory implementation for both initialization and reestimation.

So $B_{i+1} = B_0 + \sum u_j \otimes v_j$. As with L-BFGS, the limited memory implementation stores m most recent Δx_j and Δf_j for $0 \leq j < m$, and forms B_m from them. The trick is calculating $B_m x$ without forming B_m explicitly. Given that $B_m = I + \sum u_j \otimes v_j$, use dynamic programming. Consider also more clumsy recursion—intuitively, the latter preserves type by working with matrix B implicitly, and the former works with components of x directly. Maintain not only Bx but also $B_i \Delta f_i$ and $\Delta x_i B_i$ for calculating u_j and v_j . This takes $O(m^2 D)$ time and $O(mD)$ space. So represent B_m as an operator instead of a matrix.

```

class BroydenLMIInverseOperator
{
    typedef Vector<double> X;
    int m;
    Queue<pair<X, X>> updates;
    struct Result
    {
        X Bfx;
        Vector<X> Bdfs, dxBs;
    };
public:
    BroydenLMIInverseOperator(int theM) : m(theM) {}
    void addUpdate(X const& df, X const& dx)
    {
        if(updates.getSize() == m) updates.pop();
        updates.push(make_pair(df, dx));
    }
    X operator*(X const& fx) const
    {
        int n = updates.getSize();
        X Bfx = fx; //base case identity
        Vector<X> Bdfs(n), dxBs(n);
        for(int i = 0; i < n; ++i)
        {
            X Bdfi = df;
            X dxi = dx;
            for(int j = i + 1; j < n; ++j)
            {
                Bdfi += updates[j].first * updates[j].second;
                dxi += updates[j].second;
            }
            Bdfs[i] = Bdfi;
            dxBs[i] = dxi;
        }
        Bfx = fx;
        for(int i = 0; i < n; ++i)
        {
            Bfx += Bdfs[i] * dxBs[i];
        }
    }
}

```

```

        Bdfs[i] = updates[i].first;
        dxBs[i] = updates[i].second;
    }
    for(int i = 0; i < n; ++i)
    {
        X u = (updates[i].second - Bdfs[i]) *
            (1/dotProduct(updates[i].second, Bdfs[i]));
        if(!isfinite(normInf(u))) continue; //guard against div by 0
        Bfx += outerProductMultLeft(u, dxBs[i], fx);
        for(int j = i + 1; j < n; ++j)
        {
            Bdfs[j] += outerProductMultLeft(u, dxBs[i], updates[j].first);
            dxBs[j] += outerProductMultRight(u, dxBs[i],
                updates[j].second);
        }
    }
    return Bfx;
}
}

```

The limited-memory approximation can no longer be assumed to give a descent direction, don't have a way to reestimate, and backtracking may not work. But updates that contribute a bad Jacobian estimate will be removed after m steps, so no need for random steps either.

It may seem that the next best thing is to use backtracking and random steps to enforce monotonic decrease of $\|f(x)\|_\infty$, but this does poorly based on my experiments. So a reasonable goal is to prevent single bad steps. It seems best to use backtracking with step decrease by a factor of 10 to prevent $\|f(x)\|_\infty$ from increasing by \geq that factor. This is a heuristic that works slightly better than not using it. Some ideas to support it:

- Bad steps seem easy to recover from because the derivative contribution from a bad step is likely to result in reversing most of that step. So don't want to waste evaluations preventing something that will fix itself. But keep the best solution found so far in case have divergence close to the solution.
- The method seems to need to explore to be effective for large D , and decreasing steps by a factor of 10 is cheap in evaluations and seems to give more exploration than maybe using Δx that would be appropriate for a finite difference estimate of the directional derivative.

```

template<typename FUNCTION> pair<Vector<double>, double> solveLMBroyden(
    FUNCTION const& f, Vector<double> const& x0, double xEps = highPrecEps,
    int maxEvals = 1000, int m = 30)
{
    int D = x0.getSize();
    BroydenLMInverseOperator B(m);
    Vector<double> x = x0, fx = f(x);
    double s = 1, xError = numeric_limits<double>::infinity();
    while(maxEvals-- > 0)
    {
        Vector<double> dx = B * -fx * s;
        double ndx = normInf(dx)/s; //norm of full step
        //something wrong or step too small
        if(!isfinite(ndx) || ndx < xEps * (1 + normInf(x))) break;
        Vector<double> fNew = f(x + dx);
        if(normInf(fNew) > 10 * normInf(fx)) s /= 10;
        else
        {
            B.addUpdate(fNew - fx, dx);
            fx = fNew;
            x += dx;
            xError = ndx; //use last full step
            s = 1; //back to normal step
        }
    }
    return make_pair(x, xError);
}
template<typename FUNCTION> pair<Vector<double>, double> solveLMBroyden(

```

```

FUNCTION const< FUNCTION> solveBroydenHybrid(
    FUNCTION const< double> f, Vector<double> const< x0, double xEps = highPrecEps,
    int maxEvals = 1000, int m = 30)
{
    int D = x0.getSize();
    BroydenLMIInverseOperator B(m);
    Vector<double> x = x0, fx = f(x), xBest = x, fBest = fx;
    double s = 1, xError = numeric_limits<double>::infinity(), eBest = xError;
    while (maxEvals-- > 0)
    {
        Vector<double> dx = B * -fx * s;
        double ndx = normInf(dx)/s; // norm of full step
        // something wrong or step too small
        if (!isfinite(ndx) || ndx < xEps * (1 + normInf(x))) break;
        Vector<double> fNew = f(x + dx);
        if (normInf(fNew) > 10 * normInf(fx)) s /= 10;
        else
        {
            B.addUpdate(fNew - fx, dx);
            fx = fNew;
            x += dx;
            xError = ndx; // use last full step
            s = 1; // back to normal step
            if (normInf(fx) < normInf(fBest))
            {
                xBest = x;
                fBest = fx;
                eBest = xError;
            }
        }
    }
    return make_pair(xBest, eBest);
}

```

Experiments in van de Rotten (2003) suggest that larger values of m work better, need $m \geq 3$ in all cases, and 20 seems safe, but here use 30 for extra safety. The limited-memory algorithm, though useful, has flaws:

- No provable convergence
- No clear guidance for choosing m —unlike for L-BFGS, $m = 3$ isn't enough for many problems
- Potential linear dependence of m value of Δx_j questions this choice of basis—several algorithms in van de Rotten (2003) work in different bases, but need more work to pick the best one
- On my tests the algorithm often doesn't converge unlike the full-memory version—finding a descent direction isn't guaranteed

So the limited-memory algorithm makes sense only as heuristic part of a hybrid global solver (discussed later in the chapter). Here choose between the QR and it based on whether $D < 200$. For local solving should only use QR, or the more expensive hybrid.

```

template<typename FUNCTION> pair<Vector<double>, double> solveBroydenHybrid(
    FUNCTION const< double> f, Vector<double> const< x0, double xEps = highPrecEps,
    int maxEvals = 1000, int changeD = 200)
{
    return x0.getSize() > changeD ? solveLMBroyden(f, x0, xEps, maxEvals) :
        solveBroyden(f, x0, xEps, maxEvals);
}

```

For $D > 1$, having a single function with 0 gradient will cause singularity of J . This seems to expose a fundamental flaw in the presented local algorithms, which may be "fixed" by trying to step into a solution of minimum 2-norm. But this is equivalent to reducing to a minimization problem.

A single equation can create many other problems, such as skewing the norms due to lack of scale or NaNs. Black-box- f algorithms can't handle such issues and assume they don't happen. The variables should have about the same scale because otherwise $\|x\|$ is meaningless. Some implementations (Dennis & Schnabel 1996) heuristically scale implicitly, but here scaling is left to the user. Stiff ODE solvers (discussed later in the chapter) ignore this because the inputs are assumed to be well-scaled already.

The best attainable accuracy is ultimately limited by the precision of f . Given any x and Δx on the level of

$\epsilon_{\text{machine}}$, the values $f(x + \Delta x)$ needn't preserve analytic properties of f such as convexity or monotonicity. So methods that reach high accuracy and try steps of such small magnitude are effectively making random steps, and need to be careful about convergence criteria and not ask for precision that could exhaust the evaluation limit pointlessly.

To find complex single-variable equation roots, a simple method is to convert to a 2D problem (Süli & Mayers 2003). Given $z_{\text{output}} = f(z)$ for $z = x + iy$, create g such that $(x_{\text{output}}, y_{\text{output}}) = g(x, y)$ is evaluated through f .

So far the presented methods assumed some information about f , i.e., either having a sign-change interval and continuity or nonsingular J which gave sufficient descent. But need some heuristic sampling to get these algorithms started for black-box f .

Often want to get bounds from a guess point in 1D. **Exponential search**, also called **bracketing**, steps from a guess point by some distance d , doubling d after every step, until some termination condition. Can proceed in the $+$ direction, in the $-$, or in both. Usually do both because believe in the guess—first step by d , then by $-d$, then by $2d$, etc. The one-way options are good for domain-specific limits such as 0 to something.

Theorem: Two- and one-sided exponential search converges if the domain where f has the same sign as $f(x_0)$ is bounded in any search direction. In particular, two-sided exponential search converges if $g = \text{sign}(f)$ is monotone. Proof: bounded domain will eventually be stepped over; for monotone g meet the bounded domain condition $\forall x_0$ and d . \square This happens, e.g., for numerical inversion of CDF function for random variate generation by the inverse method (see the “Random Number Generation” chapter). But the algorithm must not quit too soon, which is possible in any implementation.

If these don't hold, may still be able to find a sign change by accident. So use small, scaled $d = 0.001 \max(1, |x_0|)$ as default, and allow up to 30 doublings, which seems reasonable.

```
template<typename FUNCTION> pair<double, double> find1SidedInterval0(
    FUNCTION const& f, double x0 = 0, double d = 0.001, int maxEvals = 30)
{
    assert(maxEvals >= 2 && isfinite(x0 + d) && d != 0 &&
        !isEEqual(x0, x0 + d));
    for(double xLast = x0, f0 = f(x0); --maxEvals > 0; d *= 2)
    {
        double xNext = x0 + d, fNext = f(xNext);
        if(!isfinite(xNext) || isnan(fNext)) break;
        if(haveDifferentSign(f0, fNext))
            return d < 0 ? make_pair(xNext, xLast) : make_pair(xLast, xNext);
        xLast = xNext;
    }
    return make_pair(numeric_limits<double>::quiet_NaN(),
        numeric_limits<double>::quiet_NaN());
}

template<typename FUNCTION> pair<double, double> findInterval0(
    FUNCTION const& f, double x0, double d, int maxEvals)
{
    assert(maxEvals >= 2 && isfinite(x0 + d) && isELess(x0, x0 + d));
    for(double xLast = x0, f0 = f(x0); --maxEvals > 0; d = -d)
    {
        double xNext = x0 + d, fNext = f(xNext);
        if(!isfinite(xNext) || isnan(fNext)) break;
        if(haveDifferentSign(f0, fNext)) return d < 0 ?
            make_pair(xNext, x0 - (xLast - x0)) : make_pair(xLast, xNext);
        if(d < 0)
        {
            xLast = x0 - d;
            d *= 2;
        }
    }
    return make_pair(numeric_limits<double>::quiet_NaN(),
        numeric_limits<double>::quiet_NaN());
}

template<typename FUNCTION> pair<double, double> exponentialSearch(
    FUNCTION const& f, double x0 = 0, double step = 0.001,
    double xERelAbs = highPrecEps, int maxExpEvals = 60)
```

```

pair<double, double> i0 = findInterval0(f, x0, step * max(1.0, abs(x0)), maxExpEvals);
    return !isnan(i0.first) ? solveFor0(f, i0.first, i0.second, xERelAbs) : i0;
}

template<typename FUNCTION> pair<double, double> exponentialSearch1Sided(
    FUNCTION const& f, double x0 = 0, double step = 0.001, //negate for left
    double xERelAbs = highPrecEps, int maxExpEvals = 60)
{
    pair<double, double> i0 = find1SidedInterval0(f, x0,
        step * max(1.0, abs(x0)), maxExpEvals);
    return !isnan(i0.first) ? solveFor0(f, i0.first, i0.second, xERelAbs) : i0;
}

```

As implemented, one- and two-sided bracketing is guaranteed to find a valid interval or signals an error with exact arithmetic and deterministic f . So under the same conditions exponential search is guaranteed to not call binary search with a bad interval (that causes assertion failure), except when f is randomized, in which case nothing can be done. But with inexact arithmetic can run into issues—e.g., f can have one sign during bracketing and another during binary search due to ill-conditioned nature of sign evaluation near 0, for the same reasons that repeated evaluation of f needn't produce the same value due to register allocation. It's unclear how to best solve it—perhaps only insist on sign determination with some ϵ buffer, and consider a value closer to that as a solution. But, e.g., don't have a general way to pick safe ϵ , and in practice this is rarely an issue.

Unlike binary search, bracketing isn't guaranteed to work for continuous f because can jump over a region where f changes sign twice or into a NaN domain such as $\log(-0.001)$. Finding a sign-change interval where f is continuous guarantees a solution. E.g., $f(x)=x^2$ has a root but no sign-change intervals, and $f(x)=x^2-\epsilon$ has a very small-sign change interval that's unlikely to be found. Random search, where sample until get a change of sign, is another option, but the secant method seems more useful.

When don't know a good starting point, must use some global strategy. Unlike for global optimization (see the “Numerical Optimization” chapter), the global landscape structure for equation solving gives limited information about where the solution might be because ∞ -norm mostly pays attention to a single variable. So random sampling would be the best exploration strategy, at least theoretically.

```

template<typename FUNCTION> pair<double, double> solveSecantGlobal(
    FUNCTION const& f, double xEps = highPrecEps, int nSamples = 100)
{
    assert(nSamples > 0 && xEps >= numeric_limits<double>::epsilon());
    pair<double, double> best;
    double nBestfx;
    for(int i = 0; i < nSamples; ++i)
    {
        double x = GlobalRNG().Levy() * GlobalRNG().sign();
        pair<double, double> next = solveBroyden(f, x, xEps);
        double nNextfx = normInf(f(next.first));
        if(i == 0 || nNextfx < nBestfx)
        {
            best = next;
            nBestfx = nNextfx;
        }
    }
    return best;
}

```

Function	ExpSearch		Secant		SecantGlobal	
	Err	Evals	Err	Evals	Err	Evals
Linear	-13.24	73	-10.45	5	-9.77	600
QuadE	-15.65	60	-10.24	18	-10.83	2007
Poly6	-15.65	60	-13.04	233	-13.05	23645
Sqrt2F_2_0	-13.54	71	-10.64	49	-9.85	1259
Sqrt2F_2_2	-13.54	71	-10.64	49	-12.17	1282
Average Ranks	1.0	1.8	2.6	1.3	2.4	3.0

Figure 23.6: Experiments on some f . Most have multiple roots, so the error is y error. All methods did well, though exponential search seems to have gotten lucky. Secant had fewest evaluations.

Random sampling seems only efficient for secant method in 1D, using a fat tail distribution such as Levy

to take scaled-by- x steps from 0 or current best point. Nevertheless, according to my experiments, optimization with the ∞ -norm works well. Still, it won't work for singular Jacobians.

```
template<typename FUNCTION> pair<Vector<double>, double> solveBroydenLevy(
    FUNCTION const& f, int D, double xEps = highPrecEps, int nSamples = 1000)
{
    assert(nSamples > 0 && xEps >= numeric_limits<double>::epsilon());
    typedef Vector<double> X;
    pair<X, double> best;
    double nBestfx;
    for(int i = 0; i < nSamples; ++i)
    {
        X x = GlobalRNG().randomUnitVector(D) * GlobalRNG().Levy();
        pair<X, double> next = solveBroydenHybrid(f, x, xEps);
        double nNextfx = normInf(f(next.first));
        if(i == 0 || nNextfx < nBestfx)
        {
            best = next;
            nBestfx = nNextfx;
        }
    }
    return best;
}
```

Another option is to change the problem to global optimization. On specific problems it may save evaluations to try local search from a nominal point such as 0, but usually this makes no difference. A simple error

estimate is $\max\left(\sqrt{\epsilon}, \frac{\|\nabla g(x_{\text{found}})\|_\infty}{\max(1, \|g(x_{\text{found}})\|)}\right)$, where $g = \|f\|_2$.

```
template<typename FUNCTION> struct NormFunction
{
    FUNCTION f;
    NormFunction(FUNCTION const& theF): f(theF){}
    double operator()(Vector<double> const& x) const{return norm(f(x));}
};

template<typename FUNCTION> pair<Vector<double>, double> solveByOptimization(
    FUNCTION const& f, Vector<double> const& x0, int maxEvals = 1000000)
//use scaled grad as error estimate
{
    NormFunction<FUNCTION> nf(f);
    pair<Vector<double>, double> xy = hybridBeforeLocalMinimize(nf,
        UnboundedSampler(x0), AgnosticStepSampler(),
        makeAgnosticBox(x0.getSize()), maxEvals - 2);
    double errorEstimate = max(normInf(estimateGradientCD(xy.first, nf)) /
        max(1.0, abs(xy.second)), defaultPrecEps);
    return make_pair(xy.first, errorEstimate);
}
```

This gives a reasonable solution for singular J . Use the optimization solution as the starting point for one run of local search. Hybridize random sampling with the optimization:

```
template<typename FUNCTION> pair<Vector<double>, double>
hybridEquationSolve(FUNCTION const& f, int D,
    double xEps = highPrecEps, int maxEvals = 1000000)
//first opt, then Broyden local search on opt result to improve precision
{
    int broydenEvals = max(1000, maxEvals/4),
        optEvals = maxEvals/2 - broydenEvals;
    pair<Vector<double>, double> result = solveByOptimization(f,
        Vector<double>(D), optEvals), result2 =
        solveBroydenHybrid(f, result.first, xEps, broydenEvals);
    //keep opt error estimate if Broyden did nothing
    if(!isfinite(result2.second)) result2.second = result.second;
    //do random Broyden with the other half of evals
    result = solveBroydenLevy(f, D, xEps, maxEvals/2/1000);
    if(normInf(f(result.first)) < normInf(f(result2.first)))
        result = result2;
    return result;
}
```

}

This is probably the most regret-free method for black-box f for exploration, though for routine solving of specific problems try a fast local solver.

Function	Broyden QR		LMBroyden		BroydenLevy		Opt		Hybrid	
	Err	Evals	Err	Evals	Err	Evals	Err	Evals	Err	Evals
ExtendedRosenbrock2	-15.65	8	-15.65	14	-15.65	33560	-13.57	926690	-13.66	253938
ExtendedPowellSingular4	-15.65	10	-14.89	1001	-15.65	84257	-13.48	950871	-15.65	280917
Trig2	-15.65	10	-14.52	49	-15.65	150843	-13.56	900835	-13.61	293819
MultiArctan2	-15.65	23	-15.65	25	-15.65	17972	-14.16	900524	-15.65	235169
HelicalValley	-12.73	198	1.70	1001	-15.65	121340	-13.70	908270	-13.95	291246
LinearFFullRank2	-15.65	7	-15.65	3	-15.65	8000	-15.65	950328	-14.75	229484
Wood	0.62	1002	-6.62	1001	-15.65	727857	-15.65	901484	-15.65	590109
GulfRND	-1.53	1000	-2.10	1001	-15.65	909270	-13.79	925174	-15.65	901813
BiggsExp6	-0.41	1000	-0.59	1001	-15.65	982939	-2.84	966911	-2.50	981325
ExtendedRosenbrock10	-15.65	24	-15.65	14	-15.65	55483	-13.21	946776	-13.45	266917
ExtendedPowellSingular12	-15.65	136	-15.65	410	-15.65	112309	-9.84	991448	-15.65	297937
Trig10	-14.17	76	-1.35	245	-15.65	635819	-13.76	905656	-1.75	803655
MultiArctan10	-15.65	39	-15.65	89	-15.65	59640	-15.65	903046	-14.59	254857
LinearFFullRank10	-15.65	23	-15.65	3	-15.65	24000	-15.65	902984	-14.23	238113
ExtendedRosenbrock100	-14.35	204	-15.65	14	-15.65	293157	-7.52	953481	-14.05	389808
ExtendedPowellSingular100	-15.65	488	-15.37	1001	-15.65	393638	-4.75	953476	-15.65	442253
Trig100	-14.18	923	-2.31	263	-14.23	953470	-2.88	944671	-3.62	968564
MultiArctan100	-14.64	219	0.16	1001	-15.65	272096	-15.65	904848	-15.65	365887
LinearFFullRank100	-15.65	203	-15.65	3	-15.65	204000	-14.33	904289	-15.35	328986
Average Ranks	2.2	1.4	2.7	1.6	1.0	3.3	3.4	4.8	2.8	3.9

Figure 23.7: Experiments on some f . Among local methods QR is much more reliable, but not completely so. Among global methods repeated Broyden won every case, but perhaps lucky in some cases.

No presented method checks if the found root produces f -value close enough to 0. The caller code can do this with any tolerance. Arguably should check if at least got a root to single precision, but the 0 tolerance for such checks seems to be highly application-dependent. There can also be substantial dependence on scale if typical f values are large.

23.14 Finding All Roots in 1D

Finding all roots of a normalized polynomial $\sum_{0 \leq i < n} c_i x^i$ such that $c_n = 1$ reduces to finding eigenvalues of its **companion matrix** whose subdiagonal entries = 1, last column entries = $-c_i$, and the rest = 0 (Golub &

Van Loan 2012). E.g., for $n = 3$ have
$$\begin{bmatrix} 0 & 0 & -c_0 \\ 1 & 0 & -c_1 \\ 0 & 1 & -c_2 \end{bmatrix}$$
. The matrix is in Hessenberg form, so for a bounded number of iterations need $O(n^2)$ time and space.

```
Vector<complex<double>> findAllRoots(Vector<double> const& lowerCoefs)
{
    int n = lowerCoefs.getSize();
    Matrix<double> companion(n, n);
    for(int r = 0; r < n; ++r)
    {
        if(r > 0) companion(r, r - 1) = 1;
        companion(r, n - 1) = -lowerCoefs[r];
    }
    return QREigenHessenberg(companion);
}
```

This doesn't need a special check for $x^n = 0$ (the only solution is $x = 0$), but that may not be true for other implementations of eigenvalue solvers, so would check $\|x\|_\infty < \epsilon$ for some ϵ .

Because it makes little sense to find roots of very-high-degree polynomials due to potential ill-conditioning in terms of c_i (Corless & Fillion 2013), this solution is enough and preferred in practice due to robustness.

For black-box f and finite ranges, Chebyshev polynomials allow calculating all roots, though with some robustness issues. To use a single polynomial, f must be Lipschitz to converge. Because don't know if it is,

and want to reduce the cost of eigenvalue calculations, use adaptive piecewise Chebyshev interpolation. It seems reasonable to restrict the degree to a small number, and 32 seems enough based on my tests. Per Boyd (2014), use the secant method to polish the found roots to increasing their accuracy and remove false roots.

```
template<typename FUNCTION> Vector<double> findAllRealRootsCheb(
    FUNCTION const& f, double a, double b, int maxDegree = 32,
    double duplicateXEps = highPrecEps)
{
    Vector<pair<pair<double, double>, ScaledChebAB>> pieces =
        interpolateAdaptiveHeap<IntervalCheb>(f, a, b, maxDegree).first.
        getPieces();
    PiecewiseData<EMPTY> resultFilter(duplicateXEps);
    for(int i = 0; i < pieces.getSize(); ++i)
    {
        Vector<double> rootsI = pieces[i].second.findAllRealRoots();
        for(int j = 0; j < rootsI.getSize(); ++j)
        { // range and finiteness check
            double polishedRoot =
                solveSecant(f, rootsI[j], duplicateXEps).first;
            if(isInfinite(polishedRoot) && a <= polishedRoot &&
                polishedRoot <= b) resultFilter.insert(polishedRoot, EMPTY());
        }
    }
    Vector<pair<double, EMPTY>> tempResult = resultFilter.getPieces();
    Vector<double> result(tempResult.getSize());
    for(int i = 0; i < tempResult.getSize(); ++i)
        result[i] = tempResult[i].first;
    return result;
}
```

Some analytic transformations allow extension to infinite ranges in some cases, but this isn't considered further.

Finding all roots in complex, multidimensional, or infinite ranges is problematic due to lack of good algorithms. And some of these may not make sense for a given problem—e.g., $\sin(x)$ has infinitely many roots, so something like exponential search is meaningless unless want only one or don't mind giving up after checking a large enough interval.

23.15 Ordinary Differential Equations

Assume $y(x)$ is an unknown function, with an initial condition $y(x_0) = y_0$ and $y'(x) = f(x, y)$, where f is a black box. E.g., consider $f(x, y) = x + y$; with $y(0) = 2$ the analytical solution is $y(x) = 3e^x - x - 1$ (Fausett 2003). To compare, for integration $f(x, y)$ is a function of x only, which leads to an easier problem.

Want to solve ODEs numerically for $x \in [x_0, b]$ for some b . For simplicity, assume that only want to know $y(b)$. If want intermediate values, change the codes to return them or run iteratively with successive b values.

Need extra assumptions to define a unique path in x - y space that a solution must follow (Süli & Mayers 2003). **Picard's theorem:** Let $x \in [x_0, b]$, y be such that $|y - y_0|$ is bounded, and f be continuous in $[x_0, b]$, Lipschitz in y , and bounded when $y = y_0$ (note that the variable y and the function y aren't the same just yet —need to prove it). Then in the solution domain $y(x)$ exists and is unique.

A simple algorithm to solve ODEs is **forward Euler** (often drop the “forward”):

1. Pick a small step size h
2. From $x = x_0$ until $x = b$
3. $y += h f(x, y)$
4. Return y

For integration, Euler is equivalent to the low-order left Riemann sum because of the assumption that $\Delta y = hf(x, y)$. But it's useful for stochastic differential equations, where $y'(x) = f(x, y) + \text{noise}$, because the noise cancels out any smoothness exploited by higher-order methods.

Runge-Kutta methods, of which Euler is a member, are specified succinctly with a **tableau** of the form (Wikipedia 2017d):

c	A
-----	-----

	b
	b*

Here $y_{i+1} = y_i + \sum b_j k_j$, where $k_j = hf(x_i + c_{jk}, \sum A[I, j]k_I)$. The optional b^* defines a built-in error estimate $= y_{i+1}(b) - y_{i+1}(b^*)$. Methods where only the lower-triangular part of $A \neq 0$ and $c_0 \neq 0$ are **explicit** due to being computable directly. The rest are **implicit** due to being dependent on qualities "computed in future" and defined by fixed point iteration. Any 0 entries are usually not specified in a tableau, though they may be here for illustration. E.g., for Euler, the tableau is:

0	0
	1

Convince yourself that it's algebraically equivalent to the simpler formula.

Can only evaluate f when taking steps, so can simulate the substeps directly by applying Euler. In particular, doing this with Simpson's rule leads to the 4th-order Runge-Kutta:

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

The below code is only for illustration of how a step based on such a tableaux is implemented. A somewhat better method is discussed later.

```
template<typename TWO_VAR_FUNCTION>
double RungeKutta4Step(TWO_VAR_FUNCTION const& f, double x, double y,
    double h, double f0 = numeric_limits<double>::quiet_NaN())
{
    if(isnan(f0)) f0 = f(x, y);
    double k1 = h * f0, k2 = h * f(x + h/2, y + k1/2),
        k3 = h * f(x + h/2, y + k2/2), k4 = h * f(x + h, y + k3);
    return y + (k1 + 2 * k2 + 2 * k3 + k4)/6;
}
```

One-step methods such as Euler and Runge-Kutta take one step of size h . The step **truncation error** $T_n = \frac{y_{i+1} - y_i}{h} - \frac{y(x_{i+1}) - y(x_i)}{h}$ (i.e., the applied increment vs. the correct increment). Theorem (Süli & Mayers 2003): Let $T = \max |T_i|$, and assume the equation to satisfies the conditions of Picard's theorem. Then the global error is $O(T)$. The constant factor is far more pessimistic than, e.g., the one for Simpson integration.

Applying this to Runge-Kutta, the error in $y(b)$ is $O(h^4)$; for Euler it's $O(h)$. The intuition for the error is somewhat different than for integration—individual step approximation + estimation error effectively perturbs y_0 by taking the solution on another continuous path that is hopefully close to the original one.

A method is **consistent** if $T \rightarrow 0$ as $n \rightarrow \infty$. An ODE is well-conditioned if a small perturbation to y_0 results only in a small perturbation to y_b . The conditions of Picard's theorem imply that the absolute condition number is finite. Theorem (Corless & Fillion 2013): A one-step method is convergent if it's consistent, and the condition number of the ODE is finite. Intuitively, every step must not have too much error, and the solution path must not compound the errors significantly. So, given that the conditions of Picard's theorem should be satisfied in the first place, only consistency is required for convergence.

$O(h^4)$ may be enough for good practical performance, but taking adaptive steps is more efficient (the fixed- n codes are good for graphing the solutions though, when adjusted to return all intermediate y values). Heap-based adaptivity is impossible because the intervals are dependent, so have the equivalent of recursion/stack-based adaptivity, but where the result of the left call is used by the right call. Some details:

- Start with a small h .
- Monitor local errors. Different options lead to slightly different algorithms.
- Until the local error of the current step is acceptable or reached some limit, halve h .
- When made a step, increase h by a bit in case small h no longer needed
- Keep $h \in [h_{\min}, h_{\max}]$. Need the former for efficiency and the latter for convergence, i.e., must have $h_{\max} \rightarrow 0$ as the number of intervals $\rightarrow \infty$. Too large steps are trimmed, and too small ones automati-

cally accepted and increased. By default $h_{\max} = h_{\min}^2$.

- Handle the right boundary by taking an exact step when approach it, instead of jumping over.
- As an error estimate, return $\sum \text{local errors}$. This is heuristic but tends to be useful in predicting the true error to an order of magnitude. Can have true error > linear error if the Lipschitz constant of the problem is large (as predicted by the theorems), and previous errors are magnified.
- Given some tolerance ϵ , check against $\max \left(\text{some high precision}, \epsilon \sqrt{\frac{h}{b - x_0}} \right)$. It's pessimistic to have $\sum \text{local errors} \leq \epsilon$ because errors cancel out as some steps over- and under-estimate y . So assuming the central limit theorem holds, with $n = (b - x_0)/h$, $\sqrt{n} \sum \text{local errors} \leq \epsilon$, i.e., it's enough that for a single interval local error $\leq \epsilon/\sqrt{n}$. Use high precision only as a safeguard.

A simple adaptive error estimator is comparing a single step with two half-steps. Then take the half-steps if accept h . This works with any stepping method.

But a better algorithm is the **Dormand-Prince method** (Gander et al. 2014; Press et al. 2007). It has fifth order and is the default method in MATLAB (Shampine & Reichelt 1997). The tableau is (Hairer et al. 1993):

0							
1/5	1/5						
3/10	3/40	9/40					
0							
4/5	44/45	-56/15	32/9				
8/9	19372/6561	-25360/2187	64448/6561	-212/729			
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656		
1	35/384	0	500/1113	125/192	-2187/6784	11/84	
	35/384	0	500/1113	125/192	-2187/6784	11/84	0
	5179/57600	0	7571/16695	393/640	-92097/339200	187/2100	1/40

The adaptivity control is a bit simpler because don't need to compute half-steps. A trick is **FSAL**—last step's last f evaluation = the next step's first f evaluation.

```
template<typename TWO_VAR_FUNCTION> pair<pair<double, double>, double>
RungeKuttaDormandPrinceStep(TWO_VAR_FUNCTION const& f, double x, double y,
double h, double f0)
{
    double k1 = h * f0,
           k2 = h * f(x + h/5, y + k1/5),
           k3 = h * f(x + h * 3/10, y + k1 * 3/40 + k2 * 9/40),
           k4 = h * f(x + h * 4/5, y + k1 * 44/45 + k2 * -56/15 + k3 * 32/9),
           k5 = h * f(x + h * 8/9, y + k1 * 19372/6561 + k2 * -25360/2187 +
                 k3 * 64448/6561 + k4 * -212/729),
           k6 = h * f(x + h, y + k1 * 9017/3168 + k2 * -355/33 + k3 * 46732/5247 +
                 k4 * 49/176 + k5 * -5103/18656),
           yNew = y + k1 * 35/384 + k3 * 500/1113 + k4 * 125/192 +
                 k5 * -2187/6784 + k6 * 11/84, f1 = f(x + h, yNew),
           k7 = h * f1;
    return make_pair(make_pair(yNew, f1), y + k1 * 5179/57600 + k3 * 7571/16695 +
        k4 * 393/640 + k5 * -92097/339200 + k6 * 187/2100 + k7/40);
}

template<typename TWO_VAR_FUNCTION> pair<double, double>
adaptiveRungeKuttaDormandPrice(TWO_VAR_FUNCTION const& f, double x0,
double xGoal, double y0, double localERelAbs = defaultPrecEps,
int maxIntervals = 100000, int minIntervals = -1, int upSkip = 5)
{
    if(minIntervals == -1) minIntervals = sqrt(maxIntervals);
    assert(xGoal > x0 && minIntervals > 0 && upSkip > 0);
    double hMax = (xGoal - x0)/minIntervals, hMin = (xGoal - x0)/maxIntervals,
           linearError = 0, h1 = hMax, y = y0, f0 = f(x0, y);
```

```

bool last = false;
int stepCounter = 0;
for(double x = x0; !last;)
{
    if(x + h1 > xGoal)
        //make last step accurate
        h1 = xGoal - x;
    last = true;
}
pair<pair<double, double>, double> yfye =
    RungeKuttaDormandPrinceStep(f, x, y, h1, f0);
double h2 = h1/2, xFraction = h1/(xGoal - x0);
if(h2 < hMin || isEEqual(yfye.first.first, yfye.second,
    max(highPrecEps, localERelAbs * sqrt(xFraction))))
    //accept step
    x += h1;
    y = yfye.first.first;
    f0 = yfye.first.second; //reuse last eval
    linearError += abs(y - yfye.second);
    if(++stepCounter == upSkip && h2 >= hMin)
        //use larger step after few consecutive accepted steps
        h1 = min(hMax, h1 * 2);
        stepCounter = 0;
    }
}
else
    //use half step
    h1 = h2;
    last = false;
    stepCounter = 0;
}
}
return make_pair(y, linearError);
}

```

Extending the algorithm to systems of differential equations is straightforward. The main difference is that for convergence use $\|y_1 - y_2\|_\infty$.

Function	Euler10k			DoublingRK4			DP		
	Err	Est	Evals	Err	Est	Evals	Err	Est	Evals
Step	-3.699	0.000	10000	-5.823	-5.746	512927	-5.490	-6.379	243739
Abs	-13.276	0.000	10000	-14.463	-14.347	3476	-14.477	-15.654	1897
Lin	-3.699	0.000	10000	-15.654	-14.765	3476	-15.654	-15.654	1897
Square	-7.875	0.000	10000	-14.593	-14.353	3476	-14.574	-15.654	1897
Cube	-3.699	0.000	10000	-15.654	-14.710	3476	-15.605	-15.654	1897
Quad	-7.574	0.000	10000	-11.777	-10.601	3476	-14.622	-12.074	1897
Exp	-4.000	0.000	10000	-13.481	-12.282	3476	-14.946	-13.736	1897
SqrtAbs	-6.056	0.000	10000	-1.195	-7.849	446959	-8.590	-9.026	243751
Runge	-9.973	0.000	10000	-9.213	-7.688	4105	-9.264	-9.420	2005
Log	-4.460	0.000	10000	-14.251	-13.241	3476	-14.750	-14.747	1897
XSinXM1	-5.345	0.000	10000	-6.897	-5.501	300684	-7.138	-6.900	245983
Sin	-3.774	0.000	10000	-15.487	-12.319	3476	-15.654	-13.734	1897
AbsIntegral	-4.000	0.000	10000	-15.654	-14.795	3476	-15.654	-15.654	1897
Tanh	-3.817	0.000	10000	-15.654	-11.562	3476	-15.654	-13.036	1897
NormalPDF	-15.654	0.000	10000	-15.176	-7.835	3476	-15.353	-9.308	1897
DeltaPDF	-15.176	0.000	10000	-8.623	-8.238	355785	-8.577	-9.231	255793
F575	-0.501	0.000	10000	-0.989	-1.624	889912	-0.975	-3.519	485425
Average Ranks	2.647	3.000	2.294	1.647	2.000	2	1.529	1.000	1

Figure 23.8: Performance of the methods on some f . The Dormand-Prince error estimate seems less reliable than the doubling error. So the assumptions of the error formula aren't very accurate, but this doesn't matter in practice.

23.16 Solving Stiff ODEs

Some ODEs are **stiff**. E.g. (Süli & Mayers 2003), $y' = ay$ with $y_0 = 0$ has solution $y(x) = y_0 e^{ax}$. For $a > 0$ the solution grows without bound, so the interesting case is $a \leq 0$ where $y(x) \rightarrow 0$. Using Euler's method with n steps of size $h = \frac{x}{n}$ gives $y_n = y_0(1+ah)^n$. To avoid exponential explosion need $-2 > ha > 0$. Stiffness is needing to use a much smaller h than what would be needed otherwise for the wanted accuracy.

For a linear system $y' = Ay$, the matrix A has distinct eigenvalues λ_i (assume the technical conditions hold), and for many algorithms can derive **regions of absolute stability**, i.e., subsets of the complex plane where values $z = h\lambda_i$ don't cause explosion as above. A method is **A-stable** if its region of absolute stability contains the left complex half-plane $\operatorname{Re}(z) \leq 0$.

The **stability function** $R(z)$ of a method for a single step is such that $y_1 = R(z)y_0$ (Hairer & Wanner 1996). E.g., for Euler's method $R(z) = 1 + z$; for A-stability need $|R(z)| \leq 1$ for $\operatorname{Re}(z) \leq 0$. This leads to **L-stability** (Hairer & Wanner 1996): A-stability is satisfied, and $\lim_{\operatorname{Re}(z) \rightarrow \infty} R(z) = 0$. In some sources an alternative, equivalent definition of L-stability ("L" comes from "left") definition is used that requires that $\lim_{z \rightarrow -\infty} R(z) = 0$. It's equivalent because R is a rational function for the cases of interest, and $R(\infty) = R(-\infty)$. For some practical cases A-stability isn't enough, but L-stability is. L-stability leads to a finite stability "circle" of $|R(z)| \leq 1$. For $a > 0$ and certain h it will enforce stability as the solution grows unbounded. This usually isn't a problem, but see Lambert (1991) for some examples where it is. Picking a smaller step size avoids issues (Hairer & Wanner 1996).

For stiff equations explicit step methods aren't stable, so need implicit ones. The simplest implicit step method is **backward Euler**, i.e., solving the equation $y_{i+1} = y_i + hf(x+h, y_{i+1})$. Use Broyden with y_i as the initial value for y_{i+1} . The tableau is:

1	1
	1

The formula and the tableau are the same: $y_{i+1} = y_i + k_0$, and substituting for k_0 in the tableau equations leads to the backward Euler equation.

Broyden or another generic solver isn't guaranteed to work, though it usually does. The ODE only guarantees that the solution is close to the starting value, and the solution exists, so using a local solver is justified. But the solution may not be local enough or can have bad properties, etc. As a further protection, can reduce h when the solver fails, but this isn't guaranteed to help.

A higher-order formula is the **trapezoid rule**, analogous to the integration one, given by $y_{i+1} = y_i + \frac{h}{2}(f(x, y_i) + f(x+h, y_{i+1}))$. But this is only A-stable, and backward Euler is L-stable (Hairer & Wanner 1996). The method is important for PDE solving though (briefly reviewed later in the chapter)

For backward Euler it doesn't seem to matter, but it's important to solve the equations in terms of the *fSum* basis variables and not in terms of *y*. Due to rounding in "*y_i + h(fSum)*", assuming *y* and *fSum* are of about the same magnitude, some precision in *fSum* is lost after scale and add. So if *y_i* is accurate to ϵ , it's pointless to improve accuracy of *fSum* beyond $\frac{\epsilon \|y_i\|_\infty}{h}$. But *fSum* computed to such accuracy allows *y_i*

accurate to ϵ , which is better than computing *y_{i+1}* to ϵ directly. Still, if *fSum* is much larger than *y*, the problem isn't well-scaled, and this formula won't work well, so use the specified $\|y\|_\infty$ precision. A good initial value is 0 because *y_i* is a good initial value for *y_{i+1}*.

With any stepping rule, an adaptive stepper based on halving *h*:

```
template<typename YX_FUNCTION>
Vector<double> evalYX(double x, Vector<double> const& y, YX_FUNCTION const& f)
{//assume last arg is x
    Vector<double> yAugmented = y;
    yAugmented.append(x);
    Vector<double> fAugmented = f(yAugmented);
    fAugmented.removeLast();
    return fAugmented;
}

template<typename YX_FUNCTION, typename STEP_F> pair<Vector<double>, double>
adaptiveStepper(YX_FUNCTION const& f, STEP_F const& s, double x0,
                double xGoal, Vector<double> y0, double localERelAbs = defaultPrecEps,
                int maxIntervals = 100000, int minIntervals = -1,
```

```

double upFactor = pow(2, 0.2))
//assume no reuse of h0
if(minIntervals == -1) minIntervals = sqrt(maxIntervals);
assert(xGoal > x0 && minIntervals > 0 && upFactor > 1);
int D = y0.getSize();
double hMax = (xGoal - x0)/minIntervals, hMin = (xGoal - x0)/maxIntervals,
    linearError = 0, h1 = hMax;
Vector<double> y = y0,
    y1 = Vector<double>(D, numeric_limits<double>::quiet_NaN()), f0;
bool last = false;
for(double x = x0; !last;)
{
    if(x + h1 > xGoal)
        //make last step accurate
        h1 = xGoal - x;
        last = true;
    }
    double h2 = h1/2, xFraction = h1/(xGoal - x0),
        tolERelAbs = max(highPrecEps, localeRelAbs * sqrt(xFraction)),
        solveERelAbs = tolERelAbs/10;
    if(isnan(normInf(y1))) y1 = s(f, x, y, h1, solveERelAbs);
    Vector<double> y2 = s(f, x, y, h2, solveERelAbs),
        firstY2 = y2;
    y2 = s(f, x + h2, y2, h2, solveERelAbs);
    double normError = normInf(y2 - y1), normY2 = normInf(y2);
    if(h2 < hMin || isEEqual(normY2 + normError, normY2, tolERelAbs))
        //accept step
        x += h1;
        y = y2;
        linearError += normError;
        y1 = Vector<double>(D, numeric_limits<double>::quiet_NaN());
        if(h2 >= hMin) h1 = min(hMax, h1 * upFactor); //use larger step
    }
    else
        //use half step
        y1 = firstY2;
        h1 = h2;
        last = false;
    }
}
return make_pair(y, linearError);
}

```

The 5th order **Radau II A** stepping method is good in both theory (L-stable; Hairer & Wanner 1996) and practice (see the comparisons in Hairer & Wanner 1996). The tableaux is:

$(4 - \sqrt{6})/10$	$(88 - 7\sqrt{6})/360$	$(296 - 169\sqrt{6})/1800$	$(-2 + 3\sqrt{6})/225$
$(4 + \sqrt{6})/10$	$(296 + 169\sqrt{6})/1800$	$(88 + 7\sqrt{6})/360$	$(-2 - 3\sqrt{6})/225$
1	$(16 - \sqrt{6})/36$	$(16 + \sqrt{6})/36$	1/9
	$(16 - \sqrt{6})/36$	$(16 + \sqrt{6})/36$	1/9

It's based on **Gauss-Radau quadrature** and is FSAL.

```

template<typename YX_FUNCTION> struct RadauIIA5Function
{
    Vector<double> y;
    double x, h;
    YX_FUNCTION f;
    Vector<double> operator()(Vector<double> fSumDiffs) const
    {
        assert(fSumDiffs.getSize() % 3 == 0);
        int D = fSumDiffs.getSize()/3;
        double s6 = sqrt(6), ci[3] = {(4 - s6)/10, (4 + s6)/10, 1}, A[3][3] =
        {

```

```

    {(88 - 7 * s6)/360, (296 - 169 * s6)/1800, (-2 + 3 * s6)/225},
    {(296 + 169 * s6)/1800, (88 + 7 * s6)/360, (-2 - 3 * s6)/225},
    {(16 - s6)/36, (16 + s6)/36, 1.0/9}
};

Vector<Vector<double>> ki(3);
for(int i = 0; i < 3; ++i)
{
    Vector<double> yi(y);
    for(int j = 0; j < D; ++j) yi[j] += h * fSumDiffs[i * D + j];
    ki[i] = evalYX(x + h * ci[i], yi, f);
}
for(int i = 0; i < 3; ++i)
{
    Vector<double> fSumi(D, 0);
    for(int j = 0; j < 3; ++j) fSumi += ki[j] * A[i][j];
    for(int j = 0; j < D; ++j) //convert fixed point to d
        fSumDiffs[i * D + j] = fSumi[j] - fSumDiffs[i * D + j];
}
return fSumDiffs;
}

};

struct RadauIIA5StepF
{
    template<typename YX_FUNCTION> Vector<double> operator()(YX_FUNCTION const& f, double x, Vector<double> y, double h,
    double solveERelAbs) const
    {
        RadauIIA5Function<YX_FUNCTION> r5f = {y, x, h, f};
        int D = y.getSize();
        Vector<double> fSumDiffs(3 * D, 0);
        fSumDiffs = solveBroyden(r5f, fSumDiffs, max(solveERelAbs,
            numeric_limits<double>::epsilon() * normInf(y)/h)).first;
        for(int j = 0; j < D; ++j) y[j] += h * fSumDiffs[2 * D + j];
        return y;
    }
};

```

Despite 5th order, due to limitations of nonlinear equation solving, going beyond local precision of 10^{-12} or so doesn't seem effective. This is probably because f for stiff equations is somewhat sensitive to inexact values (Hairer & Wanner 1996), and even working in the $fSum$ basis has its limits. Adaptive strategies can't recognize when decreasing h brings no further benefit (unclear how to do this effectively), but a fixed- h Radau IIA stepper can reach 10^{-10} accuracy with just 100 steps in my experiments. Also, there doesn't seem to be an embedded Radau IIA method like Dormand-Prince, so use step halving.

Using Dormand-Prince for nonstiff and Radau IIA for stiff equations is a good default choice—i.e., not necessarily the best one but never a bad one (though for large systems Radau IIA may need too much memory).

For mildly stiff ODEs solving implicit equations isn't guaranteed to work, but a nonstiff adaptive solver can be efficient and robust. So a reasonable black-box strategy is to try it for some number of evaluations, and switch to a stiff one if the reported accuracy is too small. Making this choice automatically is a natural next step but not considered further.

If f isn't as smooth as assumed, the order of convergence will be limited by the number of continuous derivatives, as for interpolation (though there don't seem to be any theorems for this in the literature). When this number is low, high-order methods waste evaluations per step because low-order methods such as Euler's get the same order with fewer. But the waste is bounded by a small constant factor, so as a general strategy higher-order methods present the best compromise in both exploiting high order where possible and not being too inefficient where not.

The presented explicit and implicit Runge-Kutta method implementations are state of the art, though good library implementations have far more engineering.

23.17 Boundary Value Problems for ODE

The ODE initial conditions allow advancing in time. A more general problem is to have conditions for both

start and end points. In standard form have a function g such as $g(y_0, y_b) = 0$ (Fausett 2003; Press et al. 2007; Heath 2018). A common solution is reducing to ODE using the **shooting method**. In a fixed point iteration formulation:

1. Derive y_0 from the initial guess for boundary conditions
2. Solve the resulting ODE for y_b
3. Adjust the boundary condition guess, and repeat

A solution doesn't exist if a root doesn't, and, in addition to the Picard's theorem conditions, need extra ones to make sure that the ODE solver doesn't break down for arbitrary initial-guess-derived y_0 . These issues aren't discussed further here—see Süli & Mayers (2003) and Gautschi (2011). Commonly recommended as the most comprehensive and detailed book on the topic is Ascher et al. (1995).

In the simplest case, which physically corresponds to shooting a cannon by picking a firing angle, have a single unknown parameter, and can use the secant method to find roots of g . Need some useful way to specify the result because now care about some number of interior points. The simplest way to do this with an adaptive ODE solver is to first solve for the boundary condition parameters. Then run a second round where given a number of interval points solve in each interval, treating the end result as the initial condition for the next interval.

```
template<typename YX_FUNCTION, typename BOUNDARY_FUNCTION>
struct BoundaryFunctor
{
    YX_FUNCTION const& f;
    BOUNDARY_FUNCTION const& bf;
    double x0, xGoal;
    double operator()(double b) const
    {
        return bf.evaluateGoal(adaptiveStepper(f, RadauIIA5StepF(),
            x0, xGoal, bf.getInitial(b)).first);
    }
};

template<typename YX_FUNCTION, typename BOUNDARY_FUNCTION>
Vector<Vector<double>> solveBoundaryValue(YX_FUNCTION const& f, double x0,
    double xGoal, Vector<double> const& xPoints, BOUNDARY_FUNCTION const& bf,
    double b0 = 0)
{
    BoundaryFunctor<YX_FUNCTION, BOUNDARY_FUNCTION> fu = {f, bf, x0, xGoal};
    double bFound = solveSecant(fu, b0).first;
    Vector<Vector<double>> result;
    if(isfinite(bFound))
    {
        Vector<double> y0 = bf.getInitial(bFound);
        for(int i = 0; i < xPoints.getSize(); ++i)
        {
            y0 = adaptiveStepper(f, RadauIIA5StepF(), x0, xPoints[i],
                y0).first;
            x0 = xPoints[i];
            result.append(y0);
        }
    }
    return result;
}
```

For several parameters use multidimensional root finding methods. Also, may want to adjust the code to allow faster nonstiff solvers and passing parameters to them. Some ODE solvers have dense output option (see the comments section) that can be used instead of asking for the solution at a finite number of points.

23.18 Partial Differential Equations—Some Thoughts

PDEs are a generalization of ODEs and BVPs. Due to the extra complexity, can solve black-box PDEs efficiently, so most methods usually work with linear PDEs, commonly classified into **parabolic**, **hyperbolic**, and **elliptic** (Heath 2018). One useful model is the **heat equation**—function u in time variable t and 1D space variable x , given by the:

- Equation $u_t = cu_{xx}$
- Space domain $0 \leq x \leq L$

- **Initial time condition** $u(0, x) = f(x)$
- **Left space boundary** $u(t, 0) = a$
- **Right space boundary** $u(t, L) = b$

The simplest solution is **the method of lines**—discretize the space variable x using a finite difference formula, and solve the resulting system of many equations using an ODE solver, configured to return the values at the wanted time points. E.g., for heat equation using 2nd derivative central difference get equation system $y_i^{(2)}(t) = c \frac{y'_{i+1}(t) - 2y'_i(t) + y'_{i-1}(t)}{\Delta x^2}$ for $y_i(t) = u(t, x_i)$. This works only because know the form of the equation.

Such systems can be stiff and large, so using >1000 space points can be slow. It's also sparse, so should exploit that. The method works for $D > 1$ in the same way, but need to be more economical with the discretization due to the curse of dimensionality. With all methods for $D > 1$, and particularly for $D > 2$, need a lot of care for computational feasibility.

Alternatively, can proceed in one of several ways:

1. **Finite difference methods**—use finite difference derivative approximations to transform the problem into a usually banded or sparse system of linear or nonlinear equations.
2. **Finite element methods**—assume the solution is given in terms of basis functions with local support such as B-splines (discussed in the comments), and solve for the coefficients of the representation by matching derivative and boundary condition values. This also leads to solving equations.
3. **Spectral methods**—similar to finite element, but use Chebyshev polynomial basis.

(1) is the simplest, (2) allows the most flexible boundary conditions, and (3) gives very high accuracy when it works, with each having a niche in particular domains. \exists a lot of theoretical and practical literature on each, and much of it is recent, so no attempt to select the most useful references is made here. Some good starting points are Heath (2018), and Press et al. (2007). For an extensive reading list for PDEs see Gautschi (2011). For convenience I converted it to an Amazon idea list <http://a.co/7Zrtmjl>.

23.19 Some Conclusions

Again, numerical algorithms don't come as full-proof solutions simply because there are none. \forall algorithm consider a checklist of issues:

- Does it converge in all cases—the basic algorithm should, at least under some known analytical conditions. Any meta-logic such as heuristic parameter picking or adaptive control generally doesn't. For deterministic algorithms think about whether the answer and the estimates make sense $\forall f$ that use the same evaluation points.
- Is the algorithm stable—should be, unless nothing else that's feasible is.
- Is the algorithm with all the meta logic reliable in all cases—usually not. But it should be in most cases, and cases when not, e.g., with the inputs “on the edge of the machine” should be known. Some algorithms such as direct matrix manipulators, FFT, and binary search equation solving are very reliable and can be considered black box. If not very reliable, at least want a well-defined subset where the algorithm is known/proven to work.
- Is the problem you are solving well-conditioned? If can, determine the condition number theoretically or estimate numerically. If it's large and the algorithm is stable, can try to solve anyway, but don't expect much, and perhaps should solve a different problem instead. Only a few problems types guarantee good conditioning in all cases—e.g., absolute condition of integration.
- Is there a reliable estimate or a bound for the forward error? Usually have a heuristic estimate that's not reliable but a good order-of-magnitude guess.
- Has the algorithm been extensively tested with a suite of problems? For numerical algorithms, unlike for regular logic ones, have many more distinct cases. Something silly such as flipped sign can be unnoticeable with a common single run test, so need to setup good automated regression testing to make sure that at least avoid silly errors and cover all special cases that are feasible to cover. A simple heuristic is to check the answer to single precision—should work in all cases.

So no numerical result should be taken for granted, and when these are used in scientific research, some expert babysitting is expected.

23.20 Implementation Notes

Every single implementation here follows a basic textbook algorithm but makes some original choices. Each also took substantial time to research. Most seem solid, though I have doubts about the limited-memory

Broyden.

When implementing Dormand-Prince, I mistyped the “64448/6561” as “64448/6581”. Because of the similarity of the digits, this was hard to notice. The debug process was as follows:

1. Realize that the error of the method is too large
2. Check the order, and estimate that it roughly varies from 2 to 3 for the tried problems
3. Make a simple nonadaptive program, and realize that it has larger error than the 4th order Runge-Kutta
4. Find a worked-out example, and discover some inaccuracy in the computation of k_5

23.21 Comments

My implementation of $O(n \lg(n))$ FFT is probably the simplest, but not the fastest in constant factors. The power-of-two decomposition idea extends to other composite numbers, so a faster algorithm would be something like this:

1. Compute the prime factorization of n ($O(n)$ with sieve of Eratosthenes—see the “Miscellaneous Algorithms” chapter)
2. Recurse using composite FFT until hit prime factors
3. Solve prime factor problems using a prime- n algorithm

For details of a state-of-the-art implementation see Frigo & Johnson (2005), where a lot of logic is spent selecting the best algorithm to minimize constant factors by considering both n and the hardware.

For Chebyshev polynomial evaluation it's more stable to use the $O(n)$ specialized barycentric formula where the expressions for the weights are simplified (Trefethen 2019). Use inverse DCT to map back to the data points and this formula (this is implemented in Chebfun; can 0-pad to a power of two for DCT efficiency, and map to “pseudo-points” to which the specialized formula applies in the same way), but Chen-shaw's algorithm is $O(n)$ and stable enough.

Another subtopic not discussed here is **Hermite interpolation** where also have f' values, which is useful in certain special applications.

For piecewise polynomials a problem is that the derivatives aren't continuous at the nodes, and in certain applications need smoothness of f and maybe f'' . **Cubic splines** are the most typical choice here, mostly due to a mechanical interpretation—both velocity and acceleration are continuous so don't have abrupt motion. See Faust (2003) for some examples and Dahlquist & Björck (2008) for theory if curious. But the use case is limited:

- The observed continuity is subject to floating point error, and observed derivative differences compared to piecewise polynomials may not be large.
- For numerical use, polynomials and piecewise polynomials work better—e.g., spline calculation need all the data to solve, whereas usually only need to work with a specific piece. Might argue that conceivably some applications need estimating first and second derivatives where must ensure continuity of estimates, but this is rare, and will use specialized techniques like nonstandard splines anyway.
- Useful spline work is multidimensional and uses numerically stable **B -splines** and generalization for curves and surfaces. The methods are usually application-specific. E.g., in graphics it's common to represent surfaces in 2D or 3D, and many special methods have been developed for that. Certain multidimensional generalizations are useful for solving PDEs. Most uses of splines fall in this category.
- Some regression methods do approximation (and not interpolation) using B -splines with a smoothness penalty, but where the definition points (called **knots**) don't correspond to the data points (Hastie et al. 2009). But unlike many other methods (see the “Machine Learning—Regression” chapter) splines don't extend efficiently to large D , and in 1D prefer other methods.
- For graphing a data set linear interpolation is the default option in MATLAB because other options like cubic splines can lead to oscillations (Moler 2013; but see their pchip program that uses Hermite splines to enforce monotonicity for perhaps a more visually pleasing look, but any visual smoothness benefits are questionable, and graphics isn't considered in this book). Most financial graphs use lines to connect market values.

Using Chebyshev polynomials is only one of several good ways to do nonspline interpolation:

- **Rational fraction interpolation** is sometimes better—see Trefethen (2019) for some particular cases, such as lack of differentiability, infinite ranges, or needing a more accurate estimate from a fixed number of known f values. But the algorithms are much more complicated. Per Boyd (2014), rational interpolation is a “definite maybe”, i.e., in future may become more stable, more robust, and

more efficient.

- **Best approximation**—done by **Remez algorithm** to compute a **minimax polynomial** with lowest worst-case approximation error. It needs to locate the extrema of f and so isn't black-box. Chebyshev interpolation will get the same error with slightly higher degree and much faster (Trefethen 2019). Best interpolation is useful only for elementary function evaluation (see Muller 2016 for some examples).

Large- D interpolation suffers from the curse of dimensionality, and even for $D > 3$ is not feasible in many applications.

Gaussian quadrature is the most accurate integration method for the number of evaluations for smooth f . Traditionally, it has been much slower than Clenshaw-Curtis (Press et al. 2007), but now have an $O(n)$ algorithm (Bogaert 2014). Still, point reuse doesn't work, and because Clenshaw-Curtis is almost as accurate (Trefethen 2019), it's the preferred general method.

For multidimensional integration, de Doncker et al. (2007) exploit the recursive error estimate by giving more resources to the base case and other small-volume integrations. This needs more experimentation to replace the simpler all-equal allocation strategy.

Hahn (2005) discusses more advanced partitioning schemes. As for 1D adaptive integration, the idea is to give more evaluations to regions with higher estimated error. Also the main deterministic approach can be either less or more efficient than iterated integration, depending on f (Li 2005).

Other cubature methods subdivide the integration region directly into small hyper-rectangles or other primitives and integrate each assuming some interpolation or approximation. E.g., for rectangular regions in 2D, can:

1. Break up into small rectangles using a grid
2. Cut each small rectangle into triangles
3. \forall triangle interpolate a plane through the corner points
4. Integrate assuming f is correctly modeled by a plane

Also can use tensor product trapezoid rule when sample all the corner points, and do iterated integration. The order doesn't matter, and all points have equal weights. This uses more points than the triangulation but gets error of same order. For examples of common regions in 2D see the cubature section of NIST (2017). One of their square rules uses 5 points (corners and center) to get $O(h^4)$ order for a square of size h , and isn't a product rule. Product Simpson's rule needs 9 points. Product rules aren't optimal but for iterated integration can adapt each dimension independently.

Can scale a hyperrectangle to a hypercube, so only need hypercube rules. Given any Lobatto-like rule that uses all corners, can create an adaptive integrator in the same way as for 1D by doubling or using the corresponding Kronrod extension. E.g., **Lyness rule** has degree 5 and uses in addition to corners points $\pm\sqrt{2}/5$ in each coordinate (Davis and Rabinowitz 1984, p379). Because break up a hypercube into 2^D equal-sized hypercubes, convergence depends on $\|\Delta x\|$ whatever the order is, and to halve it need $O(2^D)$ work. So to be better than Sobol sequence need order $> 2^D$, and f with many enough continuous derivatives. Subdivision isn't economical for $D >$ the order of convergence.

Of particular interest are formulas that integrate all multivariate polynomials up to some degree in a particular region exactly with minimal number of evaluations, like Gaussian integration in 1D (the Lyness rule is one example). They can be effective for small D and common regions such as hyperrectangles and hyperspheres. For further info see Davis and Rabinowitz (1984), Cools (2002), and references therein.

As a general rule, given a robust fast enough method, don't use faster, slightly less robust methods. This is most visible for single-variable equation solving because there is no shortage of methods that try to improve bisection. They may use 1/3 the evaluations, but this makes little difference given that bisection doesn't need many. The popular **Brent method**, which mixes bisection with parabolic interpolation, can have performance problems in the worst case (Wilkins & Gu 2013).

For limited-memory Broyden, an explicit formula (van de Rotten 2003) also allows implicit calculation of B with the same complexities, though the presented approach is simpler by computing B_{nx} directly and is equivalent to the formula. The robustness heuristics for equation-solving are mine and original.

For solving equations in large D , **Newton-Krylov method** is better than limited-memory Broyden if know the sparsity pattern of the Jacobian (Kelley 2003; i.e., which f components don't depend on which variables). Use Newton's method with backtracking, but estimate Jacobians analytically or from finite differences, and use iterative solvers such as CCNR for the equations. This works very well for many applications, such as solving PDEs where the analytical structure if known, but the method isn't fully automatic.

An idea that doesn't work is applying some form of coordinate descent (see the "Numerical Optimization" chapter) because all f components may be dependent on any subset of variables, creating a system that's not square. E.g., can solve one equation at a time with one variable at a time, but it's not guaranteed

that the equation is dependent on that variable. Also given a large system, can solve a sequence of smaller systems using QR Broyden. Here perhaps select equations and variables randomly, and iterate without random steps until hit a local maximum. Such methods are heuristic.

For finding all roots of a polynomial the best algorithm is the recent improvement of companion matrix eigenvalue solving by Aurentz et al. (2018).

Deriving Runge-Kutta formulas and proving their analytic properties is difficult. Perhaps the most extensive explanation is from the inventor of the efficient derivation method (Butcher 2016). Also see Hairer et al. (1993) and Lambert (1991).

Before the discovery of Dormand-Prince formulas, **Fehlberg formulas** were the best option (for some comparisons see Hairer et al. 1993). An 8th-order Dormand-Prince formula is available and performs very well (Hairer et al. 1993; Press et al. 2007), but for some reason it isn't included in the MATLAB suite (Shampine & Reichelt 1997). Maybe the extra complexity doesn't seem worth it.

The main historical competitor to Runge-Kutta methods are **predictor-corrector (PECE)** methods. The idea is to interpolate the f values from the last k steps, integrate the interpolation using the current y_i as the constant of integration, and use the result to calculate y_{i+1} . Given a chosen h , need several components:

- **Explicit Adams-Bashford method:** f_i is the last one. It's convenient, but has poor stability properties (Ascher & Greif 2011).
- **Implicit Adams-Moulton method:** f_{i+1} is the last one, so need a guess for y_{i+1} to evaluate it. It has good stability properties (Ascher & Greif 2011), but needs implicit equation solving.

PECE results from using:

- Adam-Bashford to guess for y_{i+1} for Adams-Moulton.
- Using one step of fixed-point iteration to solve the implicit equation—i.e., use $f_{i+1}(x_{i+1}, y_{i+1})$. Also usually use $k + 1$ last f values by not removing the oldest one.
- The correction amount of the guess of y_{i+1} is a good error estimate (called **Milne error**).

For some formulas and examples with equal- h intervals see (Fausett 2003). These aren't useful because want adaptive variable steps, so use the interpolation directly. Traditionally used older **Newton's interpolation** (Fausett 2003), but barycentric interpolation is more stable (though it's unstable for extrapolation, i.e., integral evaluation for the predictor).

This formulation also allows changing order dynamically by using the wanted number of last points. The MATLAB implementation changes k up to 13 in some cases (Shampine & Reichelt 1997), which they claim wins over Dormand-Prince for high-precision calculations with expensive-to-evaluate f .

But a practical PECE implementation seems too problematic. Hairer et al. (1993) describe some implementation details:

- As the first step use the Euler step, and build up order using the above mechanism. But to have near- $\epsilon_{\text{machine}}$ precision, this step must be similarly small, and that doesn't seem to allow quickly raising the step size by several orders of magnitude to where it should be with proper order.
- Can use barycentric or Newton form for automatic interpolation, but direct integration of either doesn't seem possible. E.g., the obvious idea of inverting the barycentric differentiation matrix fails because the latter is singular. Numerical integration that is exact for a high-degree polynomial is an obvious option but may be slow and not accurate enough. So use special methods to integrate the Newton form.
- The order-change logic uses a heuristic. It's unclear whether can do something better.

To compare with Dormand-Prince:

- Theoretically Runge-Kutta methods are more stable than multistep methods, particularly high-order ones (Suli & Mayers 2003). For the former raising the order improves stability.
- Based on tests in Hairer et al. (1993), Dormand-Prince (8th-order) is always faster on runtime than the code on which MATLAB's implementation is based. The implementation of the latter, as outlined above, has much overhead.
- Based on same tests, the PECE code does a small-constant-factor-fewer evaluations for high accuracy. So perhaps that's why MATLAB PECE code is recommended for super-expensive f at high precision. But Dormand-Prince can get close to $\epsilon_{\text{machine}}$ with not too many evaluations, and smooth f are rarely expensive to evaluate because elementary function libraries are fast, and truly expensive f from maybe simulations are hardly smooth.

Extrapolation methods, advocated for by Press et al. (2007), are similar in use case for multistep methods. They don't seem to do as well on studies and have been mostly ignored by the major libraries (Hosea & Shampine 1994; Ketcheson & bin Waheed 2014).

Using Broyden's method for stiff ODEs isn't typical. Older implementations prefer Newton's method that

doesn't update the finite-difference J . Broyden's updates are strictly better. Can save the current Broyden matrix to the next step, which may offer substantial computational savings, but this needs further study.

For stiff equations MATLAB has good implementations of implicit multistep methods, which are based on a different interpolation (Shampine & Reichelt 1997). Experiments suggest that Radau IIA code radau5 (Hairer & Wanner 1996) is slower but safer than MATLAB's multi-step method ode15s (Sandu & Sander 2006; Gonnet et al. 2012). Theoretically, by **Dahlquist's second barrier theorem**, no multi-step method of order > 2 is A-stable (Süli & Mayers 2003). But some practical ones, such as the implicit ones implemented in MATLAB, satisfy a relaxed version of A- and L-stability, and this appears to be enough for good performance (Shampine & Reichelt 1997).

Convergence analysis for explicit and implicit methods is more complicated because must take into account several starting values, and consistency alone isn't enough, though this is taken care of by conditions such as **zero-stability** (Süli & Mayers 2003). An intuitive reason for why methods with memory need more than one-step methods is that the used previous step values have some error, and this can cause further error.

23.22 Projects

- Compare trapezoid rule for integration from existing data with using piecewise barycentric interpolation that uses pieces of small degree by taking few points at a time from left to right (with the last piece using all remaining points).
- Extend barycentric interpolation to work with a collection of points dynamically, just like linear interpolation. Instead of selecting nearest points as nearest neighbors, it may be worth it to ensure that no extrapolation is done for stability at the cost of using a further but enclosing point. Test it for derivative evaluation with given points and think about how it competes with dynamic linear interpolation for function evaluation. Another option is to compute a piecewise interpolant statically from all available data (with the last or the middle point getting more points if not an exact multiple of the degree).
- Extend Monte Carlo integration to allow a generic sampler instead of only accept-reject to allow better efficiency for special cases.
- Implement adaptive integration with Lyness rule, and compare its performance with that of competing methods.
- Sort the roots for finding all roots with Chebyshev interpolation.
- Extend the Dormand-Prince code to handle a system of equations. For PECE methods this seems less straightforward. Convert the code into a stepper that allows to execute a user-defined function, a no-op by default. Might also want to pass it the local error estimate.
- Change all ODE solvers to bound the number of function evaluations as the only resource limit.
- Find or reasonably show absence of use cases for Newton form of interpolation. The custom PECE use is one possibility, but it maybe possible to update it to barycentric form with better results. Per Berrut & Trefethen (2004) barycentric form isn't the only useful one, but no justification is given. One idea is that barycentric form is known to not be stable for extrapolation.
- Markov brothers' inequality gives a very loose bound on the constant of the error from derivative approximation by interpolation. Research the literature for better bounds or better overall proofs.
- Check all presented algorithms to make sure they give appropriate errors when some inputs are NaNs or infinities.

23.23 References

- Ascher, U. M., & Greif, C. (2011). *A First Course on Numerical Methods*. SIAM.
- Ascher, U. M., Mattheij, R. M., & Russell, R. D. (1995). *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations* (Vol. 13). SIAM.
- Aurentz, J. L., Mach, T., Robol, L., Vandebril, R., & Watkins, D. S. (2018). *Core-Chasing Algorithms for the Eigenvalue Problem*. SIAM.
- Beatson, R. K. (1986). On the convergence of some cubic spline interpolation schemes. *SIAM Journal on Numerical Analysis*, 23(4), 903-912.
- Beatson, R. K., & Chacko, E. (1990). Which cubic spline should one use?
- Beebe, N. H. F. (2017). *The Mathematical-Function Computation Handbook*. Springer.
- Berrut, J. P., & Trefethen, L. N. (2004). Barycentric Lagrange Interpolation. *SIAM Review*, 46(3), 501-517.
- Bogaert, I. (2014). Iteration-Free Computation of Gauss--Legendre Quadrature Nodes and Weights. *SIAM*

- Journal on Scientific Computing*, 36(3), A1008-A1026.
- Boyd, J. P. (2014). *Solving Transcendental Equations: The Chebyshev Polynomial Proxy and Other Numerical Rootfinders, Perturbation Series, and Oracles*. SIAM.
- Butcher, J. C. (2016). *Numerical Methods for Ordinary Differential Equations*. Wiley.
- Cools, R. (2002). Advances in multidimensional integration. *Journal of Computational and Applied Mathematics*, 149(1), 1-12.
- Corless, R. M., & Fillion, N. (2013). *A Graduate Introduction to Numerical Methods*. Springer.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Dahlquist, G., & Björck, Å. (2008). *Numerical Methods in Scientific Computing*. SIAM.
- Davis, P. J., & Rabinowitz, P. (1984). *Methods of Numerical Integration*. Dover.
- de Doncker, E., Li, S., & Kaugars, K. (2007). Error distribution for iterated integrals. *WSEAS Transactions on Mathematics*, 6(1), 86.
- de Villiers, J. (2012). *Mathematics of Approximation*. Atlantis Press.
- Dennis Jr, J. E., & Schnabel, R. B. (1996). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM.
- Erdős, P., & Vértesi, P. (1980). On the almost everywhere divergence of Lagrange interpolatory polynomials for arbitrary system of nodes. *Acta Mathematica Hungarica*, 36(1-2), 71-89.
- Fausett, L. V. (2003). *Numerical Methods: Algorithms and Applications*. Pearson.
- Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 216-231.
- Gander, W., Gander, M. J., & Kwok, F. (2014). *Scientific Computing - An Introduction using Maple and MATLAB*. Springer.
- Gao, J., & Iserles, A (2016). An adaptive Filon algorithm for highly oscillatory integrals.
- Gautschi, W. (2011). *Numerical Analysis*. Birkhäuser.
- Golub, G. H., & Van Loan, C. F. (2012). *Matrix Computations*. JHU Press.
- Gonnet, P. (2009). *Adaptive Quadrature Re-revisited* (Doctoral dissertation, ETH Zürich).
- Gonnet, P. , Dimopoulos, S., Widmer, L., & Stelling, J. (2012). A specialized ODE integrator for the efficient computation of parameter sensitivities. *BMC systems biology*, 6(1), 46.
- Hahn, T. (2005). CUBA—a library for multidimensional numerical integration. *Computer Physics Communications*, 168(2), 78-95.
- Hairer, E., Nørsett, S. P., & Wanner, G. (1993). *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer.
- Hairer, E., & Wanner, G. (1996). *Solving ordinary differential equations II*. Springer.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
- Heath, M. T. (2018). *Scientific Computing: An Introductory Survey*. SIAM.
- Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. SIAM.
- Hosea, M. E., & Shampine, L. F. (1994). Efficiency comparisons of methods for integrating ODEs. *Computers & Mathematics with Applications*, 28(6), 45-55.
- Isaacson, E., & Keller, H. B. (1994). *Analysis of Numerical Methods*. Dover.
- Kelley, C. T. (2003). *Solving Nonlinear Equations with Newton's Method*. SIAM.
- Ketcheson, D., & bin Waheed, U. (2014). A comparison of high-order explicit Runge–Kutta, extrapolation, and deferred correction methods in serial and parallel. *Communications in Applied Mathematics and Computational Science*, 9(2), 175-200.
- Kopriva, D. (2009). *Implementing Spectral Methods for Partial Differential Equations: Algorithms for Scientists and Engineers*. Springer.
- Kranzer, H. C. (1963). An error formula for numerical differentiation. *Numerische Mathematik*, 5(1), 439-442.
- Kythe, P. K., & Schäferkotter, M. R. (2004). *Handbook of Computational Methods for Integration*. CRC.
- Lambert, J. D. (1991). *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*. Wiley.
- Li, S. (2005). *Online Support for Multivariate Integration*. (Doctoral dissertation, Western Michigan University).
- Lyons, R. (2015). Four Ways to Compute an Inverse FFT Using the Forward FFT Algorithm. <https://www.dsprelated.com/showarticle/800.php>. Accessed August 5, 2017.
- Martinez, J. M. (2000). Practical quasi-Newton methods for solving nonlinear systems. *Journal of Computational and Applied Mathematics*, 124(1), 97-121.
- Mason, J. C., & Handscomb, D. C. (2002). *Chebyshev Polynomials*. CRC.
- Moler, C. B. (2013). *Numerical Computing with MATLAB*. https://www.mathworks.com/moler/index_ncm.html.

- Moré, J. J., Garbow, B. S., & Hillstrom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41.
- Muller, J. M. (2016). *Elementary Functions*. Birkhäuser.
- Nocedal, J., Wright, S. (2006). *Numerical Optimization*, 3rd ed. Springer.
- NIST (2017). *NIST Digital Library of Mathematical Functions 1.0.16*. <http://dlmf.nist.gov/>.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press.
- Powell, M. J. D. (1981). *Approximation Theory and Methods*. Cambridge University Press.
- Prenter, P. M. (1975). *Splines and Variational Methods*. Dover.
- Sandu, A., & Sander, R. (2006). Technical note: Simulating chemical systems in Fortran90 and Matlab with the Kinetic PreProcessor KPP-2.1. *Atmospheric Chemistry and Physics*, 6(1), 187-195.
- Shampine, L. F., & Reichelt, M. W. (1997). The MATLAB ODE suite. *SIAM Journal on Scientific Computing*, 18(1), 1-22.
- Smith, J. O. (2007). *Mathematics of the Discrete Fourier Transform (DFT): With Audio Applications*. W3K.
- Süli, E., & Mayers, D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge University Press.
- Trangenstein, J. A. (2018a). *Scientific Computing: Vol. I - Linear and Nonlinear Equations*. Springer.
- Trangenstein, J. A. (2018b). *Scientific Computing: Vol. II - Eigenvalues and Optimization*. Springer.
- Traub, J. F., & Werschulz, A. G. (1998). *Complexity and Information*. Cambridge University Press.
- Trefethen, L. N. (2019). *Approximation Theory and Approximation Practice*. SIAM.
- Ueberhuber, C. W. (1997a). *Numerical Computation 1: Methods, Software, and Analysis*. Springer.
- Ueberhuber, C. W. (1997b). *Numerical Computation 2: Methods, Software, and Analysis*. Springer.
- van de Rotten, B. (2003). *A Limited Memory Broyden Method to Solve High-dimensional Systems of Nonlinear Equations*. (Doctoral dissertation, University of Leiden).
- Wikipedia (2017a). Discrete Fourier transform. http://en.wikipedia.org/wiki/Discrete_Fourier_transform. Accessed January 4, 2017.
- Wikipedia (2017b). Chirp Z-transform. https://en.wikipedia.org/wiki/Chirp_Z-transform. Accessed August 6, 2017.
- Wikipedia (2017c). Fubini's theorem. https://en.wikipedia.org/wiki/Fubini%27s_theorem. Accessed August 12, 2017.
- Wikipedia (2017d). Runge-Kutta methods. https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods. Accessed August 20, 2017.
- Wikipedia (2017e). Markov brothers' inequality. https://en.wikipedia.org/wiki/Markov_brothers%27_inequality. Accessed October 15, 2017.
- Wikipedia (2017f). Discrete cosine transform. https://en.wikipedia.org/wiki/Discrete_cosine_transform. Accessed September 2, 2017.
- Wikipedia (2017g). Hessian matrix. https://en.wikipedia.org/wiki/Hessian_matrix. Accessed November 26, 2017.
- Wilkins, G., & Gu, M. (2013). A modified Brent's method for finding zeros of functions. *Numerische Mathematik*, 123(1), 177-188.

24 Numerical Optimization

"If an algorithm converges unreasonably fast, it must be Newton's method" – John Dennis

24.1 Introduction

The emphasis is on unconstrained optimization algorithms, both local and global, implementations of which are described in detail. The reader has ideally taken a class in numerical methods.

24.2 Some General Ideas

Minimization problems have minor ill-conditioning (Press et al. 2007): for f with two continuous derivatives, Taylor expansion at x^* where $f'(x^*)=0$ for $x \in [x^* \pm \epsilon]$ leads to $|f(x) - f(x^*)| \leq O(\epsilon^2)$. I.e., near a minimum f can't distinguish x with relative error $O(\sqrt{\epsilon_{\text{machine}}})$ because perturbations of x of such magnitude lead to perturbations of f that are so small that many end up rounded to the same f -value. So don't expect or ask for better precision. Still, for f that don't satisfy Taylor series conditions, such as $f(x)=|x|$, this doesn't apply.

Root finding doesn't have this issue, and roots can be found to within $O(\epsilon_{\text{machine}})$. Verify that squaring the objective makes the gradient 0 at the root (by the chain rule of differentiation). This makes reduction of root finding to minimization seem unfavorable, but it really makes little difference because being able to find a root in the first place is the goal, particularly for multidimensional cases.

Minimization as a task is somewhat different from root finding. For roots x precision matters, and for optimization want any x such that $f(x)$ isn't too far from $f(x^*)$. So relative y precision is the important one, though typically settle for absolute y precision. This is easier to see for combinatorial optimization—the exact nature of the found object isn't important, and don't even have a way to evaluate it unlike for the y -value.

Reducing minimization to root finding also works—root-solve the gradient (assuming differentiable f and the ability to compute ∇f exactly). Gradients give extra information about the problem to improve conditioning. But this will only improve x -precision and not y -precision of the solution with regard to f , and need other information to distinguish minima from maxima.

An interesting question is whether x -precision is useful for termination conditions. Some ideas:

- In some cases it's the only option. E.g., golden section and coordinate descent (discussed later in the chapter) use it directly.
- Making it high is pointless at a local minimum, as discussed. But it may be useful for iterative algorithms like coordinate descent because smaller values may lead to premature termination. Golden section is special in that the default single precision becomes relevant only near a minimum, which works well.
- An algorithm should only check termination. The quality of the solution (proximity of root/gradient to 0 for respective equation solving/optimization) should be checked by the caller or a wrapper function with independent logic. Though can mostly avoid scale issues in gradient checks by proper scaling (Dennis & Schnabel 1996), coupling this logic to an algorithm is unnecessary. In particular, the user should check that the relative gradient $\frac{\|\nabla f\|_2}{\max(1, |y|)}$ is small enough. This works well even if f has a constant additive factor because precision checks are based on the f -value.
- Where y -precision works, x -precision won't be used—this seems to be a robust option. Another view is that use x -precision to detect stagnation and y -precision (estimated gradient in particular for some algorithms) to decide whether stopped successfully. But y -precision doesn't have condition problems near a minimum and seems just as robust. Using it seems to avoid the situation where Δx is small but ∇f large, which may create an issue with the alternative tests. In cases where must use x -precision, gradient check for 0 may offer extra termination, but in such case any good logic would quickly shrink x below termination precision.

All useful basic methods use some local information based on sampling f and its derivatives. So need explicit globalization logic for global search.

24.3 Single-variable Unimodal Function Minimization

Let f be unimodal on $[a, b]$, i.e., $\exists m$ such that at $x \leq m$ f is monotonically decreasing and for $x \geq m$ mono-

tonically increasing (Wikipedia 2017a). A unimodal f needn't be continuous, and that in 1D unimodal = quasiconvex (defined later in the chapter).

Golden section search maintains (a, b, c) such that $f(a) \leq f(b) \leq f(c)$:

1. Until convergence (usually of a to c)
2. Pick $x \in$ the larger of $[a, b]$, $[b, c]$
3. If $a \leq x \leq b \leq c$, use (a, x, b)
4. Else $a \leq b \leq x \leq c$, use (b, x, c)

The optimal x isn't in the middle, but 1 – the golden ratio into the larger interval from b . The algorithm terminates even with floating point arithmetic and stochastic discontinuous f .

```
template<typename FUNCTION> pair<double, double> minimizeGS(
    FUNCTION const& f, double xLeft, double xRight,
    double relAbsXPrecision = numeric_limits<double>::epsilon())
// don't want precision too low
assert(isfinite(xLeft) && isfinite(xRight) && xLeft <= xRight &&
       relAbsXPrecision >= numeric_limits<double>::epsilon());
double GR = 0.618, xMiddle = xLeft * GR + xRight * (1 - GR),
      yMiddle = f(xMiddle);
while(isELess(xLeft, xRight, relAbsXPrecision))
{
    bool chooseR = xRight - xMiddle > xMiddle - xLeft;
    double prevDiff = xRight - xLeft, xNew = GR * xMiddle + (1 - GR) *
        (chooseR ? xRight : xLeft), yNew = f(xNew);
    if(yNew < yMiddle)
    {
        (chooseR ? xLeft : xRight) = xMiddle;
        xMiddle = xNew;
        yMiddle = yNew;
    }
    else (chooseR ? xRight : xLeft) = xNew;
}
return make_pair(xMiddle, yMiddle);
```

Suppose $x > b$ without loss of generality. Let $w = \frac{b-a}{c-a}$ = the fraction of the way b is in $[a, c]$ and $z = \frac{x-b}{c-a}$

= the fraction of the way x is in $[a, c]$ after b . The next interval is $[a, x]$ of relative length $w + z$ or $[b, c]$ of relative length $1 - w$. The worst case is smallest when they are equal $\rightarrow z = 1 - 2w$. Assuming optimal choices in the past, x is the same fraction of the way in $[b, c]$ as b is in $[a, c]$, so $w = \frac{x-b}{c-b} = \frac{z}{1-w} \rightarrow w = 1 -$ the golden ratio. At the start, $b = c$ isn't optimal, but the intervals quickly become optimal. Due to geometrically shrinking intervals, the convergence is linear—i.e., need $O(\lg(1/\epsilon_{\text{absolute}}))$ evaluations.

If f isn't unimodal, this needn't work and at best will find a minimum in some unimodal region, which can even be worse than one of the endpoints. To work with nonunimodal f need global optimization strategies, which are theoretically no better than random search.

For stochastic f whose expected value is unimodal can get reasonably accurate averages, and minimize the resulting deterministic function, though this is a heuristic. The error estimate would only be accurate up to the expected noise level.

If know a starting point and not a containing interval, as for root finding, use bracketing with initial distance large enough to exceed possible noise. Here, this is well-scaled $\max(1, |x_i|)/10$. So assume unimodal f , and look for a three-point pattern $\{x_0, x_1, x_2\}$ such that $f(x_0) \geq f(x_1) \leq f(x_2)$:

1. Use the first two evaluations to determine the direction of the minimum
2. Double until overstep it
3. Apply golden section to the resulting interval

```
template<typename FUNCTION> pair<double, double> unimodalMinBracket(FUNCTION
    const& f, double x0, double fx, bool twoSided, double d, int maxEvals)
{
    assert(abs(d) > 0 && isfinite(x0) && isfinite(d)); // && maxEvals > 2?
    pair<double, double> best(x0, x0 + d);
    double fMin = f(x0 + d);
    maxEvals -= 2;
```

```

if(fx < fMin && twoSided) //check decrease direction if 2-sided
{
    d *= -1; //maximal pattern must be in the other direction
    fMin = fx;
    swap(best.first, best.second);
}
if (! (fx < fMin)) //if 1-sided can't increase current bracket
    while(d * 2 != d && maxEvals-- > 0) //handle d = 0, inf, and NaN
    {
        d *= 2;
        double xNext = x0 + d, fNext = f(xNext);
        if(fNext < fMin)
        { //shift
            best.first = best.second;
            best.second = xNext;
            fMin = fNext;
        }
        else
        { //found 3-point pattern, form interval
            best.second = xNext;
            break;
        }
    } //ensure sorted interval
if(best.first > best.second) swap(best.first, best.second);
return best;
}

template<typename FUNCTION> pair<double, double> minimizeGSBracket(
    FUNCTION const& f, double x, double fx = numeric_limits<double>::quiet_NaN(), bool twoSided = true, double step = 0.1,
    double relAbsXPrecision = defaultPrecEps, int bracketMaxEvals = 100)
{
    if(!isfinite(x)) return make_pair(x, fx);
    if(!isfinite(fx))
    {
        fx = f(x);
        --bracketMaxEvals;
    }
    pair<double, double> bracket = unimodalMinBracket(f, x, fx, twoSided,
        step * max(1.0, abs(x)), bracketMaxEvals), result = minimizeGS(f,
        bracket.first, bracket.second, relAbsXPrecision); //ensure nondecrease
    return result.second < fx ? result : make_pair(x, fx);
}

```

For the two-sided search and convex f , convergence is guaranteed \forall finite starting x and d . If f isn't convex, the algorithm always terminates, and the result is usually some local minimum, though can't make any guarantees.

24.4 Multidimensional Function Minimization—Introduction and Coordinate Descent

For $D > 1$, the optimal move direction is unknown because \exists a shrinking solution interval. All methods start with an initial guess, and guess the direction based on the values of nearby points or gradient information. No algorithm converges to a local optimum in predictably many iterations, even for sufficiently differentiable f (unless have a bound on the Lipschitz constant of f , and use very specific algorithms; see Beck 2017). Because can partition the optimization landscape into local-minimum regions, a minimum within such a region is what such local search usually finds.

A simple algorithm is **coordinate descent**:

1. Start with dimension $i = 0$ and a user-given starting point
2. Until converge or exceed the specified number of iterations, do 1D optimization on $x[i]$:
3. Find a bounding interval
4. Do golden section search
5. Set $i = (i + 1) \% D$

Instead of cycling through coordinates, it's better to randomize variable order before each cycle. For the bracketing keep starting steps for every dimension. Start with agnostic, well-scaled 0.1, then use $\min(0.1, \text{relative change in any coordinate in the last iteration relative})$. Stop iterating when this < the relative-absolute precision or make no further improvements. The algorithm is most efficient for f that can be computed incrementally, i.e., can update f incrementally from only $x[i]$ and some precomputed information. So a special interface is provided for that:

```
template<typename INCREMENTAL_FUNCTION> double unimodalCoordinateDescent(
    INCREMENTAL_FUNCTION &f, int maxEvals = 1000000,
    double xPrecision = highPrecEps)
{
    int D = f.getSize();
    Vector<int> order(D);
    for(int i = 0; i < D; ++i) order[i] = i;
    double y = f(f.getXi()), relStep = 0.1;
    while(f.getEvalCount() < maxEvals) //may be exceeded but ok
    { //use random order full cycles
        GlobalRNG().randomPermutation(order.getArray(), D);
        double yPrev = y, maxRelXStep = 0;
        for(int i = 0; i < D && f.getEvalCount() < maxEvals; ++i)
        {
            int j = order[i];
            f.setCurrentDimension(j);
            pair<double, double> resultJ = minimizeGSBracket(f, f.getXi(),
                y, true, relStep, xPrecision);
            maxRelXStep = max(maxRelXStep, abs(resultJ.first - f.getXi()) /
                max(1.0, abs(f.getXi())));
            f.bind(resultJ.first);
            y = resultJ.second;
            //done if no improvement in x or y
            if(maxRelXStep < xPrecision || !isELess(y, yPrev)) break;
            relStep = min(0.1, maxRelXStep); //take smaller first steps as converge
        }
    }
    return y;
}
```

A reasonable question is whether this is the best implementation strategy. An important use case is when the 1D step is done analytically, which is important for LSVM regression (see the "Machine Learning—Regression" chapter). The above implementation is close to this numerically. The heuristic for the control of the initial step size is reasonable and convergent. Compass search (discussed later in the chapter) uses a similar strategy. Instead, using the same 0.1 relative step size is inefficient in subsequent passes, and remembering individual step for every coordinate can shut down that coordinate prematurely with a 0 step. In general, optimization of some coordinates even by a small amount can enable optimization of other coordinates from 0 to a large change. Also, any implementation that doesn't do a complete 1D search but only one step of exponential/binary search runs into the above problem with remembered step sizes.

Can use a wrapper to work with general f :

```
template<typename FUNCTION> struct IncrementalWrapper
{
    FUNCTION f;
    mutable Vector<double> xBound;
    int i;
    mutable int evalCount;
public:
    IncrementalWrapper(FUNCTION const& theF, Vector<double> const& x0):
        f(theF), xBound(x0), i(0), evalCount(0) {}
    void setCurrentDimension(double theI)
    {
        assert(theI >= 0 && theI < xBound.getSize());
        i = theI;
    }
    int getEvalCount() const{return evalCount;}
    int getSize() const{return xBound.getSize();}
    Vector<double> const& getX() const{return xBound;}
```

```

double getXi() const {return xBound[i];}
double operator()(double xi) const
{
    double oldXi = xBound[i];
    xBound[i] = xi;
    double result = f(xBound);
    ++evalCount;
    xBound[i] = oldXi;
    return result;
}
void bind(double xi) const {xBound[i] = xi;}
};

template<typename FUNCTION> pair<Vector<double>, double>
unimodalCoordinateDescentGeneral(FUNCTION const& f, Vector<double> const&
x0, int maxEvals = 1000000, double xPrecision = highPrecEps)
{
    IncrementalWrapper<FUNCTION> iw(f, x0);
    double y = unimodalCoordinateDescent(iw, maxEvals, xPrecision);
    return make_pair(iw.xBound, y);
}

```

A seemingly undesirable feature of this wrapper is that must remember the current values, instead of leaving those to the algorithm. But need this for reducing to 1D search as a black box, so can't improve the object-oriented design.

This is applicable to all cases with $O(D \lg(1/\epsilon_{\text{golden section}}))$ time per iteration. But convergence even to a local minimum isn't guaranteed or can be very slow unless special cases apply because the f -value can improve in mixed but not coordinate directions. Practical performance is often good, particularly when the 1D search has an analytic solution—e.g., for lasso regression (see the “Machine Learning—Regression” chapter).

For nondifferentiable f convergence isn't guaranteed due to the possibility of being stuck. E.g., in 2D consider $f(x) = \frac{1}{2} \max(\|x - c\|_2^2, \|x + c\|_2^2)$ where $c = (-1, 1)$. Then \forall point (a, a) for $a \neq 0$, coordinate directions aren't descent (Conn et al. 2009). In fact, this function will theoretically defeat any incremental algorithm that takes 1D steps. So incremental optimization only works well for special f .

Coordinate descent is probably more anytime than other algorithms in a sense that it can be stopped after any number of full cycles and still give good values \forall variable. For separable f it converges in a single cycle.

24.5 Line Search

If f is differentiable, at a local minimum $\nabla f = 0$, and any direction d such that $d^\top \nabla f < 0$ is a **descent direction**, moving into which by some step reduces the value of f . Gradient-based algorithms repeatedly do 1D search in a picked descent direction until convergence, i.e., $\nabla f < \epsilon$ for some ϵ . But this can fail if estimate ∇f by finite differences, so it's more robust to stop when make no progress in f -value. Such **line search** theoretically finds the exact minimizer on the line into any direction. But often use approximate minimization that gives sufficient decrease.

Wolfe conditions for convergent line search in a descent direction d require that for step s and some constants $0 < c_1 < c_2 < 1$, the step isn't too:

- Long— $f(x + sd) \leq f(x) + c_1 sd \nabla f(x)$ —the **Armijo/sufficient descent condition**
- Short— $\nabla f(x + sd) \geq c_2 d^\top \nabla f(x)$ —the **curvature condition**

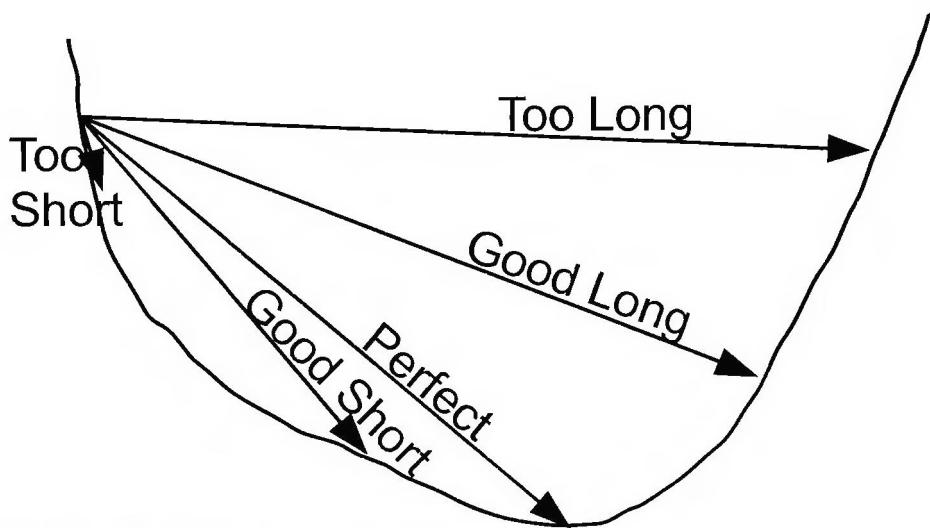


Figure 24.1: Various types of step sizes

Such c_1 and c_2 always exist given a descent direction (Nocedal & Wright 2006). **Zoutendijk's theorem:** Line search converges to a local minimum if:

- f is continuously differentiable, bounded below, and ∇f is Lipschitz
- Descent directions are picked whose angle from the gradient is bounded away from 90 degrees
- Searches satisfy Wolfe conditions

Consider **backtracking**, which worked for nonlinear equations: halve the initial step s until satisfy sufficient descent with maybe $c_1 = 10^{-4}$. Hope that it starts with s_0 large enough to satisfy the second condition and halves until the first one is satisfied. Experimentally, for algorithms that produce well-scaled directions such as Newton's (discussed later in this chapter), the initial step must asymptotically be 1 to have super-linear local convergence (Nocedal & Wright 2006). But backtracking from $s = 1$ has problems:

- It doesn't satisfy the curvature condition in general (need strong convexity of f among other things; Boyd & Vandenberghe 2004), so Zoutendijk's theorem doesn't apply.
- It may not be large enough for common algorithms other than Newton's method (Nocedal & Wright 2006).

Backtracking has own convergence theory by preventing s from getting too small (Dennis & Schnabel 1996), but this doesn't matter in practice. Still, a natural first step to a better algorithm is to double s if the first tried value satisfies sufficient descent. Stop when no longer have sufficient descent, or don't get f -value improvement. Theoretically this is almost equivalent to running a regular backtracking from a larger step. Failure to get simple decrease means that a local minimum is near, which in turn means that a nearby step satisfies the curvature condition. In particular, the selected step is at most twice as large as what would jump to a local minimum.

Looking at the gradient information adds an extra stopping condition that the directional derivative < 0 (Nocedal & Wright 2006). Go for **strong Wolfe conditions** with curvature upgraded to $|\nabla f(x+sd)| \geq c^2 d \|\nabla f(x)\|$, and use $c_2 = 0.1$ in all cases for a good quality search (Nocedal & Wright 2006 recommend 0.9 for quasi-Newton algorithms, but with finite-difference gradients this stricter value seems better).

The currently best algorithm for line search is due to (Moré & Thuente 1994). My implementation is based on their logic but uses the algorithmic framework of Nocedal & Wright (2006), which is based on an older algorithm.

Let the line search function $\phi(s) = f(x + sd)$; $\phi'(s) = f'(x)d$ is the directional derivative in the direction d . So can estimate these directly without reevaluating gradients. Make a new function $\psi(s) = \phi(s) - \phi(0) - c_1 \phi'(0)s$; $\psi'(s) = \phi'(s) - c_1 \phi'(0)$. Given an interval where a satisfactory point is present, use the following rules to shrink it:

1. Let $s = (s_{\text{low}} + s_{\text{high}})/2$
2. (Case 1) If $\psi(s) > \psi(s_{\text{low}})$, set $s_{\text{high}} = s$
3. (Case 2) Else if $\psi'(s)(s_{\text{low}} - s) > 0$, set $s_{\text{low}} = s$
4. (Case 3) Else if else if, set $s_{\text{low}} = s$ and $s_{\text{high}} = s_{\text{low}}$

For s_{low} and s_{high} the low is the current best, and they needn't be in order. This provably maintains the interval with a satisfactory point in exact arithmetic (Moré & Thuente 1994). Also use a minor modification of this algorithm for initialization:

1. Start with $s = 1$, $s_{\text{low}} = 0$, and $s_{\text{high}} = \infty$

2. While hit case 2 use $s = 2s$.

3. Continue with the regular algorithm

Implementation details:

- If case 1 fails, check if s satisfies strong Wolfe conditions to avoid unnecessary further shrinking. Numerically the bracket may become too small.
- Unlike for equation solving with backtracking, use y -precision to terminate. A simple criteria is when $|$ the directional derivative $\times (s_{\text{high}} - s_{\text{low}})|$ is too small to affect the f -value noticeably.
- Work directly with f without defining ψ or ϕ .

```
template<typename FUNCTION, typename DIRECTIONAL_DERIVATIVE> bool
strongWolfeLineSearchMoreThuente(FUNCTION const& f,
Vector<double> const& gradient, DIRECTIONAL_DERIVATIVE const& dd,
Vector<double>& x, double& fx, int maxEvals, Vector<double> const& dx,
double yEps)
{
    double dd0 = dotProduct(dx, gradient), sLo = 0, fLo = fx, s = 1,
           sHi = numeric_limits<double>::infinity(), temp = 0.0001 * dd0;
    if(!isfinite(dd0) || dd0 >= 0) return true;
    while(maxEvals > 0 && isfinite(s) &&
          (!isfinite(sHi) || isELess(fx + dd0 * abs(sHi - sLo), fx, yEps)))
    {
        double fNew = f(x + dx * s);
        if(!isfinite(fNew)) break;
        --maxEvals;
        if(fNew - s * temp > fLo - sLo * temp) sHi = s;//case 1
        else
        {
            double ddS = dd(x + dx * s, dx);
            maxEvals -= dd.fEvals();
            if(abs(ddS) <= -0.1 * dd0)//check for early termination
            {
                sLo = s;
                fLo = fNew;
                break;
            }
            if((ddS - temp) * (sLo - s) <= 0) sHi = sLo;//case 3
//case 2 and 3
            sLo = s;
            fLo = fNew;
        }
        if(isfinite(sHi)) s = (sLo + sHi)/2;//zooming
        else s *= 2;//case 2 init doubling
    }
    if(sLo > 0)
//any non-0 guaranteed to have sufficient descent
    {
        x += dx * sLo;
        double fxFirst = fx;
        fx = fLo;
        return !isELess(fx, fxFirst, yEps);//must make good progress
    }
    return true;
}
```

This algorithm is robust. Possible outcomes:

- Find a satisfactory point
- Initialization keeps doubling until hit ∞ or exceed the evaluation budget— f has no finite minima
- The interval becomes too small numerically—it still contains a good point, but don't gain from further shrinking
- The found $s \approx 0$ —close to a local minimum, or d is almost orthogonal to the steepest descent direction

The algorithm closest to exact line search is golden section. Unfortunately it:

- Works with quasiconvex f , which the line function needn't be. E.g., it can increase in the sampled

inside points or have several convex subregions, and the algorithm will converge to any of these.

- Doesn't guarantee satisfaction of either strong Wolfe condition—both should be met when f is quasiconvex because the exact minimum is found, but they may not be if not quasiconvex, or a in sufficient descent is too large.

But experimentally it works better with finite-difference gradients. In large D gradient estimation takes many more evaluations than line search. With analytical gradients, it's probably not needed. Usually it will find a strong Wolfe point directly but may not. If it:

- Finds a local minimum, curvature will be satisfied
- Finds a distant local minimum, sufficient descent may not be satisfied
- Fails due to nonconvexity, at least get an approximate step

So run strong Wolfe search on the result to avoid issues.

```
template<typename FUNCTION1D> struct EvalCountWrapper
{ // to keep track of used evaluations, which golden section doesn't
    FUNCTION1D f;
    mutable int evalCount;
    EvalCountWrapper(FUNCTION1D const& theF) : f(theF), evalCount(0) {}
    double operator()(double x) const
    {
        ++evalCount;
        return f(x);
    }
};

template<typename FUNCTION, typename DIRECTIONAL_DERIVATIVE> bool
goldenSectionLineSearch(FUNCTION const& f, Vector<double> const& gradient,
DIRECTIONAL_DERIVATIVE const& dd, Vector<double>& x, double& fx,
int& maxEvals, Vector<double> const& dx, double yEps, bool useExact = true)
{
    if(!norm(dx) > 0) return false; // this way handle NaN also
    double step = 1; // well-scaled for most algorithms
    if(useExact)
    {
        EvalCountWrapper<ScaledDirectionFunction<FUNCTION> > f2(
            ScaledDirectionFunction<FUNCTION>(f, x, dx));
        pair<double, double> result = minimizeGSBracket(f2, f2.f.gets0(), fx,
            false);
        maxEvals -= f2.evalCount; // may be exceeded but ok
        if(isELess(result.second, fx, yEps))
            step = result.first - f2.f.gets0();
    } // ensure strong Wolfe conditions
    return strongWolfeLineSearchMoreThuente(f, gradient, dd, x, fx, maxEvals,
        dx * step, yEps);
}
```

24.6 Line-search-based Algorithms

The simplest line-search-based algorithm is **gradient descent**. Given a starting point x , until convergence it performs a line search in the direction $-\nabla f$. By Zoutendijk's theorem, it converges to a local minimum with a line search that satisfies Wolfe conditions. The convergence is linear, with s a constant that depends on the condition number of $\nabla^2 f$, so it's very slow if this number is large (Nocedal & Wright 2006). In my experiments it often doesn't converge or gets low precision with the evaluation budget (by default 10^6 for most algorithms here).

Theorem (Nocedal & Wright 2006): Let B be a positive definite matrix and $\exists M > 0$ such that $\kappa(B) < M$ (κ is the condition number in any norm). Then $-B^{-1}\nabla f$ is a descent direction.

In particular, **Newton's method** uses the direction $-(\nabla^2 f)^{-1}\nabla f$ with step = 1 (Nocedal & Wright 2006). Also have scale invariance. So with strong Wolfe line search it converges to a local minimum if the Hessian is always positive definite. The convergence is quadratic in the neighborhood of the solution. But:

- The Hessian needn't be positive definite, in which case it can fail to lead to a descent direction or even not be invertible. With any correction, such as adding a multiple of identity or adjusting eigenvalues to be positive, can't solve for s without further implicit or explicit assumptions, so lose scale invariance.

- The Hessian may become ill-conditioned and give directions where the angle with the gradient is arbitrarily close to 90 degrees or even numerically slightly larger.
- Each iteration takes $O(D^3)$ time.

BFGS improves Newton's method by updating the Hessian from function and derivative values, like Broyden's method for solving equations. Newton's method for optimization is the equation solving method for $\text{gradient} = 0$, but unlike the Jacobian, the Hessian is symmetric and positive definite, which a good update rule will take into account. So the update rule ensures that H_{k+1} is positive-definite if H_k is.

$$H_{k+1} = V_k H_k V_k^T + p_k s_k^2, \text{ where:}$$

- $H = B^{-1}$
- $p_k = \frac{1}{y_k s_k}$
- $V_k = I - p_k \times (y_k \otimes s_k)$
- $s_k = x_{k+1} - x_k$
- $y_k = \nabla f_{k+1} - \nabla f_k$

For proofs see Nocedal & Wright (2006) and Dennis & Schnabel (1996). A finite-difference approximation for H_0 in general needn't be positive-definite, and any correction loses scale invariance. Also, starting with any positive-definite matrix works, unlike for Broyden's method, and the calculation would be very expensive for medium D . So the initial H_0 is typically I , possibly scaled; in all cases must ensure positive-definiteness (B is when H is).

Choosing the scale leads to the question of what the first step should be. Some approaches:

- The steepest descent step is reasonably correct in scale, and line search will adjust the resulting step size to be longer or shorter. But the gradient only gives a direction, and don't know for how long in that direction it will decrease. Also too small Δx can result in numerical issues with too small Δf , so that doubling won't happen. I.e., with convergence $\nabla f \rightarrow 0$, and in case of $\|\Delta x\|_2 = \|\nabla f\|_2$ relative y can jump below $\epsilon_{\text{machine}}$ due to ill-conditioning of optimization near the minimum. This would be an issue for starting at small-gradient points. But assuming only 2nd derivative information is relevant, the directional derivative increases linearly with an unknown constant, so after a step of size $O(\|\nabla f\|_2)$ it will become 0. Not scaling implicitly assumes this constant = 1. Scaling the gradient, as discussed later, is likely to improve this still, but for first steps the next option is safer.
- Given a user-based estimate of the size of the first step s , scale the gradient direction by $s/\|\nabla f\|_2$ (Nocedal & Wright 2006). A reasonable strategy is using $s = \max(1, \|x\|_2)/10$ or something similar to protect again large x values and be far above the ill-conditioned step size; 10 is a logical value that seems to lead to a reasonable x -change. This or something similar is also a good choice for first steps for methods that don't use gradients, though it won't work for restarts because such step is likely to be too large. But for the latter case another strategy, such as taking the size of the last improving change, works well.

For methods with state like BFGS, H_0 matters for other steps, so improve it for only the updates. After computed the step (scaled independently), but before the update, assume $H_0 = \frac{I}{p_k y_k^2}$, where k is the last step. This works well per Nocedal & Wright (2006); see Dennis & Schnabel (1996) for some theoretical justification.

L-BFGS reduces memory use by only applying m recent updates, as for limited-memory Broyden. Some differences:

- A descent direction is guaranteed because apply positive definite updates
- The update is of rank 2, and a special formula has been developed

L-BFGS stores $j \leq m$ last s_i and y_i in a queue and uses them to compute H_k recursively. BFGS would equivalently keep the entire search history. So another benefit is resistance to accumulation of ill-conditioned state—i.e., B might stop being positive-definite numerically. A general problem is that any optimization method that updates some state between iterations risks making that state ill-conditioned. It's often casually stated in the literature that experimentally BFGS and L-BFGS perform similarly (e.g., see Liu & Nocedal 1989 for some comparisons), though only BFGS inherits the superlinear convergence of the Newton's method. With $m = \infty$ the two are equivalent, but small m wins in practice per below. So L-BFGS is the method of choice, even though BFGS is often used and mentioned.

Experimentally, $m \in [3, 20]$, particularly $m = 8$, is a good choice (Nocedal & Wright 2006). Because the queue uses a vector for storage, to avoid waste make m a power of two, so 4, 8, and 16 are good values.

Compute $d = H_k \nabla f$ without explicitly forming H . Because H_{k-j} is a diagonal matrix, can expand the update

formula to calculate d from H_{k-j} and the stored s_i and y_i , so that each iteration uses $O(nm)$ time and space (Nocedal & Wright 2006):

1. $d \leftarrow \nabla f$
2. $\forall i \in [k-1, k-j]$
3. Store $a_i = p_i s_i d$
4. $d := a_i y_i$
5. $d *= H_{k-j}$
6. $\forall i \in [k-j, k-1]$
7. $d += s_i (a_i - p_i y_i d)$

As Newton's method, L-BFGS produces well-scaled directions, so line search starts with step = 1. Some extra robustness logic seems needed, beyond what is suggested in Nocedal & Wright (2006), because here the emphasis is on using finite-difference gradients instead of the actual ones, which are traditionally assumed to be available. Even when have analytical gradients, for a very ill-conditioned Hessian, nothing helps because pure gradient descent will be too slow, and Newton directions may need such large angle that will have corruption by numerical noise. For numerical gradients the problem is much worse—the basic algorithms often can't advance beyond some some precision, such as 2–4 decimal digits of accuracy. But this happens within a very small number of function evaluations (< 10000) in my experiment. So given that a hybrid algorithm (discussed later in the chapter) is a method of choice in practice, technically it's hard to argue against L-BFGS as is.

But a simple fix is to restart the algorithm when line search fails by purging history one item at a time. After all history is purged, this becomes gradient descent, which is most numerically robust against poor gradient calculation. If this doesn't work either, with the initial step size = the size of the last step, the algorithm is terminated. For many problems this gives much higher precision, though at the expense of exhausting the entire evaluation budget.

```
template<typename FUNCTION, typename GRADIENT, typename DIRECTIONAL_DERIVATIVE>
pair<Vector<double>, double> LBFGSMinimize(Vector<double> const& x0,
FUNCTION const& f, GRADIENT const& g, DIRECTIONAL_DERIVATIVE const& dd,
int maxEvals = 1000000, double yPrecision = highPrecEps,
int historySize = 8, bool useExact = true)
{
    typedef Vector<double> V;
    Queue<pair<V, V>> history;
    pair<V, double> xy(x0, f(x0));
    V grad = g(xy.first), d;
    int D = xy.first.getSize(), gEvals = g.fEvals(D);
    maxEvals -= 1 + gEvals;
    double lastGoodStepSize = max(1.0, norm(x0))/10;
    while(maxEvals > 0)
    {
        //backtrack using d to get sufficient descent
        if(history.getSize() == 0) d = grad * (-lastGoodStepSize/norm(grad));
        pair<V, double> xyOld = xy;
        if(goldenSectionLineSearch(f, grad, dd, xy.first, xy.second, maxEvals,
            d, yPrecision, useExact))
        {
            //failure
            if(history.getSize() > 0)
                //try to "restart" by purging history one at a time
                history.pop();
            continue;
        }
        else break; //gradient descent step failed
    }
    else lastGoodStepSize = norm(xy.first - xyOld.first); //success
    if((maxEvals -= gEvals) < 1) break; //out of evals
    V newGrad = g(xy.first);
    if(history.getSize() >= historySize) history.pop();
    history.push(make_pair(xy.first - xyOld.first, newGrad - grad));
    //double recursion" algorithm to update d
    d = grad = newGrad;
    Vector<double> a, p;
    int last = history.getSize() - 1;
```

```

for(int i = last; i >= 0; --i)
{
    double pi = 1/dotProduct(history[i].first, history[i].second),
           ai = dotProduct(history[i].first, d) * pi;
    d -= history[i].second * ai;
    a.append(ai);
    p.append(pi);
    //initial Hessian is scaled diagonal
    d *= 1/(dotProduct(history[last].second, history[last].second) *
             p[last]);
    for(int i = 0; i < history.getSize(); ++i)
    {
        double bi = dotProduct(history[i].second, d) * p[last - i];
        d += history[i].first * (a[last - i] - bi);
    }
    d *= -1;
}
return xy;
}

```

L-BFGS converges linearly to a local minimum if (Liu & Nocedal 1989):

- f is convex and twice continuously differentiable
- $\forall z \in \mathbb{R}^n \exists m, M > 0$ such that $m\|z\|^2 \leq z^\top \nabla^2 f z \leq M\|z\|^2$
- $\kappa(H_0) < \infty$
- The line search satisfies Wolfe conditions

These are just moral support bounds. In practice the algorithm is very robust and usually converges even if f doesn't satisfy the conditions at every point. But even if it does, \exists a useful bound on the number of iterations.

If the gradient is unavailable, compute it by finite differences. Experimentally, this performs well as long as the estimation isn't too inaccurate, but \exists supporting theory.

```

template<typename FUNCTION> struct GradientFunctor
{
    FUNCTION f;
    GradientFunctor(FUNCTION const& theF): f(theF) {}
    Vector<double> operator() (Vector<double> const& p) const
    {
        return estimateGradientCD(p, f);
    }
    int fEvals(int D) const {return 2 * D;}
};

template<typename FUNCTION> struct DirectionalDerivativeFunctor
{
    FUNCTION f;
    DirectionalDerivativeFunctor(FUNCTION const& theF): f(theF) {}
    double operator() (Vector<double> const& x, Vector<double> const& d)
    {
        const return estimateDirectionalDerivativeCD(x, f, d);
    }
    int fEvals() const {return 2;}
};

```

For finite-difference gradients numerical noise will eventually give random directions, which limits the ultimate attainable y -precision more than anything else due to early termination. Possible outcomes for a convergent algorithm:

1. f -value is locally optimal to some precision
2. ∇f is too small
3. Run out of evaluations
4. The gradient error is so large that the estimated steepest descent direction isn't descent, and stop prematurely

For (2) divide the gradient by $\max(1, |f(x)|)$ (Dennis & Schnabel 1996). Because of the "1" it gives scaled gradient estimates only for large f values. A danger is that with additive constant factors in $f(x)$, this will underestimate, but no better solutions seems possible.

Typically (4) doesn't happen. Given (1), in exact arithmetic (2) is also the case. But in my tests it's not always the case, i.e., scaled ∇f may be on the order of 0.01 but not close to what one may consider a convergent precision. So deciding convergence is problematic based on just the finite difference gradient. A more reliable indicator is that line search with estimated steepest descent direction fails, as implemented.

The test functions used here are from Dennis & Schnabel (1996) and Moré et al. (1981). Only those are used that have a known optimal value in all dimensions and don't seem to have local minimums (some exclusions are "Trig", "Wood", and "Guld R&D"). But all of these are least-squares problems, and may not be representative because least-squares has a special structure.

Gradient-based methods don't make sense in 1D due to needing line searches which are based on golden section or other approximate search methods, which can be run directly. A primary advantage of a gradient is given a descent direction—but in 1D can just cheaply try both. But can cheaply implement central-difference Newton method (with backtracking) by having the 2nd derivative estimate reuse evaluations of the 1st derivative estimate.

24.7 Some Derivative-free Methods

For a differentiable f , picking a descent direction is easy even without knowing ∇f because as long as $d\nabla f \neq 0, -d$ or d is a descent direction. But this doesn't hold for nondifferentiable f .

Nelder-Mead algorithm is among the most successful methods for black-box f , particularly for $D < 10$, where it's reported to be very efficient. It works from an initial **simplex**, which is a hyper-triangle of $D + 1$ points and consists of a guess point and its displacements into each orthogonal direction; here use $\text{uniform01} \times s \max(1, |x_i|)/10$, where $s =$ the initial step size and by default = 1. The important points are those with the best, the worst, and the second worst values. At every iteration, the algorithm pulls the simplex from the worst to the best using:

- **Scaling**—replace the worst by a convex combination of itself and the centroid of the other points. When the scale factor = -1, the result is a **reflection**.
- **Shrinking**—move all points $\frac{1}{2}$ way toward the best.

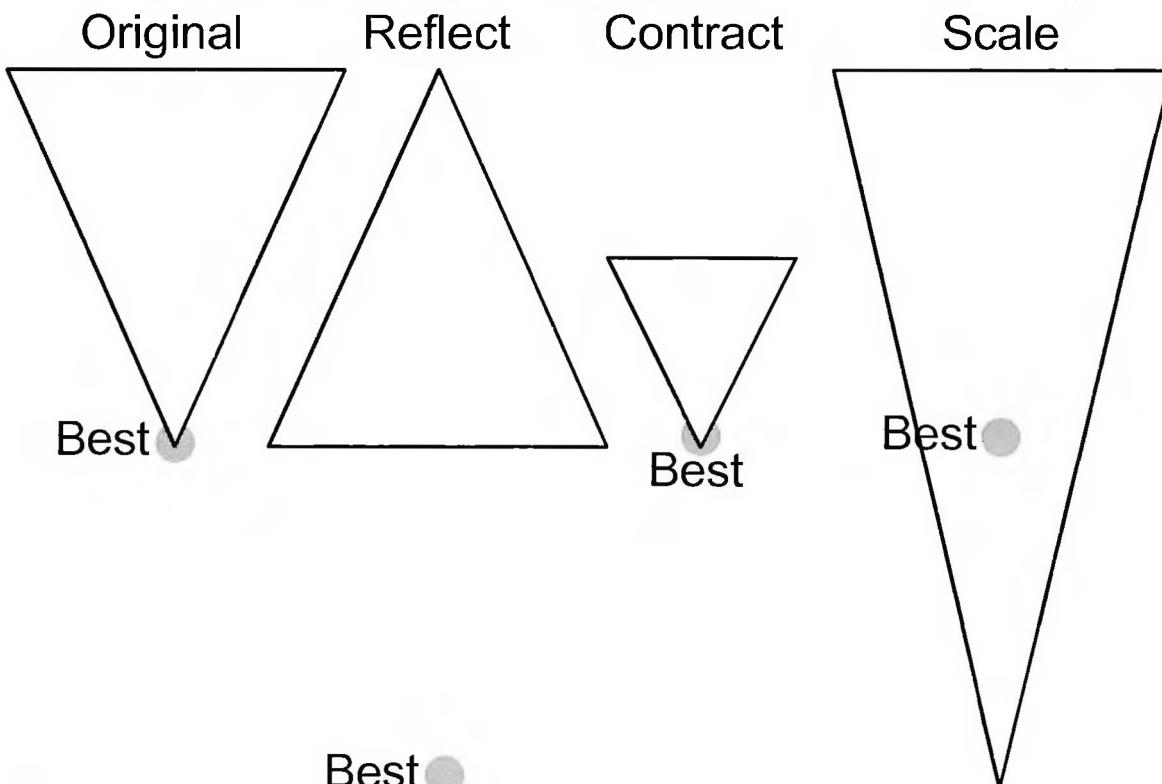


Figure 24.2: Possible transformations of the current simplex

```
template<typename FUNCTION> struct NelderMead
{
    FUNCTION f;
    int D;
    Vector<double> vertexSum; //incremental centroid
    typedef pair<Vector<double>, double> P;
    Vector<P> simplex;
    double scale(P& high, double factor, int& maxEvals)
    {
        P result = high;
        //affine combination of the high point and the
    }
}
```

```

//centroid of the remaining vertices
//centroid = (vertexSum - high)/D and
//result = centroid * (1 - factor) + high * factor
double centroidFactor = (1 - factor)/D;
result.first = vertexSum * centroidFactor +
    high.first * (factor - centroidFactor);
result.second = f(result.first);
--maxEvals;
if(result.second < high.second)
{ //accept scaling if improving
    vertexSum += result.first - high.first;
    high = result;
}
return result.second;
}
public:
NelderMead(int theD, FUNCTION const& theFunction = FUNCTION()):
    D(theD), f(theFunction), simplex(D + 1){assert(D > 1);}
};

```

The main logic:

1. Initialize the simplex
2. Until convergence
3. Find the best, the 2nd best, and the worst
4. Converged if the best ≈ the worst
5. Try to reflect
6. If the reflected point is better than the best, try to scale the new best by 2
7. Else try to scale by ½
8. If the result is worse than the next worst, shrink all points toward the best

For efficiency, if don't need high accuracy, use $\epsilon = 0.001$ or so with a properly scaled f . If the best point converges linearly and gets updated every D iterations, need $O(D \log(1/\epsilon))$ evaluations. But because by default want high precision, allow 10^6 evaluations.

```

P minimize(Vector<double> const& initialGuess, int maxEvals = 1000000,
    double yPrecision = highPrecEps, double step = 1)
{//initialize the simplex
    vertexSum = initialGuess;
    for(int i = 0; i < D; ++i) vertexSum[i] = 0;
    for(int i = 0; i <= D; ++i)
    {
        simplex[i].first = initialGuess;
        if(i > 0) simplex[i].first[i - 1] += GlobalRNG().uniform01() *
            step * max(1.0, abs(initialGuess[i - 1]))/10;
        simplex[i].second = f(simplex[i].first);
        --maxEvals;
        vertexSum += simplex[i].first;
    }
    for(;;)
    {//calculate high, low, and nextHigh, which must be all different
        int high = 0, nextHigh = 1, low = 2;
        if(simplex[high].second < simplex[nextHigh].second)
            swap(high, nextHigh);
        if(simplex[nextHigh].second < simplex[low].second)
        {
            swap(low, nextHigh);
            if(simplex[high].second < simplex[nextHigh].second)
                swap(high, nextHigh);
        }
        for(int i = 3; i <= D; ++i)
        {
            if(simplex[i].second < simplex[low].second) low = i;
            else if(simplex[i].second > simplex[high].second)
            {

```

```

        nextHigh = high;
        high = i;
    }
    else if(simplex[i].second > simplex[nextHigh].second)
        nextHigh = i;
} //check if already converged
if(maxEvals <= 0 || !isELess(simplex[low].second,
    simplex[high].second, yPrecision)) return simplex[low];
//try to reflect
double value = scale(simplex[high], -1, maxEvals);
//try to double if better than low
if(value <= simplex[low].second) scale(simplex[high], 2, maxEvals);
else if(value >= simplex[nextHigh].second)
{//try reflected/unreflected halving if accepted/rejected value
    double yHi = simplex[high].second;
    if(scale(simplex[high], 0.5, maxEvals) >= yHi)
        {//contract all to get rid of the high point
        vertexSum = simplex[low].first;
        for(int i = 0; i <= D; ++i) if(i != low)
        {
            vertexSum += simplex[i].first = (simplex[i].first +
                simplex[low].first) * 0.5;
            simplex[i].second = f(simplex[i].first);
            --maxEvals;
        }
    }
}
}

```

Nelder-Mead picks good directions for typical f and always converges to something that needn't be a local minimum. It can stall because the simplex volume becomes too small and can't recover, particularly for $D > 10$. The simplest way to handle this is by restarting at the result until stop making progress. The maximum number of restarts is 10 by default. If convergence needs more, probably Nelder-Mead isn't effective for the problem. But even if unable to converge, it usually gives substantial f -value gain after relatively few evaluations.

```
P restartedMinimize(Vector<double> const& initialGuess,
    int maxEvals = 100000, double yPrecision = highPrecEps,
    int maxRepeats = 10, double step = 1)
{
    P result(initialGuess, numeric_limits<double>::infinity());
    while(maxRepeats--)
    {
        double yOld = result.second;
        result = minimize(result.first, maxEvals, yPrecision, step);
        if(!isELess(result.second, yOld, yPrecision)) break;
    }
    return result;
}
```

Subsequent restarts are more efficient when near a solution. A good hybrid method is to run L-BFGS first. Usually get to a solution quickly and then improve precision. But because Nelder-Mead uses $O(D^2)$ memory, use it until $D = 200$ or so.

```

template<typename FUNCTION> pair<Vector<double>, double> hybridLocalMinimize(
    Vector<double> const& x0, FUNCTION const& f, int maxEvals = 1000000,
    double yPrecision = highPrecEps)
{
    GradientFunctor<FUNCTION> g(f);
    DirectionalDerivativeFunctor<FUNCTION> dd(f);
    int D = x0.getSize(), LBFGSevals = D < 200 ? maxEvals/2 : maxEvals;
    pair<Vector<double>, double> result = LBFGSMinimize(x0, f, g, dd,
        LBFGSevals, yPrecision);
    if(D > 1 && D < 200)

```

```

{
    int nRestarts = 30;
    NelderMead<FUNCTION> nm(x0.getSize(), f);
    result = nm.restartedMinimize(result.first,
        (maxEvals - LBFGSEvals)/nRestarts, highPrecEps, nRestarts);
}
return result;
}

```

This is probably the most effective general black-box minimizer. Neither component gives any theoretical guarantees—even if theoretical conditions for L-BFGS hold, still have an approximation due to finite-difference gradients. But in practice L-BFGS will quickly find the optimum, and if it fails Nelder-Mead will likely succeed from the improved f -value.

For differentiable f with unknown analytical derivatives perhaps the only methods that have theoretical convergence as implemented are **pattern search methods** (Nocedal & Wright 2006; Kolda et al. 2003, Conn et al. 2009; Audet & Hare 2017). A particular algorithm is **compass search**:

1. Start from some initial step size s , here $0.1\max(1, \|x\|_2)$
2. Until convergence when s becomes too small based on x -precision ϵ
3. \forall coordinate direction
4. If improve for $s \times$ the direction (discussed later)
5. Accept the move
6. Double s
7. Break
8. Halve s

Compass seems particularly efficient for incrementally computable f due to working with one dimension at a time. A typical implementation would try all directions in random order in a cycle, but because an improving direction can be found quickly, permutation generation becomes the bottleneck step. So a simple solution is to permute every 2D evaluations. But because need to poll all dimensions, only permute after a cycle is complete.

This algorithm is interesting theoretically, at least because convergence is guaranteed for differentiable f (Audet & Hare 2017) and a unique backtracking strategy. Unfortunately practical performance is poor—it often doesn't converge due to running out of evaluations, so the implementation isn't presented. Coordinate descent seems to be a better choice in practice for incremental f .

For large D compass and coordinate descent perform much worse than L-BFGS, so no point hybridizing. These should only be used for incrementally computable f (perhaps even as a hybrid) where they are much more efficient. Also compass works efficiently with box constraints, allowing incremental checks. It's easy to extend to general black-box constraints using rejection.

Specific f from important problems are usually minimized using customized methods such as **back-propagation** for neural networks (see the “Machine Learning—Classification” chapter).

Function	Compass		UnimodalCD		NelderMead		RestartedNelder		LBFGSMinimize		HybridLocalMinin	
	Err	Evals	Err	Evals	Err	Evals	Err	Evals	Err	Evals	Err	Evals
ExtendedRosenbrock2	-11.16	80746	-13.66	271	-13.41	193	-13.60	384	-8.08	47371	-13.66	47705
ExtendedPowellSingular4	-4.75	1000008	-4.01	1000010	-13.22	729	-13.60	1138	-4.83	1000052	-13.39	500838
HelicalValley	-11.60	27125	-11.39	398978	-13.39	489	-13.44	834	-13.03	8240	-13.87	8913
VariableDimensionF2	-15.65	200	-13.90	3738	-13.52	251	-13.60	402	-14.19	1032	-14.19	1372
LinearFFullRank2	-13.65	425	-14.47	337	-13.39	233	-13.60	399	-15.65	1000000	-15.65	500347
BrownBadScaled	-7.80	1534	-8.61	473	-13.59	435	-13.98	616	-14.00	1399	-14.15	1852
Beale	-12.66	2911	-12.60	22494	-13.92	216	-13.82	618	-10.03	1996	-13.78	2339
BiggsExp6	-2.23	1000012	-2.23	1000022	-3.69	2474	-2.22	1624	-4.61	1000055	-13.35	503479
ExtendedRosenbrock10	-5.60	1000020	-1.82	1000055	-10.82	6992	-13.19	13000	-8.45	64759	-12.98	71703
ExtendedPowellSingular12	-3.58	1000024	-3.26	1000048	0.00	8256	-11.45	27279	-4.62	1000024	-9.10	543961
VariableDimensionF10	-11.59	334075	-11.76	749456	-11.63	8405	-13.07	13357	-13.33	5741	-13.40	8140
LinearFFullRank10	-13.38	7997	-13.90	1921	-12.81	4199	-13.41	7230	-15.65	397	-15.65	3019
ExtendedRosenbrock30	-0.54	1000060	-0.63	1000067	-0.58	35433	-7.38	463063	-7.52	134706	-10.98	259814
ExtendedPowellSingular32	-2.52	1000064	-2.58	1000096	0.00	159722	-5.37	618889	-4.61	1000075	-7.19	685426
VariableDimensionF30	0.00	1000060	0.00	1000088	0.00	39674	-6.48	627111	-11.57	33397	-12.21	86511
LinearFFullRank30	-13.06	48841	-14.08	6151	0.00	49749	-11.30	374158	-5.65	601	-15.65	10209
ExtendedRosenbrock100	0.00	1000200	-0.11	1000204	-0.18	1000200	-3.49	1000203	-7.27	80115	-8.02	562807
ExtendedPowellSingular100	-1.30	1000200	-1.78	1000222	0.00	1000200	-4.37	1000201	-4.85	1000105	-5.41	1000052
VariableDimensionF100	0.00	1000200	0.00	1000246	0.00	334728	-1.41	1000200	-8.93	154946	-9.52	617026
LinearFFullRank100	-15.65	34153	-13.46	21101	0.00	1000200	-9.98	700649	-15.65	1306	-15.65	34638
Average Ranks	6.6	7.0	6.7	7.6	7.2	3.5	4.8	5.5	3.7	5.4	1.9	5.7

Figure 24.3: Performance of some of the main methods on several problems (among many more tested), with different not-too-large dimensions. The error is relative-absolute in decimal digits from the correct answer 0. The hybrid algorithm is clearly the best black-box method.

Function	Compass		UnimodalCD		LBFGSMinimize	
	Err	Evals	Err	Evals	Err	Evals
ExtendedRosenbrock1000	0.00	1002000	0.00	1002020	-6.75	1001219
ExtendedPowellSingular1000	0.00	1002000	0.00	1002014	-5.64	1001225
VariableDimensionF1000	0.00	1002000	0.00	1002005	-7.15	10365
LinearFFullRank1000	0.00	1002000	-12.97	230001	-13.85	10429
ExtendedRosenbrock10000	0.00	1020000	0.00	1020041	-5.31	1004075
ExtendedPowellSingular10000	0.00	1020000	0.00	1020069	-4.37	1003726
VariableDimensionF10000	0.00	1020000	0.00	1020051	-5.85	80309
LinearFFullRank10000	0.00	1020000	-4.17	1020009	-15.65	100315
Average Ranks	2.3	2.1	2.0	2.9	1.0	1.0

Figure 24.4: Performance of scalable methods in large D . L-BFGS does well. The rest don't, so it's hard to form a good hybrid method.

24.8 Nonsmooth Minimization

Some f such as $|x|$ are continuous but not differentiable everywhere. Derivatives/gradients don't exist, and the usual rules of calculus don't apply. But the logic is still similar. In particular, derivative and gradient generalize to **subgradient**, which in 1D is any tangent line to the function:

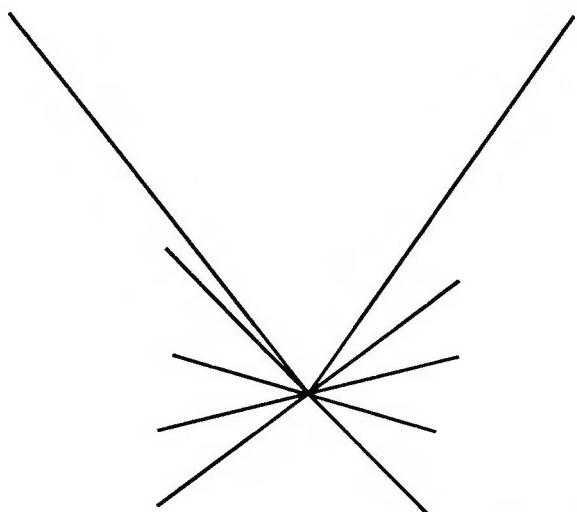


Figure 24.5: Possible subgradients at the cusp of $f(x) = |x|$

For $f(x) = |x|$, a subgradient is given by $f'(x) = x > 0 ? 1 : -1$. This choice isn't unique, but easy to program. Which particular subgradient is selected should make no material difference because some unusual event happens at a point without differentiability.

For $D > 1$, look for a tangent plane, and calculate one dimension at a time, like with gradients. In most cases, the calculation can be based on the following rule: let $f(x) = \text{if}(c(x), g(x), h(x))$; then $f'(x) = \text{if}(c(x), g'(x), h'(x))$, with an arbitrary resolution at a discontinuity point. Otherwise follow the usual chain rule of calculus.

The theory is much more complicated. For perhaps the easiest (though not easy) introduction, see Bagirov et al. (2014). Major well-studied cases are when f is convex (where subgradients are technically defined) or locally Lipschitz (**Clarke subgradient**), though in practice don't worry about it if you can deduce a subgradient as discussed. E.g., for concave function $f(x) = 1 - |x|$ follow the same logic. In a more general context, Penot (2012) gives a very mathematical description and calculus rules for several options of defining nonsmooth derivatives. The main idea is that only the convex/concave case has good calculus rules—others lack something such as having to replace equalities with inequalities, etc.

All possible subgradients at a point form the **subdifferential set**. It has a number of useful properties for optimization, in particular for a convex f it contains 0 at a local minimum.

When using subgradients with existing gradient-based algorithms, run into issues:

- The steepest descent direction based on a subgradient needn't lead to descent. E.g., in 2D let $f(x) = 2|x_1| + x_2$. Then with $\nabla f = (x_1 > 0 ? 2 : -2, 1)$, steepest descent with line search will converge to $(0, a)$ for some finite a that depends on the starting point, even though the solution is $(0, -\infty)$. The problem is that the first coordinate with larger derivative prevents the second one from moving, and, unlike for the smooth case, the first gradient component $\neq 0$. $\exists f$ and x where only directions in a small angular range give descent. Need a direction that makes a proper angle with all members of the subdifferential set, i.e., $\forall \text{member} \quad \text{dot product(direction, member)} < 0$.
- If have a descent direction, at a cusp point the sufficient descent condition makes no sense due to a possible abrupt reduction in the directional derivative.

Intuitively, that a particular subgradient doesn't lead to descent means that the problem isn't giving enough information. So could either:

- Try a random direction, but this can be very inefficient in large D
- Hope that a nearby point leads to descent, and try to use its subgradient, but it's likely to lead back to the original point given that descent wasn't found there
- Can use several subgradients as different points to deduce a better subgradient, maybe the one with the smallest norm or some other property, but it's unclear how to collect them

Several convergent strategies have been designed:

- **Subgradient descent**—don't do line search, and accept worsening steps. Under certain conditions on step sizes (similar to Robbins-Munro conditions for stochastic optimization, discussed later in the chapter), have provable convergence. For implementation, can perhaps use the same subgradient scaling as for the first step of L-BFGS, but limit the norm by a constant such as 1 to prevent divergence. Despite simplicity, this algorithm is very slow. Perhaps can use it to get descent when a regular algorithm such as L-BFGS is stuck, but this doesn't seem to have been explored in the literature.
- **Bundle methods** and variants—like L-BFGS, keep a set of past subgradients, and try to use them to

find a descent subgradient.

For a review of these and variants, see Bagirov et al. (2014); based on their tests, the algorithms aren't fully robust. So they are still being developed, e.g., see Karmitza (2015).

If f is additionally convex, **proximal algorithms** and variants are efficient; see Beck (2017) for a general overview and Hastie et al. (2015) for application to lasso regression. A competitor is coordinate descent, which also works well for convex f and is theoretically efficient in special cases such as lasso regression. See also Nesterov (2018) for further reading. The focus of current research is non-black-box algorithms that assume at least the ability to calculate a subgradient. This knowledge makes a critical difference in performance ability. For further discussion see the mentioned references.

BFGS with backtracking works well as a heuristic (Lewis & Overton 2013); per experiments in Skajaa (2010) it's better than some of the specialized methods above. Among the simpler justifications for good performance (see Lewis & Overton 2013 for more) are that:

- Lipschitz f are differentiable almost everywhere, so encountering a nondifferentiable point is unlikely unless it's a solution
- Nondifferentiable f are limits of ill-conditioned differentiable f , for which Newton's method and variants work reasonably well

So the presented hybrid algorithm for continuous optimization still seems to be the method of choice for black-box f , despite using strong Wolfe line search with L-BFGS, contrary to what is done in Lewis & Overton (2013). Using finite differences offers extra smoothness if f is locally Lipschitz, but beware that a finite-difference gradient needn't be accurate because the Taylor series remainder breaks down. But a central difference will effectively compute a finite-difference derivative at a nearby point. Also, the strong Wolfe line search implementation here will still accept a point after narrowing down the interval, which should help. But perhaps the main reason to use the hybrid algorithm as is is that for a black box f don't know whether it's fully differentiable; it may not even have subgradients, etc.

24.9 Global Minimization

The algorithms of the previous sections find local and not global minimums and usually assume differentiability. Unfortunately for arbitrarily f optimization is provably impossible, at least in worst case (Nesterov 2018). E.g., can have some arbitrarily small golf hole on a flat plateau which is effectively undetectable.

A basic useful assumption is that f is Lipschitz. Then **grid search** is effective at finding an approximate region of the global minimum for small D . Create a uniform grid given known box bounds, and try every grid point. This effectively turns the problem into discrete set optimization, discussed later in the chapter. With a uniform grid can cover the entire region up to ϵ , though this can be very inefficient. Because of the Lipschitz assumption, a solution can't be far from a nearby point. But need exponentially-many-in- D partitions to get particular ϵ . In machine learning, grid search is often the choice for $D \leq 3$.

It's usually more efficient to do random search based on some iid sampler, particularly for larger D . When have no information can use Levy variates, but often have domain-specific box bounds, which reduce the sampling region and allow uniform sampling. For an unbounded sampler want to use a larger scale factor to account for the fact that it should be a global sampler.

Assume that samplers do reasonable error checking, i.e., generate point of finite norm. But f can return NaN or ∞ , perhaps to indicate constraint violation, and algorithms must expect these and apply best effort, i.e., usually ignore the NaN points by treating them as ∞ .

```
class UnboundedSampler
{
    Vector<double> center;
    double scaleFactor;
public:
    UnboundedSampler(Vector<double> const& theCenter, double theScaleFactor =
        10): center(theCenter), scaleFactor(theScaleFactor)
    {assert(isfinite(norm(theCenter)));}
    Vector<double> operator()() const
    {
        Vector<double> next = center;
        for(int i = 0; i < next.getSize(); ++i) next[i] = (*this)(i);
        return next;
    }
    double operator()(int i) const
    //ensure finite samples
    double result = numeric_limits<double>::infinity();
```

```

        while(!isfinite(result)) result = center[i] + max(1.0, abs(center[i]))/
            10 * scaleFactor * GlobalRNG().Levy() * GlobalRNG().sign();
        return result;
    }
};

class BoxSampler
{
    Vector<pair<double, double>> box;
public:
    BoxSampler(Vector<pair<double, double>> const& theBox) : box(theBox) {}
    Vector<double> operator() () const
    {
        Vector<double> next(box.getSize());
        for(int i = 0; i < box.getSize(); ++i)
            next[i] = GlobalRNG().uniform(box[i].first, box[i].second);
        return next;
    }
    double operator()(int i) const
    {
        return GlobalRNG().uniform(box[i].first, box[i].second);
    }
};

```

Random search is good at getting close to a region with a global minimum, but usually want to find a local minimum in the best found region, so run local search right after (Zhigljavsky & Zilinskas 2007). This is done for all tested algorithms. Can divide the evaluation budget maybe 9 to 1 in favor of exploration. This method is a basic benchmark for any improvements.

Global continuous optimization is very different from global combinatorial optimization:

- For combinatorial local solutions are well-separated by a discrete change
- Continuous is more receptive to iid sampling—e.g., random before local search seems far less useful for combinatorial
- For continuous can try any random direction
- For continuous, the variables are much more likely to be separable or partially separable

Random sampling also runs into the curse of dimensionality:

- $\Pr(\text{find a point in the acceptable region}) = \frac{|\text{the acceptable region}|}{|\text{the search domain}|}$. Given that $\frac{\text{Vol(hypersphere)}}{\text{Vol(the enclosing hypercube)}} \rightarrow 0$ exponentially fast, even for large acceptable regions dimensionality is an issue. Random search is efficient only when very few variables have significant impact.
- Because want to hit the acceptable region with high probability = $1 - a$, under uniform sampling need $\frac{\log(1/a)}{|\text{the acceptable region}|} |\text{the search domain}|$ samples (Locatelli & Schoen 2013; their expression simplified using that $\log(1 - x) \approx -x$ for small x per Zhigljavsky & Zilinskas 2007).
- For $f(x) = |x|$ on $[-1, 1]$, the distribution of f -values is the standard uniform. So by order statistics (Wikipedia 2018), the minimum out of n follows beta($1, n$) distribution and has expected value $1/(n + 1)$. So here the convergence of expected value is $O(1/n)$ and is typical for 1D because in bounded domains distributions for various f are usually close to uniform.
- For $f(x) = \frac{12}{D} \sum |x_i|$ on $[-1, 1]^D$ the distribution of the sum is asymptotically normal with mean $\frac{12}{D}$ and variance 1. Normal minimum order statistic roughly $\sim \mu - \sigma \Phi^{-1}\left(\frac{1}{n}\right)$ (StackOverflow 2018).

Because the tail drops exponentially, for large D where the normal approximation is valid, $\Phi^{-1}\left(\frac{1}{n}\right)$ is effectively constant. E.g., the typical $n = 10^6$ only gives about 4.75 and smaller 10^3 about 3.09. Other distributions run into similar problems.

- As discussed in the “Optimization Algorithms” chapter, run into the central limit catastrophe where solutions tend to be close to the average. Effectively at first about $\frac{1}{2}$ of the variables will lead to a gain and the rest to a loss. As the optimization progresses, it becomes increasingly less likely to find an improvement by chance because it’s increasingly more likely to get a bad sample than a good one. So at best maybe should spend $O(n)$ evaluations on random search, perhaps \sqrt{n} or so.

So essentially in small D random sampling = grid search = find solution by looking at the graph of f . A

particular enhancement to random search is using scrambled quasi-random sequences such as Sobol. Some ideas (Bousquet et al. 2017):

- Grid search tries the same points inefficiently when some variables make no difference, and random search avoids this
- Random search can be unlucky with few evaluations
- Quasi-random sequences don't get unlucky by avoiding some regions
- Scrambling helps to reduce the chance of bad luck

On my tests scrambled Sobol sequence hybrid outperforms random search, but not other algorithms. Bousquet et al. (2017) recommend another sequence, but this makes no sense given that many studies show superiority of Sobol sequence for other tasks.

With irrelevant parameters random search wins by far by sampling more different values of relevant parameters (Bergstra & Bengio 2012), but with a very small budget it will occasionally get very unlucky by avoiding some regions. So with few parameters and a well-defined box, grid search still seems to be the methods of choice.

In larger D must use biased sampling to get better performance. By the no-free-lunch theorems no particular bias helps in all cases, so hope that the chosen bias works well for practical f . So for a particular problem various methods are only as good as their biases. Some commonly exploited biases:

- Many f are separable or almost separable, i.e., can do random search in one variable at a time
- Many f that aren't separable have interactions of only few variables
- From any particular point a better point is usually not too far away, and can have a path of such improvements that leads to the optimum
- Most f are scaled so that the optimum is close to 0, and f -value is arbitrarily bad very far away from 0

So **random coordinate descent** is another good benchmark algorithm, perhaps unbeatable for separable f . Also, it works with incrementally computable f ; here a follow-up local algorithm is the regular coordinate descent. But as for nondifferentiable local minimization, random coordinate descent can get stuck. Still, it's useful as a helper method and for incremental evaluation.

```
template<typename INCREMENTAL_FUNCTION, typename SAMPLER> struct ContinuousSMove
{
    typedef Vector<double> X;
    typedef pair<double, int> MOVE;
    INCREMENTAL_FUNCTION& f;
    SAMPLER const& s;
    ContinuousSMove(INCREMENTAL_FUNCTION& theF, SAMPLER const& theS):
        f(theF), s(theS) {}
    double getScore(X const& x) const { return f(f.getXi()); }
    pair<MOVE, double> proposeMove(X const& x) const
    {
        int i = GlobalRNG().mod(x.getSize());
        f.setCurrentDimension(i);
        double xi = s(i), yNext = f(xi), y = f(x[i]);
        return make_pair(MOVE(xi, i), //if started at nan allow all moves
                        isnan(y) ? 0 : yNext - y);
    }
    void applyMove(X& x, MOVE const& move) const
    {
        x[move.second] = move.first;
        f.setCurrentDimension(move.second);
        f.bind(move.first);
    }
};

template<typename INCREMENTAL_FUNCTION, typename SAMPLER>
pair<Vector<double>, double> randomCoordinateDescent(
    INCREMENTAL_FUNCTION& f, SAMPLER const& s, int maxEvals = 1000000)
{
    assert(maxEvals > 0);
    Vector<double> x = localSearch(ContinuousSMove<
        INCREMENTAL_FUNCTION, SAMPLER>(f, s), f.getX(), maxEvals);
    return make_pair(x, f(f.getXi()));
}
```

```
template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
RCDGeneral(FUNCTION const &f, SAMPLER const& s,
Vector<double> x0 = Vector<double>(), int maxEvals = 1000000)
{
    if(x0.getSize() == 0) x0 = s();
    IncrementalWrapper<FUNCTION> iw(f, x0);
    return randomCoordinateDescent(iw, s, maxEvals);
}
```

For nonincremental f use a wrapper:

```
template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
RCDGeneral(FUNCTION const &f, SAMPLER const& s,
Vector<double> x0 = Vector<double>(), int maxEvals = 1000000)
{
    if(x0.getSize() == 0) x0 = s();
    IncrementalWrapper<FUNCTION> iw(f, x0);
    double y = randomCoordinateDescent(iw, s, maxEvals);
    return make_pair(iw.xBound, y);
}
```

Performance of this algorithm, the benchmarks algorithms, and some others mentioned in the comments has been tested some f from Simon (2013) and Jamil & Yang (2013), with a selection made to prefer nonseparable f or those that offer various difficulties such as severe nondifferentiability. It's unlikely that any sampling method is effective beyond some medium D such as 100, particularly with many bad-quality local minima, so no testing was done for such case. Beware that mathematical separability (i.e., can decompose f into a sum) may be hidden, e.g., by a monotonic transformation of f into another function g . Here g will also be solvable by a one-variable-at-a-time process.

Another approach is using a Markovian sampler:

- It needs a starting point, such as agnostic 0 (don't use it when it's the answer to the test problems).
- The user-provided step-size sampler should be from a fat-tail distribution to allow jumping out of local minimums more efficiently. This seems to lead to good exploration/exploitation balance in that after many local moves make a large jump to a different local neighborhood. But regardless of the distribution the median step size defines scale and is essentially a discretization parameter.
- It's guaranteed to converge if every point can be generated eventually.

```
template<typename FUNCTION, typename M_SAMPLER> struct ContinuousMMove
{
    typedef Vector<double> X;
    typedef Vector<double> MOVE;
    FUNCTION const& f;
    M_SAMPLER const& s;
    ContinuousMMove(FUNCTION const& theF, M_SAMPLER const& theS): f(theF),
        s(theS){}
    double getScore(X const& x) const{return f(x);}
    pair<MOVE, double> proposeMove(X const& x) const
    {
        Vector<double> xNext = s(x);
        assert(isfinite(norm(xNext)));
        double yNext = f(xNext), currentScore = getScore(x);
        //if started at nan allow all moves
        return make_pair(xNext, isnan(currentScore) ? 0 : yNext - getScore(x));
    }
    void applyMove(X& x, MOVE const& move) const{x = move;}
};

template<typename FUNCTION, typename M_SAMPLER> pair<Vector<double>,
double> markovianMinimize(FUNCTION const& f, M_SAMPLER const& s,
Vector<double> x0, int maxEvals = 1000000)
{
    assert(maxEvals > 0 && isfinite(norm(x0)));
    Vector<double> x = localSearch(ContinuousMMove<FUNCTION, M_SAMPLER>(f, s),
        x0, maxEvals);
    return make_pair(x, f(x));
}
```

The discussed algorithms cover all types of samplers:

- Random iid search uses an iid sampler
- Markovian search uses a displacement sampler
- Random coordinate descent uses an incremental iid sampler

All of these can work with black-box constraints by rejecting infeasible points (and maybe declare failure after 1000 or so consecutive rejections, by either a return code or returning the current point). For iid samplers rejection sampling may not be needed in many cases such as domains with box constraints because can sample directly. In all cases the algorithms are encapsulated from the details, except perhaps needing to pay attention to a sampler return code.

For step-based samplers, instead of rejection could binary-search-trim the step size until satisfactory, assuming that started from a satisfactory point into a feasible direction. But this will lead to more failure in subsequent samples, so the best strategy is unclear, particularly because when starting from a high- D corner, finding a feasible direction is essentially impossible. So for constraint handling iid sampling is better.

For box constraints a simple solution is to trim individual coordinates of a proposed x to box boundaries. This avoids inefficiencies with handling of general constraints.

```
struct AgnosticStepSampler
{
    Vector<double> operator()(Vector<double> const& x) const
    { //must ensure finite samples in all components
        assert(isfinite(norm(x)));
        int maxTries = 10; //reasonable infinity protection
        while (maxTries--)
            { //correct on first attempt unless near very large numbers
                Vector<double> u = GlobalRNG().randomUnitVector(x.getSize()),
                    result = x + u *
                        (findDirectionScale(x, u)/10 * GlobalRNG().Levy());
                if(isfinite(norm(result))) return result;
            }
        return x; //can't avoid infinities
    }
};

Vector<double> boxTrim(Vector<double> x,
    Vector<pair<double, double> > const& box)
{
    for(int i = 0; i < x.getSize(); ++i)
    {
        if(x[i] < box[i].first) x[i] = box[i].first;
        else if(isnan(x[i]) || x[i] > box[i].second) x[i] = box[i].second;
    }
    return x;
}

struct BoxConstrainedStepSampler
{
    Vector<pair<double, double> > box;
    AgnosticStepSampler s;
    BoxConstrainedStepSampler(Vector<pair<double, double> > const& theBox) :
        box(theBox){}
    Vector<double> operator()(Vector<double> const& x) const
        {return boxTrim(s(x), box);}
};
```

For user convenience it may seem that any algorithm should work with only one sampler, but this prevents hybrid algorithms (discussed later). In the worst case iid samplers are just as good as Markovian samplers. But the latter allow better exploitation of various biases, which are common in practice.

In view of the NFL, the only meaningful way of comparing algorithms theoretically is checking whether some may be better at exploiting common biases than others. This is best detected experimentally because even though some biases can be explicitly designed for, it's unclear which approach is best before testing with the problem. So when making decisions based on a test set, to avoid overfitting decide as many things as possible theoretically. A conceptual comparison approach is to consider **wastefulness**—i.e., an algorithm shouldn't evaluate the same thing more often than reasonable. For low wastefulness want to balance exploration and exploitation, given assumed biases.

From regret point of view, choose the final algorithm whose typical results are considered to be as good

as those of any other, i.e., assuming none of them gets particularly lucky. So want a hybrid of several algorithms. This seems to be a good strategy even for a calculation with a fixed budget:

- Become more robust to failure of a particular algorithm
- Assuming that use a small number (< 5 maybe?) of algorithms and equal budget of evaluations for each, the loss to the best of them shouldn't be large enough to make a difference

For very small budgets, such as 100 evaluations or so, where local search isn't economical because can't afford to polish precision, hybridize RCD and Markovian search:

```
template<typename FUNCTION, typename SAMPLER, typename M_SAMPLER>
pair<Vector<double>, double> smallBudgetHybridMinimize(FUNCTION const& f,
    SAMPLER const& s, M_SAMPLER const& ms, Vector<double> x0,
    int maxEvals = 100)
{
    assert(maxEvals > 2); // no point to have fewer
    return RCDGeneral(f, s, markovianMinimize(f, ms, x0,
        maxEvals * 0.5).first, maxEvals * 0.5);
}
```

In my tests it did better than random sampling and the individual algorithms. The order makes sense because Markovian search will get to a good part of the landscape, and RCD will collect low-hanging fruit.

For larger budgets (i.e., large enough to approximate gradients for local search; $> 100D$ or so), a human is likely to approach a hard problem by trying a couple of promising algorithms to see if any of them gets a good solution. So for very large evaluation budgets it makes sense to include more algorithms, which is considered next.

An interesting strategy that forms a backbone for some step-based metaheuristics is **intermittent search** (Bénichou et al. 2011). It's related to some strategies for how a human would look for lost keys on a beach:

- If have spatial memory, usually use some pattern, assuming can scan small square patches at a time.
- Without memory, a Levy walk is a good strategy. I.e., take Levy jumps between patch scans, and don't scan during jumps. Unlike Markovian search, start from the last tried solution instead of the current best one. In some cases this has been proven optimal.

In particular, simulated annealing (see the "Optimization Algorithms" chapter) is supported by this—a simple implementation is to use Levy jump as move. The performance is good with box constraints and bad without them, at least for the Markovian version due to running away. Markovian search is able to jump around over "obstacles", potentially making the annealing process unnecessary, unlike with discrete neighborhoods of combinatorial optimization. Also some criticism is that simulated annealing seems good at jumping over small obstacles but not large ones (Locatelli & Schoen 2013). They claim that a random restart followed by local search if successful is more efficient for this. But this algorithm only seems to be a modest improvement on random restarting, which suffers from the curse of dimensionality. Zhigljavsky & Zilinskas (2007) claim that simulated annealing was eventually realized to not be as efficient as some random search methods, particularly those that use memory and statistical reasoning to prune non-promising regions, but without mentioning specific better algorithms. They claim the same for genetic algorithms (discussed later). But methods that use memory effectively are much more computationally expensive, and are generally based on metamodels (see the comments section).

A version of simulated annealing based on RCD as local search performs better, but still not as well as population-based algorithms below. So it's excluded from being a part of a hybrid approach. But it leads to a good algorithm for incremental f :

```
template<typename INCREMENTAL_FUNCTION, typename SAMPLER> pair<Vector<double>,
    double> simulatedAnnealingSMinimizeIncremental(INCREMENTAL_FUNCTION&
    f, SAMPLER const& s, Vector<double> x0, int maxEvals = 1000000)
{
    assert(maxEvals > 0 && isfinite(norm(x0)));
    Vector<double> x = selfTunedSimulatedAnnealing(ContinuousSMove<
        INCREMENTAL_FUNCTION, SAMPLER>(f, s), x0, maxEvals);
    return make_pair(x, f(f.getXi()));
}

template<typename INCREMENTAL_FUNCTION, typename SAMPLER> double
incrementalSABeforeLocalMinimize(INCREMENTAL_FUNCTION &f, SAMPLER const& s,
    int maxEvals = 1000000, double xPrecision = highPrecEps)
{
    simulatedAnnealingSMinimizeIncremental(f, s, f.getX(), maxEvals * 0.9);
}
```

```

    //if remembers evals so far
    return unimodalCoordinateDescent(f, maxEvals, xPrecision);
}
}

```

Another common set of algorithms are population based. In a hybrid, one such algorithm can be the first in the pipeline. In particular, a good approach seems to be **differential evolution**—mentioned by Nash (2014) as commonly implemented in various libraries. It also was a key algorithm for several problems in the SIAM challenge (Bornemann et al. 2004). The basic idea is to have a population of solutions and make local moves on each member independently using a block coordinate step toward a specially generated sample. Given a current population member x_i :

1. Create $x_{\text{new}} = x_i + F(x_k - x_l)$, where j, k, l are random distinct indices
2. Pick a random dimension k_{rand} , and set $x_{\text{next}} = x_i$
3. Then \forall dimension k , set $x_{\text{next}}[k] = x_{\text{new}}[k]$ if $k = k_{\text{rand}}$ or $\text{Bernoulli}(C)$ is true

Logical values are $F = 1$ and crossover rate $C = 0.5$. But $F = 0.9$ is common and used here because $F = 1$ leads to less diversity due to reducing the number of distinct random choices of j , k , and l (see Price et al. 2005 for details). The population size was chosen to be $\text{maxEvals}^{1/3}$ (same for other population algorithms discussed in the comments). It seems that need population size $\rightarrow \infty$ as $\text{maxEvals} \rightarrow \infty$ to guarantee convergence because the initial population is random based on an iid sample.

```

template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
differentialEvolutionMinimize(FUNCTION const& f, SAMPLER const& s,
Vector<pair<double, double>> const& box, int maxEvals = 1000000)
{
    assert(maxEvals > 0);
    int n = pow(maxEvals, 1.0/3);
    Vector<pair<Vector<double>, double>> population(n);
    for(int i = 0; i < n; ++i)
    {
        population[i].first = s();
        population[i].second = f(population[i].first);
    }
    maxEvals -= n;
    while(maxEvals > 0)
    {
        for(int i = 0; i < n && maxEvals-- > 0; ++i)
        { //mutate new point
            Vector<int> jkl = GlobalRNG().randomCombination(3, n);
            Vector<double> xiNew = population[i].first, xiMutated =
                boxTrim(population[jkl[0]].first + (population[jkl[1]].first -
                population[jkl[2]].first) * 0.9, box);
            //crossover with mutated point
            int D = xiNew.getSize(), randK = GlobalRNG().mod(D);
            for(int k = 0; k < D; ++k) if(GlobalRNG().mod(2) || k == randK)
                xiNew[k] = xiMutated[k];
            if(!isfinite(norm(xiNew)))
            { //enforce finite samples
                ++maxEvals;
                continue;
            }
            //select best of original and mutated
            double yiNew = f(xiNew);
            if(yiNew < population[i].second)
            {
                population[i].first = xiNew;
                population[i].second = yiNew;
            }
        }
    }
    pair<Vector<double>, double>& best = population[0];
    for(int i = 1; i < n; ++i)
        if(best.second < population[i].second) best = population[i];
    return best;
}
}

```

Theoretically it can stall if the population collapses to a single point. An interesting observation is that if the sample generation of DE is replaced by an iid sampler, the algorithm becomes parallel RCD, but with multimoves with size decided by binomial(C) sampling. So its sample generation is a good feature. DE is simple, without unnecessary logic, and frequently implemented, with established good performance. Its move mechanism also maintains scale automatically.

Another hybrid component is **genetic local search** (see the “Combinatorial Optimization” chapter). My implementation is based on:

- Uniform crossover.
- RCD as local search.

```
template<typename FUNCTION, typename SAMPLER> class GAContinuousProblem
{
    FUNCTION const& f;
    SAMPLER const& s;
public:
    typedef Vector<double> X;
    GAContinuousProblem(FUNCTION const& theF, SAMPLER const& theS): f(theF),
        s(theS){}
    X generate() const{return s();}
    void crossover(X& x1, X& x2) const
    { //uniform crossover
        assert(x1.getSize() == x2.getSize());
        for(int k = 0; k < x1.getSize(); ++k) if(GlobalRNG().mod(2))
            swap(x1[k], x2[k]);
    }
    X localSearch(X x, int nLocalMoves) const //RCD for the same sampler
    { return RCDGeneral(f, s, x, nLocalMoves).first;}
    double evaluate(X const& x) const{return f(x);}
};

template<typename FUNCTION, typename SAMPLER> pair<Vector<double>, double>
geneticLocalSearchContinuous(FUNCTION const& f, SAMPLER const& s,
    int maxEvals = 1000000)
{
    int nLocalMoves = int(pow(maxEvals, 1.0/3)), populationSize = nLocalMoves;
    return geneticLocalSearch(GAContinuousProblem<FUNCTION, SAMPLER>(f, s),
        populationSize, nLocalMoves, maxEvals);
}
```

A good hybrid approach seems to be:

1. Differential evolution for 30% of evaluations
2. Genetic local search for 30% of evaluations
3. Markovian search for 30% of evaluations on the best of the above
4. Local search with remaining 10% of evaluations

The percentages aren't tuned in any way and are based on a logical idea to give $\approx 10\%$ to the final local search and split the rest evenly. The order seems to work well in that differential evolution and GLS give a good starting solution to Markovian search to explore further, and smooth local search polishes the result if possible.

```
template<typename FUNCTION, typename SAMPLER, typename M_SAMPLER>
pair<Vector<double>, double> hybridBeforeLocalMinimize(
    FUNCTION const& f, SAMPLER const& s, M_SAMPLER const& ms,
    Vector<pair<double, double>> const& box, int maxEvals = 1000000,
    double yPrecision = highPrecEps)
{
    assert(maxEvals > 1000); //no point to have fewer
    pair<Vector<double>, double> glsSolution = geneticLocalSearchContinuous(f,
        s, maxEvals * 0.3), deSolution = differentialEvolutionMinimize(f, s,
        box, maxEvals * 0.3);
    return hybridLocalMinimize(markovianMinimize(f, ms, (deSolution.second <
        glsSolution.second ? deSolution : glsSolution).first, maxEvals * 0.3
        ).first, f, maxEvals * 0.1);
}
```

This particular hybrid seems to be the practical choice for large evaluation budgets in that it's pointless to try more algorithms to improve its answer. The particular selection isn't necessarily optimal, but seems

stable due to the simplicity of the components and their somewhat different solution strategies. Each one uses a somewhat different move strategy. It's hard to think of additional algorithms that may be beneficial as components. E.g., can replace GSL by simulated annealing based on the component sampler, but per below comparisons the former works slightly better individually, so this wasn't done. Also the first-stage algorithms don't use Markovian sampling at all, so only one such method is needed. RCD isn't used because GLS uses coordinate moves as local search.

For noisy f the discussed methods should work with small levels of noise, but the conclusions as to which perform well overall may not apply.

Function	Rando mBLM	RCDB LM	Increm entalS ABLM	Markovi anBLM	Hybrid BLM	Chol11 CMA_ ESBL M	Simulate dAnneali ngBLM	Simulat edAnnealingSB LM	Differenti alEvoluti onBL M	Geneti cLSBL M
Bukin6	-1.77	-1.68	-0.54	-1.63	-1.81	-1.79	-1.79	-1.71	-1.85	-1.72
Damavandi	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Easom	-15.65	-8.87	-4.16	-15.65	-15.65	-2.09	-15.65	-7.83	-5.65	-15.65
GulfRND	-12.72	-12.90	-5.84	-12.93	-15.55	-13.84	-12.75	-12.92	-15.65	-12.79
Price2	-15.65	-15.65	-7.11	-15.65	-15.65	-1.98	-15.65	-15.65	-15.65	-15.65
Trefethen	-5.08	-6.93	-3.03	-15.27	-15.03	-0.46	-14.41	-9.36	-15.03	-13.57
Ackley2	-15.59	-15.55	-2.75	-15.49	-15.65	-4.17	-15.52	-15.51	-15.65	-15.65
FletcherPowell2	-15.65	-15.60	-2.07	-15.64	-15.65	-15.64	-15.63	-15.64	-15.65	-15.64
Griewank2	-4.64	-7.09	-2.96	-15.65	-15.65	-1.38	-15.65	-9.34	-15.65	-10.70
Rastrigin2	-15.65	-15.65	-5.42	-15.65	-15.65	0.00	-15.65	-15.65	-15.65	-15.65
SchwefelDoubleSum2	-15.65	-15.65	-7.91	-15.63	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
SchwefelMax2	-15.61	-15.26	-2.33	-15.50	-15.65	-15.66	-15.59	-15.60	-15.65	-15.48
StepFunction2	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Weierstrass2	-13.63	-14.41	-2.28	-15.65	-15.65	-13.95	-15.65	-13.29	-15.65	-13.35
Trig2	-15.65	-15.65	-8.07	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Pinter2	-15.65	-10.49	-4.85	-15.65	-15.65	-3.19	-15.65	-14.62	-15.65	-15.65
Salomon2	-4.42	-1.90	-1.70	-15.26	-15.65	-1.40	-15.03	-3.75	-15.65	-12.18
SchaeferF62	-8.83	-1.87	-3.44	-15.65	-15.65	-1.39	-15.65	-9.74	-15.20	-12.47
Ackley6	-5.44	-14.85	-2.13	-12.44	-14.89	-2.05	-9.90	-15.02	-1.04	-14.92
StepFunction6	0.00	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Weierstrass6	-2.31	-7.83	-1.55	-15.65	-15.65	-5.71	-15.65	-8.22	-15.65	-6.57
Trig6	-7.92	-7.17	-4.20	-15.65	-15.65	-14.04	-15.65	-9.93	-15.65	-15.05
Pinter6	-4.09	-6.27	-4.33	-7.33	-15.65	0.00	-14.59	-11.56	-15.65	-15.64
Salomon6	-2.65	-0.51	-0.70	-1.00	-1.00	-1.58	-1.00	-0.68	-1.00	-0.99
SchaeferF66	-0.14	-2.53	-2.80	-0.18	-1.30	0.00	-0.51	-7.81	-0.97	-1.32
Ackley10	0.00	-14.65	-1.76	0.00	-14.69	0.00	-9.45	-14.55	-12.34	-14.60
FletcherPowell10	-9.07	-7.60	0.00	-9.52	-12.02	-10.73	-11.43	-10.12	-14.27	-11.97
Griewank10	-3.40	-2.72	-1.19	-0.40	-1.05	-0.73	-0.96	-1.28	-15.65	-2.55
Rastrigin10	0.00	-15.65	-3.27	0.00	-15.65	0.00	0.00	-15.65	-15.65	-15.65
SchwefelDoubleSum10	-14.40	-15.03	-1.39	-15.29	-14.81	-15.65	-15.17	-14.95	-15.65	-14.94
SchwefelMax10	-15.65	-15.65	-1.14	-15.59	-15.65	-15.40	-15.65	-15.65	-15.65	-15.65
StepFunction10	0.00	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Weierstrass10	-0.18	-3.19	-1.14	-15.65	-15.65	-5.77	-15.65	-2.35	-15.65	-2.59
Trig10	-7.46	-8.05	-4.22	-15.65	-15.65	-11.72	-15.65	-6.16	-15.65	-6.43
Pinter10	0.00	-1.96	-3.52	0.00	-15.64	0.00	-0.97	-11.76	-15.65	-14.57
Salomon10	-0.19	-0.29	-0.42	-1.00	-1.00	-0.57	-0.93	-0.45	-1.00	-0.76
StepFunction10	0.00	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Weierstrass10	-0.18	-3.19	-1.14	-15.65	-15.65	-5.77	-15.65	-2.35	-15.65	-2.59
Trig10	-7.46	-8.05	-4.22	-15.65	-15.65	-11.72	-15.65	-6.16	-15.65	-6.43
Pinter10	0.00	-1.96	-3.52	0.00	-15.64	0.00	-0.97	-11.76	-15.65	-14.57
Salomon10	-0.19	-0.29	-0.42	-1.00	-1.00	-0.57	-0.93	-0.45	-1.00	-0.76
SchaeferF610	0.00	-0.95	-1.59	0.00	-1.05	0.00	0.00	-2.49	-0.20	-1.06
Average Ranks	5.81	4.98	7.76	3.95	4.57	6.12	3.76	4.76	1.71	3.57

Figure 24.6: Some methods on some problems. Samplers with box constraints were used, and starting solution was a random sample from the box. 30 repetitions where used for each. All had default budgets of about 1000000 evaluations, and most used it up closely.

Function	Rando mBLM	RCDB LM	Increm entalS ABLM	Markovi anBLM	Hybrid BLM	Chol11 CMA_ ESBL M	Simulate dAnneali ngBLM	Simulat edAnnealingSB LM	Differenti alEvoluti onBLM	Genetic LSBLM
Bukin6	-1.38	-1.36	-0.49	-1.72	-1.25	-1.88	0.00	-1.39	-1.18	-1.16
Damavandi	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
Easom	-15.65	-7.31	-4.21	-2.09	-15.65	-1.57	0.00	-6.78	-15.65	-15.65
GulfRND	-12.91	-13.02	-4.92	-6.87	-12.84	-13.99	-2.85	-13.02	-12.81	-13.09
Price2	-15.65	-15.11	-6.68	-12.23	-15.65	-1.00	-1.00	-15.65	-1.00	-15.65
Trefethen	-4.92	-8.83	-2.55	-15.31	-15.07	-0.36	-1.31	-8.31	-15.07	-13.05
Ackley2	-15.49	-15.13	-2.71	-15.16	-15.65	-4.17	0.00	-15.46	-15.65	-15.65
FletcherPowell2	-14.85	-9.42	-3.33	-14.78	-11.49	-13.04	-1.18	-14.50	-14.35	-15.24
Griewank2	-6.61	-8.41	-3.18	-15.65	-15.65	-0.71	-3.14	-11.15	-15.65	-13.85
Rastrigin2	-15.65	-15.65	-3.98	-15.65	-15.65	-0.52	-4.18	-15.65	-15.65	-15.65
SchwefelDoubleSum2	-15.65	-15.65	-9.30	-15.58	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65
SchwefelMax2	-15.65	-15.31	-2.56	-15.51	-15.65	-15.65	-15.62	-15.60	-15.65	-15.45
StepFunction2	-15.13	-15.65	-15.65	-15.65	-15.65	-15.65	0.00	-15.65	-15.65	-15.65
Weierstrass2	-12.42	-13.77	-2.27	-11.74	-15.58	-15.41	-0.95	-12.92	-15.65	-13.00
Trig2	-15.48	-15.47	-7.62	-15.44	-15.38	-15.65	-4.28	-15.41	-15.65	-15.20
Pinter2	-15.65	-12.55	-4.20	-15.65	-15.65	-5.77	-15.11	-14.62	-15.65	-15.65
Salomon2	-4.91	-2.78	-1.81	-15.47	-15.65	-0.45	-7.48	-5.81	-15.65	-13.05
SchaeferF62	-2.92	-2.18	-2.42	-15.14	-15.65	-0.90	-0.30	-6.10	-15.65	-8.83
Ackley6	-0.49	-14.92	-1.68	-10.36	-14.85	-0.49	-0.52	-14.95	0.00	-14.89
FletcherPowell6	-12.23	-12.22	-0.24	-12.15	-15.00	-9.91	0.00	-12.71	-12.57	-14.16
Griewank6	-0.26	-1.62	-1.43	-0.74	-15.65	-0.46	-0.32	-2.51	-15.65	-1.71
Rastrigin6	0.00	-15.65	-2.70	0.00	-15.65	0.00	0.00	-15.65	-15.65	-15.65
SchwefelDoubleSum6	-15.65	-15.65	-2.75	-15.65	-15.65	-15.65	-15.65	-15.65	-15.65	-15.64
SchwefelMax6	-15.59	-15.65	-0.65	-15.65	-15.65	-15.65	-15.62	-15.57	-15.65	-15.65
StepFunction6	0.00	-15.65	-15.65	-15.65	-15.65	-15.65	0.00	-15.65	-15.65	-15.65
Weierstrass6	-0.92	-5.03	-1.49	-3.55	-3.32	-3.77	-0.52	-4.34	-2.30	-4.96
Trig6	-9.89	-8.52	-4.79	-9.83	-12.56	-14.44	-0.45	-10.08	-9.47	-14.77
Pinter6	-0.02	-4.75	-1.47	-7.33	-15.65	0.00	0.00	-4.89	-15.65	-15.14
Salomon6	-1.14	-0.49	-0.64	-1.00	-1.00	-0.69	-0.68	-0.59	-1.00	-0.90
SchaeferF66	-0.03	-2.55	-1.97	-0.09	-2.28	0.00	-0.52	-4.99	-0.02	-1.31
Ackley10	0.00	-13.25	-1.41	0.00	-14.65	0.00	0.00	-14.70	0.00	-14.20
FletcherPowell10	-3.68	-3.67	0.00	-3.82	-0.46	-5.12	0.00	-3.20	-2.29	-1.37
Griewank10	-2.88	-2.71	-1.00	-0.41	-15.65	-0.73	-0.45	-1.17	-15.65	-2.46
Rastrigin10	0.00	-15.65	-1.69	0.00	-15.65	0.00	0.00	-15.65	-15.65	-15.65
SchwefelDoubleSum10	-14.72	-14.65	-0.23	-15.24	-15.15	-15.65	-14.79	-14.66	-15.65	-14.92
SchwefelMax10	-15.64	-15.50	-0.30	-15.65	-15.48	-15.68	-15.65	-15.65	-15.65	-15.68
StepFunction10	0.00	-15.65	-15.65	-15.65	-15.65	-15.65	0.00	-15.65	-15.65	-15.66
Weierstrass10	-0.08	-2.52	-1.13	-0.32	-1.93	-5.29	0.00	-1.96	-1.07	-2.62
Trig10	-8.25	-9.45	-4.46	-6.53	-10.27	-9.66	0.00	-8.11	-9.07	-9.13
Pinter10	0.00	-3.40	-1.34	0.00	-15.65	0.00	0.00	-8.80	-15.65	-14.58
Salomon10	-1.78	-0.31	-0.44	-1.00	-1.00	-0.62	-0.33	-0.40	-1.00	-0.81
SchaeferF610	0.00	-1.38	-1.17	0.00	-1.00	0.00	0.00	-2.04	0.00	-1.03
Average Ranks	5.19	4.55	7.24	4.48	2.60	5.17	7.93	3.88	3.02	3.21

Figure 24.7: This time without box constraint knowledge. The performances are much worse now, and many methods failed to improve from an unlucky near-infinite initial seed. So the comparisons of performance here isn't as meaningful.

Because of the NFL, any performance comparisons at best converges to a typical practical behavior with many representative test f . But statistical conclusions hold only if these are sampled randomly. This is never the case, so at best get an observational study (see the "Computational Statistics" chapter). A good such study will include a balance of easy, medium, and hard problems that are representative in practice. Various biases such as tendency of solutions to be closer to the origin of the coordinate system, and ability to separate variables will also be eliminated from most problems. And even then a good comprehensive comparison is unlikely to be representative of that on a specific domain. Some sources recommend rotating the domain for most problems to make separable function comparisons more fair. But as Clerc (2015)

points out, this is often silly because for real-world problems the variables have clear meanings, and mixing them makes no sense. Also many algorithms intentionally or unintentionally have biases that help on biased problems. See Clerc (2015) and Clerc (2019) for a discussion of many of these issues in greater detail. Much external logic is needed to pick good algorithms—e.g., focus on those that are simple, well-established, not significantly worse than the best, etc. The hybrid algorithms presented here seem to satisfy common-sense criteria.

24.10 Discrete Set Optimization

Can generate a discrete set even from a continuous range. This works in particular for the exponential range 2^i , with $i \in$ some discrete interval, when solutions within a small multiplicative factor are indifferent, i.e., the transformed f is Lipschitz.

Grid search is the simplest such method, which discretizes and tries all possibilities. Though it's infeasible in large D , due to finding a near-optimal solution for Lipschitz f , it's very useful for small D , particularly for parameter optimization. The main task is generating all value selections. The solution idea is to order variables, keep current indices \forall variable, and:

- After advancing any variable, move to the next one. When have no variable to move to, a selection is complete.
- When the current variable loop ends, reset all higher variables. When variable-0 loop ends, the generation went through all selections.

```
template<typename FUNCTION> Vector<double> gridMinimize(
    Vector<Vector<double>> const& sets, FUNCTION const& f = FUNCTION())
{
    assert(sets.getSize() > 0);
    Vector<double> best;
    for(int i = 0; i < sets.getSize(); ++i)
    {
        assert(sets[i].getSize() > 0);
        best.append(sets[i][0]);
    }
    double bestScore = f(best);
    Vector<int> current(sets.getSize(), -1);
    current.lastItem() = 0;
    for(int level = 0; level > -1;)
    {
        if(level < sets.getSize())
        {
            if(++current[level] < sets[level].getSize()) ++level;
            else current[level--] = -1;
        }
        else
        { //process value selection
            Vector<double> values;
            for(int i = 0; i < sets.getSize(); ++i)
                values.append(sets[i][current[i]]);
            double score = f(values);
            if(score < bestScore)
            {
                bestScore = score;
                best = values;
            }
            --level;
        }
    }
    return best;
}
```

Another strategy is something similar to compass search. It tries to reduce the number of evaluations as much as possible but doesn't guarantee optimality.

1. Assume sorted values \forall variable
2. \forall variable initialize the last successful direction to +1
3. Until converge or reach the iteration limit

4. Select a variable
5. Try to move into its last successful direction once
6. If unsuccessful, try the reverse direction

```
//assumes set values are in sorted (or reverse sorted) order!
template<typename FUNCTION> pair<Vector<double>, pair<double, int>>
compassDiscreteMinimizeHelper(Vector<Vector<double>> const& sets,
Vector<int> current, FUNCTION const& f = FUNCTION(),
int remainingEvals = 100)
{//start with medians
    Vector<double> best;
    for(int i = 0; i < sets.getSize(); ++i)
    {
        assert(0 <= current[i] && current[i] < sets[i].getSize());
        best.append(sets[i][current.lastItem()]);
    }
    double bestScore = f(best);
    Vector<int> preferredSign(sets.getSize(), 1);
    for(bool done = false; !done;)
    {
        done = true;
        for(int i = 0; i < sets.getSize(); ++i)
            for(int j = 0; j < 2; ++j)
            {
                int sign = preferredSign[i];
                if(j == 1) sign = -sign;
                int next = current[i] + sign;
                if(0 <= next && next < sets[i].getSize())
                {
                    if(remainingEvals-- < 1)
                        return make_pair(best, make_pair(bestScore, 0));
                    best[i] = sets[i][next];
                    double score = f(best);
                    if(score < bestScore)
                    {
                        current[i] = next;
                        bestScore = score;
                        done = false;
                        preferredSign[i] = sign;
                        j = 2;
                    }
                    else best[i] = sets[i][current[i]];
                }
            }
    }
    return make_pair(best, make_pair(bestScore, remainingEvals));
}
```

A good starting point consists of the median of each variable set, which is reasonable assuming the ranges are chosen properly:

```
template<typename FUNCTION> Vector<double> compassDiscreteMinimize(
    Vector<Vector<double>> const& sets, FUNCTION const& f = FUNCTION(),
    int remainingEvals = 100)
{
    Vector<int> current;
    for(int i = 0; i < sets.getSize(); ++i)
    {
        assert(sets[i].getSize() > 0);
        current.append(sets[i].getSize()/2);
    }
    return compassDiscreteMinimizeHelper(sets, current, f,
                                         remainingEvals).first;
}
```

This can get stuck in a local minimum, so can adapt globalization strategies. But don't discuss this fur-

ther because in most cases can solve by mapping to a continuous problem. From my experiments the algorithm beats grid search for SVM parameter optimization (see the “Machine Learning—Classification” chapter) in terms of the number of evaluations for the same quality of learning.

24.11 Stochastic Optimization

For noisy f , want to minimize $E[f(x)]$. The expectation isn't the only way to specify qualities of a desired minimum—e.g., usually prefer x_2 with slightly larger $E[f(x_2)]$ but much smaller variance, but such formulations lead to much harder multiobjective optimization.

Can convert a stochastic problem into a low-noise deterministic one using **sample average approximation (SAA)**, i.e., minimize $g(x) = \text{the average of evaluating } f(x) n \text{ times for some } n$. For $n \rightarrow \infty$, $g(x) \rightarrow E[f(x)]$. This works with discrete, constrained, and global minimization. But it's unclear how to pick n .

Sample path approximation (SPA) first solves the problem with a small n and then many times with doubled n , using the solution of the previous round as the starting solution of the next. So heuristically, in the later rounds, though more simulations are used per evaluation, the algorithm solving the deterministic problem should converge quickly. Deem convergence when the averages and the found solutions stop changing.

Because g has small jump discontinuities, theoretically can't apply many deterministic optimization algorithms as is. E.g., finite-difference derivative will be garbage for the default choice of h . But Nelder-Mead, compass, and coordinate descent work as is. With a large enough initial step that exceeds the noise, they are likely to find a solution as close to the optimum as the noise allows.

While this is a major use case for Nelder-Mead, it's wasteful to evaluate f many times at the same x for noise reduction. More efficient algorithms optimize and reduce noise simultaneously, avoiding bad solutions without needing their accurate values.

24.12 Stochastic Approximation Algorithms

Robbins-Munro algorithm (RM) solves a system of stochastic equations $f(x) = 0$ given observations $m(x)$ of $f(x)$ such that $E[m(x)] = f(x)$:

1. Pick the starting x_0
2. For some large n and step sizes s_i , while $i < n$
3. $x_{i+1} = x_i - s_i m(x_i)$

For convergence to a root need some smoothness conditions and that $\sum_{0 \leq i \leq \infty} s_i = \infty$, and $\sum_{0 \leq i \leq \infty} s_i^2 < \infty$. The smoothness conditions are statistical or ODE-based but impossible to verify in practice. See Spall (2003) for an overview.

Under even further conditions, using $s_i = \frac{1}{i+1}$ is asymptotically optimal and leads to $O(1/\sqrt{n})$ convergence (Spall 2003). Because any algorithm, even if starting from a root x , needs to at least verify that $f(x) = 0$ using Monte Carlo, whose convergence is also $O(1/\sqrt{n})$, RM is asymptotically optimal. But finding a root is

more important than estimating its value, and a slower-decreasing $s_i = \frac{1}{(i+1)^{0.501}}$ usually leads to faster search, despite worse asymptotic convergence (Spall 2003). In some sense, asymptotic convergence matters only for polishing the precision because it kicks in after already got some. Constant s_i is also used often, but doesn't satisfy the conditions. RM is very sensitive to s_0 because too large values diverge and too small ones converge too slowly. As for SPSA (discussed later), can use grid search to find a suitable value.

Stochastic gradient descent (SGD) uses RM to solve $\nabla f = 0$. Though don't know f , can come up with an unbiased estimator of ∇f with enough knowledge of m . Let ϵ_i come from a probability distribution with PDF p . Then $\nabla f(x) = \frac{\partial}{\partial x} \int_{-\infty < t < \infty} m(x, \epsilon(t, x)) p(\epsilon(t, x)) dt$. If ϵ doesn't depend on x , and can interchange

the derivative and the integral, this simplifies to $\nabla f(x) = E \left[\frac{\partial m(x, \epsilon)}{\partial x} \right]$. In many cases, can compute

$\frac{\partial m(x, \epsilon)}{\partial x}$ analytically. SGD also works with nonsmooth f using subgradients.

E.g., given a set of points (x_i, y_i) , linear regression computes a hyperplane $y(x) = wx$ such that $\sum (y(x_i) - y_i)^2$ is minimal. When points arrive online, $m(w, \epsilon_i) = (y(x_i) - y_i)^2$ is an unbiased estimator of

the true error on point i , and $\frac{\partial m(w, \epsilon_i)}{\partial w} = x_i(y(x_i) - y_i)^2$. So the RM iteration is $w = s_i x_i (y(x_i) - y_i)^2$.

Though RM easily converges for finding roots, SGD can swing into ∞ quickly if s_i are too large, because don't check for decrease in f .

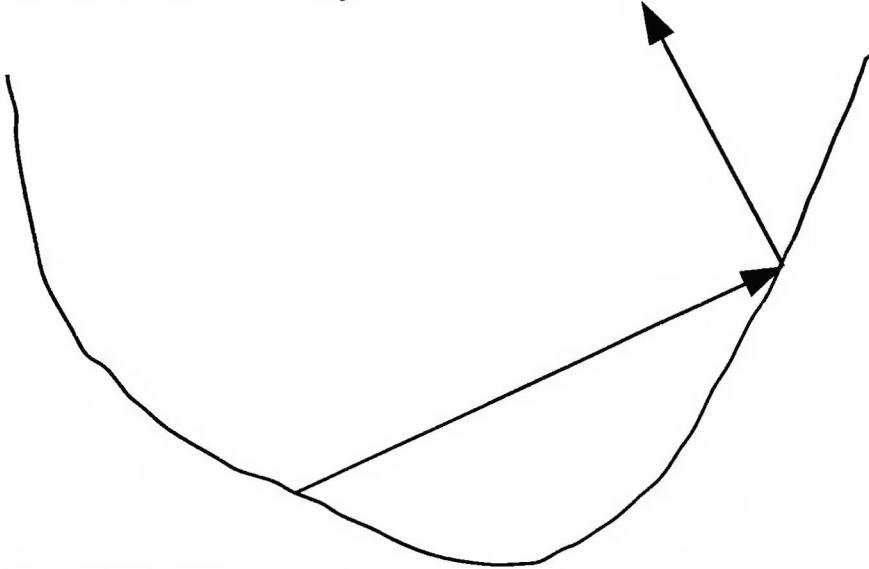


Figure 24.8: Divergence if s_i are too large

When don't have an analytical unbiased gradient estimate, **stochastic perturbation stochastic approximation algorithm (SPSA)** estimates it numerically:

1. Pick the starting point x_0 , gradient step sizes h , and displacement vectors v ,
2. While $i < n$
3. For $0 < j < D$
4.
$$x_{i+1}[j] = x_i[j] - s_i \frac{m(x_i + h_i v_i) - m(x_i - h_i v_i)}{2 h_i v_i[j]}$$

Usually v_i are vectors of Bernoulli random variables that are 1 or -1 with probability $\frac{1}{2}$. Other choices are possible but introduce complications (Spall 2003). For convergence to a local minimum, i.e., a root of ∇f need smoothness conditions and that $\sum_{0 \leq i \leq \infty} s_i = \infty$, and $\sum_{0 \leq i \leq \infty} \left(\frac{s_i}{h_i}\right)^2 < \infty$. Again, the smoothness conditions are statistical or ODE-based but impossible to verify in practice.

The asymptotically optimal choices are $s_i = \frac{1}{i+1}$ and $h_i = \frac{1}{i+1^{1/6}}$, leading to $O(n^{-1/3})$ convergence (Spall 2003), which is worse than that of SGD. Like for SGD, slower-decreasing $s_i = \frac{1}{i+1^{0.602}}$ and $h_i = \frac{1}{i+1^{0.101}}$ usually lead to faster search. The bias of SPSA gradient is $O(h_i^2)$, which is the same as that of the forward difference estimate, but needs 2 and not D evaluations of m (Spall 2003).

SPSA is very sensitive to s_i and h_i , particularly the initial values. Because it doesn't ensure descent, for rapidly changing f a few wrong steps can throw x into ∞ if s_0 is too large, and x will negligibly crawl to an optimum if too small. Want the largest s_0 that won't result in a very distant jump. Another problem is that h_i effectively don't change.

A practical approach is to use the same initial step for both s_i and h_i to cut the number of parameters and make the search more stable. Then use grid search with a range of exponentially decreasing step sizes, by default from 2^{10} to 2^{-20} , accepting only runs that give improvement. This finds a suitable initial step because too large steps will diverge to ∞ and polishes the precision because restarting the search with a smaller initial step from the found solution will overcome the effects of $s_i \rightarrow 0$ too quickly.

```
template<typename POINT, typename FUNCTION> POINT SPSA(POINT x,
    FUNCTION const& f, int maxEvals = 10000, double initialStep = 1)
{
    POINT direction = x;
    for(int i = 0, D = x.getSize(); i < maxEvals/2; ++i)
    {
        for(int j = 0; j < D; ++j) direction[j] =
            GlobalRNG().next() % 2 ? 1 : -1;
    }
    for(int i = 0, D = x.getSize(); i < maxEvals; ++i)
    {
        for(int j = 0; j < D; ++j) direction[j] =
            GlobalRNG().next() % 2 ? 1 : -1;
    }
}
```

```

        double step = initialStep/pow(i + 1, 0.101), temp =
            (f(x + direction * step) - f(x - direction * step)) /
            (2 * pow(i + 1, 0.501));
        if(!isfinite(temp)) break;
        for(int j = 0; j < D; ++j) x[j] -= temp/direction[j];
    }
    return x;
}
template<typename POINT, typename FUNCTION> pair<POINT, double> metaSPSA(
    POINT x, FUNCTION const& f, int spsaEvals = 100000, int estimateEvals =
    100, double step = pow(2, 10), double minStep = pow(2, -20))
{
    pair<POINT, double> xy(x, numeric_limits<double>::infinity());
    for(; step > minStep; step /= 2)
    {
        if(isfinite(xy.second)) x = SPSA(xy.first, f, spsaEvals, step);
        double sum = 0;
        for(int i = 0; i < estimateEvals; ++i) sum += f(x);
        if(sum/estimateEvals < xy.second)
        {
            xy.first = x;
            xy.second = sum/estimateEvals;
        }
    }
    return xy;
}

```

24.13 Linear Programming

Want to optimize a linear function given linear constraints. For simplicity, the vector transpose notation is omitted. The **simplex method** solves a linear programming problem

- $\min cx$ subject to
- $Ax = b$
- $x \geq 0$

where A is a $n \times m$ matrix with $n \geq m$, c and x vectors of size n , and b of size m .

The x are ordered so that $x = (x_b, x_n)$, where x_b is vector of m nonzero **basic variables**, and x_n of $n - m$ **zero variables**. Likewise, $C = (c_b, c_n)$ and $A = (A_b, A_n)$:

- $\min z = c_b x_b + c_n x_n$ subject to
- $A_b x_b + A_n x_n = b$
- $x \geq 0$

Can enumerate all solutions by solving $Ax = b \forall$ combination of basic variable choices. A more efficient process is greedy local search (Griva et al. 2008):

1. Start with an initial feasible basis
2. Until the solution is optimal
3. Improve the current basis by swapping a basic variable with a nonbasic one

Let $y = c_b A_b^{-1}$. Then $z = yb + (c_n - yA_n)x_n$. If $c_n - yA_n > 0$, can't decrease z by increasing any x_n , so can't improve the current basis by variable swapping. Linear programming is convex, so this local optimality test is global and allows to select the **entering nonbasic variable** as the one with smallest $c_n - yA_n$.

The **leaving variable** can be the one which allows the entering variable to take on the maximum positive value while keeping other variables ≥ 0 . Let vector a be the column of A_n corresponding to the entering variable, and $v = A_b^{-1}sa$. Because all components of x_n except for the selected s remain 0, $b = A_b(x_b + v)$. For b to remain the same, s increase, and the basis be valid, one component of x_b needs to decrease to 0, and the rest change value but remain ≥ 0 . If $v \leq 0$, the problem allows an infinitely large solution because an arbitrarily large s is possible when one of the basis variables increases. Variable i , decreasing which to 0 gives

$\max s$, is the one minimizing $\frac{x_b(i)}{v(i)}$ with $v(i) > 0$, where $x_b = A_b^{-1}b$. The implementation avoids numerically

unstable inverses using LUP decompositions of A_b and its transpose to solve equivalent equations. Remember that the inverse and the transpose of matrix are commutative. Alternatively, instead of using a transpose

can compute $yA_n = c_b A_b^{-1} A_n$ using the fact that can do the latter matrix multiplication by solving one column at a time (see the "Numerical Algorithms—Introduction and Matrix Algebra" chapter on techniques for avoiding using inverses).

```

struct LinearProgrammingSimplex
{
    Matrix<double> B, N;
    Vector<double> b, cB, cN, x;
    Vector<int> p;
    bool isUnbounded;
    bool performIteration()
    {
        LUP<double> lup(B), lupT(B.transpose());
        x = lup.solve(b);
        //check if x is optimal or find entering variable
        Vector<double> y = cN - lupT.solve(cB) * N;
        int entering = 0;
        double bestValue = y[0];
        for(int i = 1; i < y.getSize(); ++i) if(y[i] < bestValue)
        {
            bestValue = y[i];
            entering = i;
        }
        if(bestValue >= 0) return false;
        //find leaving variable
        Vector<double> a;
        for(int i = 0; i < N.rows; ++i) a.append(N(i, entering));
        a = lup.solve(a);
        int leaving = -1;
        double minRatio, maxA = -1;
        for(int i = 0; i < x.getSize(); ++i) if(a[i] > 0)
        {
            double newRatio = x[i]/a[i];
            if(leaving == -1 || minRatio > newRatio)
            {
                leaving = i;
                maxA = max(maxA, a[i]);
                minRatio = newRatio;
            }
        }
        if(maxA <= 0){isUnbounded = true; return false;}
        //swap variables
        for(int i = 0; i < N.rows; ++i){swap(B(i, leaving), N(i, entering));}
        swap(p[leaving], p[entering]);
        swap(cB[leaving], cN[entering]);
        return true;
    }
    LinearProgrammingSimplex(Matrix<double> const& B0, Matrix<double>
        const& N0, Vector<double> const& cB0, Vector<double> const& cN0,
        Vector<double> const& b0): isUnbounded(false), B(B0), N(N0), cB(cB0),
        cN(cN0), b(b0), x(b)
        {for(int i = 0; i < cB.getSize() + cN.getSize(); ++i) p.append(i);}
    Vector<pair<int, double>> solve()
    {
        while(performIteration());
        Vector<pair<int, double>> result;
        if(!isUnbounded)
            for(int i = 0; i < x.getSize(); ++i)
                result.append(make_pair(p[i], x[i]));
        return result;
    }
};

```

Simplex fails if A_b is singular, the problem is unbounded, or the problem is **degenerate**, i.e. allowing

cycling between two basis of equal value. More complicated variable selection rules avoid degeneracy, but in practice rounding errors are enough. The number of iterations is exponential in the worst case but linear in practice. Each takes $O(m^3)$ time due to LUP (but see the comments).

General linear programming minimizes a linear combination of n variables, subject to m inequality constraints on linear combinations of the variables:

- $\min cx$ subject to
- $Ax \geq b$
- $x \geq 0$

The feasible region consists of values of x that satisfy the constraints. Redundant constraints don't affect it. To make inequalities equalities, add a **slack variable** vector, and augment A with the $m \times m$ identity:

- $\min cx$ subject to
- $Ax + Is = b$
- $x \geq 0$ and $s \geq 0$

Other transformations allow further generalizations:

- If $x(i) < 0$ for some i , replace x with $x_1, x_2 \geq 0$ such that $x_1 - x_2 \geq 0$
- Replace any constraint equality $\sum A_{ji} x_i = b_j$ by both $\sum A_{ji} x_i \geq b_j$ and $-\sum A_{ji} x_i \geq -b_j$ to allow a simple initial feasible basis

Setting slack variables = 0 forms the initial basis. If it's infeasible, expand the feasible region to include the origin by adding to each violated constraint an **artificial variable** with the initial value > 0 and changing the cost of each variable to 1 if artificial and 0 otherwise. Solving makes artificial variables 0 and succeeds if the program has at least one feasible solution. Then the current values of regular variables are feasible for the original program.

Much detail goes into a robust implementation—commonly recommended references are Maros (2002) and PAN (2014).

24.14 Some Thoughts on Nonlinear Programming

The most general constrained optimization problem is:

- $\min f(x)$ subject to
- $g_i(x) \leq 0$
- $h_i(x) = 0$

If f and g_i are convex, and h_i are linear, the problem is convex. If the feasible region defined by g_i and h_i is convex but they aren't, can create an equivalent problem where they are (Boyd & Vandenberghe 2004).

The **Lagrangian of the problem** is $L(x, l, v) = f(x) + \sum l_i g_i(x) + \sum v_i h_i(x)$. Let x^* be the optimal solution. The **dual problem** $g(l, v) = \min_{x \in \text{feasible region}} L(x, l, v)$ is concave and $< x^* \forall l, v$. It asks for the maximum feasible lower bound on the solution, so negate the objective:

- $\min -g(l, v)$ subject to
- $l_i \geq 0$

It's convex. E.g., for linear program:

- $\min cx$ subject to
- $Ax = b$
- $x \geq 0$

$g(a, b) = -bv$ if $Av - b + c = 0$ and $-\infty$ otherwise. So the dual problem is:

- $\min bv$ subject to
- $-Av \leq c$

Replacing " $Ax = b$ " by " $Ax \leq b$ " replaces " $-Av \leq c$ " by " $-Av = c$ ".

If d^* is the optimal solution to the dual, $d^* \leq x^*$. This is **weak duality**. For **Strong duality** $d^* = x^*$ holds if a **constraint qualification** holds. Some are:

- $g_i(x^*)$ and $h_i(x^*)$ are linearly independent
- g_i and h_i are linear
- The problem is convex, and \exists a feasible solution

If strong duality holds, \forall optimum x^*, l^*, v^* **Karush-Kuhn-Tucker (KKT) conditions** hold:

- $g(x^*) \leq 0$
- $h(x^*) = 0$
- $l^* \geq 0$
- $l^* g(x^*) = 0$

- $\nabla L(x^*, l^* v^*) = 0$

Have a unique solution if the problem is convex. Duality and KKT conditions transform problems into possibly easier ones. E.g., the linear programming dual is easier if the problem has fewer variables than constraints because the dual has the reverse.

For specific algorithms see Nocedal & Wright (2006), Boyd & Vandenberghe (2004), and references therein; major implementations are reviewed in Andrei (2017). Terlaky et al. (2017) has many up-to-date surveys with rich bibliography and applications.

A major difficulty is that random sampling doesn't work with equality constraints, so need extra knowledge about the problem to derive a solution, e.g., by using manual preprocessing to solve for the equality constraints. With non-0-volume feasible regions (i.e., no equality constraints) can use random sampling for adapting the discussed global methods.

A general method is to run unconstrained algorithms as is, but change the unconstrained function to return ∞ at infeasible points. Then for convex f and convex feasible region with no equality constraints, many algorithms give reasonable answers even without explicit projections of their search path onto the active constraint subset. Having a large, finite penalty is also an option, and can increase it gradually.

Convexity is crucial in the ability to solve efficiently, i.e., don't have efficient methods for nonconvex programs.

24.15 Implementation Notes

Almost every implementation has something original, and in most cases algorithm selection has been the main challenge.

- L-BFGS is the only gradient-based algorithm, and I gave it many robustness heuristics
- Fibonacci search with extensions is the only 1D-search algorithm and is very robust
- Global optimization algorithms are as much as possible extensions of the corresponding combinatorial ones

The linear programming implementation is very basic and could use more tricks for efficiency, but I choose to make it simple and not present more complicated algorithms because of the major effort needed to implement them well.

24.16 Comments

Though golden section is worst-case optimal, **Brent search** (which is much more complicated; see Press et al. 2007) may be faster for smooth f .

For strong Wolfe line search more sophisticated doubling/zooming strategies use quadratic or cubic interpolation based on recent f and directional derivative values, but they need to protect from the worst case by doing bisection occasionally (Nocedal & Wright 2006; Moré & Thuente 1994). These aren't considered further, mostly because with finite-difference gradients it makes sense to do "exact" search first. Also don't consider the supposedly better line search based on approximate Wolfe conditions (Hager & Zhang 2005) that are designed to account for floating point computations. It's not as popular as the presented version, and errors in numerical gradients defeat the point.

For robust L-BFGS it may seem better to check the angle between the step and the steepest descent. E.g., can require that $\cos(\text{angle}) < 0.01$. But in poorly conditioned cases such tests fail because might need small angle steps (Nocedal & Wright 2006).

For BFGS an appealing idea seems to be setting B to the gradient-finite-difference approximation of the Hessian. Then update its Cholesky decomposition with the changes, like QR for Broyden's method. This also allows to enforce positive-definiteness and monitor conditioning of B and restart or take another corrective step if needed (Dennis & Schnabel 1996). This seems to lead to more stability compared to the BFGS formula (which uses Sherman-Morrison formula). But, as already mentioned, this is complicated and unnecessary inefficient. In particular, it doesn't allow using limited memory, the $O(D^3)$ time of factoring B at the start doesn't scale, and clumsy monitoring of potentially outdated state for numerical issues is worse than forgetting it. Even logically, no amount of condition monitoring can guarantee preservation of good numerical state, and would occasionally need to restart. This isn't worth it even for BFGS (Nocedal & Wright 2006) because starting with the identity B works well. In general, trust region methods (discussed later) are better at maintaining a full model.

Conjugate gradient methods (Nocedal & Wright 2006) are a slightly simpler alternative to L-BFGS, but with issues:

- The performance is worse. Nash (2014) reports that from his experience classical conjugate gradient methods aren't particularly good. The newer **Hager-Zhang** formula (Hager & Zhang 2005) is

claimed to be faster in runtime than L-BFGS despite doing more evaluations. Also, the latest version of this method (Hager & Zhang 2013) uses memory and is claimed to match L-BFGS on evaluations while being more economical on the state update. But for expensive f the state update isn't the bottleneck, and the L-BFGS update is only a constant factor more expensive, so need more research to establish the merits of this method. In general, the number of evaluations should be the only metric of interest.

- The convergence theory is very different. L-BFGS is ultimately based on Newton's methods with the positive-definite enforcement. Conjugate gradients take their own approach by maintaining orthogonality, based on the assumptions that they minimize a quadratic exactly after D iterations. But with large deviations from the quadratic or ill-conditioned Hessians, this is subject to numerical issues, so need occasional restarting. How much the memory-use version is able to avoid this remains to be researched. Also in this chapter the emphasis is on when finite difference is used to estimate gradients, which seems to be more stable with Newton-based methods because positive-definiteness is easier to maintain than orthogonality. But this is something that needs further research.

Conjugate gradient needs a bit less from the problem to guarantee convergence (e.g., no bound on $\kappa(H_0)$; for details with an older formula see Nocedal & Wright 2006), but this doesn't seem to make a difference in practice. For further reading also see Pytlak (2008).

Trust region methods (Nocedal & Wright 2006) are an alternative to line search for controlling global convergence of algorithms. The idea is to assume that the model of a method (such as the implicit quadratic model of Newton's method) is valid within some small radius and pick the next point as the best point in the model subject to such restriction. This is similar to what compass search does. Though have convergence under some simple conditions, and the can adapt the trust radius, have some problems:

- Solve the radius-constrained problem by running a nested solver until some acceptable precision, but this is more difficult than line search.
- For the Newton model the approach is scalable only if the Hessian is sparse and can be computed as such. One particular algorithm is **truncated Newton**—use linear preconditioned conjugate gradients until non-positive-definiteness is detected.

Scalable line-search algorithms like L-BGFS win over trust region methods in general. Nor do the latter seem to have advantages for equation solving. But they don't run into the problem of dealing with non-positive-definite Hessian, so have advantages in application-specific cases when have 2nd derivative information, and D is small or the Hessian is sparse. This is often the case for least-squares problems, where specialized algorithms such as **Levenberg-Marquardt** are used (Nocedal & Wright 2006).

A generalization of compass search is **MADS** (Audet & Hare 2017). It allows a dense set of directions in the limit if stalled, bypassing the limitation of coordinate descent and compass with nondifferentiable f . But the mechanics is to clumsy to ensure convergence. Compass search is convergent on differentiable f by implicitly being on a mesh of powers of two of the initial step (see Kolda et al. 2003 for a detailed description of the theory). MADS is maintaining that mesh with a lot of cleverness. Instead can simply use random directions, and decrease the step by a factor of 2^{-2n} if unsuccessful, otherwise doubling. This technically isn't convergent because don't impose any form of a sufficient descent condition (unless use a hard-to-scale **forcing function**—see Kolda et al. 2003), but has the same convenience as MADS of flawlessly handling constraints.

For global optimization I tried several other metaheuristics, but for the problems tested they were outperformed by the hybrid and its components. The comparisons here aren't meant to be statistically meaningful, only to rule out bad algorithms. A proper comparison is tricky, see Clerc (2015) for some ideas about what makes sense, and what doesn't. All selections and combinations are based on my own experiments.

Given lack of statistical evidence, prefer simple methods that are easier to understand, converge, and make sense from worst-case reasoning. Methods that don't have such characteristics aren't necessarily bad, but there is no reason for them to be better. Many metaheuristics come with marketing, where a proposed algorithm is shown to be a bit better on some set of problems and then applied to solve practical problems in subsequent papers. Because the designers have substantial commercial interest, must ask:

- Would the method do better on other f , and were the competitors tuned properly?
- Would another algorithm do just as well or better on the mentioned practical applications?

Some general ideas (in addition to what is discussed in the “Optimization Algorithms” chapter):

- Many algorithms are inspired by some natural process. This might lead to good biased sampling during an otherwise random walk, and not because of some grand logic that's correct by definition. Such algorithms need to be simplified or improved using mathematical analysis and experimentation, and the natural process only occasionally gives hints about certain choices.
- Due to the NFL and the difficulty of proper testing, numerous algorithms and variations have been

proposed. To cut down the search:

- Only algorithms that have a history of good performance and significant use are worth considering
- Algorithms must not have too much extra setup logic, and their mechanics must be close to the problem
- All parameters other than the limit evaluation budget must have good defaults
- Only one version of the algorithm is tried—usually the basic one with the most logical parameter settings
- Algorithms sensitive to minor changes are unstable and can't be a robust choice

See Weise (2011) for an illustrative discussion of optimization landscapes from the perspective of continuous optimization.

For unbounded iid and Markovian samplers a question is how fat the tail should be, which is unclear. Pareto distribution (Wikipedia 2017b) can give tail with $O(x^a)$ probability of $x > 0$. Maybe $a = 0.3$ or so is perhaps the fattest useful tail, but unclear if anything is gained by this because at least must respect the floating point range with very high probability. Here, to get negative step and those in range $[-1, 1]$ can use a mixture distribution if needed for some reason. Levy distribution has stability, i.e., several Levy steps are equivalent to a single step with different parameters, which can be useful. But no tail can be fat enough for a truly global search because any such distribution has a certain median (fixed to be 1 in the implementations here). So points inside a hypersphere of radius = the median are as likely as the points outside, which is a local bias. So even though the local Markovian search technically converges globally as is because at any sample it can get lucky, this isn't particularly meaningful.

Some specific algorithms (for details see the references if curious—the descriptions here only specify implementation choices for completeness):

- **Particle swarm optimization** was not considered because it needs far more setup and parameters than seems reasonable. Also in some of the studies such as Civicioglu, & Besdok (2013), it and some other recent algorithms didn't outperform differential evolution, which has been the same for 20 years, while all other algorithms in the study have recent not-yet-validated improvement proposals.
- **Cholesky CMA-ES**. The **CMA-ES** algorithm and variants have been doing well in many recent benchmarks. Cholesky is the simplest and needs $O(D^2)$ update (most others needs $O(D^3)$). The implementation is exactly based on the pseudocode and parameter recommendations in Bäck et al. (2013). The parameter “ σ ” is set to the usual initial x -scale = $\max(1, \|x\|_2)/10$. Surprisingly, it performed worse than random-before-local search (yes, I checked the implementation for correctness). Its use case appears to be only noisy local minimization where it would replace Newton-based methods.

The selection is representative, though certainly not comprehensive. Any good algorithm must not try to look for a needle in a haystack because it's wasteful and unlikely to succeed.

An open problem is whether for practical problems population methods have overall advantage over single-solution methods, which currently doesn't appear to be true. It seems that population methods just run interacting state-based samplers and technically shouldn't be able to outperform Markovian search and some of its obvious extensions, such as taking multiple steps. Also can have population collapse and waste evaluations on poor quality population members (iid samples should be less wasteful). It seems that a population offers memory, but this is useful only when it's exploited well. Convergence for single-solution methods is generally easier to prove. For population-based methods a collapse sometimes leads to complete stagnation.

Many such interesting questions may not be impacted by the NFL. E.g., is it always better to sample from fat tail distributions? Not if the global minimum is nearby. But in this case the loss of efficiency is small, and when it's far, fat-tail sampling should do better. f where no biases can be exploited are called **deceptive**, and don't expect good performance from any algorithm. E.g., when optimizing a function in 2D by looking at a graph, usually don't think that it can have a singularity between the pixels, though it can. The possibility of encountering mildly deceptive f increases the need for hybrid algorithms that work with several common biases.

It appears that the most useful result of metaheuristic research is discovering better ways to explore and exploit. Hope that eventually will use some of these in algorithms of choice. Some particular discoveries include:

- DE mutation operator for global sampling
- Levy samples for fat-tail local sampling
- “Bare-bones” version of many algorithms

Almost every known metaheuristic currently has a proposed improvement which however isn't clearly

accepted as such. Designing algorithms purely to do well on standard test sets leads to overfitting.

Metamodel optimization is promising. Try to find some multidimensional interpolation to the problem, and optimize it to find the next sample point. In 2D a good approach is **Kriging/Gaussian process/Bayesian optimization**, but it's computationally expensive and doesn't seem popular beyond small D . For very expensive f , such as car crash experiments or long simulations, this is by far the method of choice. The metamodel can be any regression method (i.e., random forest and not Kriging), and can optimize it using the black-box methods in this chapter. Because typically start a metamodel with a few random/grid points, need a budget of more than a small number of evaluations.

If f isn't fully black box, and have some useful information such as an upper bound on the Lipschitz constant, might solve its global optimization exactly using branch-and-bound methods (Locatelli & Schoen 2013). In some cases can heuristically estimate the Lipschitz constant, such as by using extreme order statistics to deduce bounds on min an max f -values. Also see Pintér (1996) for many implementation guidelines.

For the simplex method, using sparse matrix algebra and maintaining LUP dynamically is complicated, but substantially speeds up the calculations (Griva et al. 2008). The result is an example of "too technical" algorithms because need clever tricks to get good performance.

Interior point methods are an alternative for solving linear programs. Simplex traverses the edges of the feasible region, but they go through the interior. This takes more time per iteration, but converge in $O(\text{polynomial})$ number of iterations, usually constant in practice.

24.17 Projects

- Does golden section search work on a discrete f ? What modifications are needed, and what can be proved about it's correctness and termination?
- For 1D implement a finite-difference Newton's method where reuse f evaluations for the 2nd derivative estimate. Compare the performance with that of other methods.
- Implement the artificial variable method for linear programming.
- Improve efficiency of the linear programming simplex method by updating a matrix factorization instead of recomputing it at every iteration. Use dense matrix algebra.
- Research and implement some common interior point methods for linear programming.
- Research and implement some common interior point methods and other methods for nonlinear programming.
- Investigate the MADS method from Audet & Hare (2017) as an alternative to compass. Extend the result to work with box constraints.
- Research model-based derivative-free algorithms. See Audet & Hare (2017) and Conn et al.(2009) for an introduction to the concept. From experiments in Hansen et al. (2009) the **NEWUOA algorithm** of Powell seems most promising. It builds as quadratic model incrementally from few evaluations, but from the comments in Conn et al. (2009) might use some improvements.
- For Markovian search, instead of the Levy move try normal move of similar scale. How does performance differ?
- Make local search change x by reference for efficiency with few moves and large n .
- Reasearch and implement the simulated annealing variant from the GenSA (Xiang et al. 2013), which performs well according to Nash (2014).
- Explore other variants of CMA-ES. IPOP-CMA-ES is considered a benchmark among them. Try also ILS with the version implemented here, which is essentially a local search. Particularly interesting are the limited-memory versions, inspired by L-BFGS—see Varelas et al. (2018) and references therein. This is still a developing area.
- Research and implement Bayesian optimization. Try the result for machine learning parameter selection instead of the currently used optimization technique.
- Research and compare found and presented methods for noise optimization. Use the existing test set with relatively low-level generated noise. Random search may be a good benchmark.
- For differential evolution and genetic algorithm crossover moves try to mix on average 10% or so and not 50% of the two solutions. Does this improve performance by retaining good building blocks better?
- For small evaluation budgets it seems better to compare performance by the improvement from the initial value to check percent decrease, possibly on a log scale. Change the performance comparison to do this. Are the conclusions more informative?

24.18 References

- Andrei, N. (2017). *Continuous Nonlinear Optimization for Engineering Applications in GAMS Technology*. Springer.
- Audet, C., & Hare, W. (2017). *Derivative-Free and Blackbox Optimization*. Springer.
- Bäck, T., Foussette, C., & Krause, P. (2013). *Contemporary Evolution Strategies*. Springer.
- Bagirov, A., Karmitsa, N., & Mäkelä, M. M. (2014). *Introduction to Nonsmooth Optimization: Theory, Practice and Software*. Springer.
- Beck, A. (2017). *First-Order Methods in Optimization*. SIAM.
- Bénichou, O., Loverdo, C., Moreau, M., & Voituriez, R. (2011). Intermittent search strategies. *Reviews of Modern Physics*, 83(1), 81.
- Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb), 281-305.
- Bornemann, F., Laurie, D., Wagon, S., & Waldvogel, J. (2004). *The SIAM 100-digit Challenge: a Study in High-accuracy Numerical Computing*. SIAM.
- Bousquet, O., Gelly, S., Kurach, K., Teytaud, O., & Vincent, D. (2017). Critical Hyper-Parameters: No Random, No Cry. *arXiv preprint arXiv:1706.03200*.
- Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.
- Civicioglu, P., & Besdok, E. (2013). A conceptual comparison of the Cuckoo-search, particle swarm optimization, differential evolution and artificial bee colony algorithms. *Artificial intelligence review*, 1-32.
- Clerc, M. (2015). *Guided Randomness in Optimization*. Wiley.
- Clerc, M. (2019). *Iterative Optimizers: Difficulty Measures and Benchmarks*. Wiley.
- Conn, A. R., Scheinberg, K., & Vicente, L. N. (2009). *Introduction to Derivative-Free Optimization*. SIAM.
- Dennis Jr, J. E., & Schnabel, R. B. (1996). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM.
- Griva, I., Nash, S. G., & Sofer, A. (2009). *Linear and Nonlinear Optimization*. SIAM.
- Hager, W. W., & Zhang, H. (2005). A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on optimization*, 16(1), 170-192.
- Hager, W. W., & Zhang, H. (2013). The limited memory conjugate gradient method. *SIAM Journal on Optimization*, 23(4), 2150-2168.
- Hansen, N., Auger, A., Ros, R., Finck, S., & Pošík, P. (2010, July). Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation* (pp. 1689-1696).
- Hastie, T., Tibshirani, R., & Wainwright, M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- Jamil, M., & Yang, X. S. (2013). A literature survey of benchmark functions for global optimization problems. *International Journal of Mathematical Modeling and Numerical Optimization*, 4(2), 150-194.
- Karmitsa, N. (2015). Diagonal bundle method for nonsmooth sparse optimization. *Journal of Optimization Theory and Applications*, 166(3), 889-905.
- Kolda, T. G., Lewis, R. M., & Torczon, V. (2003). Optimization by direct search: New perspectives on some classical and modern methods. *SIAM review*, 45(3), 385-482.
- Lewis, A. S., & Overton, M. L. (2013). Nonsmooth optimization via quasi-Newton methods. *Mathematical Programming*, 1-29.
- Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3), 503-528.
- Locatelli, M., & Schoen, F. (2013). *Global Optimization: Theory, Algorithms, and Applications*. SIAM.
- Maros, I. (2002). *Computational Techniques of the Simplex Method*. Springer.
- Moré, J. J., Garbow, B. S., & Hillstrom, K. E. (1981). Testing unconstrained optimization software. *ACM Transactions on Mathematical Software (TOMS)*, 7(1), 17-41.
- Moré, J. J., & Thuente, D. J. (1994). Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software (TOMS)*, 20(3), 286-307.
- Nash, J. C. (2014). *Nonlinear Parameter Optimization Using R Tools*. Wiley.
- Nesterov, Y. (2018). *Lectures on Convex Optimization*. Springer.
- Nocedal, J., Wright, S. (2006). *Numerical Optimization*. Springer.
- PAN, P (2014). *Linear Programming Computation*. Springer.
- Penot, J. P. (2012). *Calculus Without Derivatives*. Springer.
- Pintér, J. D. (1996). *Global Optimization in Action: Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications*. Springer.
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Science*.

- tific Computing, 3rd ed. Cambridge University Press.
- Price, K., Storn, R. M., & Lampinen, J. A. (2005). *Differential Evolution: A Practical Approach to Global Optimization*. Springer.
- Pytlak, R. (2008). *Conjugate Gradient Algorithms in Nonconvex Optimization*. Springer.
- Simon, D. (2013). *Evolutionary Optimization Algorithms*. Wiley.
- Skjaa, A. (2010). Limited memory BFGS for nonsmooth optimization. *Courant Institute of Mathematical Science, New York, Master's thesis*.
- Spall, J. C. (2003). *Introduction to Stochastic Search and Optimization: Estimation, Simulation, and Control*. Wiley.
- StackOverflow (2018). Approximate order statistics for normal random variables. <https://stats.stackexchange.com/questions/9001/approximate-order-statistics-for-normal-random-variables>. Accessed January 6, 2018.
- Terlaky, T., Anjos, M. F., & Ahmed, S. (Eds.). (2017). *Advances and Trends in Optimization with Engineering Applications*. SIAM.
- Varelas, K., Auger, A., Brockhoff, D., Hansen, N., ElHara, O. A., Semet, Y., Kassib, R. & Barbaresco, F. (2018). A comparative study of large-scale variants of CMA-ES. In *International Conference on Parallel Problem Solving from Nature* (pp. 3-15). Springer.
- Weise, T. (2011). *Global Optimization Algorithms: Theory and Application*, 3rd ed. Downloadable from <http://www.it-weise.de/projects/bookNew.pdf>.
- Wikipedia (2013). Stochastic approximation. http://en.wikipedia.org/wiki/Stochastic_approximation. Accessed May 18, 2013.
- Wikipedia (2017a). Unimodality. <https://en.wikipedia.org/wiki/Unimodality>. Accessed December 10, 2017.
- Wikipedia (2017b). Pareto distribution. https://en.wikipedia.org/wiki/Pareto_distribution. Accessed December 10, 2017.
- Wikipedia (2018). Order statistic. https://en.wikipedia.org/wiki/Order_statistic. Accessed January 11, 2018.
- Xiang, Y., Gubian, S., Suomela, B., & Hoeng, J. (2013). Generalized Simulated Annealing for Global Optimization: The GenSA Package. *R Journal*, 5(1).
- Zhigljavsky, A., & Zilinskas, A. (2007). *Stochastic Global Optimization*. Springer.

25 General Machine Learning

"Ga-ga-ga" – Amelia Kedyk, at age 1, when looking at a toy penguin after having seen geese

25.1 Introduction

The emphasis is on general themes, such as data preparation, that occur in most machine learning tasks. The relevant theory is also discussed. A prerequisite is a good understanding of statistics.

The main takeaway is that machine learning is far from automatic. Assume that a pattern is noise unless proven otherwise. In most domains, where the data is only marginally related to the wanted information, getting a hint from the data is the best that can be expected. Also the performance of a deployed learning system will likely drop compared to the testing one because the data slowly evolves, and users try to game the system. Still, when stable patterns are present in the data and properly learned by an algorithm, the result might be valuable.

25.2 What Machine Learning Is

Imagine that you would like to get a mortgage. You see a banker on the street, greet him, and asked for a mortgage. Can he give you one? No because he doesn't know anything about your situation.

So let's say you go to a bank instead and bring some documentation:

- Your credit score = 680
- Your down payment amount = 30%
- The home's monthly total maintenance $\approx \$1000$

This is great, but what does the bank do with this data? Is it good enough or no? There must be some reliable reference data to base this decision on.

So let's say the bank has payed for a database of past mortgage applications with expert banker decisions to approve or reject. What now? The simplest idea is nearest neighbor—scan the database to find the "closest" application to your and decide based on that. Here base "closest" on some distance function such as Euclidean, with the data scaled the data into $[0, 1]$ range to not compare apples with oranges (discussed later in the chapter).

What if this decision is a "no"—by law mortgage application decisions must be fully transparent so this approach won't do. Another approach is to come up with a decision tree from the data (discussed in the "Machine Learning—Classification" chapter) such as:

- If credit score < 700
 - If down payment < 50%, reject
 - Else approve
- Else
 - If down payment < 20%, reject
 - Else approve

Machine learning is deriving a decision structure automatically from data. It produces decisions, typically called predictions, on data points such as individual mortgage applications.

25.3 Mathematical Learning

Want a predictor f that decides correctly in specific situations, represented as objects in some **feature space** X . X is usually a vector space but doesn't have to be—e.g., a metric space that gives distances between objects is enough for effective learning. In a vector space, every object is a vector of **features**, selected based on domain knowledge. Learning finds f that **partitions** X into possibly uncountably many pieces, assigning $\forall x \in X$ a partition label ID , which can be bounded discrete or continuous. x is assumed to represent all useful information about an object. E.g., to detect free riders in a team shooter game, can use the number of met enemies and the explored % of game space as features, and consider a player who met 0 enemies and explored < 5% of the space a free rider. Here $ID \in \{\text{yes, no}\}$. Assume $x \sim \text{distribution } P$, and $ID \sim \text{conditional distribution } P|x$. Depending on the task, the latter may be continuous, discrete, or Dirac delta. Only the last one describes deterministic learning where $\forall x \exists$ a unique ID .

Instead of explicitly defining f (which might be practically impossible, e.g., how to locate a face in a photo?), an algorithm A sees a finite set S of n examples z_i , from which it hopefully learns f that **generalizes** to unseen z . $\forall z_i$ have x_i and **hint information** $y_i \in Y$. $\forall A$ need to understand its partitioning strategy. For

prediction tasks, $ID_i = y_i$ = a sample from $P|x_i$, but in general y_i can be arbitrary or not exist for some or all i ; y is a function of x , ID , and possibly other information.

Would like to measure learning performance in terms of x and ID , but in general have only x and y . A **loss function** $L_f(z) = L_f(x, y)$ measures errors of f on z . For perfect performance, L must return the minimum possible finite value, usually 0. **Risk** $R_f(P) = E_z[L_f(z)]$ measures the generalization error of f . Usually assume that all z are equally important. Z and L determine a **learning task**, and P a problem within it. A is usually designed to minimize the risk for a specific task, doing which is the goal of learning. Typical tasks:

- **Classification**— $ID = y$ = **class label** $\in [0, k - 1]$. Assuming right decisions have profit 0 and wrong ones cost 1, $L_f(x, y) = (f(x) \neq y)$, with $R_f = E[\% \text{ of misclassified } z]$.
- **Regression**— $ID = y$ are real-valued. Usually $L_f(x, y) = (f(x) - y)^2$, i.e., assume quadratic cost of errors.
- **Clustering**—same as classification, but don't have hint information, i.e., $y = x$. Often use $L_f(x, y) = \text{distance}(f(x), x)$ to group similar x .
- **Semi-supervised learning**—like clustering, but some examples have hint information, which helps improve the learned partition. L is conditional on whether have a hint.
- **Value-function reinforcement learning**—classification or regression where a hint is assigned to a sequence of decisions. E.g., for games like chess some logic tells what is a good move, after a game such logic wins or loses, and this is the hint \forall move, along with extra information such as how far the move was from the end of the game.

E.g., for handwritten digit recognition, want to recognize a handwritten digit from its image, represented by an 8×8 array of cells with gray-scale colors $\in [0, 16]$:

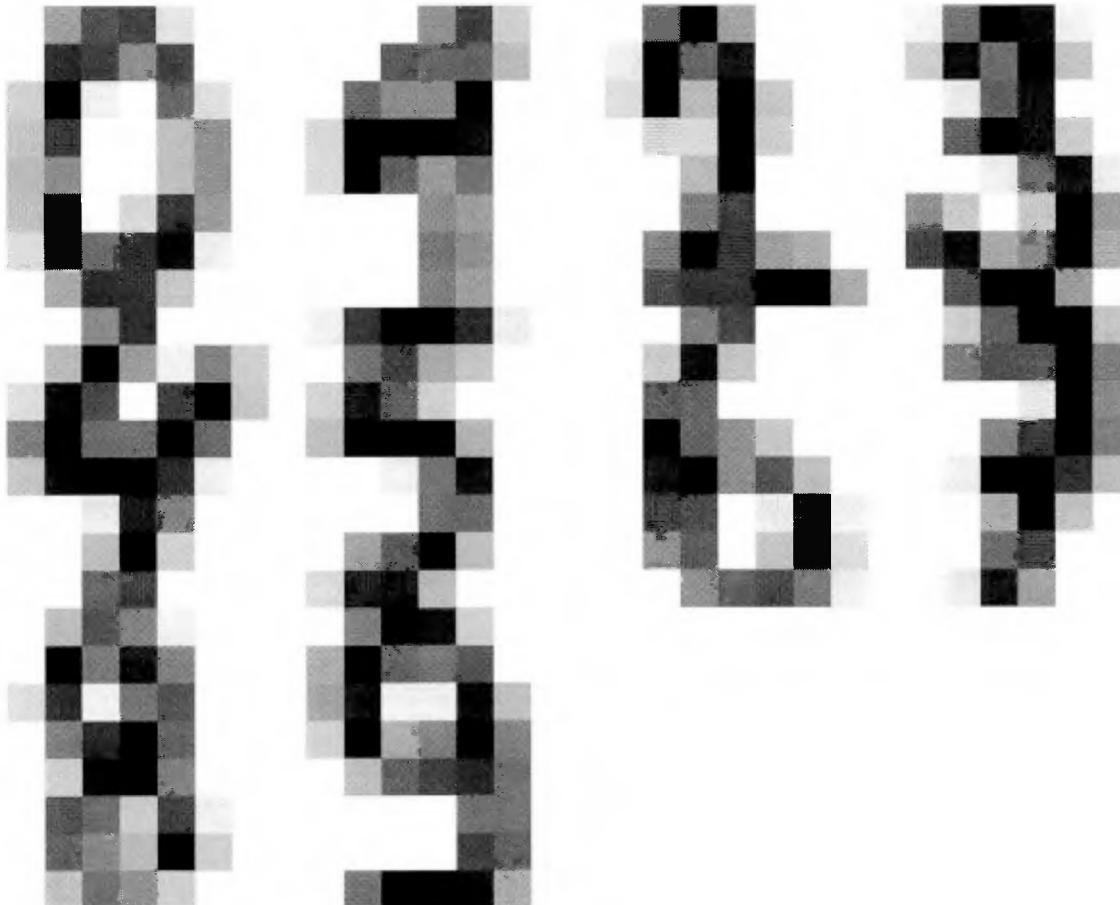


Figure 25.1: Digit data examples

The **iris flower data set** gives measurements of iris leaves and asks to decide what type of iris it is.

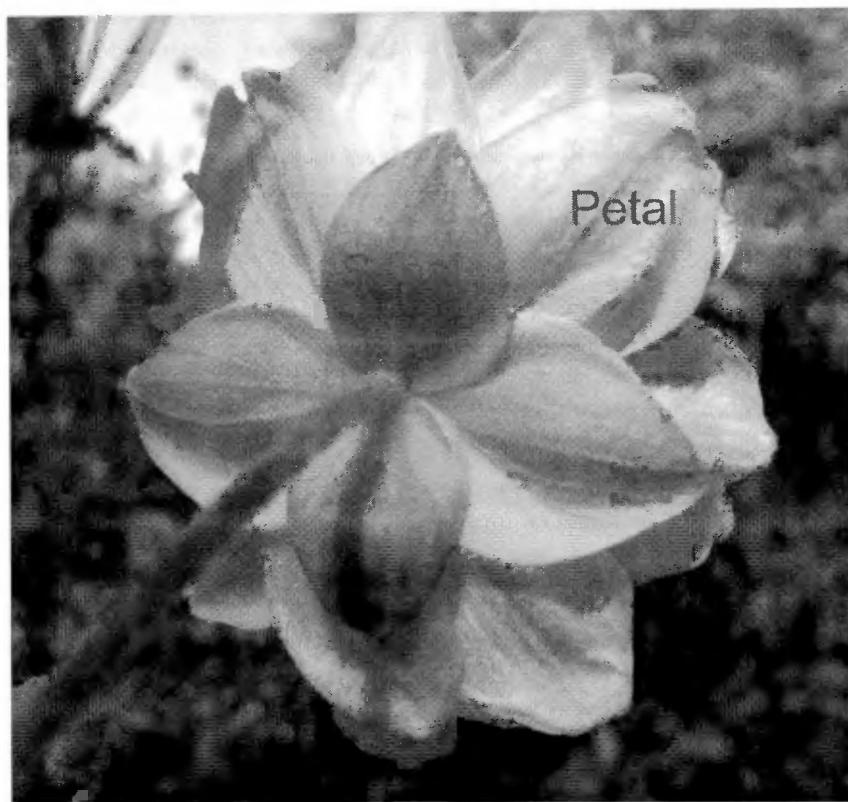


Figure 25.2: A sepal is similar to a petal, but supports it and is usually green

Type	Sepal Length	Sepal Width	Petal Length	Petal Width
Setosa	5.1	3.5	1.4	0.2
Setosa	5.4	3.9	1.7	0.4
Versicolor	6.4	3.2	4.5	1.5
Versicolor	5.6	2.9	3.6	1.3
Virginica	6.2	3.4	5.4	2.3
Virginica	7.2	3.6	6.1	2.5

Predicting resource use of A based on its explicit and implicit inputs is a typical regression problem. An industrial problem is predicting heating load inefficiency based on data such as below:

Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution	Heating Load
0.98	514.5	294	110.25	7	2	0	0	15.55
0.9	563.5	318.5	122.5	7	2	0.1	1	29.03
0.86	588	294	147	7	3	0.1	2	27.02
0.79	637	343	147	7	4	0.1	3	36.97
0.76	661.5	416.5	122.5	7	5	0.1	4	32.31
0.69	735	294	220.5	3.5	2	0.1	5	11.21

The UCI repository (Bache & Lichman 2013) contains these and many other data sets in an easy-to-read format. In the real world, collection and organization of data is usually unique to each task and done by parsing various log files and/or querying databases. Such data usually needs thoughtful cleaning. Though must avoid data snooping by looking at the data before making inferences, also must look at the data to parse it correctly. Though it may be hard, don't use that do get an idea about the data itself!

The data is best represented by a model-view-controller-like accessor interface, which allows creating a data transformation pipeline to avoid excessive copying.

```
template<typename X, typename Y> struct InMemoryData
{
    Vector<pair<X, Y>> data;
```

```

typedef X X_TYPE;
typedef Y Y_TYPE;
typedef X const& X_RET;
InMemoryData() {}
template<typename DATA> InMemoryData(DATA const& theData):
    data(theData.getSize())
{
    for(int i = 0; i < data.getSize(); ++i)
    {
        data[i].first = theData.getX(i);
        data[i].second = theData.getY(i);
    }
}
void addZ(X const& x, Y const& y) {data.append(make_pair(x, y));}
X_RET getX(int i)const
{
    assert(i >= 0 && i < data.getSize());
    return data[i].first;
}
double getX(int i, int feature)const
{
    assert(i >= 0 && i < data.getSize() && feature >= 0 &&
           feature < data[i].first.getSize());
    return data[i].first[feature];
}
Y_TYPE const& getY(int i)const
{
    assert(i >= 0 && i < data.getSize());
    return data[i].second;
}
int getSize()const return data.getSize();
};

```

For vector data need to be able to retrieve the dimension, which works with any data accessor:

```

template<typename DATA> int getD(DATA const& data)
{
    assert(data.getSize() > 0);
    return data.getX(0).getSize();
}

```

The pipeline can get deep sometimes, making access inefficient:

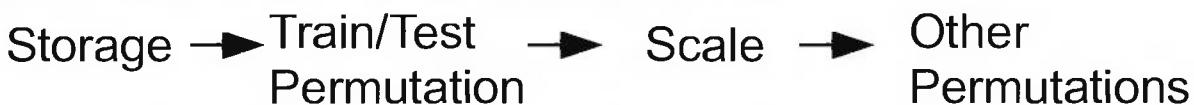


Figure 25.3: A typical data processing pipeline (the specific transformations are discussed later in the chapter)

So can buffer the data before applying A. A buffer is most justified for A that already use much memory, so the extra due to a buffer doesn't change their use cases.

```

template<typename LEARNER, typename BUFFER, typename PARAMS =
    EMPTY> class BufferLearner
{
    LEARNER model;
public:
    template<typename DATA> BufferLearner(DATA const& data,
        PARAMS const& p = PARAMS()): model(data, p) {}
    typename BUFFER::Y_TYPE predict(typename BUFFER::X_TYPE const& x)const
        return model.predict(x);
};

```

Though machine learning isn't useful \forall real world problem, for many it's very successful, in particular (Witten et al. 2016):

- Face detection—implemented in most smart phones and identifies faces quickly and reasonably reliably.

- Re-identification—recovering redacted personal information in published records. E.g., can uniquely identify most people by zip code, birth date, and gender. Famously, the governor who published anonymized health records, was mailed his own.

Learning aims to discover knowledge. Given a problem:

1. Get the data
2. Clean it if needed
3. Run A on it to get f
4. Deploy/use f

(4) is the most critical part of this process despite seeming least so. The final system must be ready to work under many conditions and not be unduly influenced by lets say some minor bias in the training data. Must do validation experiments for proper testing. Some sensitivity analysis is also a good idea—make arbitrary or even illogical changes to the inputs to see how the system reacts to find out more about its robustness. One way to get such data is by using dimension mixing on existing data. None of the above is discussed in machine learning books, but as with all software, careful deployment is a must.

25.4 Predictive vs Structural Inference

In both statistics and machine learning try to estimate something. Perhaps the main difference is that for the former the emphasis is on learning parameters of the model, and for the latter is about getting a model with good predictive performance.

In some cases, such as linear regression, parameter values are derived from usual statistical considerations such as maximum likelihood. But the model is then used for prediction. So predictive inference is in some sense more general than statistical one—minimizing risk is a well-defined problem unlike maximum likelihood, which requires some conditions. In principle can find parameters by risk minimization directly, and much machine learning theory explores how this would work.

25.5 Risk Estimation of a Predictor

F , the space of functions that map X to task-dependent IDs, defines all possible f . $\forall f \in F, R_f$ is well-defined but unknown due to not knowing P . Because randomly sampled S is the only information, at best can know that \exists error ϵ and probability p such that with probability $\geq 1 - p$ have one of:

- $R_f \leq \epsilon$: **f is probably approximately correct (PAC(ϵ, p))**
- $R_f \in [\epsilon_-, \epsilon_+]$: have two-sided confidence

If know such bound, and ϵ is small enough, with $1 - p \approx 95\%$ learning succeeded. Can't guarantee $p = \epsilon = 0$ unless X is finite, and see all of Z instead of S , in which case don't need learning because memorization will do. So must be careful when deploying f as part of a critical system. E.g., can't know what a trained self-driving car will do when an animal suddenly jumps in front of it, and there is a motorcycle behind—such case was certainly not part of training. In many cases need human judgment, e.g., an obsolete work zone sign can cause a car to reduce speed and upset the drivers behind it. Conflicting lane marks, when one is clearly faded to a human, could be another problem.

Inferring general from particular is **induction**, which isn't as well understood as **deduction**, i.e., inferring particular from general. The problem is that a true particular doesn't imply true general, i.e., \exists reverse modus ponens, and PAC is the next best thing. Can have rare events which are genuinely unpredictable, such as the chance of a network failure due to malicious activity. So \exists the true right f , only good enough ones. In a quantum-mechanical world, almost all natural phenomena have some noise. Not being able to get 0 or almost 0 (e.g., 10^{-4}) risk is a problem for many tasks because in many cases otherwise excellent risk such as 1% (assuming classification) isn't enough.

For a set of examples, **empirical risk** $R_{f,n} = \frac{1}{n} \sum L_f(z_i)$. Computing it needs $O(n)$ calls to f .

```
template<typename Y, typename DATA, typename LEARNER> Vector<pair<Y, Y>>
evaluateLearner(LEARNER const& l, DATA const& test)
{
    Vector<pair<Y, Y>> result;
    for(int i = 0; i < test.getSize(); ++i)
        result.append(pair<Y, Y>(test.getY(i), l.predict(test.getX(i))));
    return result;
}
```

When z_i are iid, $L_f(z_i)$ are iid functionals with some unknown distribution. Can use the CLT to get confidence bounds. The result is valid only asymptotically and doesn't give a proper PAC bound but is often used

in practice anyway. If $L \in [0, 1]$, use Hoeffding or empirical Bernstein to get PAC bounds.

Unless found f by a lucky guess without looking at S , $L_f(z_i)$ aren't iid, and the resulting bounds are overly optimistic due to multiple testing. $\exists f$ that learn S well but have poor performance on other examples, which is **overfitting**. E.g., given n phone numbers and credit cards, can fit a polynomial that predicts the credit card number from a phone number and works $\forall z_i$. \exists uncountably many f with 0 risk on any finite data set. Overfitting happens with humans too—e.g., given a set of input-output examples, occasionally interviewees write code that works only for the examples and not in general. But generalization is what matters because f will be used on new, unseen data.

One solution is the **holdout method**—reserve part of the data for risk estimation, and get an estimate and a PAC bound from it using Hoeffding's inequality or the CLT (for an asymptotic bound). Typically use 80% for learning and the other 20% for testing. The implementation uses a permutation to index into the data and deterministically permute it to break any possible sort order. E.g., could have gotten the data from a SQL query which sorted by some database key.

```
template<typename DATA> struct PermutedData
{
    DATA const& data;
    Vector<int> permutation;
    typedef typename DATA::X_TYPE X_TYPE;
    typedef typename DATA::Y_TYPE Y_TYPE;
    typedef typename DATA::X_RET X_RET;
    PermutedData(DATA const& theData) : data(theData) {}
    int getSize() const { return permutation.getSize(); }
    void addIndex(int i) { permutation.append(i); }
    void checkI(int i) const
    {
        assert(i >= 0 && i < permutation.getSize() &&
               permutation[i] >= 0 && permutation[i] < data.getSize());
    }
    X_RET getX(int i) const
    {
        checkI(i);
        return data.getX(permutation[i]);
    }
    double getX(int i, int feature) const
    {
        checkI(i);
        return data.getX(permutation[i], feature);
    }
    Y_TYPE const* getY(int i) const
    {
        checkI(i);
        return data.getY(permutation[i]);
    }
};

template<typename DATA> pair<PermutedData<DATA>, PermutedData<DATA> >
createTrainingTestSetsDetPerm(DATA const& data,
                               double relativeTestSize = 0.8)
{
    int n = data.getSize(), m = n * relativeTestSize;
    assert(m > 0 && m < n);
    pair<PermutedData<DATA>, PermutedData<DATA> > result(data, data);
    Vector<int> perm(n);
    for(int i = 0; i < n; ++i) perm[i] = i;
    permuteDeterministically(perm.getArray(), n);
    for(int i = 0; i < n; ++i)
    {
        if(i < m) result.first.addIndex(perm[i]);
        else result.second.addIndex(perm[i]);
    }
    return result;
}
```

Any minor tweak to a mathematical model can invalidate its assumptions and any dependent analysis. This is particularly problematic for machine learning, where simplified models are analyzed due to the infeasibility of analyzing the actual ones. The conclusions are unlikely to be affected much by minor changes, but rigorous validity disappears. Violation of the iid assumption is probably the most common issue because the data is almost always not completely independent. But it's OK to rely on heuristics when the alternative is to have almost nothing useful. In many cases informally assume some kind of stability, i.e., that the results with mildly violated assumptions are at most negligibly different.

25.6 Sources of Risk

Picking L allows defining the **optimal Bayes learner** that knows P and $\forall x$ picks $ID = \text{argmax}_z(\Pr(y(ID)|x))$. E.g., for predicting a biased coin flip with 60% chance of heads and $L = (\text{mistake? } 1 : 0)$, $R_{oB} = 0.4$. This decision is optimal, so $R_{oB} = E_z[\min_f L_f(z)]$. Because P is usually unknown, oB is useful only theoretically:

- $\forall f R_{oB} \leq R_f$
- In artificial cases where it's known it provides insight into behavior of other A

Let $\min_f L_f(z) = 0$. Then for nonoverlapping problems where $\forall x$ have a single corresponding ID , $P|x$ puts all probability on a single point or a discrete interval, and $R_{oB} = 0$. An overlapping problem has a random component, so the chosen features don't give enough information to always make correct decisions. Need more informative features, or don't have any.

A general strategy to find a good A is to reduce f search to optimization, i.e., optimize over some **search space** or **model class** $G \subseteq F$. Usually G is much smaller than F because F is uncountable but can only consider $G = \{f \text{ that don't take too much memory to represent}\}$. Limiting precision of numbers doesn't cause problems because f different only at precision limits are statistically indistinguishable. $A = G + \text{a strategy for picking } f \in G \text{ by minimizing } R_f \text{ based on } S$.

$R_f \leq$ the sum of:

- **Feature informativeness error** $R_{oB} - 0$; due to not having informative features
- **Approximation error** $\min_{h \in G} R_h - R_{oB}$; due to not using G with best h
- **Estimation error** $R_g - \min_{h \in G} R_h$, for $g = \text{argmin}_{h \in G} \text{SearchObjective}(h, n)$; due to not knowing Z
- **Optimization error** $R_f - R_g$, for $f = \text{approx argmin}_{h \in G} \text{SearchObjective}(h, n)$; due to optimizing approximately (exact optimization is usually hard)

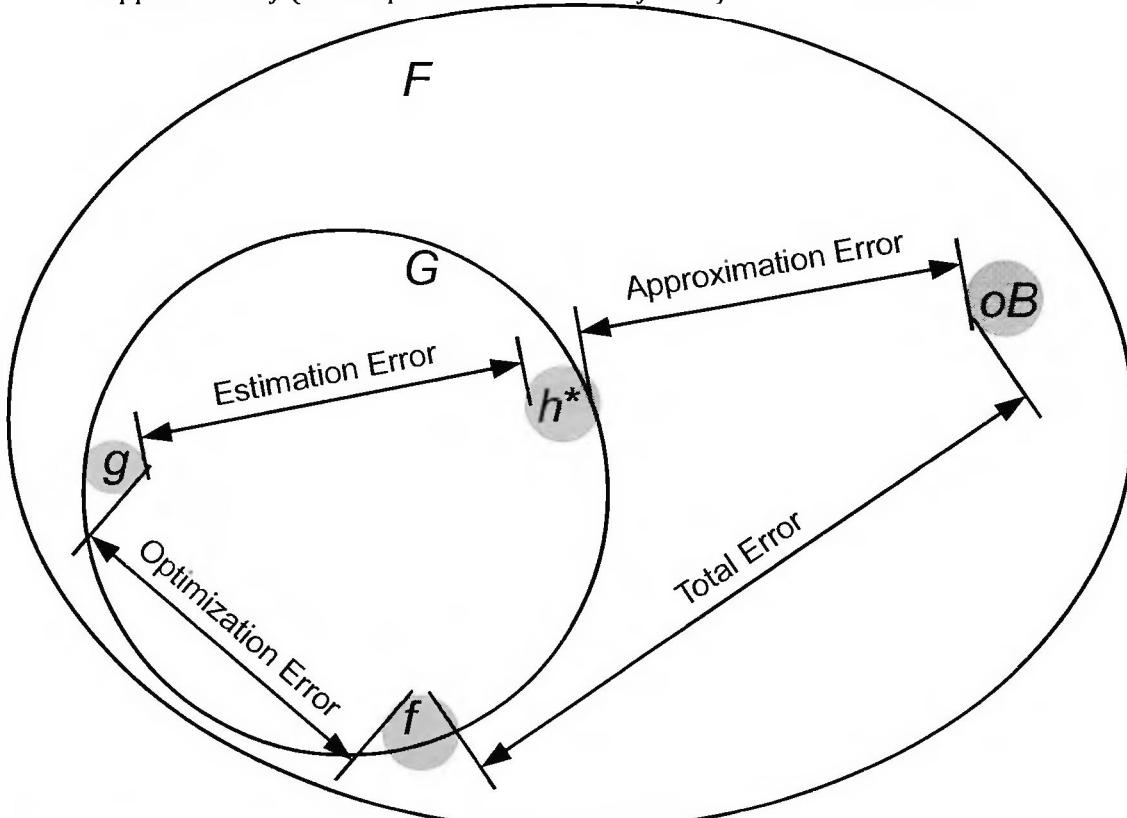


Figure 25.4: Errors generated by unavoidable restrictions

Assuming perfect features and optimization error = 0, approximation and estimation errors are what

matters. Might have a bound on the latter in terms of n , i.e., $\forall f \in G |R_f - R_{f,n}| \leq B(n, f, p)$ with probability $\geq 1 - p$ such that $\forall \epsilon > 0$ and $p > 0 \exists n(\epsilon, f, p)$ such that $B \leq \epsilon$; B is independent of P . If B is the same $\forall f, G$ has **uniform convergence**, and $B(n, f, p) = B(n, C(G), p)$, where C measures G 's **capacity to overfit**. Finite sample bounds hold simultaneously over $\forall f \in G$, in particular, over any f returned by A . Restricting capacity prevents overfitting by excluding f capable of accurately modeling noise in S , as in the credit card example.

A heuristic C is number of parameters to be estimated. It doesn't lead to general bounds but is usually close to a C that does. E.g., the credit card example doesn't work if $S = \text{polynomials of degree } < n$. Hoeffding gives uniform convergence for singleton $S = \{f\}$ for tasks with bounded L .

Vapnik's principle suggests solving a simpler problem directly instead of first solving a more difficult intermediate one. The idea is to minimize estimation error.

25.7 Complexity Control

Using $R_{f,n}$ as the search objective is **empirical risk minimization (ERM)**. If G has uniform convergence, B is the same $\forall f \in G$, and the found one will have a finite-sample bound. ERM is only useful for a few special cases such as linear regression due to needing small C or very large n to come up with small B . Uniform convergence defines sufficient **sample complexity** of S , i.e., $\forall \epsilon > 0$ and $p > 0$ can solve for n such that the estimation error $< \epsilon$. To allow efficient learning, need $n = O(\text{polynomial})$. But usually don't know G , so the use of sample complexity is essentially restricted to linear regression.

Can construct G with a finite-sample bound. Consider $G = \cup H_i$ and the corresponding $w_i \in (0, 1)$, picked before seeing the data so that $\sum w_i \leq 1$, and $\forall H_i$ have uniform convergence with bound B . Due to multiple testing, $R_f \leq B$ doesn't hold, but by Bonferroni correction $\forall f \in S |R_f - R_{f,n}| \leq B(n, C(H_i(h)), w_i(f)p)$ with probability $\geq 1 - p$ (Shalev-Shwartz & Ben-David 2014).

For G with a finite-sample bound, using search objective $= R_{f,n} + B(n, f, p)$ is **structural risk minimization (SRM)**. Usually it starts by looking at simpler models first and stops when for the best found f and all h not yet considered $R_{f,n} + B(n, f, p) \leq B(n, h, p)$. Let $g = \min_{h \in S} R_h$; because the search picked f and not g , $R_{f,n} + B(n, f, p) \leq R_{g,n} + B(n, g, p)$, and so $R_{f,n} - R_{g,n} \leq B(n, g, p) - B(n, f, p)$. Then the estimation error $= R_f - R_g \leq (R_{f,n} + B(n, f, p)) - (R_{g,n} + B(n, g, p)) = R_{f,n} - R_{g,n} + B(n, f, p) + B(n, g, p) \leq 2B(n, g, p)$, which is a finite-sample bound. \square

From the proof see that need a two-sided bound because if only have an upper bound, $R_{h,n}$ can be bounded away from R_h and give no information about g . In SRM, the distinction between finding model structure and optimizing its parameters is lost. A conclusion is that even if the true model is known to be complex, due to not having enough data to estimate it, may need to settle for a simpler one.

w_i represent prior information but aren't significant—e.g., for H_i in increasing $C(H_i(h))$ order can use $w_i = \frac{6}{\pi(i+1)^2}$, which decline slowly and sum to 1 in the limit. Can use whatever p is wanted for the final answer, such as 0.05 or 0.01, in all cases, and stop searching after the bounds prevent a better f from existing in H_i with of higher capacity. Must pick p, S , and w_i before seeing the data. But usually pick at least some of the above after seeing data, in which case the result is only a heuristic.

Assume H_i consists of a single hypothesis h and $L \in [0, M]$. Plugging $w_i p$ into two-sided Hoeffding, $|R_f - R_{f,n}| \leq M \sqrt{\frac{\ln(1/w_i) + \ln(2/p)}{2n}}$ with probability $\geq 1 - p$. So can do SRM over any countable hypothesis set after assigning weights. In particular, let every h be represented in binary using some prefix-free universal code such as gamma (see the "Compression Algorithms" chapter) using size $|h|$. Then for $w_i = 2^{-|h|}$, by Kraft inequality $\sum w_i \leq 1$. This leads to **Occam risk minimization (ORM)** of finding $f = \operatorname{argmin}_{h \in G} R_{h,n} + M \sqrt{\frac{|h| \ln(2) + \ln(2/p)}{2n}}$. For encoding a real number, use ad hoc "good enough" or a systematic $O(1/\sqrt{n})$ precision, based on the typical variance of many estimators as predicted by the Cramer-Rao bound (see the "Computational Statistics" chapter). The latter is more reasonable because even in the best case of taking an average have an estimate accurate to about that precision. The more economical the code, the better the bound.

A minor issue for ORM is that scaling y to change M doesn't affect the code for the model by much. Needing to define G before seeing the data isn't a problem because even defining a data structure able to represent f defines a prefix-free code. ORM is very general—it applies to any task with bounded L , and it's usually easy to define and search a suitable G . But using problem-specific information can give much better bounds for a complex enough h —ORM (usually very loose) lower bounds don't prevent this. So in practice ORM's value

is to show design flaws that prevent generalization. E.g., for best generalization don't specify unnecessary precision for parameters.

In philosophy, **Occam razor** advises to not make things more complex than needed—i.e., a simple explanation is more likely to be right than a complicated one, and ORM follows this. Benefits:

- Simplicity is useful in itself—e.g., a human can understand and inspect a simple model
- \exists many more complex models than simple ones, so finding a complex one that fits the data by chance is more likely. I.e., picking a simple G limits overfitting by reducing the “number” of f that can be tried. Alternatively, given data and two models that fit it well, purely by chance much more needs to go right in the complex model in term of parameter choice than in the simple one to get a good fit.

25.8 Approximation Error

G can't always compute the oB partition, which for a particular problem may be **arbitrarily complex**, as defined by its Kolmogorov complexity, i.e., the uncomputable smallest number of bits that can represent it (see the “Compression Algorithms” chapter), or a heuristic measure such as size in some simple encoding. So a partition may need arbitrarily many examples to specify. Some complexities:

- Some $z_i \in S$ can appear to be **outliers** when it's hard to distinguish **hint noise** (some y_i are wrong) from valid information. Complex models should be able to handle local patches of X with different behavior, especially if such patches are supported by enough examples. Noise is problematic in case of nearby examples with different hints. Usually, if have enough **example support**, a patch is assumed to be locally different, else it's an outlier. Noise is more problematic in smaller local patches than in larger ones because the latter compensate for it more.

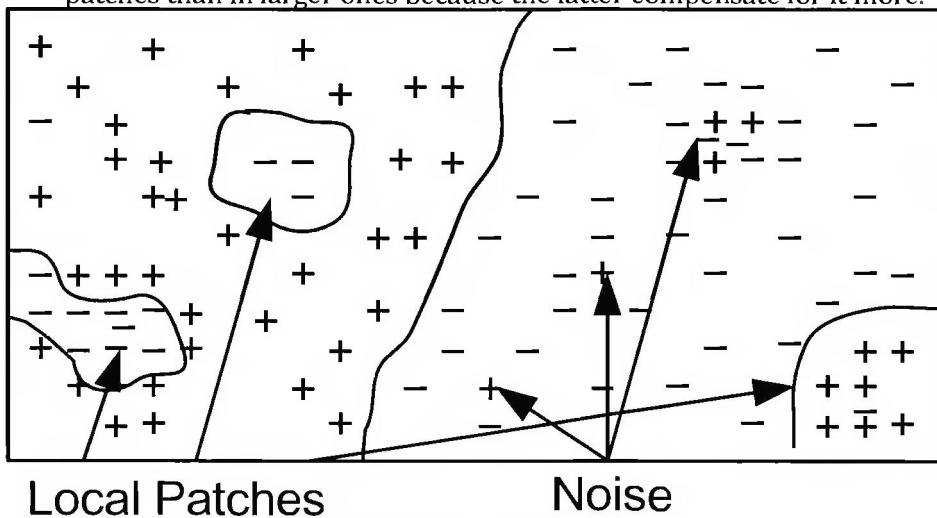


Figure 25.5: Various types of examples

- **Curse of dimensionality**—for vector-space data, learning gets harder as D increases. If x is a vector of binary variables, all 2^D possibilities are equally likely, so even getting enough data for estimation is a problem because a particular x says nothing about the others, e.g., for the multidimensional xor problem. More generally, a function can consist of many local regions, each behaving differently, and their number usually increases with D . Perhaps the best explanation for why theoretical bounds don't cause problems is the **manifold hypothesis** that for practical data sets the Z part with non-negligible support is much smaller than Z . This helps control estimation error by focusing the data on regions that matter and ignoring unlikely ones, which is often enough for satisfactory performance.

So in general it's hopeless to try to discover the exact best partition, at least better than asymptotically. At best can approximate it, and the more expressive G is, the smaller is the approximation error. So usually pick the most expressive G where have satisfactory control of estimation error. Because any training data is contained in a hypersphere of finite radius, making predictions on anything outside has no example support. But encountering such x is exponentially unlikely in n .

\exists finite-sample bounds on approximation error, at least without some assumptions such as bounded range or that the partition has Lipschitz continuity or other exploitable regularity (i.e., similar x lead to similar y). E.g., \exists a shape that most closely resembles both 1 and 7, but which one is genuinely unclear, so in practice assume that such shape is unlikely appear. A human would say “don't know because this doesn't conclusively look like either one”. Also in games like chess grandmasters only have intuition about a tiny subset of all possible positions. Partition complexity is the main cause of approximation error. One way to

measure it for a problem is to see how complex is the partition logic of a particular sufficiently low-risk A .

A is **consistent** for a problem if $R_f(A, n) \rightarrow R_{\text{OB}}$ as $n \rightarrow \infty$, and **universally consistent** if it's consistent \forall problem within a task (Devroye et al. 1996). This is different from statistical consistency which only needs convergence to the limiting parameter of the selected model without any regard to its approximation error. Some points:

- Many A are universally consistent.
- For problems where simple partitions do poorly, a universally consistent A needs arbitrarily large n and memory to represent f .
- Strengthening the definition by requiring finite-sample bounds leads to an impossible request because some problems need arbitrarily complex partitions. In the general case **no free lunch** (NFL) theorems prove this (e.g., see the "Machine Learning—Classification" chapter).

The intuition behind other (more popularly known) NFL theorems for prediction tasks is that \exists problems where finite training data is harmful and not helpful, so that over all problems no A wins. The basic proof idea (Lattimore & Hutter 2013) is that if learning is deterministic and X and Y are bounded, can put a uniform distribution on possible P so that every P is equally likely. Then $\forall A$ and finite training, if the training data is excluded from risk calculation, $E_p R_f(A, \text{data}) = E_{x \notin \text{data}} E_p L(f(x), y)$. Because under a uniform distribution of P , given x , all y are equally likely, and $E_p L(f(x), y) = E_p L(f(x), \text{uniformly random } y)$. For classification, the latter = $1/k$, i.e., the expected risk of a random guess. \square

So:

- Over all problems, learning is impossible
- Because some A do better than others on some problems, they must do worse on other problems
- In practice assuming at least some implicit bias is essential for generalization

More general NFL theorems (Wolpert 2002) lead to the same conclusions. Classification is the easiest learning problem both in terms of hint information and the range of y . So NFL theorems hold for more general tasks as well (Steinwart & Christmann 2008). This logic should apply to other impossibility or computational hardness results. NFL doesn't contradict universal consistency, which allows arbitrarily much training data. The most problematic NFL assumption is that want to solve all possible problems. Some interpretations:

- It's pointless to compare different A assuming that some are better. Only particular assumptions distinguish them on various problems. Experimentally, different types of problems prefer different learners, so some partial NFL must be true in that the assumptions of a particular A are more appropriate for certain problems than for others. E.g., a linear function can do better than a piece-wise one on some problems and worse on others. But some A consistently do better on most tested problems, though it's conceivable that on other they will do poorly.
- Must supply domain knowledge about the particular problem to avoid the NFL. Domain knowledge helps substantially in many cases, e.g., for image and natural language processing. Because must select at least a model class from which to do model selection, how to do so is generic domain knowledge.
- Don't expect to solve problems where a training data is harmful, i.e., hint info for S isn't related to that of $z \notin S$. The NFL doesn't work if only want to solve some P , so can ignore it. I.e., must assume something, at least "nice" distributions only. E.g., most A have a bias by assuming that nearby-in-some-way z are related. The above interpretations can hold only partially.

A particular A needn't be consistent, even if its underlying knowledge representation allows consistency. If can't have consistency, high approximation power is the next best thing.

Finite-sample bounds on estimation error can be too large with small fixed n when G is expressive enough to have small approximation error. E.g., G = polynomials of degree 1000 has finitely many parameters and should have uniform convergence for many problems, but n must be very large for useful bounds. So SRM and ORM are preferred over ERM.

Choosing a knowledge representation data structure usually determines G and suggests a search strategy. E.g., using a hyperplane means G = sets of all hyperplanes and a natural search is optimization of parameters. **Because must choose knowledge representation, must make at least some assumptions.**

No induction principle can always automatically come up with a good A . Theoretical training set bounds only offer guidance, and not useful error estimates. Most A have been created ad hoc:

1. Pick a knowledge representation that doesn't prevent consistency and is economical for simplicity and estimation error reduction
2. Control estimation error in some way, either through simplicity, stability, lack of multiple testing, or a combination

3. Respect other constraints, e.g., might need human-friendliness or scalability, to get which may need to give up some risk

No top-performing A in their exact implemented form give rigorous finite-sample bounds, and known except-for-minor-violation bounds are loose. Given a good A , it's unclear if \exists a better one for the problem. E.g., ellipses must be complex to human intuition because it took centuries before Kepler discovered elliptical orbits. Every 10 years or so at least one very good A has been discovered. More often than not, in practice the choice of A is arbitrary and doesn't involve careful logic, despite what one is tempted to believe from successful applications.

25.9 Stability

Intuitively, a stable A produces similar f from different data sets (Mohri et al. 2018). Let training sets T_1 and T_2 with n iid examples differ in only one example. A is **uniformly b -stable** if given f from training on T_1 and h on T_2 , $\forall z |L_f(z) - L_h(z)| \leq b$.

Theorem (Mohri et al. 2018): If $L \leq M$ and A is b -stable, $\forall T$ consisting of n iid samples, if A trained on T produces f , then $R_f \leq R_{f,n} + b + (2nb + M)\sqrt{\frac{\ln(1/p)}{2n}}$ with probability $\geq 1 - p$.

So if $b = O(1/n)$, as is the case for several algorithms (some discussed in the later chapters), the estimation error is $O(1/\sqrt{n})$. This is a very strong result, given its generality. Unfortunately, currently know b only for a few useful A . Nevertheless, this underscores the importance of stability—in particular, unstable A can't estimate their parameters to a reasonable accuracy, and are expected to have a high estimation error.

25.10 Risk Estimation of a Learning Strategy

Can estimate risk of f using holdout. But in many cases want to estimate risk of A (Vanwinckelen & Blockeel 2014; Dietterich 1998) on:

1. P when use only a specific S —for certain data-based decisions during training such as tuning parameters
2. P —for comparing several A independently of S but on fixed n —because typically will train on some data set of size n and use f thereafter
3. Several P —for comparing many A in general—for deciding which A to use for a new data set.

Will address (3) later, and (1) is a special case of (2). For (2), it's important to understand what must be independent and the sources of variance:

- A may be randomized
- The training data must be iid
- The test data must be iid among itself, and independent from the training data

Ideally know P , and can m times train on a random sample of size n and test the resulting f on a randomly sampled example. This creates m iid unbiased performance estimates for statistical inference. To do this in practice, need a data set of size $m(n+1)$, which isn't efficient use of data, but can't do better without introducing bias. Here can run each training several times with random example order and average the test results. This reduces the variance due to A 's own randomization and dependence on example order still gives an iid sample of size m of batch averages.

For (1), need an additional independent data set of size m . Then m times train A on the original data in random order, and evaluate on a single example, getting m iid unbiased estimates. This is similar to holdout, except training is redone \forall test example. If A has no own randomization and doesn't depend on the example order, both will give the same result, with holdout being substantially more efficient and, in this case, the method of choice with a lot of data.

Cross-validation: partition the data into k equal subsets, k times train A on $k-1$ subsets, and test on the remaining one, returning the average risk as the performance estimate:

Fold	Data Use				
1	Train	Train	Train	Train	Test
2	Train	Train	Train	Test	Train
3	Train	Train	Test	Train	Train
4	Train	Test	Train	Train	Train
5	Test	Train	Train	Train	Train

Then train f on all data. This way each example is tested once (up to round-off when k isn't a factor of n , but this doesn't matter for a typical small k). Experimentally, in typical cases $k = 10$ gives the best estimate, but 5 or 20 is just as good (Kohavi 1995). The best value depends on the application (Arlot & Celisse 2010); here for efficiency by default $k = 5$.

```
template<typename LEARNER, typename Y, typename DATA, typename PARAM>
Vector<pair<Y, Y>> crossvalidateGeneral(PARAM const& p,
DATA const& data, int nFolds = 5)
{
    assert(nFolds > 1 && nFolds <= data.getSize());
    Vector<pair<Y, Y>> result;
    int testSize = data.getSize() / nFolds; // roundoff goes to training
    for(int i = 0; i < nFolds; ++i)
    {
        PermutatedData<DATA> trainData(data), testData(data);
        int testStart = i * testSize, testStop = (i + 1) * testSize;
        for(int j = 0; j < data.getSize(); ++j)
        {
            if(testStart <= j && j < testStop) testData.addIndex(j);
            else trainData.addIndex(j);
        }
        LEARNER l(trainData, p);
        result.appendVector(evaluateLearner<Y>(l, testData));
    }
    return result;
}
```

Understanding many aspects of cross-validation is an open problem due to lack of independence:

- The training sets overlap in 60% or 80% of the data, depending on k
- If any two folds contain very different data from A 's point of view, in all runs the training and the testing data can be very different

Other and related problems:

- k repetitions may not be enough to remove the variance due to randomness in A .
- The estimate is for A trained on $n(k - 1)/k$ examples and is usually pessimistic for A trained on n examples because more data helps A .
- \nexists an unbiased estimate of the variance of cross-validation's risk estimate (Bengio & Grandvalet 2004). Also, though not proven, it's likely that \nexists a useful upper bound on it.

Some ways to improve cross-validation:

- Stratify by y —simple for discrete Y . This gives better estimates where applicable (Kohavi 1995).
- Deterministically scramble the example order using a random generator with a fixed seed—generally makes different folds more similar, e.g., when the data is sorted by some property, and is probably sufficient when stratification isn't applicable.
- Repeat cross-validation many times, maybe 10 or 20, with random permutations of data, average the test scores of each instance, and use the batch averages as samples for heuristic variance calculation. The repetition will remove variance due to randomness of A and, more importantly, the particular selection of the train sets from the data.

```
template<typename PARAM, typename Y, typename DATA, typename LEARNER>
Vector<pair<Y, Y>> repeatedCVGeneral(PARAM const& p, DATA const& data,
```

```

int nFolds = 5, int nRepeats = 5)
{
    Vector<pair<Y, Y>> result;
    PermutedData<DATA> pData(data);
    for(int i = 0; i < data.getSize(); ++i) pData.addIndex(i);
    for(int i = 0; i < nRepeats; ++i)
    {
        GlobalRNG().randomPermutation(pData.permutation.getArray(),
            data.getSize());
        result.appendVector(crossValidateGeneral<PARAM, Y>(p, data, nFolds));
    }
    return result;
}

```

- Consider switching to holdout when n is large ($>10^6$ maybe). With large n , if A is reasonably stable, the results will be similar, and will save considerable time. Can gradually reduce k to 2 before switching.

For (1), cross-validation and its improved versions effectively give an almost-iid sample, which heuristically consider iid. For efficiency, usually use stratification or deterministic scrambling, even though repetition is likely to be more accurate. Also, averaging in the repetition may be inconvenient for computing some other metrics. So repeat only if need a more accurate estimate, or A is known to be randomized.

Can get a biased estimate even in nonobvious ways. E.g., if f is the best on a test set, its estimate on the same test set is too optimistic because part of its best performance is by chance and favored by that test set. Even for validation (discussed later in the chapter), overfitting can lead to selecting suboptimal parameters.

f trained using A with a given estimated risk may not have the same risk. If A has large variance, R_f will vary accordingly. An accurate estimate of $R_{f(A, n)}$ is all that can hope for. For online learning, need somewhat different algorithms.

For (3):

- By the NFL, comparisons may be flawed in that the overall best A needn't be the best \forall data set. Specific A do best on specific types of problems, and which does well where is useful domain knowledge. E.g., digit and iris recognition are very different. Using domain knowledge often gives much better results. E.g., for digit recognition, including rotations of images gives extra useful data.
- Pick a metric and calculate it \forall problem and A . The metric is usually $R_{f, n}$, a theoretical upper bound, or something else. If A is randomized, run it several times, and take the average.
- Because different problems can have different difficulty, to make them comparable, transform the metrics into curved grades or ranks (see the "Computational Statistics" chapter). When the metric is risk, curved grades are particularly suitable because $R \geq 0$. This assumes that the transformed scores are iid and unbiased.
- $\forall A$ calculate the average transformed metric, and use that for comparison.

Can do this for several metrics such as risk and resource use and both curved grades and the rank transformation. But for metrics such as runtime, the rank transformation is probably the only reasonable solution because the slowest A can be much slower than the fastest ones, and scaling makes little sense. Also D and n matter, but it's unclear how to take them into account by scaling, and runtimes from various domains aren't iid.

But small data sets count as much as large ones. Data set choice matters too because it's easy to find data sets where a particular A does well. To avoid biases, it's best to use different data sets, preferably from different domains and of different difficulty. Also, need enough data sets for statistical significance.

To compare independent samples for (1), (2), and (3), can use (see the "Computational Statistics" chapter):

- Hypothesis tests:
 - The sign test for two A .
 - Nemenyi test for more—it can compare all pairs, which includes the best A against the rest.
- Bootstrap—effective for many types of comparisons, particularly to find the percentage of good performance for A . It's also more flexible, e.g., by allowing comparison of multiple metrics simultaneously in the wanted way.

When evaluating f and A using well-known data sets such as those from the UCI, beware that many A have been designed and tuned on them in the first place. It's likely that many algorithm and parameter configurations were tried and the unsuccessful ones were discarded, leading to bias. A reasonable but imperfect solution is to use more data sets, particularly the newer ones. Still, it's hard to correctly update existing

studies with new data sets or A because some multiple testing usually occurs.

25.11 Making Choices and Model Selection

For A that have parameters, reserve part of the training data for **validation** to properly optimize them. A parameter is anything that restricts the considered set of predictors, such as a choice of strategy. In statistics, a **model** is a set of predictors—e.g., linear model is the set of all hyperplanes and is fit to the data to create a **predictor**. Don't confuse with the common use, where intuitively *model* = *predictor* (many sources do though, so these are effectively synonyms). A **model class** is a set of models such as the set of polynomials \leq degree 3. Can also think of submodels, and often the difference is blurred.

The idea is the same as for risk estimation—to have good generalization, parameter optimization needs to look at different data. The simplest effective strategy for few parameters is exponential range grid search (see the “Numerical Optimization” chapter) with 5-fold cross-validation risk estimate as score. Cross-validation is particularly suitable because errors in estimates of differences between A are usually much smaller than errors in performance estimates (Kohavi 1995). Model selection is easier than performance estimation, because can reduce the former to the latter. Can also use the n test results as iid samples and calculate their variance for use in rules such as selecting the simplest A with performance within one standard deviation of the best A , though the accuracy of such variance calculation is heuristic. For this rule and cross-validation, the standard error is calculated based on a sample of size k , i.e., assuming independent folds (Hastie et al. 2009). After picking parameters, retrain on all data.

Capacity-based finite-sample bounds aren't affected by parameter selection, but estimation error is usually due to multiple testing. So for parameters to which A isn't very sensitive, it's best to have reasonable defaults, which also saves runtime. I.e., when possible, use logic (i.e., prior knowledge) and not search over many possibilities to select structure and parameters. E.g., use grid search for parameter ranges that work for most domains. Typically, make such ranges a little wider than needed because missing a good value for some problem is worse than minor estimation error and inefficiency in most problems. Use custom ranges for particular domains if possible.

Limiting optimization tends to reduce estimation error further. Complexity control and multiple testing avoidance are the main tools for estimation error control. For supervised learning, have more direct methods, discussed in the later chapters.

To get an accurate risk estimate after parameter selection, need to evaluate the choice again on new data, used only when final predictor is computed. Nevertheless, though the risk estimate of a chosen parameter configuration on selection data is biased, the differences between the candidates should be much less so, so the selection itself should be valid.

Can generalize to **nested model selection**, which applies when selecting multiple parameter sets. E.g., can use cross-validation to select a model, but A can do its own cross-validation with the data it has for whatever internal parameters it needs. Nested selection generally overfits less than selection out of a multi-dimensional set but requires enough data.

Sometimes want to pass no parameters, so the below works with the standard API that expects a single-parameter (possibly a complex object) interface.

```
template<typename LEARNER, typename Y, typename X = NUMERIC_X>
struct NoParamsLearner
{
    LEARNER model;
    template<typename DATA> NoParamsLearner(DATA const& data,
        EMPTY const& p): model(data) {}
    Y predict(X const& x) const{return model.predict(x);}
    double evaluate(X const& x) const{return model.evaluate(x);}
};
```

For some learning tasks can't define a good L , so methods like cross-validation don't work. Usually have specialized parameters selection methods.

For a user a basic research rule is to not consider any A that depends on parameters that can't be automatically selected, unless domain knowledge is enough for selection, which is usually not the case.

25.12 Some General Model Selection Strategies

Many heuristic induction principles have been proposed. Some are consistent, i.e., capable of producing consistent A in general cases, but this is only a sign of good design. Most formalize Occam's razor in some way.

- Model selection based on hypothesis testing—same as validation-based selection, but instead of validating on separate data use pairwise hypothesis tests. I.e., assume the simplest model as the null and use something like the sign test to move on to more complex models. This is usually much faster than validation, but suffers from multiple testing due to making many inferences from the same data. 95% confidence has little relevance to good future performance though, so this tends to pick overly simple models and isn't used often.
- **Regularization**—“fix” SRM by using aB instead of B , for $a \in [0, 1]$ and determined by cross-validation. This way the data decides how much to penalize complexity. A further relaxation is using the search objective = $R_{h,n} + aC(h)$, for some heuristic C such as $|h|$ or the number of parameters, and $a \in [0, \infty)$. Many A have some form of regularization, whereas SRM/ORM are mostly conceptual. But good regularization will enforce asymptotic simplicity. The main goal of SRM is to show that a brute force search for a learner works. Regularization is the practical aspect. So give up the idea of worst-case estimation error control of SRM.
- **Minimum description length (MDL)** principle—reduce learning to compression, i.e., pick $f = \operatorname{argmin}_h (|h| + |(y|h)|)$ using some code for both (Grunwald 2007). The assumption is that being able to compress the data means having learned something about it, i.e., need less information to correct mistakes of h than to come up with the data. ! \exists a known finite-sample inequality to justify this, though how useful for a distribution is a code learned from samples surely depends on n . It seems as valid as wanting simplicity and is consistent in some cases (Grunwald 2007). Unlike ORM, MDL applies even if L is unbounded. **Crude MDL** is using an ad hoc code for $|h|$ and $|(y|h)|$. Can code $|h|$ as for ORM. For $|(y|h)|$, only code the differences between the predicted and the actual y , which is enough to reconstruct the latter. The math simplifies in some cases because constant-in-model-choice additive terms don't matter. As for ORM, can use universal codes. E.g., for real y encode errors and their signs. The signs take 1 bit each and drop out as constants. When assuming that the errors follow a distribution with PDF p and that errors are discretized to some tiny interval d , need about $-\lg(dp(\text{error}))$ bits to encode, so d drops as a constant additive term, and $-\lg(p)$ remains. E.g., for normal p with mean 0, after some further simplifications $|(y|h)|$ is just the L_2 error. For classification, coding is more complicated—with k classes, prediction is either right or makes one of the $k(k-1)$ mistakes. Using a uniform code for these and counting the number of possibilities gives a multinomial, so with m correct answers and e_i wrong ones, for mistake i need $\lg\left(\frac{n!}{m! \prod e_i!}\right)$ bits to correct mistakes. This is a little clumsy, so with bounded L ORM wins over MDL. Usually, counting the number of possibilities and using universal codes on the result is optimal for MDL. Though crude MDL is intuitively appealing, designing codes for general A is messy, and the complexity penalty seems too strong, so its practical use is essentially limited to suggesting penalty terms for regularization.
- **Greedy space partitioning**—mirror some multidimensional data structure, and split X recursively using a simple h for each split. Greed considers only a small part of f at every step, which reduces the number of considered h and so overfitting.

25.13 Bias, Variance, and Bagging

Approximation–estimation–optimization isn't the only useful decomposition. Intuitively, A can err due to:

- **Bias**—preference for certain functional relationships over others due to prior knowledge. Unlike approximation error, bias takes into account all decisions such as optimization error in using local search.
- **Variance**—producing different f from different random S of the same size due to not being able to estimate parameters effectively, mostly due to overfitting. Unlike estimation error, it takes part of optimization error and other randomness in decisions into account.

Intuitively, bias is like partial blindness, and variance like hallucination—for a clear view, want to minimize both. To measure them need to use L for the problem. Let p be a random predictor variable and its **optimal combination** $C(p) = \operatorname{argmin}_m E_p[L(p, m)]$. E.g., for L being L_2 regression loss, $C(r) =$ the mean, for L_1 loss the median, and for classification's binary loss the mode. Note that $\text{oB}(x) = C(y|x)$, i.e., oB can generate arbitrarily many samples from $y|x$ and combine them.

Because work with a data set of size n , can only sample S of size n , and the next best thing is the **main predictor** $M(x) = C(f_S(x))$, i.e., combining predictions of trained f on a randomly sampled S . Beyond the optimality of the optimal combination, the main predictor has no direct ideological meaning. But it's a good

anchor point, around which variance is small due to in some sense being what is expected to be learned. To compare, uniform stability doesn't use an anchor point. Then define (Domingos 2000):

- $\text{Bias}(x) = L(\text{oB}(x), M(x))$ —the main predictor performance relative to oB
- $\text{Variance}(x) = E_S[L(f_S(x), M(x))]$ —the cost of performance differences from the main predictor
- $\text{Noise}(x) = E_S[L(\text{oB}(x), y(x))]$ —can't avoid noise, i.e., $E_x[\text{noise}(x)] = R_{\text{oB}}$

Theorem (Domingos 2000): $\forall x$ and metric L , $L(x, y) \leq \text{noise}(x) + \text{bias}(x) + \text{variance}(x)$. For L_2 loss for regression, have equality. This justifies the definitions of bias and variance, which are generalizations of the classical statistical ones for regression. Want A with both of them small. Can estimate variance using **bagging**, which simulates the main predictor using bootstrap resampling:

1. ***T* times for some *T* such as 300**
2. **Create a resample of *S* of size *n* from the data set**
3. **Train *A* on it to get *f***
4. **To predict *x*, form the main predictor out of all the *f*, and return its answer**

To estimate the expected-over- x bias and variance, use **out-of-bag** samples, obtained by running the main predictor on S and, for a particular example, using only f in training which it wasn't included. For bias estimation, because don't know oB, use the loss of the main predictor as a combined bias + noise measure. Because a problem has fixed noise $\forall A$, this still allows detecting bias differences among various A .

Bagging is considered accurate but imperfect. It injects randomness by using bootstrap samples for training and out-of-bag samples for evaluation. Both are different from using true random samples.

A with a high enough variance is deemed **unstable**. Many A have been studied using bagging and similar methods, with some interesting conclusions (Domingos 2000; Valentini & Dietterich 2004). E.g., bias and variance vary for different parameter choices and other decisions. Usually have a trade-off in that must lose one to gain the other.

Due to excellent performance, bagging with $A = \text{decision tree}$ (see the "Machine Learning—Classification" chapter) is itself a good A for classification, though superseded by better random forest (same chapter). Due to bootstrap samples, bagging increases the bias of A a little and reduces the variance by much, so using it usually improves A with high variance. The ability to form the main predictor \forall task makes bagging and improvements very extendable. A is **unbiased** iff $\forall x \text{bias}(x) = 0$. An important conclusion is that decision tree tends to be low-bias and unstable.

An interesting property of such **randomization ensemble** is that combining more base learners doesn't overfit more, contrary to complexity-based reasoning. Though formally proven for a special case (Breiman 2001; the idea is that for classification \forall class the proportion of votes converges to a fixed value), this holds generally. Semiformal proof: A specific m leads to a specific unknown general risk, and nearby m should lead to similar values of risk. Consider the risk of the optimal combination $R(p) = \min_m E_p[L(p, m)]$ for an ensemble. Assume L is bounded. Then given T base learners, making any one of them give an arbitrary prediction will only change R by $\leq L/T$, regardless of whether the current $\min m$ changes. By McDiarmid inequality (Mohri et al. 2018), R converges to its expected value exponentially fast in T . This R value corresponds to a possibly infinite set of m . As more base learners are added, R stops changing, but the set shrinks. For discrete m , it converges to one or more values which equal R . More generally and for continuous m , it converges to a countable set of neighborhoods of equal R minima. After some T , the neighborhood count becomes fixed, and the size of each is small enough to not matter for which m is eventually selected out of each. So in both cases after a finite T , the set of materially distinct considered m stops changing, and adding arbitrarily more base learners, the effect of which is more than that of the first T ones, makes no difference. \square \exists some technical issues here—e.g., for some L the limits may not exist, but for bounded L they do and, due to taking resamples from the same data and only considering reasonable m , for reasonable L always have the limits.

So randomization ensembles don't have drift where adding more base learners overfits more. Intuitively, no multiple testing is done by combining, and stability is controlled indirectly to some degree.

Bias and variance depend on n and aren't asymptotic, though noise is. The variance part due to n will decrease as n increases. But bias and variance comparisons are still meaningful for particular data sets because reached conclusions are likely to hold for a variety of n , though maybe not for the asymptotic case. It's not a contradiction for A to be both biased and consistent because can have bias and variance $\rightarrow 0$. This is the case for k -NN (see the "Machine Learning—Classification" chapter; Domingos 2000). Many consistency proofs actually show that bias $\rightarrow 0$ by slowly enlarging G so that also variance $\rightarrow 0$.

For losses other than L_2 , variance can cancel out part of the bias. An alternative decomposition is into the effects of bias and variance (James 2003), which together with noise exactly account for all loss instead of just bounding it. But bagging doesn't work for estimating effects, and by trying to explain interaction of bias

and variance they lose simple intuitive meaning.

Structural bias and variance are an interesting ideological concept. Combining predictors combines and smooths out their partitions. But this is only relevant in as much as L can measure. E.g., even with full knowledge of the distribution on Z , can have lack of example support in certain areas, meaning that distinct partitions would be indistinguishable. Also certain partition differences lead to the same outcomes, e.g., for classification, so L can't make accurate distinctions even with needed example support. This suggests that in general, the partition of oB isn't unique. Also, different partitions may have different variance but the same variance effect.

Using ensembles might reduce (Dietterich 2000):

- Approximation error—combining base learners widens G by including functions $\notin G$
- Estimation error—combining poorly estimated base learners can produce well-estimated averages
- Optimization error—reduce the risk that a single unlucky heuristic optimization will produce a bad result

25.14 Design Patterns

Have certain recurring solutions to common problems, including:

- **Bet on sparsity**—a complex model with many parameters is unlikely to be effective, so set as many as possible to an ineffective value, usually 0, so that they have no influence, reducing complexity (Hastie et al. 2009).
- Optimize approximately—due to estimation error, there is usually very little difference between approximate and optimal solutions. **Early stopping** is stopping optimization after some logical milestone or when the monitored validation set error starts increasing; a potential problem is when the increase is premature.
- **Convexification**—if optimization for certain L is hard because it's not convex, find a **surrogate loss function** that bounds L and leads to a convex optimization problem. Because of the bound the risk of the solution may be very close to the risk of the original problem's optimal solution.
- **Get more data**—this is usually the simplest and most effective way to reduce risk. A mediocre model with a lot of data often outperforms great models with little data (Domingos 2012). Learning has been observed to fail particularly for x that are very different from all seen examples. Any learned f is practically incomplete, i.e., doesn't apply to $\forall x \in X$, in the same way that numerical interpolation is very different from extrapolation.
- Use domain knowledge—any information A doesn't have to learn can be very helpful. E.g., in image processing, neighboring pixels are related.
- Use SGD (see the “Numerical Optimization” chapter) with a regularization penalty—convenient for models that have subgradients because training is online, efficient, and its under-optimization tends to keep estimation error small.
- Trust only validated A —for many tasks far too many A have been and keep being proposed, and it's practically impossible to conclusively compare more than a reasonable number of the most promising ones. It's a form of estimation error that qualities of each A aren't known completely, and finding them out is costly in terms of researcher time. A prudent strategy is to consider only A experimentally shown to have top performance on a variety of problems by someone other than the creators, or those with other desirable qualities such as interpretability, ease of use and parameter setting, efficiency, and a good design that justifies performance. Though other A may get better R_f , at least for particular problems, the effort should be focused on data preparation. Nevertheless, for problems of high economic significance, want to do extensive research.

All known A have issues, but it's hard to fix them without breaking something else. The end goal is to deploy f as part of a decision making system. Commodity reasoning applies as usual, so want A with:

- A track record of getting low-risk f on common data sets, particularly the ones similar to the problem being solved
- Reasonable efficiency—ideally training takes at most a few days, and making a decision is instant
- The ability to work as a black box, with at most minor help from the user—any parameters should be tuned automatically, but it's reasonable to have users give domain-specific information such as approximate parameter ranges or customized features
- Simplicity—e.g., Netflix chose not to deploy the overly complex winner of its recommendation competition (<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>)
- Interpretability—a learned partition should be understandable by a human, and often prefer f with simple knowledge representation over slightly more accurate complicated f

25.15 Data Preparation

Feature selection is the most important step in the learning process. Uninformative or random features like record id (an easy mistake to make when the data is automatically read from some database) may influence f by appearing to have predictive power by chance. Though certain domains have natural features, for best results should do feature engineering based on domain knowledge as much as possible. E.g., for trying to predict a person's level of health, it's better to include the body-mass index instead of height and weight because, given the latter, the model won't necessarily learn that the former is useful. Also, for games like chess, humans decide the quality of a position using piece count, mobility, pawn structure, etc., and it's impractical to learn these from piece locations (or so it was believed until recently—see <https://chessprogramming.wikispaces.com/Deep+Learning>).

Falsely informative features are even more dangerous. E.g., a well-known case is when a military organization tried to train a tank recognizer and, to save costs, used photos taken over only two days for training—one with tanks and one without. It happened to be cloudy on one of the days, and the algorithm learned to predict based on sky color.

It's usually easy to apply one of several good A to any data in a supported format. The hard part is collecting data from various often poorly maintained sources such as free-format logs or databases with data that needs cleaning, i.e., removing common-sense-encoded invalid values such as age = 0, gender = "N/A", etc. Given such noisy sources and a large n , coming up with derived features may be practically impossible without at least domain knowledge for cleaning. A simple heuristic to detect bad data is to check frequently occurring values because those are likely to be the invalid ones.

Generally, x is **numeric** (continuous or **discrete**), **ordinal**, **categorical**, mixed, or arbitrary but with a distance or a kernel function (discussed later). The difference between ordinal and categorical is that the latter gives no order relation.

Can convert between categorical and numeric features. For categorical \rightarrow numeric, define a binary 0/1 variable \forall category value. This transformation gives the necessary gradient or linear separability for many algorithms.

For numeric \rightarrow categorical, define subranges called **bins**, and for a given feature value return the corresponding bin number. Usually assume that a feature has bounded range, and put $z \notin S$ and outside this range in the end-of-interval bins. A simple binning strategy is **equal width**, with maybe $\lg(n)$ bins of the same size:

```
typedef Vector<int> CATEGORICAL_X;
class DiscretizerEqualWidth
{
    ScalerMinMax s;
    int nBins;
    int discretize(double x, int i) const
    {
        x = s.scaleI(x, i);
        if(x < 0) return 0;
        if(x >= 1) return nBins - 1;
        return nBins * x;
    }
public:
    template<typename DATA> DiscretizerEqualWidth(DATA const& data,
        int theNBins = -1): s(data), nBins(theNBins)
    {
        assert(data.getSize() > 1);
        if(nBins == -1) nBins = lgCeiling(data.getSize());
    }
    CATEGORICAL_X operator()(NUMERIC_X const& x) const
    {
        CATEGORICAL_X result;
        for(int i = 0; i < x.getSize(); ++i)
            result.append(discretize(x[i], i));
        return result;
    }
};
```

For online learning, can use constant 5-10 bins. But binning loses information and is best avoided by picking A that doesn't need discretization. To minimize estimation error, each bin must be supported by

enough examples, but even if that is the case, still have approximation error by putting dissimilar examples in the same bin.

If have a mix of numeric and categorical features, it's usually best to convert the latter into the former because this doesn't lose information. So commonly represent data as a numeric vector:

```
typedef Vector<double> NUMERIC_X;
```

25.16 Scaling

Many algorithms give equal priority to features when combining them, so those with large scale can have more influence. Only some A are scale-oblivious. One way to scale is mapping all feature values into the

same range, usually $[0, 1]$, using $\text{scaled}_X = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$; x_{\max} and x_{\min} are computed from the training data.

Scaling is done by a transformation in the data pipeline.

```
class ScalerMinMax
{
    NUMERIC_X minX, maxX;
public:
    ScalerMinMax(int D) : minX(D, numeric_limits<double>::infinity()),
        maxX(D, -numeric_limits<double>::infinity()) {}
    template<typename DATA> ScalerMinMax(DATA const& data)
    {
        assert(data.getSize() > 0);
        minX = maxX = data.getX(0);
        for(int i = 1; i < data.getSize(); ++i) addSample(data.getX(i));
    }
    void addSample(NUMERIC_X const& x)
    {
        assert(minX.getSize() == x.getSize());
        for(int j = 0; j < x.getSize(); ++j)
        {
            minX[j] = min(minX[j], x[j]);
            maxX[j] = max(maxX[j], x[j]);
        }
    }
    double scaleI(double xi, int i)const
    {
        double delta = maxX[i] - minX[i];
        return delta > 0 ? (xi - minX[i])/delta : 0;
    }
    NUMERIC_X scale(NUMERIC_X x)const
    {
        for(int i = 0; i < x.getSize(); ++i) x[i] = scaleI(x[i], i);
        return x;
    }
};
```

Only training data is normalized into $[0, 1]$, and other data may be mapped outside this range, but this shouldn't cause problems.

Another popular way to scale is **studentization**, i.e., \forall feature making the mean 0 and the variance 1. It appears to perform worse overall, probably because variance as a measure of scale is meaningful only when data \sim normal, but better for A that need mean 0 (usually for SGD to help convergence).

```
class ScalerMQ
{
    Vector<IncrementalStatistics> ic;
public:
    ScalerMQ(int D) : ic(D) {}
    template<typename DATA> ScalerMQ(DATA const& data) : ic(getD(data))
    {
        for(int i = 0; i < data.getSize(); ++i) addSample(data.getX(i));
    }
    void addSample(NUMERIC_X const& x)
    {
        for(int j = 0; j < x.getSize(); ++j) ic[j].addValue(x[j]);
    }
    double scaleI(double xi, int i)const
    {
```

```

        double q = ic[i].stdev();
        return q > 0 ? (xi - ic[i].getMean()) / q : 0;
    }
    NUMERIC_X scale(NUMERIC_X x) const
    {
        for(int i = 0; i < x.getSize(); ++i) x[i] = scaleI(x[i], i);
        return x;
    }
}

```

Can scale online by dynamically updating the relevant parameters. Though scales will adjust with data, and a first few $z_i \in S$ may be poorly scaled, this shouldn't cause problems.

Scaling isn't a free lunch. It usually improves performance but loses information when scales between features are dependent. E.g., neighboring pixels in images are related and having a light pixel next to a dark one signals contrast. Need $O(nD)$ time to scale S .

Both of these popular scaling methods have apparent problems that seem to be ignored:

- For range scaling, the boundary estimates aren't statistically stable and change depending on the data. It would be better to use maybe 5th and 95th percentiles as 0 and 1 respectively because these converge and have lower estimation variance. To compute these efficiently can use two heaps.
- For studentization the variance depends on the shape. E.g., the uniform and the binomial data both have the same range, but different variances.

The practical standard seems to be to use range scaling unless A needs studentization.

25.17 Handling Missing Values

Collected vector data can be incomplete if some x contain null values—e.g., survey respondents can fail to fill in some items. Despite not bothering oB, this is an issue for practical A . Can discard samples with missing values if n is large. But dropping missing value examples can introduce bias when values aren't **missing at random** and follow a pattern (Garcia et al. 2014).

Another option is discarding features with missing values. This is effective if mostly values of several specific features are missing—e.g., if respondents don't want to admit something.

With small n it makes sense to use the information from incomplete examples. Some solutions:

- Use the mean or the median of values of other examples—the simplest option, but can introduce bias.
- Use the value of x 's nearest neighbor, calculated using a distance that uses only non-missing-value features—probably the most practical replacement strategy.
- Do preliminary learning to guess missing values—more complex to setup, but probably the least biased replacement strategy.
- Use A that can handle missing features—doesn't lose information, but such A aren't necessarily good choices. For when dropping missing value examples removes too many, this seems to be the best solution.

That a value is missing can be useful information. E.g., might be able to make an accurate diagnosis based on only the names of tests requested by a doctor.

A different issue is predicting based on a sample with missing values. Even though f that work with missing values can give an answer, their performance estimates lose accuracy. May need to maintain replacement logic based on training data.

25.18 Feature Selection

Creating a useful feature needs human insight, but may be able to remove useless features automatically, and keep the best subset. This has several goals:

- Gain simplicity—and the corresponding benefits. May even want to simplify at expense of slightly increased risk
- Reduce the cost of data collection—no need to measure removed features, e.g., by giving patients extra medical tests
- Improve efficiency—fewer features = less computation

Noisy features are a problem because A will try to use the noise. E.g., if have many coin flips as features, some of them may seem useful and will be kept, resulting in a more complex, worse model.

A feature is (Kohavi & John 1997):

- **Strongly relevant** if removing it from any subset reduces R_{oB}

- **Weakly relevant** if not removing it from some but not all subsets reduces R_{oB}
- **Irrelevant** if removing it from any subset doesn't reduce R_{oB}

Optimal subsets consist of all strongly relevant, possibly some weakly relevant, and no irrelevant features. E.g., for the iris data, every variable allows reasonably accurate classification on its own, so for perfect classification might not need all four. Need weakly relevant features because any one of several can give the needed info. E.g., for learning the xor function, neither variable alone is helpful, but knowing both is.

Reducing risk by feature selection is usually hopeless. $\exists 2^D$ feature subsets, and each could be best. Attempting to discover which one needs reserving some validation data and increases the estimation error because the subset found on reduced data needn't be the best for all data, in turn because both useful and useless features may not appear so. Theoretically, to claim that a subset of k features is best, must consider all subsets of 2^k features (Devroye et al. 1996). Good A have built-in overfitting control, which benefits little from separate feature selection. A **realistic goal is to reduce the number of used features with minimal risk increase**. So usually use all features included by a domain expert, and try to select only if efficiency and data collection cost are a problem. A good success metric is the % of retained features.

Embedded methods use A with built-in feature selection. Some A are usually very efficient and don't use some features, attach importance to features, or explicitly try to not use some features. These are task-specific—e.g., decision/regression trees and methods based in L_1 regularization (discussed later in the corresponding chapters). They train and select features simultaneously, avoiding estimation error and search inefficiency for free. But such A aren't typically the best performing (otherwise feature selection would be solved). Chose embedded methods when n is too small to use a validation set or too large for other methods to be efficient.

Feature selection is implemented as a filter in the data pipeline.

```

class FeatureSelector
{
public:
    Vector<int> fMap;
public:
    FeatureSelector(Bitset<> const& selection)
    {
        for(int i = 0; i < selection.getSize(); ++i)
            if(selection[i]) fMap.append(i);
    }
    NUMERIC_X select(NUMERIC_X const& x) const
    {
        NUMERIC_X result;
        for(int i = 0; i < fMap.getSize(); ++i) result.append(x[fMap[i]]);
        return result;
    }
    double select(NUMERIC_X const& x, int feature) const
    {
        assert(feature >= 0 && feature < fMap.getSize());
        return x[fMap[feature]];
    }
};

template<typename DATA> struct FSData
{
    DATA const& data;
    FeatureSelector const & f;
    typedef typename DATA::X_TYPE X_TYPE;
    typedef typename DATA::Y_TYPE Y_TYPE;
    typedef X_TYPE X_RET;
    FSData(DATA const& theData, FeatureSelector const & theF): data(theData),
        f(theF) {}
    int getSize() const{return data.getSize();}
    X_RET getX(int i) const{return f.select(data.getX(i));}
    double getX(int i, int feature) const
    {return f.select(data.getX(i), feature);}
    Y_TYPE const& getY(int i) const{return data.getY(i);}
};

template<typename LEARNER> struct FeatureSubsetLearner
{

```

```

FeatureSelector f;
LEARNER l;
public:
    template<typename DATA> FeatureSubsetLearner(DATA const& data,
        Bitset<> const& selection): f(selection), l(FSData<DATA>(data, f)) {}
    int predict(NUMERIC_X const& x) const{return l.predict(f.select(x));}
};

```

For implementation, it's convenient to use one algorithm to find a number of good subsets and order them by the number of used features, and another to select the lowest-risk one. A typical validation search picks the best-performing subset, resolving ties in favor of smaller ones.

```

template<typename RISK_FUNCTOR> Bitset<> pickBestSubset(
    RISK_FUNCTOR const &r, Vector<Bitset<>> const& subsets)
    {return valMinFunc(subsets.getArray(), subsets.getSize(), r);}

```

The realistic goal is to reduce the number of used features and not improve performance. So, alternatively can pick the smallest subsets with risk \leq risk of the full subset. In case of some variance in the base A , the resulting multiple testing prefers much simpler subsets, offsetting estimation error and gaining some efficiency.

```

template<typename RISK_FUNCTOR> Bitset<> pickBestSubsetGreedy(
    RISK_FUNCTOR const &r, Vector<Bitset<>> const& subsets)
{
    int best = subsets.getSize() - 1;
    double fullRisk = r(subsets[best]);
    for(int i = 0; i < best; ++i) if(r(subsets[i]) <= fullRisk) best = i;
    return subsets[best];
}

```

The subset with all features is the benchmark, included by the subset-creation algorithm, and is often selected. E.g., for digit recognition, though some image areas are more important, each pixel carries similar info, so only a few may be unneeded, but not being able to remove any isn't a failure—probably all are useful. But if the base A has high variance, the risk of the full subset may not be estimated well, leading to selection of an overly simple subsets. Consider repeated cross-validation for only the full subset.

With many subsets, to reduce computation time and estimation error can subsample the ranked subsets using grid search. Can iterate this to zoom in on promising ranges (not implemented here).

```

Vector<Bitset<>> subSampleSubsets(Vector<Bitset<>> const& subsets,
    int limit)
{
    assert(subsets.getSize() > 0 && limit > 0);
    Vector<Bitset<>> result;
    int skip = ceiling(subsets.getSize(), limit);
    for(int i = subsets.getSize() - 1; i >= 0; i -= skip)
        result.append(subsets[i]);
    result.reverse();
    return result;
}

```

Wrapper methods use one or more A to search through some subsets, using a validation set to estimate their risk. A minor problem with wrappers is that they tend to pick features that are good only for the used base A (Guyon 2008). But because usually use the same A to relearn on all data and the found subset, this doesn't matter. Also, due to estimation error, can't select best subsets to satisfy all possible learners because each can use different features and be confused by others, etc.

Complete enumeration is feasible up to maybe $D \leq 12$, depending on n and the base A . Generate them in order of size:

```

struct SubsetLengthComparator
{
    bool operator()(Bitset<> const& lhs, Bitset<> const& rhs) const
        {return lhs.popCount() < rhs.popCount();}
    bool isEqual(Bitset<> const& lhs, Bitset<> const& rhs) const
        {return lhs.popCount() == rhs.popCount();}
};

Vector<Bitset<>> selectFeaturesAllSubsets(int D)
{
    assert(D <= 20); //computational safety
}

```

```

int n = pow(2, D) - 1;
Vector<Bitset<>> result(n, Bitset<>(D));
for(int i = 0; i < n; ++i)
{
    int rank = i + 1;
    for(int j = 0; rank > 0; ++j, rank /= 2)
        if(rank % 2) result[i].set(j);
}
quickSort(result.getArray(), 0, result.getSize() - 1,
    SubsetLengthComparator());
return result;
}

```

Forward search starts with no features and greedily adds the most useful one at a time until meeting the full subset's performance. It needs $O(D^2)$ evaluations so is useful for maybe $D \leq 40$.

```

template<typename RISK_FUNCTOR>
Bitset<> selectFeaturesForwardGreedy(RISK_FUNCTOR const &r, int D)
{
    Bitset<> resultI(D);
    resultI.setAll();
    double fullRisk = r(resultI);
    resultI.setAll(0);
    for(int i = 0; i < D; ++i)
    {
        double bestRisk;
        int bestJ = -1;
        for(int j = 0; j < D; ++j) if(!resultI[j])
        {
            if(i == D - 1)
            {
                bestJ = j;
                break;
            }
            resultI.set(j, true);
            double risk = r(resultI);
            resultI.set(j, false);
            if(bestJ == -1 || risk < bestRisk)
            {
                bestRisk = risk;
                bestJ = j;
            }
        }
        resultI.set(bestJ, true);
        if(r(resultI) <= fullRisk) return resultI;
    }
    resultI.setAll();
    return resultI;
}

```

Used with oB, it finds a maximal subset that can't be increased to improve performance. But it can't find xor-like subsets, where single features don't appear useful. Nevertheless, this usually isn't a problem for practical data, particularly because useful subsets are discovered eventually.

A much more scalable approach is to consider the prediction ability of each feature individually, and create subsets by adding one feature at a time from lowest to highest risk. With large D this could at least give an approximate ranking of subsets. Combine with subsampling for efficiency during evaluation at the next stage.

```

template<typename RISK_FUNCTOR> Vector<Bitset<>> selectFeatures1F(
    RISK_FUNCTOR const &r, int D)
{
    Vector<Bitset<>> selections;
    Vector<double> risks(D);
    for(int i = 0; i < D; ++i)
    {

```

```

        Bitset<> temp(D);
        temp.set(i);
        risks[i] = r(temp);
    }
    Vector<int> indices(D);
    for(int i = 0; i < D; ++i) indices[i] = i;
    IndexComparator<double> c(risks.getArray());
    quickSort(indices.getArray(), 0, D - 1, c);
    Bitset<> resultI(D);
    for(int i = 0; i < D; ++i)
    {
        resultI.set(indices[i]);
        selections.append(resultI);
    }
    return selections;
}

```

This ignores interactions between features and does poorly on xor-type problems. In general, can't tell whether a feature is useful based on a simple test. E.g., Zodiac sign may be useful in making predictions about a person if the person tries to live up to it. The xor problem leads to many counter-examples. Nevertheless, most problems are very different from xor, and efficiency makes single feature search useful. It's usually good at removing irrelevant features (Guyon 2008).

For wrappers, the overall strategy is to use:

- Complete enumeration if $D \leq 12$
- Greedy forward search if $D \leq 40$
- Single-feature search with subsampling otherwise

A rough justification for 12 and 40 is runtime control. Assuming the base A runs in $O(D)$ time with respect to D , complete enumeration and forward search respectively take $O(D2^D)$ and $O(D^3)$ time, which are close for the selected numbers.

```

template<typename RISK_FUNCTOR> Bitset<> selectFeaturesSmart(
    RISK_FUNCTOR const& r, int D, int subsampleLimit = 20)
{
    if(D <= 12) return pickBestSubset(r, selectFeaturesAllSubsets(D));
    else if(D <= 40) return selectFeaturesForwardGreedy(r, D);
    else return pickBestSubsetGreedy(r, subSampleSubsets(
        selectFeatures1F(r, D), subsampleLimit));
}

```

25.19 Kernels

Kernels allow efficiently adding features that are combinations of others. This is done by a feature-mapping function F . E.g., $F = \text{identity}$ corresponds to no mapping, and a simple F can include all pairs of feature products.

Such mapping can drastically increase D and make learning computationally infeasible, so is usually done only when the **kernel trick** applies, i.e., when A only uses dot products of something with x . E.g., for linear regression wx for some weight vector w becomes $F(w)F(x) = K(w, x)$, where K is a **kernel** function specific to F . K is computable directly, usually in $O(D)$ time, without mapping to the enhanced space first. This allows the dimension of the enhanced space to be very high, even ∞ .

By definition, any F corresponding to a dot product in the enhanced space is a valid K , but not every function K is valid kernel. Theorem: (Aggarwal 2014), K is valid iff the all-example $n \times n$ matrix M such that $M[i][j] = K_{ij}$ is **symmetric and positive definite (SPSD)**; i.e., $M[i][j] = M[j][i]$, and $\forall u \in \mathbb{R}^n uMu \geq 0$. Proof: M is SPSD $\rightarrow \exists$ matrix B such that $M = B^T B \rightarrow F(x_i) =$ the i^{th} column of B is a feature map $\rightarrow K$ is valid. Also $K_{ij} = F(x_i)F(x_j) \rightarrow K_{ij} = F(x_j)F(x_i)$, and $uMu = \sum u_i u_j F(x_i)F(x_j) = \|\sum u_i F(x_i)\|^2 \geq 0 \rightarrow M$ is SPSD. \square

So every valid K generates a numeric feature vector of some dimension $\forall x$, x needn't be a vector. Because K is an inner product of such vectors, it's a Hilbert space functional—e.g., it's bilinear. Intuitively, K measures similarity. In particular, the L_2 distance in the enhanced space is well-defined because using bilinearity, $\text{distance}^2 = \|F(x_i) - F(x_j)\|^2 = K_{ii} - 2K_{ij} + K_{jj}$.

M represents all available information about the training data. Conversion to M loses some information —e.g., because a dot product measures the cosine of an example to the origin, so any rotation information is lost. The choice of K represents prior domain knowledge about the data, and none work for all types of

data. For numeric vectors, useful K include **linear** (plain dot product) and **Gaussian** (also called **radial basis**), defined by $K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|}{\sigma}\right)$, where σ is the width parameter, similar to the standard deviation. The latter can represent any continuous partition boundary and is usually the method of choice due to good experimental performance for classification with SVM (discussed in the “Machine Learning—Classification” chapter; Bottou et al. 2007):

```
struct GaussianKernel
{
    double a;
    GaussianKernel(double theA): a(theA) {}
    double operator() (NUMERIC_X const& x, NUMERIC_X const& y) const
    {// beware - if x - y is too large, result = 0
        NUMERIC_X temp = x - y;
        return exp(-a * dotProduct(temp, temp));
    }
};
```

\exists useful K for other types of data such as strings and graphs. Using kernels allows learning nonvector data directly. SVM (see the “Machine Learning—Classification” chapter) uses kernels directly in a special way. In general, the **representer theorem** makes using kernels easier (Mohri et al. 2018): Let g be a nondecreasing function, H a Hilbert space corresponding to any K , and R any risk function. Then \exists reals a_i such that $f(x) = \sum a_i K(x_i, x)$ is a solution of $\min_{h \in H} g(\|h\|) + R(h(x_0), \dots, h(x_{n-1}))$.

25.20 Online Training

Many top-performing A need too much time or memory for large n . Want A that don't need all data before training and learn one example at a time.

SGD is often the optimization algorithm of choice. In most cases need to cross-validate its initial starting rate, otherwise it can quickly swing into ∞ . Gradients are scale-sensitive, and using the same initial rate such as 1 only works in special cases, mostly for classification, where gradients can't get too large. Also:

- SGD comes with some automatic stability by starting from parameters at 0 or some other low-complexity values.
- SGD naturally handles missing data—in the update equation, omit any missing values.
- Estimation error is much less of a problem because seeing an example once usually isn't enough to overfit.

To tune parameters use **racing**—works for parameters that can be tuned offline by grid search. \forall considered parameter combination, it creates an online A . After learning is almost complete, the remaining data is used for validation to pick the best f . In a fully online case where learning never ends, use **sequential error**—first evaluate on an example and then learn from it, keeping the running average. Prediction uses the best f so far, and can discard or discount earlier evaluations for estimating R_f . Such simple estimates are probably not accurate enough as performance estimates, but they are to pick the best or near-best parameters. Also can drop poorly performing racers after some time (not implemented here).

```
struct BinaryLoss
{
    double operator() (int predicted, int actual) const
    {return predicted != actual;}
};

template<typename LEARNER, typename PARAMS = EMPTY, typename Y = int,
typename LOSS = BinaryLoss, typename X = NUMERIC_X> class RaceLearner
{
    Vector<LEARNER> learners;
    Vector<double> losses;
    LOSS l;
};

public:
    RaceLearner(Vector<PARAMS> const& p): losses(p.getSize(), 0)
    {for(int i = 0; i < p.getSize(); ++i) learners.append(LEARNER(p[i]));}
    void learn(X const& x, Y y)
    {
        for(int i = 0; i < learners.getSize(); ++i)
        {
```

```

        losses[i] += l(learners[i].predict(x), y);
        learners[i].learn(x, y);
    }
}
Y predict(NUMERIC_X const& x) const
{
    return learners[argMin(losses.getArray(), losses.getSize())].
        predict(x);
}
;

```

E.g., this is applied to linear SVM.

25.21 Dealing with Nonvector Data

Converting some types of data into vector form is common in many domains but can be clumsy. E.g., for

- Text—try a **bag-of-words model**, where have a universe of D words, and represent a document by D counts of occurred words
- Audio—typically use Fourier transform frequencies as features
- Image—use pixels, possibly reducing the color range

For text, vector conversion loses information such as relationships between word positions. **Nonvector data** is any data that isn't easily converted into vector form.

For most domains, need to do some knowledge engineering, i.e., describe the domain knowledge formally by means of an **ontology**, i.e., an object model that describes what is happening. E.g., can formalize geography by defining regions, countries, cities, etc., along with the appropriate containment relationships. Once a domain is formalized, it becomes easier to define features using common patterns. E.g., for chess, a game state is usually represented by a combination of:

- The difference in material from assigning a value to each piece type
- Positional factors such as king safety, mobility, center control, pawn structure, etc.

Without domain knowledge, the only features would be piece locations.

Instead of vector conversion, can use distance and kernel functions. Doing so for various A is discussed in the later chapters. The former is usually simpler, but, given distance d , something like e^{-d} or $\frac{1}{d+1}$ is usually a good kernel.

25.22 Large-scale Learning

The primary challenge is efficiency—common good A usually take too long. Can:

- Use an online A —works, but these are usually aren't the most accurate.
- Sample the data to create a smaller but still representative S , and use the rest for validation and evaluation—usually a good strategy because most A are efficient at evaluation and can use all remaining data for that. But training with all data should produce more accurate models. Sampling works cleanly with ensembles (see the “Machine Learning—Classification” chapter). E.g., can change random forest to use $\leq m$ (for some reasonable m such as 10^4) examples for bags, or partition the data into independent chunks—sometimes this is called the **software alchemy method**.
- Keep the data on disk—works only with A that scale in n and access data cache-efficiently. Others are slow because of random access and/or superlinear runtime. Still, using a buffer can help. E.g., using permuted data directly causes I/Os—to avoid them, use external memory sorting to compute a buffer from the permutation. Random or deterministic permutation can also be done efficiently this way.

```

template <typename X, typename Y> struct DiskData
{
    EMVector<pair<X, Y>> data;
    DiskData(string const& filename): data(filename) {}
    template <typename DATA> DiskData(DATA const& theData,
        string const& filename): data(filename)
    {
        for (int i = 0; i < theData.getSize(); ++i)
            addZ(theData.getX(i), theData.getY(i));
    }
}

```

```

void addZ(X const& x, Y const& y){data.append(make_pair(x, y));}
typedef X X_TYPE;
typedef Y Y_TYPE;
typedef X const& X_RET;
X_RET getX(int i) const
{
    assert(i >= 0 && i < data.getSize());
    return data[i].first;
}
double getX(int i, int feature) const
{
    assert(i >= 0 && i < data.getSize() &&
           feature >= 0 && feature < data[i].first.getSize());
    return data[i].first[feature];
}
Y_TYPE const& getY(int i) const
{
    assert(i >= 0 && i < data.getSize());
    return data[i].second;
}
int getSize() const{return data.getSize();}
};

```

- Explicitly handle sparse feature vectors with many 0's. E.g., linear SVM SGD can handle sparse models such as bag of words using maps instead of vectors. This leads to a substantial efficiency gain.

"Big data" and related appealing terms are given to large-scale learning. \exists nothing magic about it because the best known approach is using the techniques above and clusters of parallel computers. A challenge is that the standard parallel programming data decomposition technique is usually impossible, because most A need to have access to all data, unless small subsamples suffice. Currently, big data research concerns mostly:

- Efficient storage systems such as Hadoop and Spark—these already allow working with much larger data sets
- Scalable algorithms that compute acceptable approximate answers
- Methods for large D —big data is still isn't enough to overcome the curse of dimensionality and the resulting estimation errors, despite occasional claims that it is

25.23 Conclusions

Every A has some problems. If have the computation time, usually use several A , and have cross-validation select the best one. Machine learning is only part of data mining—the goal is to do good science using data, and machine algorithms are just tools. An important aspect that's more art than method is finding a good data set, asking it the right questions, and determining if get well-supported conclusions. See Skiena (2017) for an entertaining general overview and Nolan & Lang (2015) for several detailed case studies. For random data, from incompressibility reasoning a probability of a clever pattern is tiny, so need data that shows stable patterns.

It's in human nature too see patterns in random data and be convinced that they are right, to be influenced by personal biases and ideologies, or to make externally motivated but inaccurate predictions. It's rational to make extreme predictions regularly because usually gain much when occasionally right and lose little otherwise ("experts" never admit luck). In professional sports and other big-money-prediction areas the deciding factor is often having more domain-specific information than the competition to allow slightly better results on average.

For many day-to-day tasks that can benefit from data analysis, can do much useful analysis by common sense and simple methods from classical statistics such as linear and logistic regression.

25.24 Implementation Notes

The idea of a data pipeline is original though obvious. This way algorithms look at data the same way and can read it on demand even from disk, which makes learning feasible even for very large data sets.

The feature selection algorithms were difficult to select because of many possible choices and little guidelines as to which are the best. So I went for the simplest algorithms and those whose good performance is easily explained.

Other data-preprocessing algorithms follow textbook descriptions.

25.25 Comments

The hint information concept is new but useful because A must take x as input and return some ID as output.

The name *ORM* is new. Somewhat misleadingly, it has also been called MDL (Shalev-Schwartz & Ben-David 2014), though it doesn't minimize the data description. The bound itself is **Occam razor bound** and is similar to **PAC-Bayesian bounds** with $\Pr(h) = 2^{-|h|}$ (Langford 2002).

Beware of terminology—in classical mathematics have uniform convergence for a sequence, uniform convergence in probability for an estimator, and here uniform convergence for G .

G is **agnostic PAC learnable** (Shalev-Schwartz & Ben-David 2014) if $\exists A$ such that $\forall \epsilon > 0$ and $p > 0 \ \exists n$ such that A returns a $\text{PAC}(R_{\text{ob}} + \epsilon, p)$. Though this defines an appealing complexity class, it's not useful. The original **PAC learnability** is less general, and assumes binary classification and that $R_{\text{ob}} = 0$. Class label 1 forms a **concept** $\subset X$, and the task is to identify it. This is less general than imposing a distribution on X and Y . A further assumption is that, because all computer information is binary with limited memory, G consists of a finite number of Boolean functions. This allows proving that certain Boolean learning tasks are learnable in polynomial time and certain others aren't. PAC is important only for historical reasons because its creation developed significant interest in statistical learning theory.

But the “concept” concept is intuitively useful for classification because this is how humans learn. A don't benefit from the idea because estimating a partition is a less general problem than estimating concepts \forall class. But concepts have an advantage in deciding that, e.g., a certain digit image isn't a digit at all and doesn't belong to any class. Technically, can handle this by creating a “don't know” class, but this is rarely done. To compare, regular classification requires a complete partition of X , though some parts of it won't correspond to x with nonnegligible probability or to x valid for the domain. Knowing the number of classes k is also useful domain knowledge.

\exists several similar definitions of stability, i.e., can bound $E[R_f - R_{f,n}]$, and use the Markov inequality to derive a tail bound from this (Shalev-Schwartz & Ben-David 2014).

Statistical rules for model selection such as AIC and variants (see Burham & Anderson 2002) are effective for some statistical models. Here a model with the minimum AIC = the number of parameters – the log-likelihood is considered best. Flaws:

- Only applies to models with a clearly defined number of parameters and log-likelihood
- With small n need corrections such as AIC_c
- Assumes that all considered models are reasonably effective
- The rules are asymptotic, often needing specific conditions for theoretical effectiveness

MDL doesn't have these problems and is more intuitive. Same for Bayesian methods because it's easier to reason in terms of universal codes than specific distributions. Here have:

- Bayesian model selection— $\Pr(h|y) = \text{O}(\Pr(h)\Pr(y|h))$. $\Pr(h)$ is the **prior likelihood** that model h explains the data relatively well before seeing it and $\Pr(y|h)$ is the likelihood of seeing the data if it were reasonably explained by h , i.e., any errors are considered to be noise. E.g., for real y usually put a normal distribution on the residuals. Then $f = \text{argmax}_h \Pr(h|y)$ is a **maximum a posteriori (MAP)** model. Usually the only prior knowledge is preference for simpler models, but this is automatic because a proper prior distribution can't be uniform over an unbounded range, and so must make some assumptions, which tend to favor simplicity. I.e., without enough data, might as well assume that a parameter ≈ 0 . While selecting a prior out of the blue might seem wrong, it has a stabilizing effect by reducing variance. Interpreting the prior is somewhat tricky, because almost always the true function $\notin G$; the prior only specifies relative belief in quality of considered models —no paradoxes here. Of course, the prior shouldn't be dogmatic (e.g., a delta function)—the posterior distribution must allow any data with at least some tiny probability. So the prior = $\Pr(\text{the structure}) \Pr(\text{the parameters})$. The biggest problem with MAP is that need to decide how much to believe in the prior, i.e., pick initial variances of the parameters. Regularization handles this using cross-validation and so is preferred. Most forms of regularization are equivalent to some form of MAP, e.g., L_2 penalty on coefficients means Gaussian prior, and L_1 Laplace.
- Bayesian **posterior inference** without picking a model—similar to MAP, except $f = E[h] = \int_h \Pr(h|y)$. The idea is to avoid committing to a particular h , which by itself is usually highly unlikely, and do inference directly. This follows the **Epicurus principle** of keeping all good explanations. The integral is usually approximated by MCMC, which is particularly suitable because know the posterior up to a constant (see the “Random Number Generation” chapter). Regulariza-

tion is implicit in that all models have probability > 0 , but with a simple model nonnegligible probability is put on fewer data than with a more complex one. So the most likely model depends on data, enough data means that a not-too-complex and not-too-simple model is preferred, and the more data there is, the more complex model can be preferred. But complexity doesn't always correspond to what model is chosen by data (Murray & Ghahramani 2015).

Around 2000, much progress has been made in learning theory due to applications of **concentration inequalities** and **Rademacher complexity** (Mohri et al 2018), which give useful finite-sample bounds when used together.

Many things are known about cross-validation, and many aren't (Arlot & Celisse 2010). The lack of fold independence hasn't prevented improper use in statistical tests. E.g., for comparing two A , an occasionally recommended solution is to average the results of each fold, obtaining k supposedly iid normal estimates, and apply the paired t -test. A heuristic estimate of variance (Nadeau & Bengio 2003) can be reasonably accurate, but comes with no guarantees and underestimates variance in a simple problem (Kuncheva 2014). Hidden dependencies cause problems in many ways—e.g., 2-fold cross-validation may appear independent, but have similarity dependence because instances in one fold are too similar or too dissimilar to those in another, relative to a random sample. An interesting question for cross-validation is whether a deterministic permutation improves the stratified version. The logic is that despite stratification, the data may have some sort order.

Repeated holdout is partitioning data randomly into test and train sets of the same size many times. Repeated cross-validation with the same number of trainings has less variance.

For cross-validation, the limiting case $k = n - 1$, called **leave-one-out cross-validation (LOOCV)**, is appealing due to giving an estimate for training on almost all n examples and not needing repetition (except when A is randomized or depends on the example order), but usually ineffective:

- Need considerably more time than even repeated cross-validation.
- Due to using test examples from the same data set instead of independent ones, the held out instances are either very similar/dissimilar to the training set instances, resulting in high variance estimates for many A . For regular cross-validation, the test set is more likely to have balance between similar and dissimilar instances. Intuitively, smaller k is faster and gives smaller estimate variance, and larger reduces estimate bias. LOOCV has high variance with respect to small changes to S . Unless A is random or depends on example order, LOOCV is deterministic by producing the same result on the same data, but the choice of S still gives it randomness.

For small data sets and very stable A , LOOCV can be the method of choice. But regular cross-validation does almost as well and is usually used regardless.

A natural statistical alternative to cross-validation is using bootstrap to repeatedly train on n resamples and test on the whole data. But about 63.2% of the training data would be in the test set, leading to a too optimistic risk estimate. Testing only on data not picked for training removes this bias but overestimates risk because the training set has about 0.632 of the data. Intuitively, bootstrap isn't appropriate because its assumption that the distribution of the functional of the sample \approx the distribution of the resamples is questionable due to strong influence of repeated data.

Many bias-variance decompositions have been proposed and used to make various conclusions (James 2003). Despite technical differences, all of them are similar enough so that general conclusions made by one carry out to another, particularly about stability and low bias.

For parameter selection, a theoretical question is whether using a selection strategy is consistent (Devroye et al. 1996). In practice, this tends to not matter because consistency applies in the limit, and many biases are negligible with large enough n .

For crude MDL, using Cauchy PDF for errors will give logarithmic error, but more like compression because the latter is similar to universal gamma code. Also, for real y can discretize errors as $\lceil \text{error}/d \rceil$ for some small discretization interval d , and encode with gamma directly, but this is clumsy. **Refined MDL**, a supposedly improved version of MDL, particularly in using optimal codes (Grunwald 2007), is complicated, and deriving its codes is nontrivial even for simple parametric models such as linear regression (with the result being similar to BIC) and doesn't work for nonparametric ones.

A more expensive but interesting way to scale is to remember S and use it to transform values into ranks, which are truly scale-invariant.

Another main binning method is **equal frequency**, with bin ranges defined so that the number of data points in each bin is about the same. This way, every interval is supported by some examples. Want the number of bins such that each has enough examples for accurate estimation, so use \sqrt{n} , which presumably gives a good balance between approximation and estimation errors. This seems to slightly outperform other binning methods (Garcia et al. 2014). An interesting idea is to make each bin have maybe ≥ 30 items.

\exists many other solutions to filling in missing values (Garcia et al. 2014), but all of them are questionable due to introducing bias. Another preprocessing method that isn't worth it is instance selection, either for efficiency or noise removal, because \exists nonheuristic ways to identify bad instances, and good A are good at taking the right info from each instance.

For small D it's feasible to use human insight to improve learning. One technique is to plot all pairs of variables, including the hint, and see if \exists a visual pattern. Also, can plot each feature individually and see its distribution.

In some cases a feature definitely isn't useful:

- If a subset of features is perfectly linearly correlated, none of them can be strongly relevant, and can remove all except one. In practice, perfect correlation is unlikely, but can use maybe $|\text{correlation}| \geq 0.95$. Can compute the correlation matrix, single out sufficiently correlated entries, and greedily select variables correlated with most other variables to minimize the total number of them, though this is expensive computationally.
- If a feature has ≈ 0 variance, it's irrelevant, and can remove it.

Generally these aren't worth checking because computing the correlation matrix needs $O(D^2)$ time and space, and features rarely have ≈ 0 variance. Also have some estimation error—e.g., highly correlated features in S needn't be so in general. This is magnified by multiple testing because some features can be highly correlated by chance.

For feature selection, **backward search** starts with a subset containing all features and greedily removes the least useful feature at a time. Used with oB, it finds a minimal subset that can't be further reduced without increasing risk. But with many weakly relevant features in the initial stages, it can easily remove some very good ones due to their seemingly small marginal benefit. Nevertheless, on typical data sets this doesn't seem to be a problem, and "improvements" such as using single feature scores to resolve ties give worse results. Still, greedy forward search results in similar risk, is much faster, and finds smaller subsets. **Bidirectional search** combines results of both backward and forward searches but is slower than both and seems to result in similar risk.

Can try to improve single-feature search by taking feature interactions into account. Evaluate $A \forall$ pairs of features, and, starting from an empty subset B , greedily add feature $\underset{i}{\operatorname{argmin}} \left(R_i - \sum_{j \in B} \frac{R_j}{|B|} \right)$ to it. The idea is that this should select the next feature as the most useful and least redundant one. But need $O(D^2)$ time and space, and experimentally this doesn't pay off in terms of the number of selected features or risk.

A frequently mentioned class of feature selection algorithms is **filter methods**, which quickly heuristically evaluate the importance of a feature to eliminate or rank it. Those that look at one variable at a time usually aren't useful because single-feature search is more general, can use a good A , and has similar efficiency with many decent A .

Classical statistics suggests single-variable tests which consider one feature at a time. Assuming numeric vector x , can use for:

- Numeric y —rank correlation—detects monotonic relationships.
- Categorical y —ANOVA—checks if the differences between category-specific means of continuous variables are significant. Can't use a more robust Friedman test because usually don't have equal number of $z_i \forall$ category.

Can use the results to directly remove insignificant features or create a sequence of subsets by greedily ranking them by significance, as in single feature search. Despite a somewhat appealing nature of this:

- Neither a difference between means nor rank correlation imply that a feature is useful for the used A if they are significant and not if not
- The tests make assumptions that may not hold and pay no attention to multiple testing

It's tempting to generate a feature report for the user, containing maybe range, mean, and median \forall feature, along with some test results, but this isn't scalable, and wrappers do the job faster and better than the user, though occasionally a human can notice something useful.

A more advanced and perhaps the best-known filter is **mRMR** (Garcia et al. 2014). Similarly to the all-pairs wrapper search, it computes mutual information MI between all pairs of features and all features and y , and computes a collection of subsets by greedily adding feature $\underset{i}{\operatorname{argmin}} \left(MI(x[i], y) - \sum_{j \in S} \frac{MI(x[i], x[j])}{|B|} \right)$. It performs similarly too, with an additional problem that currently the best practical way to estimate MI (Kraskov et al. 2004) needs using a 2D k -NN query, which can be made efficient in time but still needs $O(n)$ extra space.

Relief (Garcia et al. 2014) is similar, but also needs a k -NN query, suffers from similar flaws as mRMR,

and is less general. With filters, an occasionally suggested alternative to subset selection is keeping some % of the best features, hoping they contain the best subset. This is dangerous because can easily select a subset that's too small due to not having feedback from A because filters don't know how the features will be used.

Due to lack of published experimental comparisons, many other feature selection algorithms have been proposed but aren't useful.

Feature extraction (Garcia et al. 2014) maps all features into a presumably more informative feature space and uses some or all features in that space. Usually scale first. The quality usually depends on how well can reconstruct the original features or how well properties such as relative distances are preserved. Potential benefits:

- Smaller D and increased efficiency
- Smaller estimation error due to approximate representation of data, which averages out noise

But it's unclear how to do this well. Popular techniques have flaws:

- **Principal component analysis (PCA)** retains only $k < D$ vectors responsible for at least 95% of the variance in X (perhaps k is an interesting measure of data complexity?). This needs computing a $D \times k$ transformation matrix in $O(nD^2)$ time and is reasonable for up to a medium D . But using variance directions effectively assumes a multivariate normal distribution of the data, so the result isn't effective for multipeak or other nonnormal distributions because PCA effectively loses information that a good A can use. Techniques such as **kernel PCA** (Hastie et al. 2009) and sparse PCA variants (Hastie et al. 2015) try to improve it, but the former has $O(n^3)$ runtime, and the latter doesn't help intelligent A .
- The **random projection method** computes a random projection matrix (Witten et al. 2016), which tends to preserve distances between x . But most A lose more from the information loss than gain from the reduction in D , so don't have good use cases.

Feature transformations can hide irrelevant variables. They also go against the idea of solving the wanted problem directly, but in this reasoning may want to use them for semi-supervised learning (see the "Machine Learning—Other Tasks" chapter). Need extensive experimental evidence to justify general use of feature extraction.

An interesting model for online learning allows **drift**, i.e., changes of the distribution on Z (Gama 2010). E.g., the chance of someone hitting a dart board in a correct place improves as they learn. This is complex to handle both theoretically and computationally. A simple practical strategy is to occasionally refit the model to protect against distribution changes, but this doesn't work in all cases.

For online evaluation, more complex strategies than prequential are possible (Gama 2010). A simple one is discounting earlier example error by some small factor to account for progress due to learning, but it's unclear how to pick a robust discounting factor and how exactly this is beneficial. For drift, a possible solution is to use a sliding window of some number of last examples.

25.26 Projects

- Would it make more sense to use C++ inheritance instead of templates for the data pipeline? Does it make a difference in convenience and performance?
- Try a robust version of mean-variance scaling, using perhaps median as location and **MADN** (replace means in variance calculation by medians) or **IQR** (75th percentile – 25th one) as scale (see the "Computational Statistics" chapter). Test with classification with SVM.
- Create a generic CVS file parser to avoid making a custom one for each test data set. It should work on most test sets from UCI in the "matrix" format with a specified delimiter character. Use appropriate robustness check, e.g., that the read numbers are finite.
- Research and implement Shapley feature selection.

25.27 References

- Anguita, D., Ghelardoni, L., Ghio, A., & Ridella, S. (2013). A survey of old and new results for the test error estimation of a classifier. *Journal of Artificial Intelligence and Soft Computing Research*, 3(4), 229–242.
- Arlot, S., & Celisse, A. (2010). A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4, 40–79.
- Bache, K. & Lichman, M. (2013). *UCI Machine Learning Repository* [<http://archive.ics.uci.edu/ml>]. University of California. Accessed 10/19/2014.
- Bengio, Y., & Grandvalet, Y. (2004). No unbiased estimator of the variance of k-fold cross-validation. *The*

- Journal of Machine Learning Research*, 5, 1089–1105.
- Bolón-Canedo, V. (2014). *Novel Feature Selection Methods for High Dimensional Data*. PhD Thesis.
- Blum, A., Kalai, A., & Langford, J. (1999). Beating the hold-out: Bounds for k-fold and progressive cross-validation. In *Proceedings of the Twelfth Annual Conference on Computational Learning Theory* (pp. 203–208). ACM.
- Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5–32.
- Devroye, L., Gyorfi, L., & Lugosi, G. (1996). *A Probabilistic Theory of Pattern Recognition*. Springer.
- Dietterich, T. G. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10(7), 1895–1923.
- Domingos, P. (1999). The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4), 409–425.
- Domingos, P. (2000). A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning*. Morgan Kaufmann (pp. 231–238).
- Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10), 78–87.
- Gama, J. (2010). *Knowledge Discovery from Data Streams*. CRC.
- García, S., Luengo, J., & Herrera, F. (2014). *Data Preprocessing in Data Mining*. Springer.
- Grünwald, P. D. (2007). *The Minimum Description Length Principle*. MIT Press.
- Guyon, I. (2008). Practical feature selection: from correlation to causality. *NATO Science for Peace and Security*, 19, 27–43.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
- James, G. M. (2003). Variance and bias for general loss functions. *Machine Learning*, 51(2), 115–135.
- Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI* (Vol. 14, No. 2, pp. 1137–1145).
- Kohavi, R., & John, G. H. (1997). Wrappers for feature subset selection. *Artificial intelligence*, 97(1), 273–324.
- Kraskov, A., Stögbauer, H., & Grassberger, P. (2004). Estimating mutual information. *Physical Review E*, 69(6), 066138.
- Lattimore, T., & Hutter, M. (2013). No free lunch versus Occam's razor in supervised learning. In *Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence* (pp. 223–235). Springer.
- Langford, J. (2002). *Quantitatively Tight Sample Complexity Bounds*. PhD thesis, Carnegie Mellon.
- Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning*. MIT Press.
- Murray, I., & Ghahramani, Z. (2005). A note on the evidence and Bayesian Occam's razor.
- Nadeau, C., & Bengio, Y. (2003). Inference for the generalization error. *Machine Learning*, 52(3), 239–281.
- Nolan, D., & Lang, D. T. (2015). *Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving*. CRC.
- Rissanen, J. (2008). Minimum description length. *Scholarpedia*, 3(8), 6727.
- Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- Skiena, S. S. (2017). *The Data Science Design Manual*. Springer.
- Steinwart, I., & Christmann, A. (2008). *Support Vector Machines*. Springer.
- Valentini, G., & Dietterich, T. G. (2004). Bias-variance analysis of support vector machines for the development of SVM-based ensemble methods. *The Journal of Machine Learning Research*, 5, 725–775.
- Vanwinckelen, G., & Blockeel, H. (2014). Look before you leap: some insights into learner evaluation with cross-validation. In *JMLR: Workshop and Conference Proceedings* (pp. 1–17).
- Witten, I. H., Frank, E., & Hall, M.A. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.
- Wolpert, D. H. (2002). The supervised learning no-free-lunch theorems. In *Soft Computing and Industry* (pp. 25–42). Springer.

26 Machine Learning—Classification

26.1 Introduction

Classification is the main task of machine learning, with most research and data. Need a basic understanding of statistics to understand many A . All the major ones are discussed in detail, often with original implementation choices. A major omission is deep learning.

For some A , with $k = 2$ it's more convenient to have $y \in \{-1, 1\}$. Also many algorithms and theoretical results are for binary classifiers, but such results might apply to $k > 2$ at least heuristically because \exists simple methods to reduce such problems to a collection of binary ones.

To find k from data (assume that all are represented):

```
template<typename DATA> int findNClasses(DATA const& data)
{
    int maxClass = -1;
    for(int i = 0; i < data.getSize(); ++i)
        maxClass = max(maxClass, data.getY(i));
    return maxClass + 1;
}
```

26.2 Stratification by Class Label

Some A are sensitive to example order; particularly, want to prevent sorting by y . Randomizing the data before partitioning and/or using stratified sampling usually corrects this. The latter usually suffices and gives repeatable partitioning by being deterministic.

```
template<typename DATA> pair<PermutatedData<DATA>, PermutatedData<DATA> >
createTrainingTestSetsStratified(DATA const& data,
                                 double relativeTestSize = 0.8)
{
    int n = data.getSize(), m = n * relativeTestSize;
    assert(m > 0 && m < n);
    pair<PermutatedData<DATA>, PermutatedData<DATA> > result(data, data);
    Vector<int> counts(findNClasses(data)), p(n); //need p for legacy only
    for(int i = 0; i < n; ++i){++counts[data.getY(i)]; p[i] = i;}
    for(int i = 0; i < counts.getSize(); ++i) counts[i] *= relativeTestSize;
    for(int i = 0; i < p.getSize(); ++i)
    {
        int label = data.getY(p[i]);
        if(counts[label])--counts[label]; result.first.addIndex(p[i]);}
        else
        {
            result.second.addIndex(p[i]);
            p[i--] = p.lastItem();
            p.removeLast();
        }
    }
    return result;
}
```

For cross-validation without stratification, need to randomize the data in case the examples are sorted by class because breaking them up in folds naively creates substantial class imbalance. With stratification the implementation is tricky. It computes class sizes and, keeping track of the last used example \forall class, \forall fold marks the examples to be used for testing. Then the test set is created from these and used, and the original data is restored. For splitting and restoration to work, the included example numbers are sorted.

```
template<typename LEARNER, typename DATA, typename PARAMS>
Vector<pair<int, int> > crossValidationStratified(PARAMS const& p,
                                                 DATA const& data, int nFolds = 5)
{
    assert(nFolds > 1 && nFolds <= data.getSize());
    int nClasses = findNClasses(data), testSize = 0;
    Vector<int> counts(nClasses, 0), starts(nClasses, 0);
```

```

PermutedData<DATA> pData(data);
for(int i = 0; i < data.getSize(); ++i)
{
    pData.addIndex(i);
    ++counts[data.getY(i)];
}
for(int i = 0; i < counts.getSize(); ++i)
    counts[i] /= nFolds; //roundoff goes to training
for(int i = 0; i < counts.getSize(); ++i) testSize += counts[i];
Vector<pair<int, int>> result;
for(int i = 0;; ++i)
{//create list of included test examples in increasing order
    Vector<int> includedCounts(nClasses, 0), includedIndices;
    for(int j = valMin(starts.getArray(), starts.getSize());
        includedIndices.getSize() < testSize; ++j)
    {
        int label = data.getY(j);
        if(starts[label] <= j && includedCounts[label] < counts[label])
        {
            ++includedCounts[label];
            includedIndices.append(j);
            starts[label] = j + 1;
        }
    }
    PermutedData<DATA> testData(data);
    for(int j = testSize - 1; j >= 0; --j)
    {
        testData.addIndex(includedIndices[j]);
        pData.permutation[includedIndices[j]] =
            pData.permutation.lastItem();
        pData.permutation.removeLast();
    }
    result.appendVector(evaluateLearner<int>(LEARNER(pData, p),
        testData));
    //put test data back into data in correct places
    if(i == nFolds - 1) break;
    for(int j = 0; j < testSize; ++j)
    {
        pData.addIndex(includedIndices[j]);
        pData.permutation[includedIndices[j]] =
            testData.permutation[testSize - 1 - j];
    }
}
return result;
}

```

26.3 Risk Estimation

The most common R_f is expected error = $E[\text{the \% of misclassified examples}]$. It has a binomial distribution with some unknown mean, so use Wilson score confidence interval for test set results (see the “Computational Statistics” chapter). **Empirical accuracy** = $1 - R_{f, n}$. For equally likely classes, the random f has expected $1/k$ accuracy.

Accuracy is sensitive to **class imbalance**, i.e., the data needn't come from a random sample that represents all classes well, which can easily happen due to human error or unequal costs of getting the data. E.g., if the digit data test set only had 90 0 and 10 1 digits, f that always returns 0 has 90% accuracy despite being useless. **Balanced error rate** (and correspondingly **balanced accuracy**) weighs each class equally, effectively assuming that the examples in each class are iid but not between classes. It's no longer binomial, but finite-sample inequalities apply because, by a few algebraic manipulations, equal class weights are equivalent to using a generalized version of the inequality on the average of the example results, with class

“counts” for class i ranging from 0 to $w_i = \frac{n}{n_i k}$. E.g., $0.75a + 0.25b = 0.5(1.5a + 0.5b)$. With these weights,

use range $\left[0, \sqrt{\sum \frac{w_i^2}{n}}\right]$, and apply the usual inequality. But this is only useful as a diagnostic evaluation metric. All learning algorithms assume and often strongly depend on having iid data. Any strong violations will result in learners that don't generalize well, and no amount of evaluation metric tuning will help.

Have limited knowledge about how much to trust each prediction. E.g., for the iris data, because *seposa* is linearly separable from others, predicting *seposa* is less likely to be wrong than *versicolor* or *virginica*. No *R* can account for this confidence. Think of it as conditional accuracy, i.e., accuracy on the predicted class. Models with very high confidences for some classes are useful—can trust confident enough decisions, and send the rest to humans for review. E.g., for credit card fraud detection, should be able to automatically flag most transactions as not suspicious, and use human labor only for investigating what is marked suspicious.

Some *A* can give reliable probability estimates for individual examples. Most don't because estimating probabilities is a more general problem, and the results tend to be less accurate than *y* prediction.

For *f* that only returns labels, the **confusion matrix** *M* gives complete information about the performance, containing the number of assigned examples \forall predicted and actual label pair. A predicted label is a row coordinate, and an actual one a column one. Need $O(k^2)$ space to represent and $O(n + k^2)$ time to compute *M*.

```
Matrix<int> evaluateConfusion(Vector<pair<int, int>> const& testResult,
    int nClasses = -1)
{
    if(nClasses == -1)
        //calculate nClasses if unknown
        int maxClass = 0;
        for(int i = 0; i < testResult.getSize(); ++i) maxClass =
            max(maxClass, max(testResult[i].first, testResult[i].second));
        nClasses = maxClass + 1;
    }
    Matrix<int> result(nClasses, nClasses);
    for(int i = 0; i < testResult.getSize(); ++i)
        ++result(testResult[i].first, testResult[i].second);
    return result;
}
```

From *M* can compute many useful metrics by summing over its rows and columns:

- $t = \sum M[r, c]$ = the total number of examples in the test set
- $a = \frac{\sum_{r \neq c} M[r, c]}{t}$ = accuracy
- $t_l = \sum M[r, l]$ = the number of examples with label *l*
- $a_l = \frac{\sum_{r \neq l} M[r, l]}{t_l}$ = accuracy on examples with label *l*, also called **recall**
- $p_l = \sum M[l, c]$ = the number of examples with predicted label *l*
- $c_l = \frac{\sum_{r \neq l} M[r, l]}{p_l}$ = confidence in prediction of label *l*, also called **precision**

These allow defining derived metrics, such as balanced accuracy = $\frac{\sum a_l}{k}$. E.g., consider a single-feature

binary data set with decision boundary 00100|111. Here, precision(0) < recall(0) but precision(1) > recall(1). For the averages, the 0/0 NaNs are skipped. Can have $c_l = \text{NaN}$ but not $t_l = \text{NaN}$ because a label should always be represented unless the data set is incomplete, though a classifier might not assign any examples to it.

Confidence intervals are computed only for accuracy and balanced accuracy. No multiplicity adjustment is done, assuming the user will only pick the needed one. No intervals are computed for class-specific metrics due to multiple testing.

```
struct ClassifierStats
{
    double acc, bac;
    pair<double, double> accConf, bacConf;
    Vector<double> accByClass, confByClass;
    int total;
    ClassifierStats(Matrix<int> const& confusion) : total(0)
```

```

// same row = same label, same column = same prediction
Vector<int> confTotal, accTotal;
int k = confusion.getRows(), nBac = 0, actualK = 0;
IncrementalStatistics accS, bassSW;
Vector<IncrementalStatistics> precS(k);
Vector<double> weights(k);
for(int r = 0; r < k; ++r)
{
    int totalR = 0;
    for(int c = 0; c < k; ++c)
    {
        totalR += confusion(r, c);
        weights[r] += confusion(r, c);
        total += confusion(r, c);
    }
    accTotal.append(totalR);
    actualK += (totalR > 0);
}
double M = 0;
for(int r = 0; r < k; ++r)
{
    weights[r] = total/weights[r]/actualK;
    IncrementalStatistics bacS;
    for(int c = 0; c < k; ++c)
    {
        int count = confusion(r, c);
        bool correct = r == c;
        while(count--)
        {
            accS.addValue(correct);
            bassSW.addValue(correct * weights[r]);
            M += weights[r] * weights[r];
            bacS.addValue(correct);
            precS[c].addValue(correct);
        }
    }
    accByClass.append(bacS.getMean());
}
M = sqrt(M/total);
for(int c = 0; c < k; ++c)
{
    int totalC = 0;
    for(int r = 0; r < k; ++r) totalC += confusion(r, c);
    confTotal.append(totalC);
    confByClass.append(precS[c].getMean());
}
acc = accS.getMean();
accConf = wilsonScoreInterval(acc, accS.n);
bac = bassSW.getMean();
bacConf = HoeffFunctor::conf(bac, bassSW.n);
bac *= M;
bacConf.first *= M;
bacConf.second *= M;
};

}

```

```

acc * total 1746
total 1798
Accuracy: 0.97107897664071186 95% interval: 0.96206872498450946 0.97799786769029617
Balanced Accuracy: 0.97111191719888246 95% interval: 0.95905336988373613 0.98060008111632324
accuracy by class:
0.9943820224719101
0.99450549450549453
0.98870056497175141
0.96174863387978138
1
0.96703296703296704
0.97790055248618779
0.93296089385474856
0.94857142857142862
0.9444444444444442
Confidence by class:
0.98882681564245811
0.96276595744680848
1
0.96703296703296704
0.98369565217391308
0.97777777777777775
0.98882681564245811
0.98235294117647054
0.93785310734463279
0.92391304347826086

```

Figure 26.1: Random forest (discussed later in this chapter) results for the digits data

Probably all of them are useful for describing f 's performance on a particular task. Balanced accuracy is good for the final evaluation for most cases because it:

- Forces f to treat all cases equally, which may be important even if the examples represent a true biased distribution.
- Is equal to overall accuracy with perfectly balanced classes.
- Is a special case of risk over an input distribution that expects all labels to occur equally.
- Not always \leq accuracy. When performance on minority classes is better, have average accuracy $>$ accuracy.

But balanced accuracy has higher variance than accuracy in case of strong class imbalance because minor-class examples have large impact.

Use accuracy for optimizing parameters and comparing A because it's built into many algorithms and allows influencing the input distribution by resampling or assuming S is representative. In particular, the implementation of cross-validation uses accuracy as a single metric to pick parameters. Intuitively, using accuracy moves margins (discussed later in this chapter) further from the majority and using balanced accuracy from the minority.

```

template<typename LEARNER, typename DATA, typename PARAMS> double
crossValidation(PARAMS const* p, DATA const* data, int nFolds = 5)
{
    return ClassifierStats(evaluateConfusion(
        crossValidationStratified<LEARNER>(p, data, nFolds))).acc;
}
template<typename LEARNER, typename PARAM, typename DATA>
struct SCVRiskFunctor
{
    DATA const* data;
    SCVRiskFunctor(DATA const* theData): data(theData) {}
    double operator()(PARAM const* p) const
        {return 1 - crossValidation<LEARNER>(p, data);}
};

```

One problem with accuracy and balanced accuracy is not distinguishing predictors that have mediocre confidence \forall label and high confidence on some labels and poor on others. Look at M for the final evaluation.

26.4 Reducing Multiclass to Binary

\exists two simple ways to extend binary learners for $k > 2$. **One vs all (OVA)** works for f that output probabilities.

ties. It trains k binary learners to output 1 iff $y = k$, and the result is the largest output label.

One vs one (OVO) trains $O(k^2)$ learners \forall combinations of binary classifiers and chooses the class with most votes. It's usually better:

- If the actual class wins all the time, and the rest are arbitrary, the result is always correct and fairly robust if the rest are random. E.g., for digits, 2 vs 3 wouldn't know what to do with a 7 and make a somewhat random choice.
- Asymptotic efficiency—though need more trainings, each is much faster due to solving a smaller problem; so faster overall as most A are superlinear.
- Decision boundaries between any two classes are simpler, and class imbalance is less of a problem. E.g., for the iris data, versicolor, which is between seosa and virginica, can't be separated by a single line with OVA, but can be with OVO. Theoretically, OVO leads to a smaller worst case approximation error than other methods (Daniely et al. 2012).

The implementation stratifies two-class data to improve learning for many online A such as SGD. In the data transformation pipeline create a relabel stage as needed.

```
template<typename DATA> struct RelabeledData
{
    DATA const& data;
    typedef typename DATA::X_TYPE X_TYPE;
    typedef typename DATA::Y_TYPE Y_TYPE;
    typedef typename DATA::X_RET X_RET;
    Vector<Y_TYPE> labels;
    RelabeledData(DATA const& theData) : data(theData) {}
    int getSize() const {return data.getSize();}
    void addLabel(Y_TYPE y) {labels.append(y);}
    void checkI(int i) const
    {
        assert(i >= 0 && i < data.getSize() &&
               labels.getSize() == data.getSize());
    }
    X_RET getX(int i) const
    {
        checkI(i);
        return data.getX(i);
    }
    double getX(int i, int feature) const
    {
        checkI(i);
        return data.getX(i, feature);
    }
    Y_TYPE const& getY(int i) const
    {
        checkI(i);
        return labels[i];
    }
};

template<typename LEARNER, typename PARAMS = EMPTY, typename X = NUMERIC_X>
class MulticlassLearner
{ //if params not passed, uses default value!
    mutable ChainingHashTable<int, LEARNER> binaryLearners;
    int nClasses;
public:
    Vector<LEARNER const*> getLearners() const
    {
        Vector<LEARNER const*> result;
        for(typename ChainingHashTable<int, LEARNER>::Iterator i =
            binaryLearners.begin(); i != binaryLearners.end(); ++i)
            result.append(&i->value);
        return result;
    }
    template<typename DATA> MulticlassLearner(DATA const* data,
                                                PARAMS const&p = PARAMS()): nClasses(findNClasses(data))
```

```

{
    Vector<Vector<int>> labelIndex(nClasses);
    for(int i = 0; i < data.getSize(); ++i)
        labelIndex[data.getY(i)].append(i);
    for(int j = 0; j < nClasses; ++j) if(labelIndex[j].getSize() > 0)
        for(int k = j + 1; k < nClasses; ++k)
            if(labelIndex[k].getSize() > 0)
            {
                PermutatedData<DATA> twoClassData(data);
                RelabeledData<PermutatedData<DATA>>
                    binaryData(twoClassData);
                for(int l = 0, m = 0; l < labelIndex[j].getSize() || m < labelIndex[k].getSize(); ++l, ++m)
                {
                    if(l < labelIndex[j].getSize())
                    {
                        twoClassData.addIndex(labelIndex[j][l]);
                        binaryData.addLabel(0);
                    }
                    if(m < labelIndex[k].getSize())
                    {
                        twoClassData.addIndex(labelIndex[k][m]);
                        binaryData.addLabel(1);
                    }
                }
                binaryLearners.insert(j * nClasses + k,
                    LEARNER(binaryData, p));
            }
    }
int predict(X const& x) const
{
    Vector<int> votes(nClasses, 0);
    for(int j = 0; j < nClasses; ++j)
        for(int k = j + 1; k < nClasses; ++k)
    {
        LEARNER* s = binaryLearners.find(j * nClasses + k);
        if(s) ++votes[s->predict(x) ? k : j];
    }
    return argMax(votes.getArray(), votes.getSize());
}
int classifyByProbs(X const& x) const
{ //for probability-output learners like neural network
    Vector<double> votes(nClasses, 0);
    for(int j = 0; j < nClasses; ++j)
        for(int k = j + 1; k < nClasses; ++k)
    {
        LEARNER* s = binaryLearners.find(j * nClasses + k);
        if(s)
        {
            double p = s->evaluate(x);
            votes[k] += p;
            votes[j] += 1 - p;
        }
    }
    return argMax(votes.getArray(), votes.getSize());
}
};

```

26.5 Complexity Control

A good measure of complexity C of G , consisting of binary classifiers for vector data, is **VC dimension** = maximum d such that $\exists d \ x$, arranged so that \forall assignment of binary labels to them, $\exists f \in G$ that can separate (shatter) them.

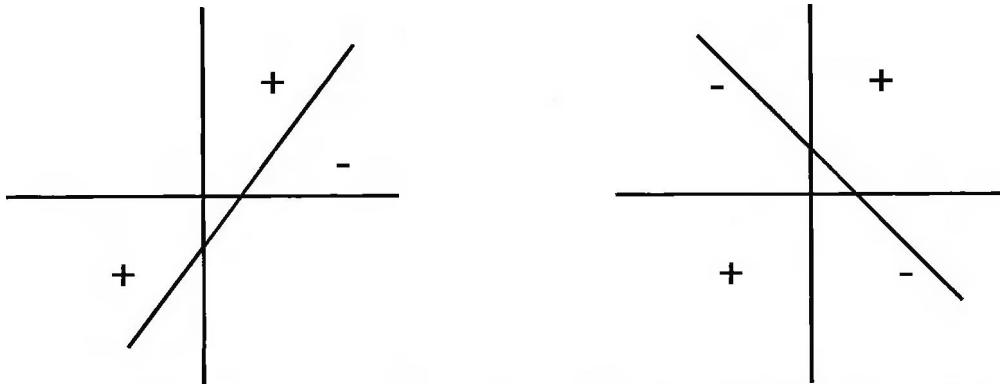


Figure 26.2: The xor problem—a line can separate 3 points in 2D, but not 4

So for 2D lines, $d = 3$. For D -dimensional hyperplanes, $d = D + 1$. E.g., can solve the xor problem by a split along the z-coordinate.

Can bound generalization error by a function of $R_{f,n}$, n , and d . Theorem (Mohri et al. 2018): Let G have VC dimension d . Then with probability $\geq 1 - p$, $R_f \leq R_{f,n} + \sqrt{\frac{2d \ln(en/d)}{n}} + \sqrt{\frac{-\ln(p)}{2n}}$. So if $n/d \rightarrow \infty$ as $n \rightarrow \infty$, $R_{f,n} \rightarrow R_f$. E.g., even hyperplanes can overfit for $D \approx n$. The intuition is that though can have uncountably many $f \in G$, each can classify n examples in finitely many ways. Occam bounds are usually worse than VC dimension ones, but not always—e.g., for $G = \{\text{sine functions}\}$, VC dimension = ∞ , but the description is quite small.

For SRM, can consider H_i with $i+1=d$; maybe with $w_i = \frac{6}{(nd)^2}$ to sum to 1. Then the search objective = $R_{h,n} + \sqrt{2d \frac{\ln(en/d)}{n}} + \sqrt{\frac{2\ln(nd) - \ln(6p)}{2n}}$; the 3rd term is tiny relative to the 2nd for reasonable p . If know d , this is easy to calculate but applies only to $k = 2$, and the bound is loose unless n is large enough to counter the pessimism of d . In particular, model selection using theoretical upper bounds based on VC dimension is less effective than cross-validation (Hastie et al. 2009).

Margin is distance of a patch of examples of a particular class to partition boundaries. E.g., in case of 2D data and a line that splits it, the margin = the smallest distance of any example to the line. Intuitively, the larger the margin, the more certain is the partition, so in some cases can bound risk as a function of the margin, which usually gives much better bounds than using d or ORM. A very complex f can have large margins and much smaller risk bounds than suggested by complexity bounds. Many successful algorithms implicitly or explicitly maximize margins in some way. Still, for known A , margins don't fully explain R_f . They are equivalent to more general bias-variance decomposition (Domingos 2000) but more intuitive in most cases.

Borderline examples are close to partition boundaries and involved in forming the margins. Label noise in these is more problematic than in interior examples due to affecting margins (Garcia et al. 2014). Can't detect noise without enough support from correct examples.

Class boundary complexity is the source of approximation error. It can have arbitrarily large Kolmogorov complexity. \exists some metrics of data complexity (Orriols-Puig et al. 2010), but they don't seem useful. E.g., the number of support vectors in a SVM or the size of an unpruned decision tree (both discussed later in this chapter) seem more informative and are simpler. The amount of linear SVM margin violation is a good measure of linear separability of classes.

Have an impossibility result, which however doesn't seem concerning. Theorem (Devroye et al. 1996): $\forall A$ that uses n examples for learning f and $\epsilon > 0$, \exists a problem with $R_{OB} = 0$ such that $R_f \geq \frac{1}{2} + \epsilon$. This **no free lunch (NFL)** theorem confirms that some problems need arbitrarily many examples (and doesn't contradict consistency of some algorithms). Also it suggests that no particular A is best in all cases because for one that is bad for some problem another may do well. Some A are better than others in most practical cases, which this NFL allows.

Another impossibility result is interesting conceptually but irrelevant: Theorem (Mohri et al. 2012; Anthony & Bartlett 1999): Let G have VC dimension $d > 1$. Then $\forall n$ such that $\epsilon = \sqrt{\frac{d}{320n}} \leq \frac{1}{64}$, $\forall A$ that returns $f \in G$, \exists a problem such that $\Pr(R_h - \min_{f \in G} R_f > \epsilon) \geq \frac{1}{64}$. This holds even for finite G and A that remember all examples because examples are sampled and \exists a problem that withholds enough examples

from A during training to meet the bound. This doesn't contradict the upper bound on risk—can make probability of a larger ϵ (which can be made arbitrarily small) arbitrarily small.

The general problem with lower bounds is that they apply only to the restricted model considered. E.g., radix sort still works in $O(n)$ time, despite the comparison model's $n \lg(n)$ lower bound. Likewise, this bound assumes that know d for the problem exactly, which usually isn't true. E.g., for $G = \{\text{sine functions}\}$ $d = \infty$ because it can shatter any number of points, but not if frequency and amplitude take $O(1)$ bits to represent. In general, considering numerical restrictions and X boundaries gives lower d . No bounds are preventing estimation error from being 0 for most problems, but the lower bound is conceptually interesting because always $d > 0$, even if data that invokes it is unlikely in practice.

26.6 Naive Bayes

Assume x consists of independent categorical features (equivalently discrete features with bounded range). By the independence, $\Pr(x|y) = \prod_j \Pr(\text{the value of feature } j|y)$, which is called **likelihood**. Estimate $\Pr(\text{value}|y)$ by (the number of examples with the value \in class/the number of examples \in class). These counts start from 1 to not divide by 0 and act as a prior. To avoid underflow, compute likelihood as **log likelihood**. This doesn't change comparison results because log is monotonically increasing.

1. \forall class and feature value initialize the count to 1
2. \forall example
3. \forall feature value increment the count associated with it and y
4. At prediction \forall class i
5. Find $LL_i = \sum_j \ln(\text{estimated } \Pr(\text{the value of feature } j|\text{class } i))$ using counts for class i
6. Return $\text{argmin}_i(LL_i)$

```
class NaiveBayes
{
    struct Feature
    {
        int count;
        LinearProbingHashTable<int, int> valueCounts;
        Feature(): count(0) {}
        void add(int value)
        {
            ++count;
            int* valueCount = valueCounts.find(value);
            if (valueCount) ++*valueCount;
            else valueCounts.insert(value, 1);
        }
        double prob(int value)
        {
            int* valueCount = valueCounts.find(value);
            return (valueCount ? 1 + *valueCount : 1) / (1.0 + count);
        }
    };
    typedef ChainingHashTable<int, Feature> FEATURE_COUNTS;
    typedef ChainingHashTable<int, FEATURE_COUNTS> CLASS_COUNTS;
    mutable CLASS_COUNTS counts;
public:
    typedef Vector<pair<int, int>> SPARSE_CATEGORICAL_X;
    static SPARSE_CATEGORICAL_X convertToSparse(CATEGORICAL_X const& x)
    {
        SPARSE_CATEGORICAL_X result;
        for (int i = 0; i < x.getSize(); ++i)
            result.append(make_pair(i, x[i]));
        return result;
    }
    void learn(SPARSE_CATEGORICAL_X const& x, int label)
    {
        for (int i = 0; i < x.getSize(); ++i)
        {
            FEATURE_COUNTS* classCounts = counts.find(label);
            if (!classCounts) classCounts = &counts.insert(label,

```

```

        FEATURE_COUNTS() -> value;
    Feature* f = classCounts->find(x[i].first);
    if(!f) f = &classCounts->insert(x[i].first, Feature() ->value;
    f->add(x[i].second);
}
}

int predict(SPARSE_CATEGORICAL_X const& x) const
{
    double maxLL;
    int bestClass = -1;
    for(CLASS_COUNTS::Iterator i = counts.begin(); i != counts.end();
        ++i)
    {
        double ll = 0;
        for(int j = 0; j < x.getSize(); j++)
        {
            Feature* f = i->value.find(x[j].first);
            if(f) ll += log(f->prob(x[j].second));
        }
        if(bestClass == -1 || maxLL < ll)
        {
            maxLL = ll;
            bestClass = i->key;
        }
    }
    return bestClass;
}
};

```

Features are rarely independent, and need a lot of data to sufficiently cover every feature-class-value combination. So naive Bayes isn't competitive with the best methods, except some problems. But it's important to understand conceptually and has many advantages:

- Learning n examples with D features, k classes, and v values takes $O(nD)$ time and $O(Dkv)$ space, and classifying takes $O(kD)$ time.
- Very reliable by ORM because the complexity term $O\left(\sqrt{\frac{Dkv}{n}}\right)$ is very small for large n .
- Learning is online, allowing appearance of new classes, features, and feature values.
- Natural handling of sparse features and missing values. E.g., this makes naive Bayes the method of choice for e-mail spam classification.

Can handle numeric data using equal-width binning:

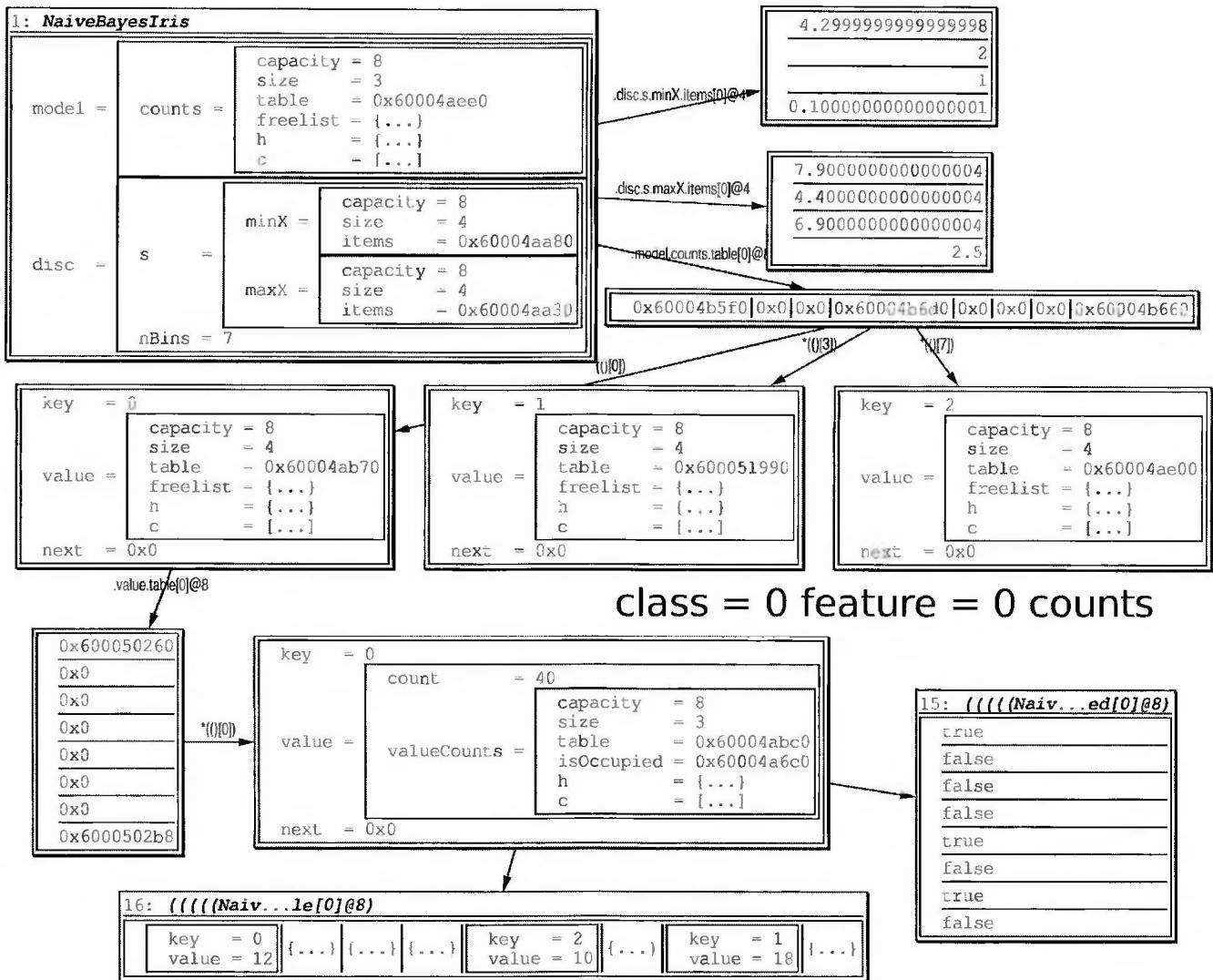


Figure 26.3: Memory layout of naive Bayes learned structure

```

struct NumericalBayes
{
    NaiveBayes model;
    DiscretizerEqualWidth disc;
    template<typename DATA> NumericalBayes(DATA const& data) : disc(data)
    {
        for(int i = 0; i < data.getSize(); ++i) model.learn(NaiveBayes:::
            convertToSparse(disc(data.getX(i))), data.getY(i));
    }
    int predict(NUMERIC_X const& x) const
        { return model.predict(NaiveBayes::convertToSparse(disc(x))); }
};

```

But this needn't work well and isn't online; naive Bayes is useful only for discrete or easily discretized data.

The main benefit is that learning is online, allowing appearance of new classes, features, and feature values. But the assumption of completely independence and equally relevant features is far from realistic, so the use case is limited. But conceptual understanding of this basic algorithm is important.

26.7 Nearest Neighbor

When X has a distance function d , can remember S , and classify by returning y of the nearest example. E.g., children can learn letters after being shown each once (though they see it many times). The calculation reduces to the nearest neighbor problem (see the “Computational Geometry” chapter):

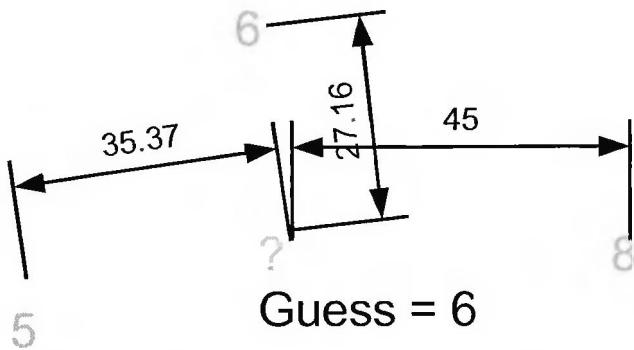


Figure 26.4: "?" = the class of nearest example

d must be meaningful and properly scaled. E.g., Euclidean distance in age-height space doesn't mean much. Equivalently, d can do the scaling.

More generally, can use the majority class of k (don't confuse with the number of classes) nearest neighbors (k -NN) for $k > 1$. If X is a vector space with $D < \infty$, as $n \rightarrow \infty$ (Devroye et al. 1996):

- If $k \rightarrow \infty$, and $k/n \rightarrow 0$, k -NN is universally consistent— k must increase with n
- $R_{k\text{-NN}} \leq R_{\text{OB}}(1 - R_{\text{OB}}) \leq 2R_{\text{OB}}$ — k -NN with lots of data has certain optimality
- For odd k , $R_{(k+1)\text{-NN}} = R_{k\text{-NN}} \leq R_{\text{OB}} + \frac{1}{\sqrt{ke}}$ —use odd k

These don't hold if $D = \infty$ or for general metric spaces (Cerou & Guyader 2006). Scaling is theoretically irrelevant because a change of scale is equivalent to a change of P . Because want odd k , $k=2\lfloor \lg(n)/2+1 \rfloor$ seems to be a good default. It satisfies consistency, and agrees with the typically recommended $k = 5$.

For fixed n , as k increases, bias increases and variance decreases (Domingos 2000), which also supports increasing k with n . The runtime of a k -NN query also increases with k due to less effective bounds, but this is a concern only beyond small k . The algorithm is online (though not implemented here as such).

```
template<typename X = NUMERIC_X, typename INDEX = VpTree<X, int, typename
EuclideanDistance<X>::Distance> > class KNNClassifier
{
    mutable INDEX instances;
    int n, nClasses;
public:
    KNNClassifier(int theNClasses): nClasses(theNClasses), n(0) {}
    template<typename DATA> KNNClassifier(DATA const& data): n(0),
        nClasses(findNClasses(data))
    {
        for(int i = 0; i < data.getSize(); ++i)
            learn(data.getY(i), data.getX(i));
    }
    void learn(int label, X const& x){instances.insert(x, label); ++n;}
    int predict(X const& x) const
    {
        Vector<typename INDEX::NodeType*> neighbors =
            instances.kNN(x, 2 * int(log(n))/2 + 1);
        Vector<int> votes(nClasses);
        for(int i = 0; i < neighbors.getSize(); ++i)
            ++votes[neighbors[i]->value];
        return argMax(votes.getArray(), votes.getSize());
    }
};
```

For the digit data, nearest neighbor is one of the most accurate methods, getting 97.96% accuracy in very little time.

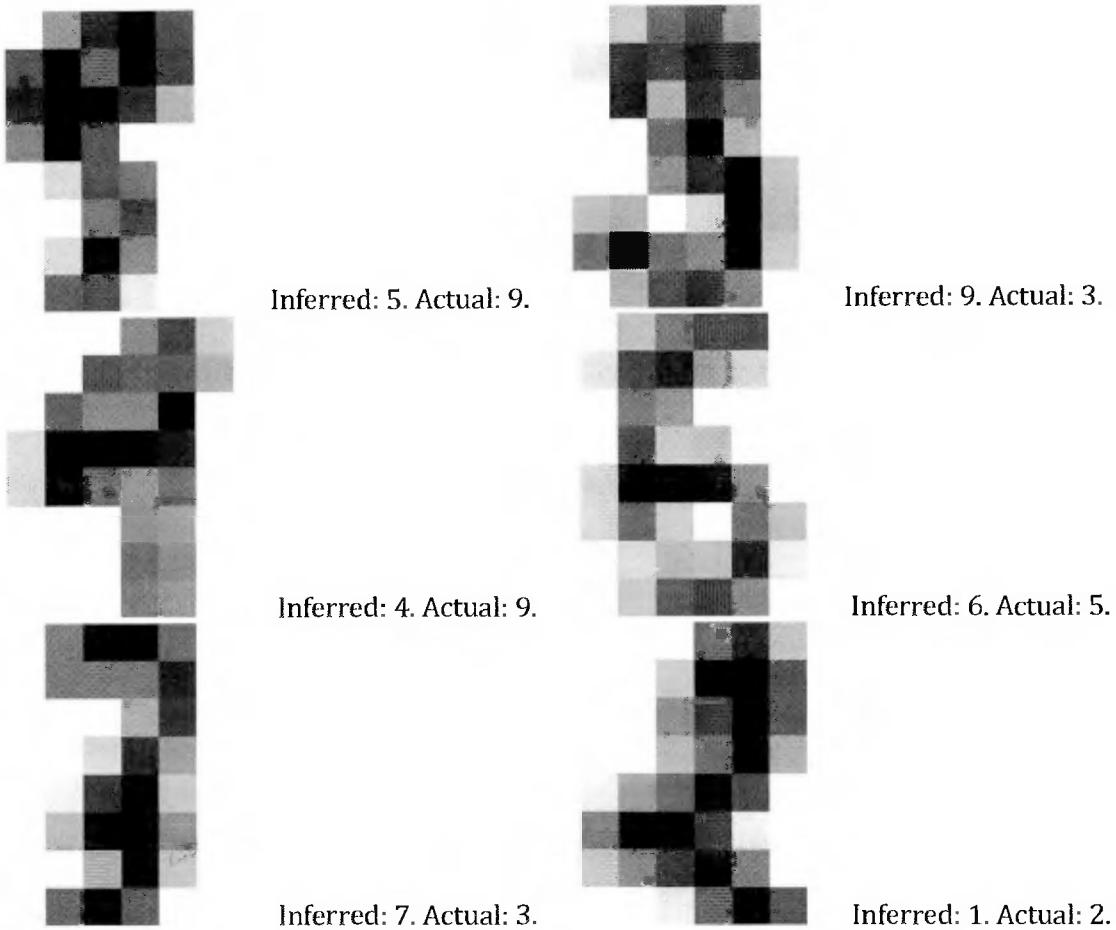


Figure 26.5: Some k -NN errors

On training data, 1-NN has perfect accuracy, but its VC dimension = n , and $|h|$ is $O(n)$, so neither VC dimension nor ORM bounds guarantee generalization. But further assumptions give a finite-sample bound. Theorem (Shalev-Schwartz & Ben-David 2014): Let $P(y|x)$ be Lipschitz with constant c , X a vector space, and $k > 1$. Then for Euclidean distance $R_{k\text{-NN}} \leq \left(1 + \sqrt{\frac{8}{k}}\right)R_{\text{ob}} + \frac{6c\sqrt{D+k}}{n^{1/(D+1)}}$; c is usually unknown, so this doesn't even help select k . But see the curse of dimensionality—learning becomes more difficult with large D . For points $\sim \text{uniform}([0, 1]^D)$, the distribution of distances is highly skewed to the maximum distance, so it's easier to pick a wrong neighbor due to noisy features. But for typical practical data the performance is usually good because the underlying dimension of that data is much smaller, and the distances in its feature space don't have the same problem because the distribution in X is far from uniform.

In practice, k -NN has high accuracy even with little training data, but uses too much memory and can be slow for large D , perhaps enough to not qualify for real time applications (in early 1990's for digit recognition k -NN with special tangent distance was famously rejected in favor of a worse-performing neural network). So k -NN is useful only for nonvector data, where one can come up with a good distance function such as edit distance between strings. Based on my experiments VP tree is fast even for $n = 10^6$, so memory limit seems to be the bottleneck.

A somewhat different approach is **nearest mean**—compute centroids of each class and assign x to the class of the nearest centroid. It's not competitive but interesting because it corresponds to **k -means** for clustering (see the “Machine Learning—Other Tasks” chapter), and its mediocre performance ($\approx 84\%$) suggests a similar problem for k -means.

26.8 Decision Tree

Create a binary tree where nodes look at specific feature values to decide which branch to take, and leaves give the resulting class. This is similar to what k -d tree (see the “Computational Geometry Algorithms” chapter) does. The latter typically uses the next-in-order feature with the value of a random example as split, while the former greedily picks the best feature and value. **Decision trees are easily interpretable**, so with unknown data they are the first A to use for getting some intuition.

Creating an optimal tree using any reasonable error measure is NP-complete (Hyafl & Rivest 1976). Practical construction greedily proceeds top-down, picking the best feature and its value as the split point

for the root, and recursively constructing its children.

A good split criteria is entropy = $\sum H(p_i)$, with p_i = the % of examples of class i ($H(p) = -\sum p \log(p)$; see the “Compression” chapter). Minimize the total entropy after split = the number of left child examples \times entropy(left child examples) + the number of right child examples \times entropy(right child examples). With equally accurate splits, entropy favors those where in the resulting nodes some classes are more concentrated than others, so prefer it over accuracy. Intuitively, with every split, entropy gains more information about the labels by being able to compress them better. Eventually, all examples in a leaf have the same label, entropy = 0, and splitting stops.

For a numerical feature have $n - 1$ possible splits at the root. An efficient way to pick the best is sorting the values and considering all splits one-by-one from left to right, updating after each split the left and the right count tables \forall class, and recomputing entropy from these. Considering all splits at the root takes $O(n \lg(n) D)$ time.

The worst-case depth is n , and a tree that reaches it would overfit and take much longer to build, so restrict the depth. The best case depth = $\lg(n \text{Classes})$ due to similarity of examples. A typical depth for a modestly effective tree is probably based on the fact that most data remains in a single large chunk, and every split peels off a small fraction a of it. Here depth = $\frac{\lg(n)}{\lg(1/a)}$ and, because the maximum depth mostly controls runtime and not complexity, the default limit = 50 seems robust—smaller values wouldn't improve the runtime by much and larger ones won't improve generalization. Depth restriction also makes it unnecessary to use a stack instead of recursion. Some problems need greater depth, i.e., D -dimensional xor needs a complete binary tree of depth D , but meaningful data usually doesn't have such complexity. A larger number such as 100 buys little safety at the expense of efficiency loss when for difficult tasks deep nodes are created and later pruned (pruning is explained later).

Decision trees have no error on the training set (unless depth restriction stops growth), and so overfit. \forall function \exists a decision tree that represents it arbitrarily well. For $k = 2$, VC dimension of a tree = the number of leaves (Shalev-Schwartz & Ben-David 2014), which isn't a specific function of depth but is bounded by a function of the depth restriction. **Pruning** replaces a subtree by a majority leaf.

A simple way to decide whether to prune a subtree is the sign test (see the “Computational Statistics” chapter). A correctly/incorrectly classified example by both the tree and the node is a draw, otherwise it's a win for whoever got it right. Because the subtree isn't worse than the node, the number of wins for the node and the subtree is respectively $n\text{Draws}/2$ and $(n\text{Draws}/2 + \text{the difference in the correctly classified examples})$. By default, pruning uses z-score = 1, which overprunes a little but gives a much simpler tree. Experimentally, 0.5 and 0.25 give the best results, and values > 1 do poorly for most tested data sets. So the usually statistical $z = 2$ corresponding to 95% confidence is too conservative. Selecting z-score using cross-validation doesn't improve and is slower.

DT_zcv 50	DT-z0_50	DT-z0.25_50	DT-z0.5_50	DT-z1_50	DT-z2_50
0.889	0.892	0.896	0.892	0.871	0.779

Figure 26.6: Performances of some decision tree options. All performance measures use curved balanced accuracy, averaged across several data sets.

Pruning is done recursively, on the way back after a tree is constructed and not before to avoid a situation like the xor problem, where any first feature isn't effective, and any second one is. The overall algorithm:

1. Find the best split with incremental calculation
2. Split the data based on it into left and right parts
3. Recurse on the parts until get a pure node or exceed some depth m
4. Try to prune

The implementation assumes numerical x and has several safeguards to avoid issues with bad data. It supports random forest mode (discussed later in this chapter).

```
struct DecisionTree
{
    struct Node
    {
        union
        {
            int feature; //for internal nodes
            int label; //for leaf nodes
        };
    };
}
```

```

double split;
Node *left, *right;
bool isLeaf(){return !left;}
Node(int theFeature, double theSplit): feature(theFeature),
    split(theSplit), left(0), right(0) {}
} *root;
Freelist<Node> f;
double H(double p){return p > 0 ? p * log(1/p) : 0;}
template<typename DATA> struct Comparator
{
    int feature;
    DATA const& data;
    double v(int i)const{return data.data.getX(i, feature);}
    bool operator()(int lhs, int rhs)const{return v(lhs) < v(rhs);}
    bool isEqual(int lhs, int rhs)const{return v(lhs) == v(rhs);}
};
void rDelete(Node* node)
{
    if(node)
    {
        rDelete(node->left);
        f.remove(node->left);
        rDelete(node->right);
        f.remove(node->right);
    }
}
typedef pair<Node*, int> RTYPE;
template<typename DATA> RTYPE rHelper(DATA& data, int left, int right,
    int nClasses, double pruneZ, int depth, bool rfMode)
{
    int D = data.getX(left).getSize(), bestFeature = -1,
        n = right - left + 1;
    double bestSplit, bestRem, h = 0;
    Comparator<DATA> co = {-1, data};
    Vector<int> counts(nClasses, 0);
    for(int j = left; j <= right; ++j) ++counts[data.getY(j)];
    for(int j = 0; j < nClasses; ++j) h += H(counts[j] * 1.0/n);
    int majority = argMax(counts.getArray(), nClasses),
        nodeAccuracy = counts[majority];
    Bitset<> allowedFeatures;
    if(rfMode)
        //sample features for random forest
        allowedFeatures = Bitset<>(D);
        allowedFeatures.setAll(0);
        Vector<int> p = GlobalRNG().sortedSample(sqrt(D), D);
        for(int j = 0; j < p.getSize(); ++j) allowedFeatures.set(p[j], 1);
    }
    if(h > 0) for(int i = 0; i < D; ++i) //find best feature and split
        if(allowedFeatures.getSize() == 0 || allowedFeatures[i])
        {
            co.feature = i;
            quickSort(data.permutation.getArray(), left, right, co);
            int nRight = n, nLeft = 0;
            Vector<int> countsLeft(nClasses, 0), countsRight = counts;
            for(int j = left; j < right; ++j)
                //incrementally roll counts
                int label = data.getY(j);
                ++nLeft;
                ++countsLeft[label];
                --nRight;
                --countsRight[label];
            double fLeft = data.getX(j, i), hLeft = 0,
                fRight = data.getX(j + 1, i), hRight = 0;
        }
}

```

```

        if(fLeft != fRight)
        {//don't split equal values
            for(int l = 0; l < nClasses; ++l)
            {
                hLeft += H(countsLeft[l] * 1.0/nLeft);
                hRight += H(countsRight[l] * 1.0/nRight);
            }
            double rem = hLeft * nLeft + hRight * nRight;
            if(bestFeature == -1 || rem < bestRem)
            {
                bestRem = rem;
                bestSplit = (fLeft + fRight)/2;
                bestFeature = i;
            }
        }
    }

    if(depth <= 1 || h == 0 || bestFeature == -1)
        return RTYPE(new(f.allocate())Node(majority, 0), nodeAccuracy);
    //split examples into left and right
    int i = left - 1;
    for(int j = left; j <= right; ++j)
        if(data.getX(j, bestFeature) < bestSplit)
            swap(data.permutation[j], data.permutation[++i]);
    if(i < left || i > right)
        return RTYPE(new(f.allocate())Node(majority, 0), nodeAccuracy);
    Node* node = new(f.allocate())Node(bestFeature, bestSplit);
    //recursively compute children
    RTYPE lData = rHelper(data, left, i, nClasses, pruneZ, depth - 1,
        rfMode), rData = rHelper(data, i + 1, right, nClasses, pruneZ,
        depth - 1, rfMode);
    node->left = lData.first;
    node->right = rData.first;
    int treeAccuracy = lData.second + rData.second, nTreeWins =
        treeAccuracy - nodeAccuracy, nDraws = n - nTreeWins;
    //try to prune
    if(!rfMode &&
        signTestAreEqual(nDraws/2.0, nDraws/2.0 + nTreeWins, pruneZ))
    {
        rDelete(node);
        node->left = node->right = 0;
        node->label = majority;
        node->split = 0;
        treeAccuracy = nodeAccuracy;
    }
    return RTYPE(node, treeAccuracy);
}
Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate())Node(*node);
        tree->left = constructFrom(node->left);
        tree->right = constructFrom(node->right);
    }
    return tree;
}
public:
    template<typename DATA> DecisionTree(DATA const& data, double pruneZ = 1,
        bool rfMode = false, int maxDepth = 50): root(0)
    {
        assert(data.getSize() > 0);
    }
}

```

```

int left = 0, right = data.getSize() - 1;
PermutedData<DATA> pData(data);
for(int i = 0; i < data.getSize(); ++i) pData.addIndex(i);
root = rHelper(pData, left, right, findNClasses(
    data), pruneZ, maxDepth, rfMode).first;
}
DecisionTree(DecisionTree const& other)
    {root = constructFrom(other.root);}
DecisionTree& operator=(DecisionTree const& rhs)
    {return genericAssign(*this, rhs);}
int predict(NUMERIC_X const& x) const
{
    assert(root); //check for bad data
    Node* current = root;
    while(!current->isLeaf()) current = x[current->feature] <
        current->split ? current->left : current->right;
    return current->label;
}
};

```

The resulting tree for the iris data uses only petal length (feature 2) and gets 100% accuracy on seosa (label 0) and versicolor (label 1), but only 60% on virginica (label 2):

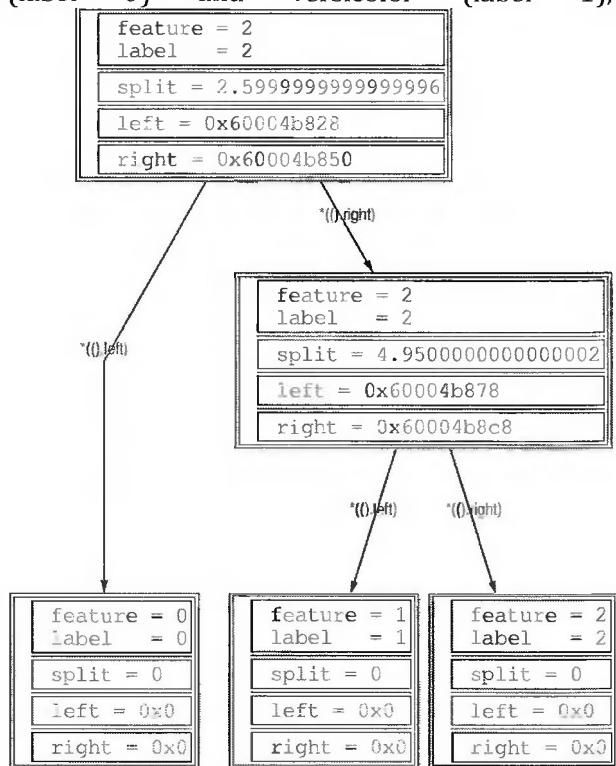


Figure 26.7: Memory layout of a decision tree on the iris data

The unpruned tree is a bit more complex due to also using petal and sepal widths but has higher accuracy:

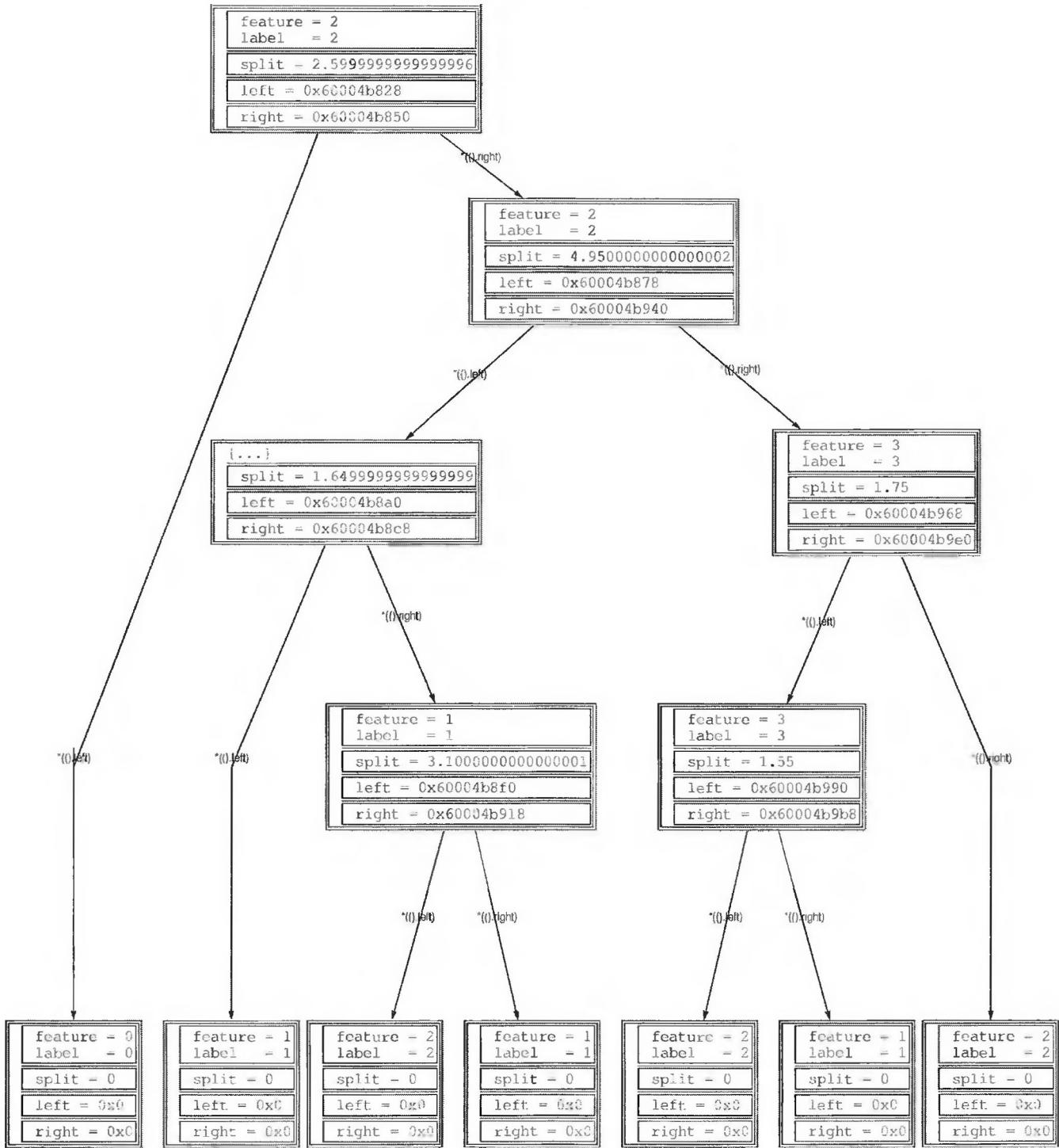


Figure 26.8: Memory layout of an unpruned decision tree on the iris data

Assuming balanced splits, because for the root (and recursively \forall node with n examples) picking the split takes $O(n \lg(n) Dk)$ time, the total work for the root $C(n)=2C(n/2)+O(n \lg(n) Dk)$, so by the master theorem $C(n)=O(n \lg(n)^2 Dk)$. Because the maximum depth is m , the worst case is $O(m n \lg(n) Dk)$. Pruning doesn't affect this.

For applying ORM, can encode tree structure using 2 bits per node, i.e., using depth-first traversal to write a node, then 1 bit for the left child, and 1 bit for the right. Feature id needs $\lceil \lg(D) \rceil$ bits, and label $\lceil \lg(k) \rceil$. For a split, can scale the data into $[0, 1]$ and represent to precision 0.001, so about 10 bits is enough; scaling has no influence on the logic because decision trees are scale-invariant. Have $m/2$ internal nodes and $m/2 + 1$ leaves, so $|h| \approx m \left(10 + \frac{\lg(kD)}{2} \right)$, which, assuming $\lg(kD) < 20$ and $p = 0.05$, leads to complexity term $\approx 3.7 \sqrt{\frac{m}{n}}$.

For $k = 2$, getting a tree with d leaves means (by the VC dimension bound) that with $p = 0.05$ have the

complexity term $\approx \sqrt{\frac{d \ln(2en/d)}{n}} + \sqrt{\frac{1.5}{n}} \approx 1.8\sqrt{\frac{m}{n}}$, assuming $n/d < 100$ in the log. For either complexity measure, for a three-node leaf subtree want $R_{\text{root}} < R_{\text{tree}} + a\sqrt{\frac{1}{n}}$, which is what the sign test checks with a z-score as a .

The number of used features $\leq n$, and usually the same features are selected, making decision tree a good embedded feature selector.

Some flaws of a decision tree:

- Not universally consistent because \exists distributions where splitting based on entropy fails to make progress on S of size ∞ (Devroye et al. 1996). But practical data sets don't have such structure.
- Decision boundaries tend to be very coarse, consisting of piecewise functions. If the data is linearly separable, this doesn't matter if the margin is wider than the coarseness. Random forest smooths out the boundary, usually increasing accuracy.
- Instability (Breiman 1996). Small changes in the data lead to large changes in the structure, but little change in overall accuracy. With pruning, this is less of a problem.

26.9 Support Vector Machine

SVM is a theoretically good classifier with very good practical performance on many types of data. The basic version works with $k = 2$. If the data is linearly separable, \exists a set of hyperplanes $f(x) = wx + b$ such that $\forall x_i \in \text{data}, f(x_i) < 0$ if $y_i = -1$ and > 0 if $y_i = 1$. SVM computes a hyperplane separating the two classes, with w and b such that the margin = minimum distance(x_i , hyperplane) is maximal. I.e., the x_i closest to the hyperplane are equidistant to it. Intuitively, when walking in a minefield, it's best to walk halfway between the mines.

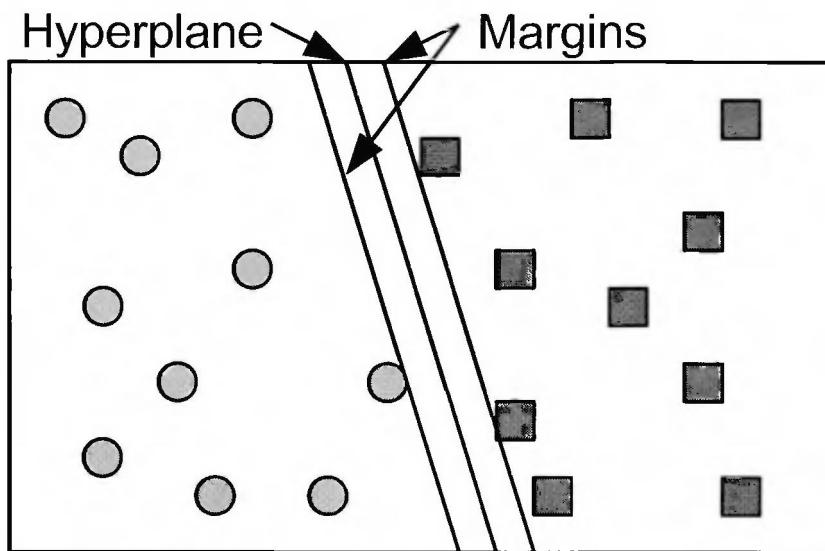


Figure 26.9: A typical SVM separation

$\text{Distance}(x, \text{hyperplane}) = |f(x)| / \|w\|$. Because can scale f to produce arbitrary margins, normalize to get a **canonical hyperplane** where $\min|f(x)| = 1$. So for support vectors, $|f(x)| = 1$, and the margin = $1/\|w\|$ —maximizing it is equivalent to minimizing w^2 . General SVM works for linearly nonseparable data by:

- Mapping x to a higher dimensional enhanced feature space using a feature-mapping function F . The hyperplane becomes $f(x) = wF(x) + b$.
- Allowing some x_i to be outside the margins—**soft SVM**. Use **slack variables** ϵ_i to decide to what degree x_i can be outside.

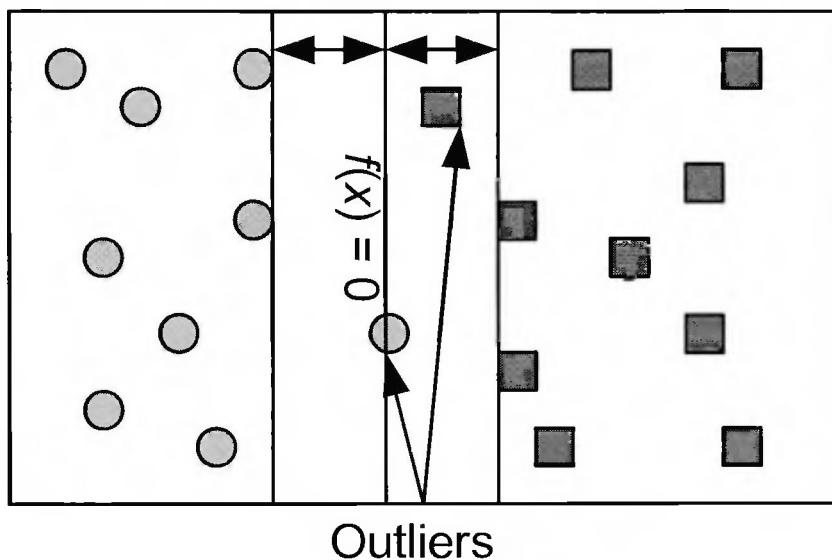


Figure 26.10: A typical soft SVM separation

Get a constrained optimization problem (Bottou et al. 2007):

- $\min \frac{1}{2}w^2 + C\sum \epsilon_i$ subject to
- $y_i f(x_i) \geq 1 - \epsilon_i$
- $\epsilon_i \geq 0$

The constant $C \geq 0$ defines the trade-off between margin size and accuracy; $C = 0$ means no x_i can be outside. The cases, depending on $y_i f(x_i)$:

- > 1 —correctly classified and away from but on the right side of the margin
- $= 1$ —correctly classified and on the margin
- > 0 and < 1 —correctly classified and inside the margin
- < 0 —incorrectly classified and on the wrong side of the margin

Though can plug in VC dimension d of a hyperplane to bound risk, better analysis is possible. The simplest case is when the data is linearly separable. Theorem (Mohri et al. 2018): Assume that the coordinate plane is shifted so that the optimal $b = 0$. Let r be the radius of the smallest hypersphere containing the data

in the enhanced space and Q such that $\min_i |f(x_i)| = 1$, and $\frac{1}{\|w\|} \leq Q$ is the size of the margin. Then \forall hyperplane with such w , $d \leq (rQ)^2$. So parameter values and not just their number bound d , and maximizing the margins is a legitimate goal.

26.10 Linear SVM

Using identity F results in a linear SVM, which is restricted to linear class boundaries but much faster to train and online. SVM constraints imply that $\epsilon_i = \max(0, 1 - y_i f(x_i))$, leading to an equivalent unconstrained problem $\min \frac{1}{2}lw^2 + \sum \max(0, 1 - y_i f(x_i))$, with $l = 1/C$. Prefer penalty $l\|w\|$ because using L_1 regularization of the weights (also called **lasso**) tends to produce a **sparse solution**, with many $w_i = 0$ (Hastie et al. 2015). This is because in the equivalent constrained problem the feasible region contains cusps corresponding to some variables set to 0, and one of them is likely to attain the best feasible level set:

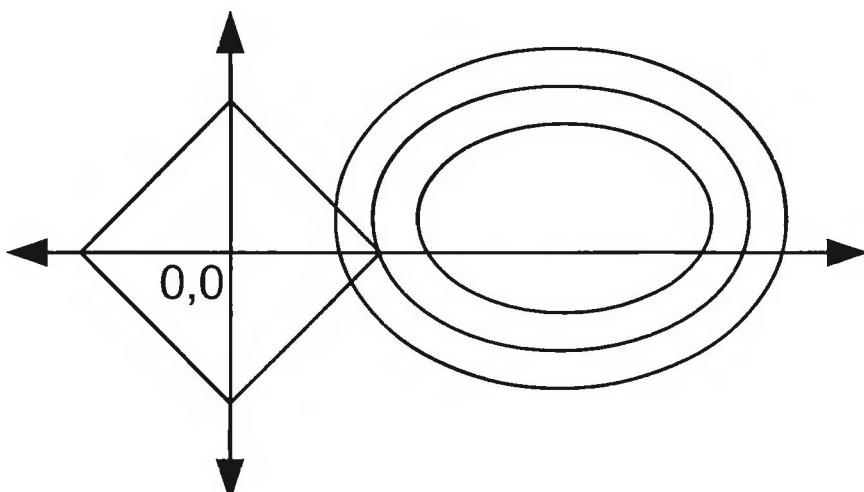


Figure 26.11: A level set at a cusp is maximal

This problem is convex, but not differentiable. Using SGD gives a good approximate solution with regard to estimation error (Montavon et al. 2012; Bottou 2010) and allows online learning. Need an expression whose expected value is a subgradient of the function. Here, the subgradient is a sum, and use $n \times$ the value of example i , for random i . The update equations for observing example i :

- $\forall 0 \leq j \leq D, w_{ij} := r_i(w_{ij} > 0 ? 1 : -1) I - n(y_i f(x_i) > 1 ? 0 : y_i x_i)$
- $b += r_i(y_i f(x_i) > 1 ? 0 : y_i)$

So examples outside the margin increase it, and examples on the correct side shrink it. Presenting close-to-margin examples first or last gives a different answer, so the order matters. For parameters:

- The initial learning rate = 1, which seems to work well for many problems, without divergence.
- The number of passes over the data = $[10^5/n]$ because SGD convergence asymptotically is $O(\sqrt{n})$. This reaches relative precision $O(0.003)$, which seems reasonable and performs well in my experiments. For very large n , only do a single pass, and it may not be necessary to go through all the data. For small n , a single pass isn't enough.
- Order = stratified by class or random. The default order can have examples sorted by class, so stratified is usually the best, but random may be simpler and has known theoretical performance.

Though SGD is good for generalization, it's not for deciding which coefficients are 0 to get a sparse solution due to slow convergence. Doing the latter is an active research topic (Hastie et al. 2015), but coordinate descent (see the "Numerical Optimization" chapter) usually does well in batch mode. To optimize one weight at a time efficiently, use $f(x_i) = \text{sum}_j + w_{ij} x_{ij}$ or $\text{sum}_i + b$, with sum being the sum of the untouched weights. So can evaluate the impact of variable j in $O(n)$ time, independently of D . The 1D optimizations are solved using bracketing and golden section (see the "Numerical Optimization" chapter).

Coordinate descent isn't guaranteed to converge or be fast, but usually no problems happen. For the implementation:

- Evaluation budget = 10^5 .
- Use the defaults for the termination precision and the initial step size. These work well for getting irrelevant variables to 0.
- Because SGD is good at initial convergence and for robustness, when coordinate descent gets stuck, do it first.

1. **forall pass process every example with SGD**
2. **Run coordinate descent to get better precision**
3. **For prediction, given x , compute the hyperplane margin, and classify accordingly**

```
class BinaryLSVM
{
    Vector<double> w;
    double b, l;
    int learnedCount;
    static int y(bool label){return label * 2 - 1;}
    static double loss(double fxi, double yi){return max(0.0, 1 - fxi * yi);}
    double f(NUMERIC_X const& x) const{return dotProduct(w, x) + b;}
    template<typename DATA> class GSL1Functor
    {
        DATA const& data;
    };
}
```

```

mutable Vector<double> sums;
Vector<double> &w;
double &b, l;
int j, D;
mutable int evalCount;
double getSumI(double wj, int i) const
{
    return sums[i] + (j == D ? wj - b : (wj - w[j]) * data.getX(i, j));
}
public:
GSL1Functor(DATA const& theData, double& theB, Vector<double>& theW,
double theL): data(theData), sums(theData.getSize(), theB),
w(theW), b(theB), l(theL), j(0), D(getD(data)), evalCount(0)
{
    for(int i = 0; i < data.getSize(); ++i)
        sums[i] += dotProduct(w, data.getX(i)) + b;
}
void setCurrentDimension(int theJ)
{
    assert(theJ >= 0 && theJ < D + 1);
    j = theJ;
}
int getEvalCount() const {return evalCount; }
int getSize() const {return D + 1; }
Vector<double> getX() const
{
    Vector<double> x = w;
    x.append(b);
    return x;
}
double getXi() const {return j == D ? b : w[j]; }
double operator() (double wj) const
{
    ++evalCount;
    double result = j == D ? 0 : l * abs(wj);
    for(int i = 0; i < data.getSize(); ++i)
        result += loss(getSumI(wj, i), y(data.getY(i)));
    return result / data.getSize();
}
void bind(double wjNew)
{//first update sum
    for(int i = 0; i < data.getSize(); ++i)
        sums[i] = getSumI(wjNew, i);
    (j == D ? b : w[j]) = wjNew;
}
};
public:
BinaryLSVM(pair<int, double> const& p): w(p.first), l(p.second), b(0),
learnedCount(0) {}
template<typename DATA> BinaryLSVM(DATA const& data, double theL,
int nGoal = 100000, int nEvals = 100000): l(theL), b(0),
w(getD(data)), learnedCount(0)
{//first SGD
    for(int j = 0; j < ceiling(nGoal, data.getSize()); ++j)
        for(int i = 0; i < data.getSize(); ++i)
            learn(data.getX(i), data.getY(i), data.getSize());
//then coordinate descent
GSL1Functor<DATA> f(data, b, w, l);
unimodalCoordinateDescent(f, nEvals);
}
int getLearnedCount() {return learnedCount; }
void learn(NUMERIC_X const& x, int label, int n = -1)
{//online mode uses SGD only
}

```

```

if(n == -1) n = learnedCount + 1;
double rate = RMRate(learnedCount++), yl = y(label);
for(int i = 0; i < w.getSize(); ++i)
    w[i] -= rate * (w[i] > 0 ? 1 : -1) * 1/n;
if(yl * f(x) < 1)
{
    w -= x * (-yl * rate);
    b += rate * yl;
}

int predict(NUMERIC_X const& x) const{return f(x) >= 0;}
template<typename MODEL, typename DATA>
static double findL(DATA const& data)
{//used for regression as well
    int lLow = -15, lHigh = 5;
    Vector<double> regs;
    for(double j = lHigh; j > lLow; j -= 2) regs.append(pow(2, j));
    return valMinFunc(regs.getArray(), regs.getSize(),
        SCVRiskFunctor<MODEL, double, DATA>(data));
}
};

1: LSVMIris

```

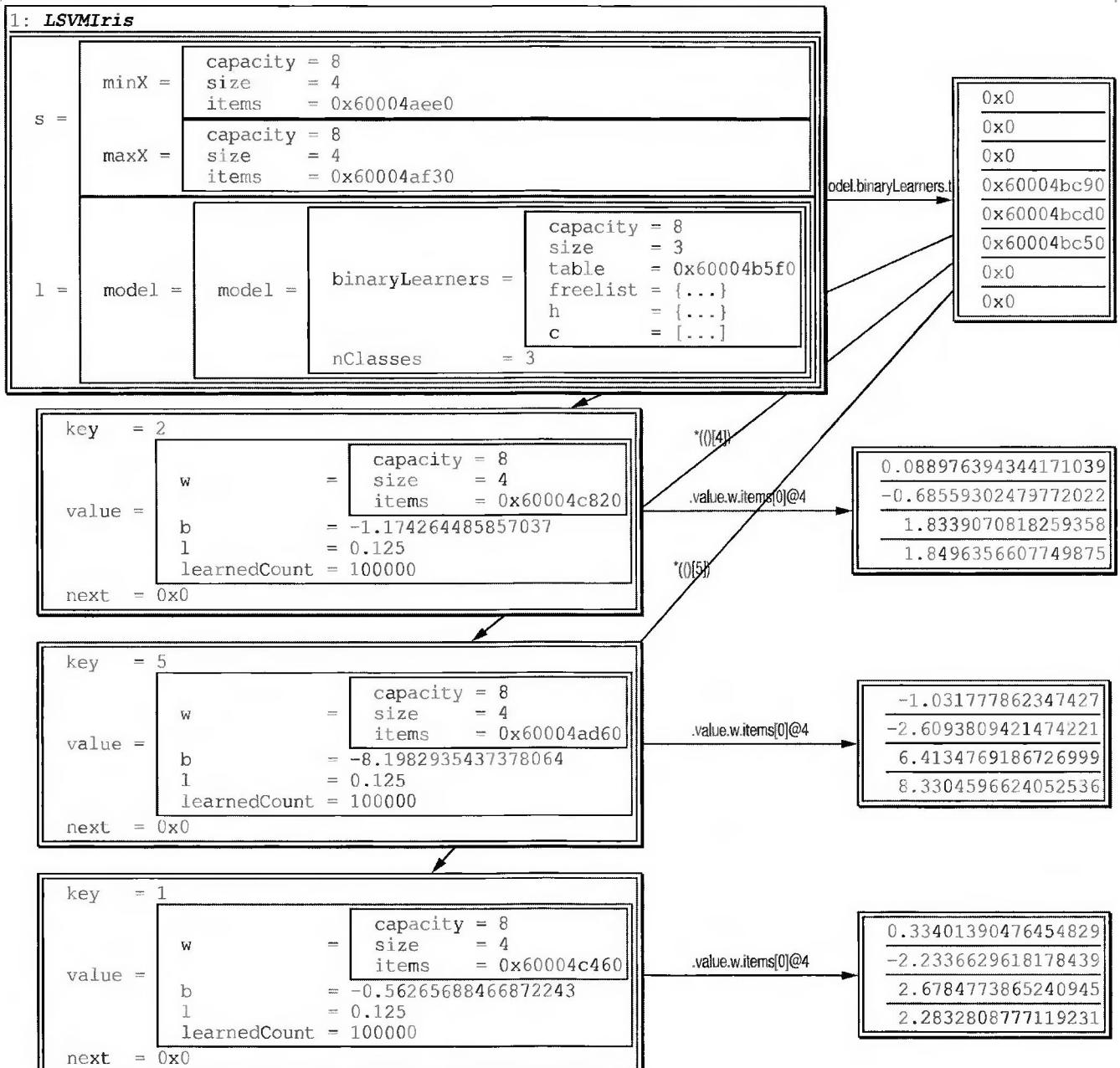


Figure 26.12: Memory layout of a LSVM on the iris data

No simplification of the model is done, though some coefficients will be on the level of machine precision. This and any inspection of the model are left to the user because some common-sense tolerance such as 10^{-4} needn't work for all domains.

Coordinate descent is slow when the data is stored on disk example by example because of noncontiguous access. Here store the data in D contiguous pieces, each containing all examples from single feature. Even in memory, if have extra, should use a buffer to remove the slowdown due to repeated data pipeline evaluation, particularly scaling.

SLSVM100k	SLSVMcd100	mqlSVMsgd100kcd100	SLSVMsgd100kcd100
0.874	0.780	0.895	0.917

Figure 26.13: Performance comparison of some choices ("S" means default [0, 1] range scaling)

LSVM can't solve the xor problem with $k = 2$ because \exists a line that can. Despite such lack of modeling power, it's a useful method for many problems where D is high or the classes are mostly separable.

For $k > 2$ use OVO. For better estimation use a single l for all binary learners. Pick C using grid search with cross-validation among values 2^i , with odd $i \in [-15, 5]$. For best generalization start with large l first.

```
template<typename MODEL, typename DATA>
static double findL(DATA const& data)
{
    int lLow = -15, lHigh = 5;
    Vector<double> regs;
    for(double j = lHigh; j > lLow; j -= 2) regs.append(pow(2, j));
    return valMinFunc(regs.getArray(), regs.getSize(),
        SCVRiskFunctor<MODEL, double, DATA>(data));
}

struct NoParamsLSVM
{
    typedef MulticlassLearner<BinaryLSVM, double> MODEL;
    MODEL model;
    template<typename DATA> NoParamsLSVM(DATA const& data) : model(data,
        BinaryLSVM::findL<MODEL, DATA>(data)) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};

typedef ScaledLearner<NoParamsLearner<NoParamsLSVM, int>, int> SLSVM;
```

26.11 Kernel SVM

For problems like xor \exists nonlinear separators. To introduce nonlinearity include artificial features that are nonlinear functions of the original features ones using kernels. L_1 regularization would remove features in the enhanced space without affecting the original space ones, so minimize margins instead.

With kernels the problem is still convex because both the objective and the constraints are convex, so strong duality holds (Mohri et al. 2018). The Lagrangian $L = \frac{1}{2}w^T w + C \sum \epsilon_i - \sum a_i(y_i f(x_i) - 1 + \epsilon_i) - \sum c_i \epsilon_i$. The KKT conditions at the optimum (see the "Numerical Optimization" chapter) are:

1. $0 = \nabla_w L = w - \sum a_i y_i f(x_i) \rightarrow w = \sum a_i y_i f(x_i)$
2. $0 = \nabla_b L = -\sum a_i y_i \rightarrow \sum a_i y_i = 0$
3. $0 = \nabla_c L = C - a_i - c_i$
4. $0 = a_i(y_i f(x_i) - 1 + \epsilon_i)$
5. $0 = c_i \epsilon_i$
6. $a_i, c_i \geq 0$

(3) and (6) imply $0 \leq a_i \leq C$. The cases:

- $a_i = C \rightarrow 0 < \epsilon_i = 1 - y_i f(x_i) \rightarrow$ inside or on the wrong side of the margin
- $a_i = 0 \rightarrow \epsilon_i = 0 \rightarrow$ correctly classified and beyond the margin
- $0 < a_i < C \rightarrow 0 = \epsilon_i = 1 - y_i f(x_i) \rightarrow$ correctly classified and on the margin, can compute b from these

Plugging in the dual values for w leads to

$L = \frac{1}{2}(\sum a_i y_i f(x_i))^2 - \sum a_i y_i f(x_i)(\sum a_j y_j f(x_j)) + \sum a_i - b \sum a_i y_i + \sum \epsilon_i(C - a_i - c_i)$. The first two parts combine and last two are 0, so $L = \sum a_i - \frac{1}{2} \sum a_i y_i a_j y_j K_{ij}$. $\forall i$ such that $0 < a_i < C$, $b = y_i - \sum a_j y_j K_{ij}$. Directly using K and simplifying, get:

- $\max V(a) = \sum a_i - \frac{1}{2} \sum a_i y_i a_j y_j K_{ij}$ subject to
- $0 \leq a_i \leq C$
- $\sum a_i y_i = 0$

Give optimal a_i^* for this quadratic programming problem, $f(x) = \sum a_i y_i * K(x_i, x) + b$. **Support vectors** are x_i for which $a_i^* > 0$. Numerically, use the typical precision $\sqrt{\epsilon_{\text{machine}}}$ (such small value may seem surprising, but most examples are never touched during optimization). The solution stores them with the corresponding x_i and y_i .

The special structure of the problem allows a simpler and more efficient solution than by general quadratic programming. Let $d_i = y_i a_i$ (also $a_i = y_i d_i$) and $[L_i, H_i] = [0, C]$ if $y_i = 1$ and $[-C, 0]$ otherwise. Get:

- $\max \sum y_i d_i - \frac{1}{2} \sum d_i d_j K_{ij}$ subject to
- $L_i \leq d_i \leq H_i$
- $\sum d_i = 0$

By **Osuma decomposition theorem** (Bottou et al. 2007), iteratively optimizing over any subset of \geq two variables converges to the solution. In particular, $\forall i$ and j , d_i and d_j are in a box defined by the constraints:

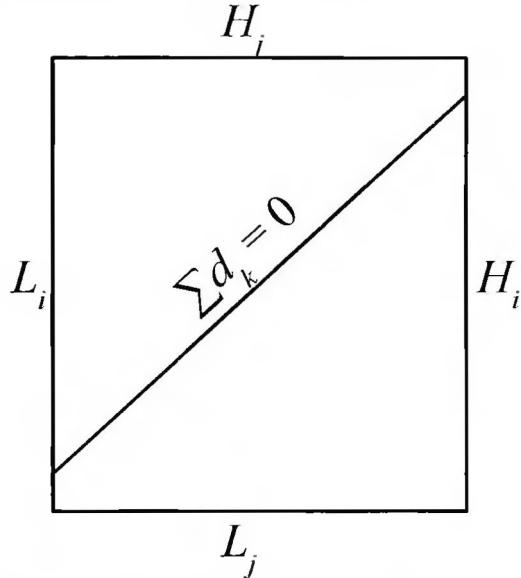


Figure 26.14: The constraints for two variables

Let $g_k = \nabla V(d_k)[k] = y_k - \sum d_l K_{kl}$ (the “ $\frac{1}{2}$ ” disappears due to symmetry). If the solution isn't optimal, $\exists i$ such that $d_i < H_i$ and j such that $d_j > L_j$, so that can increase V by increasing d_i and decreasing d_j by some step

s. Without the box constrains, $d_k(s) = d_k + s_k$, where $s_k = \begin{cases} s, & \text{if } k=i \\ -s, & \text{if } k=j \\ 0, & \text{otherwise} \end{cases}$. Then

$$V(s) = C_1 + \sum_{k \in \{i, j\}} d_k(s)(y_k - \sum_{l \in \{i, j\}} d_l(s)K_{kl}) - \frac{1}{2} \sum_{k, l \in \{i, j\}} d_k(s)d_l(s)K_{kl} = C_2 + \sum_{k \in \{i, j\}} s_k(y_k - \sum_{l \in \{i, j\}} d_l K_{kl}) - \frac{1}{2} \sum_{k, l \in \{i, j\}} (s_k d_l + s_l d_k) K_{kl} - \frac{1}{2} \sum_{k, l \in \{i, j\}} s_k s_l K_{kl}.$$

The symmetric $\frac{1}{2} \sum_{k, l \in \{i, j\}} (s_k d_l + s_l d_k) K_{kl} = \sum_{k \in \{i, j\}} s_k (\sum_{l \in \{i, j\}} d_l K_{kl})$, so can combine the l partitions:

$$V(s) = C_2 + \sum_{k \in \{i, j\}} s_k g_k - \frac{1}{2} \sum_{k, l \in \{i, j\}} s_k s_l K_{kl}. \text{ Maximize } V(s) \text{ when } 0 = \frac{\partial V}{\partial s} = g_i - g_j - s(K_{ii} - 2K_{ij} + K_{jj}). \text{ So}$$

$$s_{\text{opt}} = \frac{g_i - g_j}{K_{ii} - 2K_{ij} + K_{jj}}.$$

This formula is the basis of the **SMO algorithm**:

1. Use the dual representation
2. Initialize the support coefficients to 0 and their gradients to 1
3. For some number of iterations
4. Pick two variables to optimize, using greedy selection
5. Check for convergence
6. Solve the resulting optimization problem analytically
7. Update the gradients
8. Store the nonzero support coefficients and the corresponding vectors
9. To predict x , compute the margins, and choose the maximum-margin class

By positive definiteness of the kernel matrix, the denominator > 0 (it's actually the squared L_2 distance between the examples in the enhanced space) unless K is invalid, numerical cancellation is strong enough, or examples with the same x have different y . For robustness, use dummy $s_{\text{opt}} = \infty$, which will make one such example a support vector (a clumsy alternative is to remove both). Taking the box constraints into

account, $s = \min(H_i - d_i, d_j - L_j, s_{\text{opt}})$. To update g , use $\frac{\partial g_k}{\partial s} = -(K_{ik} - K_{jk})$. The simplest choice of i and j is such that the gap $= g_i - g_j$ is maximal (this is called the **maximal violating pair heuristic**). So the solution is optimal if the gap $= 0$, but for numerical reasons use some small precision, usually 0.001 (Chang & Lin 2011). Because $b = g_k \forall k$ such that $L_k < d_k < H_k$, $g_j < g_k < g_i$. So can set b to g_i or g_j (or for numerical stability to $\frac{g_i + g_j}{2}$) at termination. Can't find proper i and j only if all y are the same, in which case $w = 0$, and $\forall k b = y_k$ is a solution because $\forall k \epsilon_k = 0$.

In practice, the runtime bottleneck is computing K , values of which are cached for efficiency. LRU cache with a memory limit is good for this, but for simplicity the implementation caches all values. Asymptotically, the number of support vector is $O(n)$ with a properly scaled C (Bottou et al. 2007). SMO converges to any precision in finite number of iterations (Bottou et al. 2007), but for safety the implementation sets a budget of $\max(10n, 10000)$. The idea is that each iteration updates two (usually) support vectors, so small data sets get more precision, and larger ones have $O(n)$ support vectors, though typically much less than n .

```
template<typename KERNEL = GaussianKernel, typename X = NUMERIC_X> struct SVM
{
    Vector<X> supportVectors;
    Vector<double> supportCoefficients;
    double bias;
    KERNEL K;
    template<typename DATA> double evalK(LinearProbingHashTable<long long,
        double*& cache, long long i, long long j, DATA const& data)
    {
        long long key = i * data.getSize() + j;
        double* result = cache.find(key);
        if(result) return *result;
        else
        {
            double value = K(data.getX(i), data.getX(j));
            cache.insert(key, value);
            return value;
        }
    }
    int makeY(bool label){return label * 2 - 1;}
    double lowDiff(bool label, double C, double d){return label ? d : d + C;}
    double highDiff(bool label, double C, double d){return label ? C - d : d;}
public:
    template<typename DATA> SVM(DATA const& data, pair<KERNEL, double> const&
        params, int maxRepeats = 10, int maxConst = 10000): K(params.first)
    {
        double C = params.second;
        assert(data.getSize() > 0 && C > 0);
        bias = makeY(data.getY(0)); //just in case have 1 class only
        LinearProbingHashTable<long long, double> cache;
        int n = data.getSize(), maxIters = max(maxConst, n * maxRepeats);
        Vector<double> d(n, 0), g(n);
        for(int k = 0; k < n; ++k) g[k] = makeY(data.getY(k));
        while(maxIters--)
        { //select directions using max violating pair
            int i = -1, j = -1; //i can increase, j can decrease
            for(int k = 0; k < n; ++k)
            { //find max g[i] and min g[j]
                if(highDiff(data.getY(k), C, d[k]) > 0 && (i == -1 || g[k] > g[i])) i = k;
                if(lowDiff(data.getY(k), C, d[k]) > 0 && (j == -1 || g[k] < g[j])) j = k;
            }
            if(i == -1 || j == -1) break;
            bias = (g[i] + g[j])/2; //avb for stability
            //check optimality condition
            double optGap = g[i] - g[j];
```

```

    if(optGap < 0.001) break;
    //compute direction-based gminimum and box bounds
    double denom = evalK(cache, i, i, data) -
        2 * evalK(cache, i, j, data) + evalK(cache, j, j, data),
        step = min(highDiff(data.getY(i), C, d[i]),
        lowDiff(data.getY(j), C, d[j]));
    //shorten step to box bounds if needed, check for numerical
    //error in kernel calculation or duplicate data, if error
    //move points to box bounds
    if(denom > 0) step = min(step, optGap/denom);
    //update support vector coefficients and gradient
    d[i] += step;
    d[j] -= step;
    for(int k = 0; k < n; ++k) g[k] += step *
        (evalK(cache, j, k, data) - evalK(cache, i, k, data));
    //determine support vectors
    for(int k = 0; k < n; ++k) if(abs(d[k]) > defaultPrecEps)
    {
        supportCoefficients.append(d[k]);
        supportVectors.append(data.getX(k));
    }
}
int predict(X const& x) const
{
    double sum = bias;
    for(int i = 0; i < supportVectors.getSize(); ++i)
        sum += supportCoefficients[i] * K(supportVectors[i], x);
    return sum >= 0;
}
};

```

The runtime is $O(\text{the number of iterations} \times n)$. The radial basis kernel effectively averages only nearby margin examples due to exponential weight decrease in distances. A small C results in fewer support vectors because the margin is less accommodating to isolated examples.

For nonlinearly separable data can't bound the VC dimension as a function of the margin, but a customized analysis produces a similar bound. Let the p -margin loss be $L_p(x) = \begin{cases} 1, & \text{if } x < 0 \\ 0, & \text{if } x < p \\ 1-x/p, & \text{otherwise} \end{cases}$. Then have an empirical p -risk R_p on the training data. Theorem (Mohri et al. 2018): Let $b = 0$, and $\text{Tr}(G) =$ the trace of the kernel matrix. Then $\forall p, q > 0$ with probability $\geq 1 - q$, $R(h) \leq R_p + 2\sqrt{\frac{\text{Tr}(G)}{m} \frac{Q}{p}} + 3\sqrt{\frac{\ln(2/q)}{2m}}$.

In particular, $p = 1$ is useful because hinge loss bounds p -loss, and $R_1 \leq \sum \epsilon_i$. This justifies SVM design as trying to both minimize the sum of margin violations and maximize the margin. $b \neq 0$ is only a minor assumption violation in practice (Yuan et al. 2012) and theory because other parameters should make up for it.

In some sense, using kernels is optimal because it allows to express any boundary shape with enough example support, exploiting the manifold assumption.

26.12 Multiclass Nonlinear SVM

Use OVO for its general advantages and because it needs less memory for the kernel cache:

```

template<typename KERNEL = GaussianKernel, typename X = NUMERIC_X>
class MulticlassSVM
{//need buffer for speed
    typedef pair<KERNEL, double> P;
    MulticlassLearner<BufferLearner<SVM<KERNEL, X>,
        InMemoryData<X, int>, P>, P> mcl;
public:
    template<typename DATA> MulticlassSVM(DATA const& data,
        pair<KERNEL, double> const& params): mcl(data, params) {}
    int predict(X const& x) const{return mcl.predict(x);}
};

```

};

A good but slow approach to picking y and C is exponential range grid search (see the “Numerical Optimization” chapter) with cross-validation. All else equal, want small C and large y . Grid points used by LIBSVM are 2^i with odd $i \in [-15, 3]$ for y and $[-5, 15]$ for C , with 110 values to try (Bottou et al. 2007). This works well in practice, but assumes the data is scaled in a way that this range is reasonable. But simple discrete compass search with $\max = 10$ evaluations (see the “Numerical Optimization” chapter) gives similar accuracy and is several times faster based on my experiments.

```

struct NoParamsSVM
{
    MulticlassSVM<> model;
    struct CCSVMFunctor
    {
        typedef Vector<double> PARAMS;
        MulticlassSVM<> model;
        template<typename DATA> CCSVMFunctor(DATA const& data,
            PARAMS const& p):
            model(data, make_pair(GaussianKernel(p[0]), p[1])) {}
        int predict(NUMERIC_X const& x)const{return model.predict(x);}
    };
    template<typename DATA> static pair<GaussianKernel, double>
        gaussianMultiClassSVM(DATA const& data, int CLow = -5,
        int CHigh = 15, int yLow = -15, int yHi = 3)
    {
        Vector<Vector<double>> sets(2);
        for(int i = yLow; i <= yHi; i += 2) sets[0].append(pow(2, i));
        for(int i = CLow; i <= CHigh; i += 2) sets[1].append(pow(2, i));
        Vector<double> best = compassDiscreteMinimize(sets,
            SCVRiskFunctor<CCSVMFunctor, Vector<double>, DATA>(data),
            10);
        return make_pair(GaussianKernel(best[0]), best[1]);
    }
    template<typename DATA> NoParamsSVM(DATA const& data): model(data,
        gaussianMultiClassSVM(data)) {}
    int predict(NUMERIC_X const& x)const{return model.predict(x);}
};

typedef ScaledLearner<NoParamsLearner<NoParamsSVM, intint

```

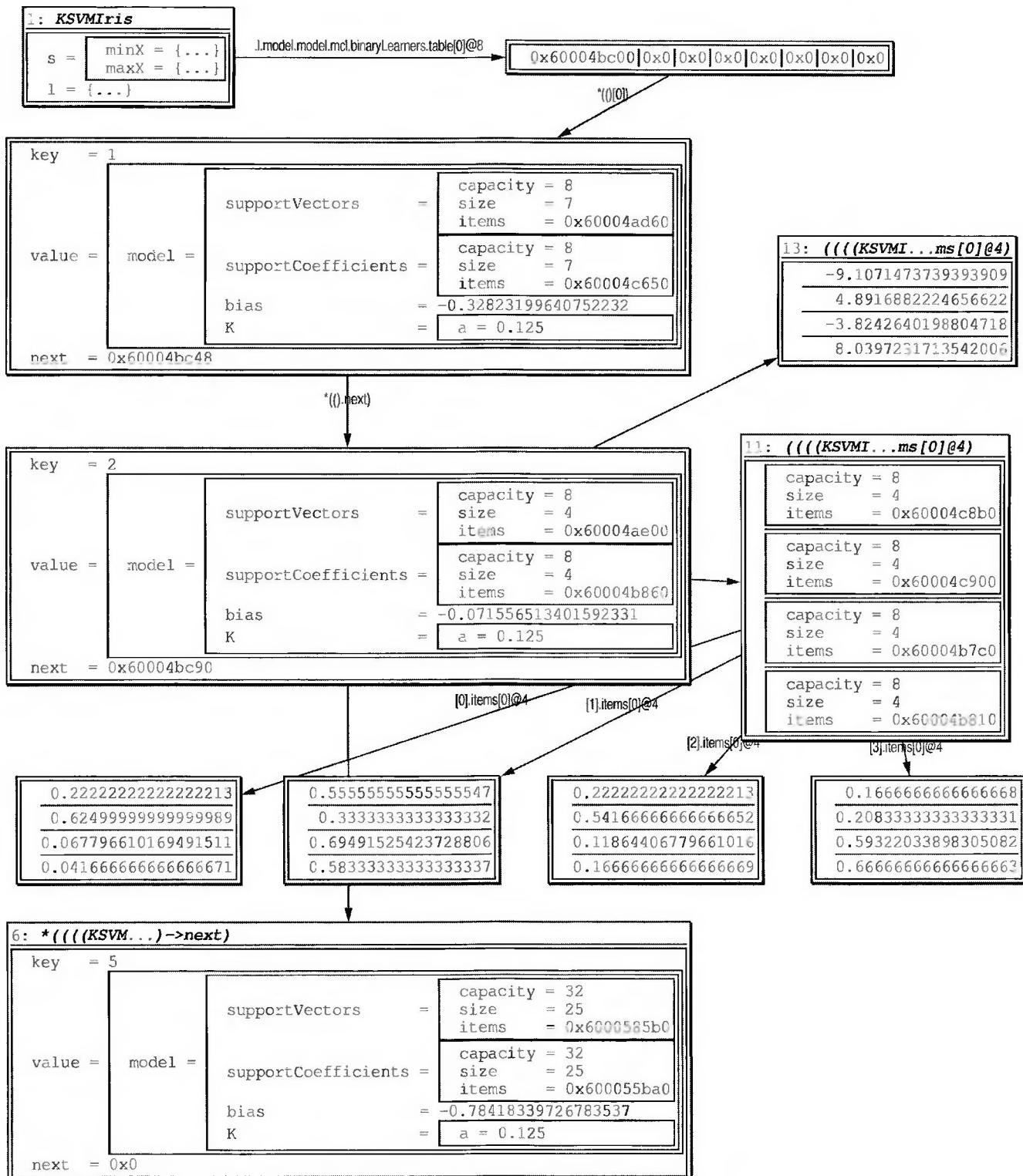


Figure 26.15: Memory layout of a KSVM on the iris data

The authors of LIBSVM recommend scaling features into [0, 1] (Hsu et al. 2010).

SSVM_10n10k_001_seps_c10	SSVM_10n10k_001_seps_r20	GSVM_10n10k_001_seps_c10	MQSVM_10n10k_001_seps_c10
0.940	0.911	0.943	0.920

Figure 26.16: Some performance comparisons for parameter settings and scaling

Kernel SVM is slow with parameter selection, and scales poorly for large n . So its use is restricted to small or medium n .

26.13 Neural Network

A neural network is designed for regression. For classification, predict $\Pr(f(x)=y)$. A **neuron** has an **activation function** $g(x)$ such as the logistic $\frac{1}{1+e^{-x}}$, which normalizes x into a simple range such as (0, 1), a

list of weights $w \forall$ feature f , and a **bias feature** with constant value 1. w form a hyperplane in X . A neuron computes a given point's distance to that hyperplane as $g(\sum w_i f_i)$, which is a distance because $\sum w_i x_i = a$ $\forall a$ is a set of hyperplanes. So a neuron splits X into $a \geq 0$ and $a < 0$.

A network is a DAG of neurons, arranged in **layers** so that inputs to the first layer are features, but to any other outputs of the previous layer. The last layer is the **output layer**. A network is usually fully connected but doesn't have to be.

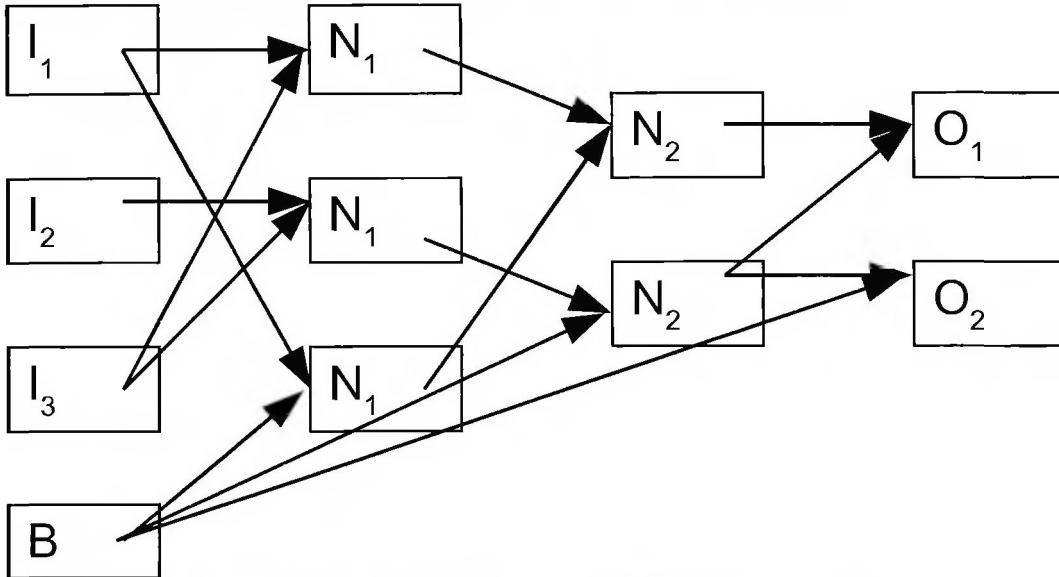


Figure 26.17: A network with 2 hidden layers ("B" is the bias unit)

A **feed-forward network** with L layers is a function defined recursively by $f(x) = f_{k,L-1}$, $f_{k,j+1} = \sum g(w_{kj} f_{kj})$, $f_{k,0} = x[k]$. It usually has a different g , depending on the task:

- Output layer, regression—identity.
- Output layer, classification—logistic. It's symmetric about (0, 0.5).
- Hidden layer—tanh.

Intuitively, each subsequent layer is a transformed feature space that is closer to Y . Effectively, the nonoutput layers find an arrangement of X (in 2D, it's a set of possibly intersecting lines that splits 2D into regions), a particular value of which defines a specific region for x , which is then used by the output layer to make a decision. Typically, but not always, every neuron connects to every neuron of the previous layer and the bias.

Theorem (Hornik 1991; Leshno et al. 1993): A network with a single identity g output and one hidden layer with bounded and nonconstant activation function can approximate arbitrarily well with enough hidden neurons in:

- L_∞ norm—any continuous function on a compact domain, if g is also continuous
- L_p norm— $\forall p$ any function whose L_p distance to the 0 function $< \infty$

To be useful for training, g must be also differentiable. Common sigmoids such as logistic or tanh satisfy this. But the theorem technically doesn't apply to classification because, even with $k = 2$, the wanted function isn't continuous and usually not finitely close to 0. But in practice this doesn't seem to cause problems. The number of hidden neurons needed for getting a specific approximation error is unknown and depends on the problem.

```

class NeuralNetwork
{
    struct Neuron
    {
        Vector<int> sources;
        Vector<double> weights;
        double output, error;
    };
    mutable NUMERIC_X inputs;
    bool isContinuousOutput;
    mutable Vector<Vector<Neuron>> layers;
    double actOutput(double x) const
    { return isContinuousOutput ? x : 1/(1 + exp(-x)); }
    double actInner(double x) const { return tanh(x); }
}
  
```

```

public:
    NeuralNetwork(int D, bool theIsContinuousOutput = false,
                  double theInitialLearningRate = 1): inputs(D), learnedCount(1),
                  initialLearningRate(theInitialLearningRate),
                  isContinuousOutput(theIsContinuousOutput) {}
    void addLayer(int nNeurons){layers.append(Vector<Neuron>(nNeurons));}
    void addConnection(int layer, int from, int to, double weight)
    {
        Vector<Neuron>& last = layers[layer];
        last[from].sources.append(to);
        last[from].weights.append(weight);
    }
};

```

The inputs are propagated to the next layers, and the outputs of the last layer have the result:

```

void propagateInputs(NUMERIC_X const& x) const
{
    inputs = x;
    for(int i = 0; i < layers.getSize(); ++i)
        for(int j = 0; j < layers[i].getSize(); ++j)
        {
            Neuron& n = layers[i][j];
            double sum = n.error = 0;
            for(int k = 0; k < n.sources.getSize(); ++k)
                sum += n.weights[k] * getInput(i, n.sources[k]);
            n.output = i == layers.getSize() - 1 ?
                actOutput(sum) : actInner(sum);
        }
}
double getInput(int layer, int source) const
{
    return source == -1 ? 1 : layer == 0 ?
        inputs[source] : layers[layer - 1][source].output;
}
Vector<double> evaluate(NUMERIC_X const& x) const
{
    propagateInputs(x);
    Vector<double> result;
    for(int i = 0; i < layers.lastItem().getSize(); ++i)
        result.append(layers.lastItem()[i].output);
    return result;
}

```

Learning adjusts weights to minimize the L_2 error of the output neurons from evaluating every training example. This isn't convex, so use local optimization. The simplest, scalable, and effective training method is SGD, with derivatives calculated using **back-propagation**. \forall neuron only its weights contribute to the error, so can use gradient descent. \forall output neuron, error $\epsilon = \frac{1}{2}(g(w_f) - y)^2$. With respect to w , $\epsilon' = tf$ for $t = og'(w, f)$, and output error $o = g(wf) - y$. For neuron j in the layer feeding the output layer, $\epsilon_j = \sum$ errors of neurons that use its output as feature $= \frac{1}{2} \sum (g(w_k f_j) - y_k)^2$. With respect to w , $\epsilon'_j = \sum (g(w_k f_j) - y_k) g'(w_k f_j) w_k[j] g'(w_j f_j) f_j = t_j f_j$ for $o_j = \sum t_k w_k[j]$. Inductively, this works \forall nonoutput neuron, leading to gradient descent update $w_{i+1} = w_i - s_i tf$, where s_i is some step size that satisfies Robbins-Munro conditions (see the "Numerical Optimization" chapter). SGD converges slowly, so usually don't need explicit regularization. Use $[10^5/n]$ or $[10^6/n]$ passes over the data to get reasonable accuracy (about 10^{-3} to 10^{-2} asymptotically).

For $g(x) = \frac{1}{1+e^{-x}}$, $g'(x) = g(x)(1-g(x))$; for tanh, $g'(x) = 1 - g(x)^2$. These relations allow computing the derivatives directly from neuron outputs, avoiding the extra storage needed otherwise.

```

long long learnedCount;
double initialLearningRate;

```

```

double learningRate()
    {return initialLearningRate/pow(learnedCount++, 0.501);}
double actOutputDeriv(double fx) const
    {return isContinuousOutput ? 1 : fx * (1 - fx);}
double actInnerDeriv(double fx) const{return 1 - fx * fx;}
void learn(NUMERIC_X const& x, Vector<double> const& results)
{
    assert(results.getSize() == layers.lastItem().getSize());
    propagateInputs(x);
    Vector<Neuron>& last = layers.lastItem();
    for(int j = 0; j < last.getSize(); ++j) last[j].error =
        last[j].output - results[j];
    double r = learningRate();
    for(int i = layers.getSize() - 1; i >= 0; --i)
        for(int j = 0; j < layers[i].getSize(); ++j)
    {
        Neuron& n = layers[i][j];
        double temp = n.error * (i == layers.getSize() - 1 ?
            actOutputDeriv(n.output) : actInnerDeriv(n.output));
        for(int k = 0; k < n.sources.getSize(); ++k)
            //update weights and prev layer output errors
            int source = n.sources[k];
            if(i > 0 && source != -1)
                layers[i - 1][source].error += n.weights[k] * temp;
                n.weights[k] -= r * temp * getInput(i, source);
    }
}

```

For $k = 2$, only know VC dimension d bounds for networks with particular g . Let W = the number of weights in a network with sigmoid hidden layer activations and threshold output activation. Then (Shalev-Schwartz & Ben-David 2014; Anthony & Bartlett 1999) $O(W^2) < d < O(W^4)$. But these don't consider weight and data sizes and make assumptions on activations. Assuming that each weight takes $O(1)$ bits to represent, a network needs $O(W)$ bits, leading to a better ORM upper bound. Probably \exists better, currently unknown bounds that consider data and weight sizes. Nevertheless, these show that every weight matters. Learning from a single example takes $O(W)$ time.

Applying SGD needs some care (Montavon et al. 2012). A problem is **saturation**, i.e., when ∇x the sums at one or more neurons are too small/large, and $g'(x) \approx 0$. Nevertheless, though for some examples the sums can be huge, resulting in negligible updates, for others they should lead to reasonably large $g'(x)$. A typical classification recipe for structure and parameters of a classical shallow architecture:

- For $k > 2$ use OVO; binary problems have a 0/1 output neuron. A neural network is better at discriminating between two classes than between many. Count the output layer outputs as fractions for the OVO voting, without rounding to 0 or 1. (Alternatively, for OVA \forall class use a 0/1 output neuron with logistic activation, and pick the class corresponding to the highest value neuron. Unlike for SVM, OVO is less efficient by a factor of $O(k)$, but the decision boundaries are simpler, and few hidden neurons suffice.)
 - Use one hidden layer with 5 neurons. This tends to work well experimentally (Fernández-Delgado et al. 2014) and allows online operation. For 5, the nonlinearity is more than with logistic regression and not too much to overfit. Other $2i + 1$ numbers such as 3 or 9 or cross-validation give similar results.
 - Scale inputs to mean 0 and standard deviation 1. This helps keep the sums small.
 - Initialize output neuron and bias weights to 0, and the rest to uniform($-a, a$) samples for $a = \sqrt{3/D}$. Then the latter weights have mean 0 and standard deviation $1/\sqrt{D}$, and the sum mean 0 and standard deviation 1. This makes saturation unlikely and gives different hidden neurons different weights, so that they don't learn the same thing (using 0 for everything causes this because of symmetry). Each hidden unit essentially does a random projection at initialization.
 - For hidden layers, use g symmetric around 0. Most sources recommend $g = \tanh$ —the same logic as for scaling inputs to 0—e.g., with logistic $g(0) = 0.5$, so the sums at the next layer neurons will be $0.5 \times$ the number of inputs, which could saturate immediately.
 - The initial learning rate = 1 seems to work well.

- Use 5 networks, differing only in initial weights, and average their outputs. This is to remove variance and tends to work well experimentally (Fernández-Delgado et al. 2014).

```

class BinaryNN
{
    Vector<NeuralNetwork> nns;
    void setupStructure(int D, int nHidden)
    {
        double a = sqrt(3.0/D);
        for(int l = 0; l < nns.getSize(); ++l)
        {
            NeuralNetwork& nn = nns[l];
            nn.addLayer(nHidden);
            for(int j = 0; j < nHidden; ++j)
                for(int k = -1; k < D; ++k) nn.addConnection(0, j, k,
                    k == -1 ? 0 : GlobalRNG().uniform(-a, a));
            nn.addLayer(1);
            for(int k = -1; k < nHidden; ++k) nn.addConnection(1, 0, k, 0);
        }
    }
    public:
        BinaryNN(int D, int nHidden = 5, int nNns = 5):
            nns(nNns, NeuralNetwork(D)){setupStructure(D, nHidden);}
        template<typename DATA> BinaryNN(DATA const& data, int nHidden = 5, int
            nGoal = 100000, int nNns = 5): nns(nNns, NeuralNetwork(getD(data)))
    {
        int D = getD(data), nRepeats = ceiling(nGoal, data.getSize());
        setupStructure(D, nHidden);
        for(int j = 0; j < nRepeats; ++j)
            for(int i = 0; i < data.getSize(); ++i)
                learn(data.getX(i), data.getY(i));
    }
    void learn(NUMERIC_X const& x, int label)
    {
        for(int l = 0; l < nns.getSize(); ++l)
            nns[l].learn(x, Vector<double>(1, label));
    }
    double evaluate(NUMERIC_X const& x)const
    {
        double result = 0;
        for(int l = 0; l < nns.getSize(); ++l)
            result += nns[l].evaluate(x)[0];
        return result/nns.getSize();
    }
    int predict(NUMERIC_X const& x)const {return evaluate(x) > 0.5;}
};

class MulticlassNN
{
    MulticlassLearner<NoParamsLearner<BinaryNN, intpublic:
    template<typename DATA> MulticlassNN(DATA const& data): model(data) {}
    int predict(NUMERIC_X const& x)const {return model.classifyByProbs(x);}
};

typedef ScaledLearner<NoParamsLearner<MulticlassNN, int, int, EMPTY,
    ScalerMQ> SNN;

```

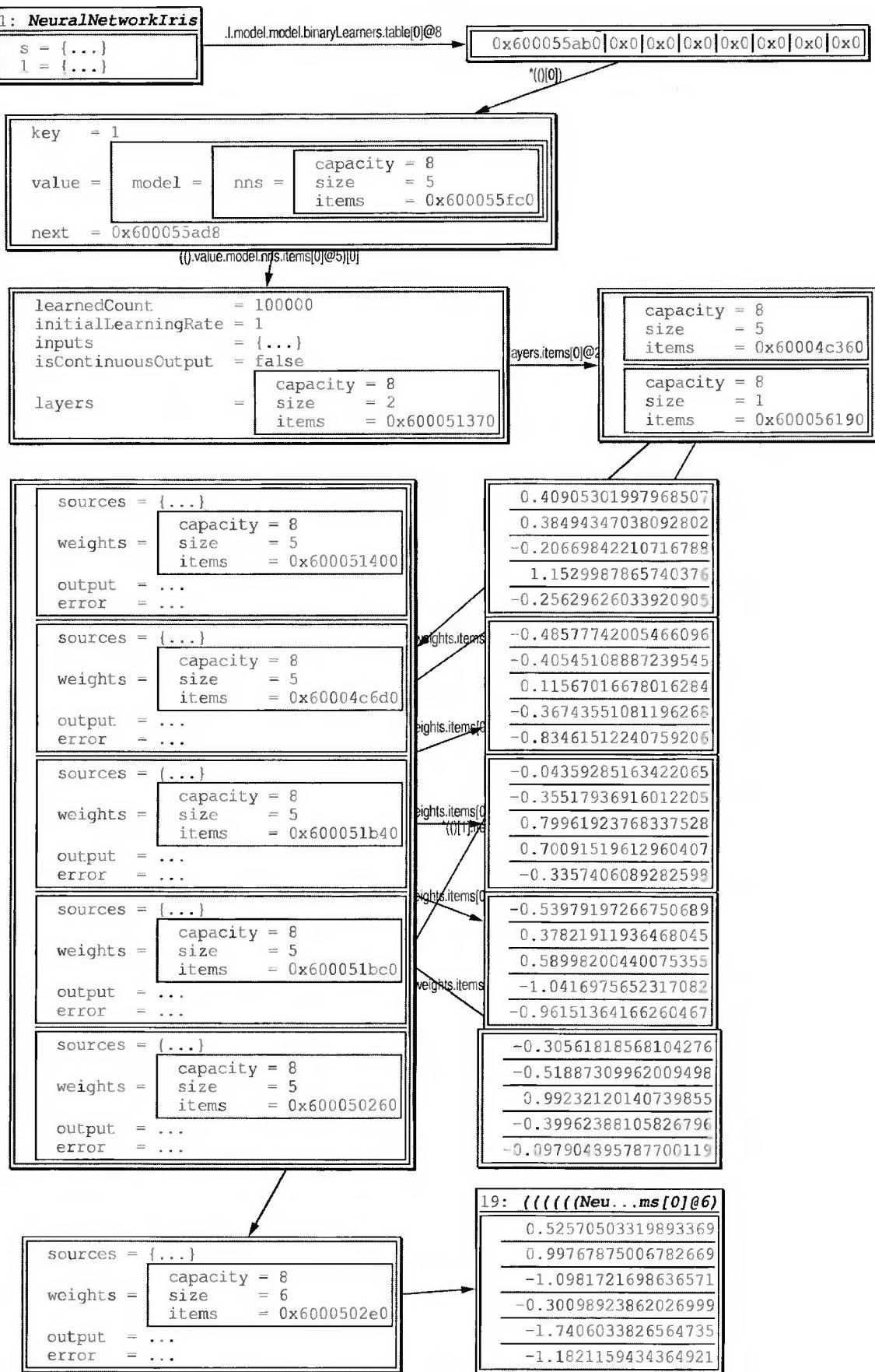


Figure 26.18: Memory layout of a neural network on the iris data

Neural networks have many problems, which haven't been solved by years of research:

- Search (SGD or other methods) easily gets trapped in local minima of arbitrary quality. Averaging several randomly trained networks essentially gives in to the central limit catastrophe (see the "Optimization" chapter), but don't have better alternatives. Can try global optimization methods, but need explicit regularization for them not to overfit—can use the L_2 penalty, which for neural

networks is called **weight decay**. Still, this takes too long, and the results are unlikely to be reproducible. So kernel SVM mostly replaced neural network as the black-box method when need nonlinearity. But in some applications such as game playing, a globally trained neural network can give good results (Mandziuk 2010); nevertheless, other methods might have given better results if used instead.

- Training is slow even with SGD—about $O(k^2D \times \text{the number of examples} \times \text{the number of hidden neurons})$. Essentially don't have room for parameter selection. Some implementations cross-validate decay (Fernández-Delgado et al. 2014; Caruana & Niculescu-Mizil 2006; Caruana et al. 2008), but faster low-precision SGD gives automatic regularization.

Black-box single hidden layer networks generally do worse than top-performing *A* such as SVM or random forest (discussed later), but are useful in specific cases such as online learning. **Top-performing networks use domain knowledge**. E.g., the famous convolution network for digit recognition (Hastie et al. 2009) groups neighboring pixels (more generally, highly correlated features), and years of design went into it. Another domain-specific information is training on slightly rotated images, which improved performance substantially (Montavon et al. 2012).

Deep learning is an advance in neural networks, with two fundamental changes, by the use of (Montavon et al. 2012):

- More than one hidden layer (deep architecture). A single hidden layer may need exponentially many neurons to learn some functions. Having more layers avoids this.
- Unsupervised pretraining, which makes deep architecture training feasible. For a standard network with more than one hidden layer, have **vanishing gradient** (errors sent to lower layers ≈ 0) and too many local minimum problems.
- Use of **ReLU activation function**, which is a trimmed line (technically unbounded but apparently without convergence issues; Bengio et al. 2016). It's currently believed to be better than the classical tanh.

A simple way to use unsupervised data is **auto-encoding**:

- \forall hidden layer attach an extra decoder layer, which tries to reconstruct the input. Can use standard SGD with back-propagation.

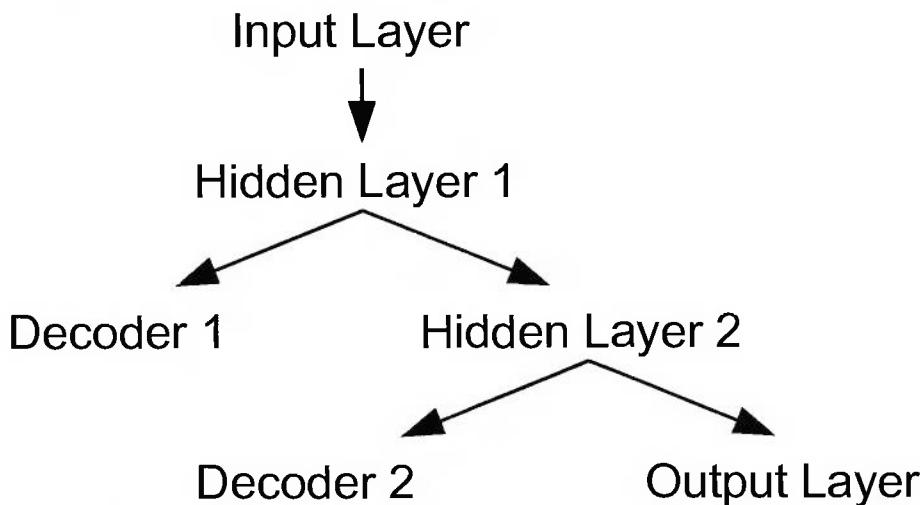


Figure 26.19: Hidden layers with decoders

- Starting from the first layer, train each layer using a decoder as last layer, but without modifying already trained layers. Need to cross-validate the initial learning rate for the decoder units because activation derivatives are no longer bounded.
- Use y to back-propagate as usual.
- A simple architecture is using the same number of hidden neurons (cross-validated) \forall layer, and maybe 5 layers. This is to avoid overfitting, but have many probably better choices (Bengio et al. 2016).
- Due to using unsupervised data, can't decompose into binary problems, so use k output units with OVA. This requires learning more complex boundaries but is faster, and a deep architecture supposedly makes up for this by giving better features.

Deep learning is an active research field and hasn't produced a competitive black-box *A* yet (this is debatable and remains to be experimentally validated). But many application-specific results are state of the art, to the extent that the main inventors received the ACM Turing award. How to setup a deep neural

network to be an effective black box like random forest and why does it not have problems with overfitting are open problems.

26.14 Randomization Ensembles

Combining multiple f may do better than using a single one. The simplest way to do so is by using majority vote, i.e., the most popular class is the answer. **Jury theorem:** If T combined f_i are independent, and for each $\Pr(\text{the chosen class is right} > 0.5)$, as $T \rightarrow \infty$, $\Pr(\text{the majority is wrong}) \rightarrow 0$. With $k > 2$, only need correctness $> 1/k$, assuming no wrong answer is more likely than the correct one. Proof: The probability of the correct answer will increase much faster than that of another answer, which can only increase by chance. \square

But a set of f can't be fully independent because all are trained on the same data. Also, variance of the noninformative f_i will bring significant noise to close decisions by the informative ones; so should make sure that f stay informative.

A **bagged decision tree** is an application of bagging (see the “General Machine Learning” chapter) to get the benefits of randomization. Trees aren't pruned to reduce bias, but the depth restriction is still enforced. **Random forest** improves it by reducing bias at the expense of higher variance by reducing correlation between trees. It makes each tree more random by disallowing a random subset of features from being used for splits; a different subset is sampled \forall node. Usually, f_i with little correlation to each other have little in common.

The number of allowed features a should be small enough to decorrelate the trees but large enough to not weaken each one. The original paper default choice $a = \sqrt{D}$ seems robust. An alternative frequent suggestion $a = \lfloor \lg(D) \rfloor$ gets less correlation, and logical $a = D^{\frac{1}{3}}$ (8 out of 16, 27 out of 81, etc.) more strength. According to my tests the former is worse, and the latter about the same, so no reason to prefer either. Generally, to improve the:

- Diversity of trees—want smaller a
- Strength of each tree—want larger a
- Efficiency—want smaller a and T

Choosing T depends on the problem and is difficult:

- Too large is inefficient in time and memory use
- Too small isn't enough to stabilize predictions and results in a randomized algorithm with high variance

$T = 1000$ is suggested in the original paper and frequently used. It's reasonable to do grid search from this number using approximately factor $\sqrt{10}$ step. Based on the results, 300 seems to be better:

- Prior knowledge is in favor of 1000 due to use; experimentally, using more than some number stops making a difference (Breiman 2001)
- 100 is the smallest with good enough performance
- 100 is close to mediocre performance, and 1000 to unnecessary inefficiency
- The relative standard deviations of all three are close and slightly favor more trees
- Prediction needs $O(T)$ base learner predictions
- Random forest is useful as a black box and, in several cases, with augmentations, so neither efficiency nor variance should be a problem with its default choice
- Using cross-validation on doubling numbers is slow, variance in performance can lead to bad selection, and T , as long as large enough, makes little difference

Bag_1k	RF_Ig_1000	RF_175_1k	RF_05_30	RF_05_100	RF_05_300	RF_05_1k	RF_05_3k
0.941	0.957	0.971	0.953	0.962	0.964	0.967	0.968

Figure 26.20: Some performance comparisons for choosing a and T

Random forest estimates probabilities reasonably well using count proportions. Bagging does too, but only with unstable base A .

1. Pick T (a hardcoded)
2. T times
3. Create a resample of size n from S
4. Learn a decision tree from the resample, randomly selecting features \forall node and not pruning
5. To predict x , classify it using every tree, and find the majority class, or use count proportions to get probabilities

```
class RandomForest
{
```

```

Vector<DecisionTree> forest;
int nClasses;
public:
    template<typename DATA> RandomForest(DATA const& data, int nTrees = 300):
        nClasses(findNClasses(data))
    {
        assert(data.getSize() > 1);
        for(int i = 0; i < nTrees; ++i)
        {
            PermutedData<DATA> resample(data);
            for(int j = 0; j < data.getSize(); ++j)
                resample.addIndex(GlobalRNG().mod(data.getSize()));
            forest.append(DecisionTree(resample, 0, true));
        }
    }
    template <typename ENSEMBLE> static int classifyWork(NUMERIC_X const& x,
        ENSEMBLE const& e, int nClasses)
    {
        Vector<int> counts(nClasses, 0);
        for(int i = 0; i < e.getSize(); ++i) ++counts[e[i].predict(x)];
        return argMax(counts.getArray(), counts.getSize());
    }
    int predict(NUMERIC_X const& x) const
    {
        return classifyWork(x, forest, nClasses);
    }
    Vector<double> classifyProbs(NUMERIC_X const& x) const
    {
        Vector<double> counts(nClasses, 0);
        for(int i = 0; i < forest.getSize(); ++i)
            ++counts[forest[i].predict(x)];
        normalizeProbs(counts);
        return counts;
    }
};

```

One extra feature is an **out-of-bag** risk estimate—i.e., $\forall z_i \in S$, compute y based on the subforest consisting of trees not trained on z_i . This is as accurate as the estimate on an independent test set (Breiman 2001). Because about $1/e$ trees don't include a particular example, and ≈ 200 trees gives good performance, this is best done with > 600 trees.

A theoretical problem is that random forest based on a greedily trained decision tree isn't consistent due to unlikely special cases (Biau et al. 2008). But this doesn't seem to cause problems in practice.

26.15 Online Training

Nearest neighbor, linear SVM, and a neural network are online. But nearest neighbor takes too much memory, and linear SVM and neural network might not have the top performance. Also, from consistency point of view, all except nearest neighbor keep a model of constant complexity with respect to the number of learned examples, whereas top offline algorithms adapt their models.

Online A that need $k = 2$ can also work with $k > 2$. After seeing a new example, adjust k if needed, and new binary learners are created. If some classes are unknown for a long time, the new learners will have less data to train on. E.g., if 0 and 1 appeared first and 2 later, 0 vs 2 and 1 vs 2 would have missed out examples with $y = 0$ and 1. This isn't a problem as long as the data isn't sorted by class.

```

template<typename LEARNER, typename PARAMS = EMPTY, typename X = NUMERIC_X>
class OnlineMulticlassLearner
{
    mutable Treap<int, LEARNER> binaryLearners;
    int nClasses;
    PARAMS p;
    int makeKey(short label1, short label2) const
    {
        return label1 * numeric_limits<short>::max() + label2;
    }
public:
    OnlineMulticlassLearner(PARAMS const& theP = PARAMS(),
        int initialNClasses = 0): nClasses(initialNClasses), p(theP) {}

```

```

void learn(X const& x, int label)
{
    nClasses = max(nClasses, label + 1);
    for(int j = 0; j < nClasses; ++j) if(j != label)
    {
        int key = j < label ? makeKey(j, label) : makeKey(label, j);
        LEARNER* s = binaryLearners.find(key);
        if(!s)
        {
            binaryLearners.insert(key, LEARNER(p));
            s = binaryLearners.find(key);
        }
        s->learn(x, j < label);
    }
}
int predict(X const& x)const
{
    assert(nClasses > 0);
    Vector<int> votes(nClasses, 0);
    for(int j = 0; j < nClasses; ++j)
        for(int k = j + 1; k < nClasses; ++k)
        {
            LEARNER* s = binaryLearners.find(makeKey(j, k));
            if(s) ++votes[s->classify(x) ? k : j];
        }
    return argMax(votes.getArray(), votes.getSize());
}
};

```

This enables implementation of online linear SVM. Though don't know n , using the number of processed examples instead works well. For scaling, studentization tends to work better than range, probably because of better estimation of variance than of range. For testing on offline data, sample from $S 10^6$ examples and learn them online. Random sampling gives much better performance than using the original order for most data sets because it improves balance. Ensuring balance is a critical problem in online learning because A estimates partition boundaries, for which a learnable unit is a strata of examples with different labels.

```

class SRaceLSVM
{
    ScalerMQ s;
    typedef pair<int, double> P;
    RaceLearner<OnlineMulticlassLearner<BinaryLSVM, P>, P> model;
    static Vector<P> makeParams(int D)
    {
        Vector<P> result;
        int lLow = -15, lHigh = 5;
        for(int j = lHigh; j > lLow; j -= 2)
        {
            double l = pow(2, j);
            result.append(P(D, l));
        }
        return result;
    }
public:
    template<typename DATA> SRaceLSVM(DATA const& data):
        model(makeParams(getD(data))), s(getD(data))
    {
        for(int j = 0; j < 1000000; ++j)
        {
            int i = GlobalRNG().mod(data.getSize());
            learn(data.getX(i), data.getY(i));
        }
    }
    SRaceLSVM(int D): model(makeParams(D)), s(D){}
    void learn(NUMERIC_X const& x, int label)

```

```

    {
        s.addSample(x);
        model.learn(s.scale(x), label);
    }
    int predict(NUMERIC_X const& x) const{return model.predict(s.scale(x));}
};

```

The performance strongly depends on having enough examples (≥ 10000) in balanced order. This allows SGD to find margins quickly, and not find distant illusory margins and adjust them back too far or too little. Neural network extends similarly:

```

class SOnlineNN
{
    ScalerMQ s;
    OnlineMulticlassLearner<BinaryNN, int> model;
public:
    template<typename DATA> SOnlineNN(DATA const& data): model(getD(data)), s(getD(data))
    {
        for(int j = 0; j < 1000000; ++j)
        {
            int i = GlobalRNG().mod(data.getSize());
            learn(data.getX(i), data.getY(i));
        }
    }
    SOnlineNN(int D): model(D), s(D) {}
    void learn(NUMERIC_X const& x, int label)
    {
        s.addSample(x);
        model.learn(s.scale(x), label);
    }
    int predict(NUMERIC_X const& x) const{return model.predict(s.scale(x));}
};

```

SraceLSVM_10r6	MqraceLSVM_10r6	SonlineNN_10r6
0.908	0.923	0.926

Figure 26.21: Some performance comparisons

26.16 Boosting

Boosting tries to create a set of f_j such that each next one tries to improve the performance on the examples misclassified by the ensemble so far by giving them more weight. E.g., for regression, boosting is intuitively similar to **Tukey twicing**, i.e., first do regression for y , then again for the errors. Assuming $k = 2$ with $y \in \{-1, 1\}$, want to find base classifiers h_i and weights a_i , such that the combined classifier $\text{sign}(F = \sum a_i h_i)$ has the minimum risk = $\sum L(F_j, y_j)$, where L is some loss function, and F_j is the sum for the example j .

The optimization is NP-hard when L is binary loss, so instead must use a **surrogate loss function** L that:

- \geq binary loss—boosting reduces the latter using an upper bound
- Is convex—then $\sum L_j$ is convex and easy to optimize
- Is monotonically decreasing— F effectively determines the confidence of a correct decision, so examples with large F are safer to classify

Finding F that minimizes L is numerical minimization in the space of functions, and can be done using gradient descent. Starting from no-information $F = 0$, each next h is a step toward the minimum:

1. $F_j = 0 \forall \text{example } j$
2. T times
3. Find the h closest to $d = -dL/dF$
4. Pick a_j analytically or numerically
5. $F_j += a_j h_j \forall \text{example } j$

To find h , minimize one of:

- **AnyBoost**—dot product $h \times d$ —convenient for classification and needs weighing of examples. For L such that $d \geq 0$, which holds for all popular L , this is equivalent to weighing each example j by d_j . To avoid training base A on weighed examples, resample using the distribution specified by the weights. Usually, the resample size = n .

- **Gradient boosting**— L_2 norm $\|h - d\|$ —convenient for regression with L_2 loss and doesn't need weighing. Need to train a regression algorithm to predict real-valued y to minimize L . This is what Tukey twicing does roughly.

AdaBoost is derived by using exponential loss $L(F, y) = e^{-yF}$. If $r = \sum(h \neq y)d$ is the weighted error of h , the optimal $a = \frac{1}{2} \ln\left(\frac{1-r}{r}\right)$ (Schapire & Freund 2012).

Theoretically need a **weak learner** base A (Schapire & Freund 2012), which has error $\epsilon < 0.5$ with probability $\geq 1 - p \forall p > 0$ and large enough (depending on p) n . For the implementation to work, only need $\epsilon < 0.5 \forall$ resample. Theorem (Mohri et al. 2018): Assume:

- The base A is weak, and its G has VC dimension d
- a_i are normalized such that $\sum a_i = 1$
- L_1 margin $m = \min_{y \in \{-1, 1\}} a_i h(x_i) > 0$
- Each h_j achieves $\epsilon_j < \frac{1}{2}$

Then after T rounds and any fixed achieved m , with probability $\geq 1 - p$, $R_f \leq 2^T \prod \sqrt{\epsilon_j^{1-m} (1-\epsilon_j)^{1+m}} + \sqrt{\frac{8d \ln(en/d)}{nm} + \frac{-\ln(p)}{2n}}$.

In particular, for a fixed p and enough rounds, the generalization error is $O(1/\sqrt{n})$ and $\rightarrow 0$. This is impossible for $R_{\text{OB}} > 0$ —the weak A assumption is too strong, so it's more intuitive to assume that the distribution on Z is trimmed so that any undecided label examples have 0 support. Then, observing a noisy example from the real distribution in the:

- Testing data leads to a single error
- Training data leads to a margin-based “error hole” around the example

The second problem is much worse—experiments with AdaBoost show that generalization error ≈ 0.5 for training on very noisy-label data (Schapire & Freund 2012). More generally (Long & Servedio 2010), any boosting algorithm with any convex loss, under the assumption of constant fraction of noisy examples in training data, can be made to perform almost randomly. But most practical data sets aren't such worst case, and AdaBoost performs well (Schapire & Freund 2012). Still, theoretically \exists any middle ground between great performance for $R_{\text{OB}} = 0$ and poor for $R_{\text{OB}} > 0$.

An important conclusion is that the base A should balance having low complexity and strength to get good margins. A heavily pruned decision tree is a natural base A . For base A , generalization and training errors should be close, and performance maximally better than random. The former is more important—e.g., an unpruned decision tree is highly likely to have training error 0 even with resampling. It may seem that decision stumps, used in face recognition, are the best A . But experimentally (Caruana & Mizil 2006) and theoretically (Schapire & Freund 2012) stumps do poorly. Also, for something like the xor problem stumps aren't weak learners. Finally, stumps put more margin around noisy-label examples, leading to larger holes. Ideally, base A should be strong enough to isolate the holes as much as possible. Face recognition uses very specifically engineered stumps, with which boosting happened to work well due to its bias-decreasing property. For vector x , \exists a reason to not use decision trees as base A .

SAMME extends AdaBoost for $k > 2$, needing base learner accuracy $> 1/k$ (Zhu et al. 2009). As for random forest, use $T = 300$. A smaller T is less likely to overfit, which isn't a problem for random forest. The implementation follows gradient descent with exponential loss but makes simplifications:

- Negative gradients d are maintained directly, and data weights and sums F aren't.
- The weights are $\log\left(\frac{1-r}{r}\right) + \log(k-1)$, as for AdaBoost for $k = 2$ (constant factors don't matter), and the weight update formula uses $y \in \{0, 1\}$ to simplify algebra.

If any accuracy $< 1/k$, for robustness the ineffective h is removed. If error = 0, $\log(1)/0 = \text{NaN}$, and resampling will fail \forall subsequent round. A robust solution is to replace the ensemble by h (removing it may remove all effective h for very easy problems).

Certain modifications make sense and improve extendability, but not performance in general:

- Use L that put less weight on misclassified examples—in particular, due to its probabilistic interpretation, many variants use **binomial deviance** (also called **logistic**) loss = $\lg(1+e^{-\text{margin}})$, scaled to bound binary loss (constant factors don't matter though). Because it's almost linear on the left, the worst case weight $\approx 1 \forall$ misclassified example. But experimentally this doesn't help with either noise or overfitting (Schapire & Freund 2012). Deriving analytic weights works only with exponential and logistic losses (Schapire & Freund 2012)—for many others, the only choice is numerical optimization, which is hard to analyze and needs robust implementation. Hinge loss doesn't work

because correctly classified examples over margin = 1 have weight 0, and too many examples are ignored in subsequent rounds.

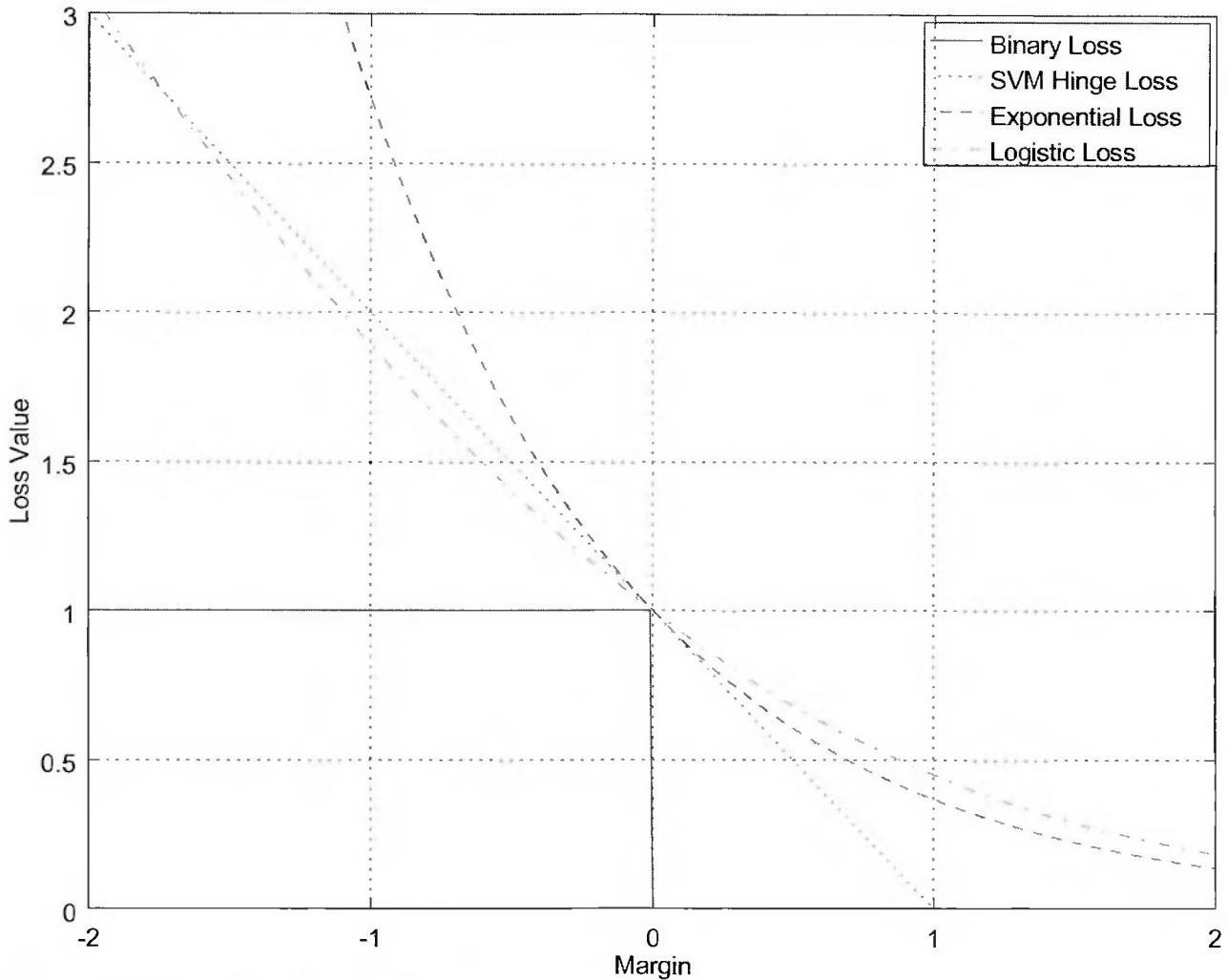


Figure 26.22: Common loss functions

- For $k > 2$, the proper margin to use is $F(\text{the label}) - F(\text{the best false label})$ because this is what classification uses. But many variants use $F(\text{the label}) - \sum_{i \neq \text{the label}} F(i)$, which is too pessimistic (majority instead of plurality) but bounds the proper margin and is easier to analyze.
- Analytic weights may overfit. Small constant weights are suggested instead (Hastie et al. 2009), and Robbins-Munro weights seem better still, guaranteeing convergence of the gradient descent. But such uninformed choice may underfit by over-weighting learners with poor performance.

The resulting algorithm is **RMBoost** (discussed later) $T = 1000$ by default, based on experimental comparisons; 100 performs slightly worse. In general, should use SAMME for boosting, but RMBoost is easily extended to cost-sensitive classification. This is useful because for boosting count proportions don't give accurate probabilities in my experiments, probably because of stronger dependence between base f . The code is presented in the cost learning section later in the chapter.

Intuitively, while random forest decreases variance, boosting decreases both, mostly bias in the first few iterations and mostly variance in the latter ones (Schapire et al. 2012).

26.17 Large-scale Learning

Many algorithms are naturally parallelizable, e.g.:

- Cross-validation—the folds and tested parameters are independent
- Nearest neighbor—keep several indices, and combine the answers
- Random forest—the trees are independent
- Multiclass transform—the binary learners are independent

But many aren't:

- Boosted learners depend on each other
- Can't train binary SVM trained in parallel

26.18 Cost-sensitive Learning

A general problem is considering costs of errors. E.g., for weather prediction, the loss of carrying an umbrella < the loss of not having one during a storm, but f can say *no umbrella needed* if its $\text{Pr}(\text{rain}) = 49\%$. Also, for a secure facility, incorrectly denying entry is much cheaper than letting in someone who shouldn't be.

With costs, right decisions have cost 0 or some negative cost profit value, and wrong ones cost C (the predicted class, the actual class) for some cost matrix C , which assigns a cost \forall confusion matrix cell. If every error is equally costly, all costs = 1. Usually C is normalized:

- The diagonal = 0, i.e., a correct answer costs nothing.
- All other entries $\in [0, 1]$. This ensures no range effect for algorithms that are affected by it. Multiplying C by a scalar leads to an equivalent problem.

```
void scaleCostMatrix(Matrix<double>& cost)
{
    double maxCost = 0;
    for(int r = 0; r < cost.getRows(); ++r)
        for(int c = 0; c < cost.getRows(); ++c)
            maxCost = max(maxCost, cost(r, c));
    cost *= 1/maxCost;
}
```

Theorem (Margineantu 2001): Adding a constant to any column of C leads to an equivalent problem. This makes normalization simple for negative-cost diagonals—add -the diagonal value to every corresponding column.

Can calculate cost risk for a model using cost and confusion matrices. Usually use **cost risk**, with the corresponding confidence bounds.

```
double evalConfusionCost(Matrix<int> const& confusion,
    Matrix<double> const& cost)
{
    int k = confusion.rows, total = 0;
    assert(k == confusion.columns && k == cost.rows && k == cost.columns);
    double sum = 0;
    for(int r = 0; r < k; ++r)
        for(int c = 0; c < k; ++c)
        {
            total += confusion(r, c);
            sum += confusion(r, c) * cost(r, c);
        }
    return sum/total;
}
```

Some solutions for minimizing cost risk:

- Ignore costs—acceptable for highly accurate A and without huge cost differences, because mistakes are rare and, due to normalization, cost risk \leq binary risk \times maximum cost of any mistake.
- Use A that output probabilities. This is particularly convenient with random forest, where probability estimates are reasonably accurate. The idea of other methods is to reduce estimation error by adjusting margins to respect costs.

```
template<typename LEARNER = RandomForest> class CostLearner
{
    Matrix<double> cost;
    LEARNER model;
public:
    template<typename DATA> CostLearner(DATA const& data,
        Matrix<double> const& costMatrix): model(data), cost(costMatrix) {}
    int predict(Numeric_X const& x) const
        {return costClassify(model.classifyProbs(x), cost);}
};
```

- Change A individually to take costs into account. Want to replace binary risk by cost risk in all decisions, including parameter choice by cross-validation and internal objective optimization or greedy decision making. This is sometimes clumsy—e.g., for decision tree \exists cost-weighted entropy.
- Use generic methods to make any A take costs into account. A simple solution is resampling, e.g., picking an example with weight proportional to its expected cost. This works well for $k = 2$. **Cost**

folk theorem (Zadrozny et al. 2003): For $k = 2$, $\forall f$ the expected risk from classifying with costs is equivalent to the expected risk from classifying without costs on samples from the cost-proportional distribution. So can run A on samples from the latter, obtained by resampling. For $k > 2$ this breaks down because costs depends on unknown predicted labels. A heuristic trick is using the average cost (Marginantu 2001).

```
template<typename LEARNER, typename PARAMS = EMPTY,
         typename X = NUMERIC_X> class AveCostLearner
{
    LEARNER model;
    template<typename DATA> static Vector<double> findWeights(
        DATA const& data, Matrix<double> const& costMatrix)
    { // init with average weights
        int k = costMatrix.getRows(), n = data.getSize();
        assert(k > 1 && k == findNClasses(data));
        Vector<double> classWeights(k), result(n);
        for(int i = 0; i < k; ++i)
            for(int j = 0; j < k; ++j)
                classWeights[i] += costMatrix(i, j);
        for(int i = 0; i < n; ++i) result[i] = classWeights[data.getY(i)];
        normalizeProbs(result);
        return result;
    }
public:
    template<typename DATA> AveCostLearner(DATA const& data,
                                              Matrix<double> const& costMatrix, PARAMS const& p = PARAMS()):
        model(data, findWeights(data, costMatrix), p) {}
    int predict(X const& x) const{return model.predict(x);}
};
```

A simple way to create a weighted A is using a small weight-based bagging ensemble of size 5–15. The base A need no changes because ensembles are easily tweaked to support weights directly. Bagging-based resampling is better than classical oversampling, which generates too much data.

```
template<typename LEARNER, typename PARAMS = EMPTY>
class WeightedBaggedLearner
{
    Vector<LEARNER> models;
    int nClasses;
public:
    template<typename DATA> WeightedBaggedLearner(DATA const& data,
                                                 Vector<double> weights, PARAMS const& p = PARAMS(), int nBags = 15):
        nClasses(findNClasses(data))
    {
        assert(data.getSize() > 1);
        AliasMethod sampler(weights);
        for(int i = 0; i < nBags; ++i)
        {
            PermutedData<DATA> resample(data);
            for(int j = 0; j < data.getSize(); ++j)
                resample.addIndex(sampler.next());
            models.append(LEARNER(resample, p));
        }
    }
    int predict(NUMERIC_X const& x) const
    { return RandomForest::classifyWork(x, models, nClasses); }
};
```

Boosting is naturally adapted to handle costs because it uses feedback in subsequent rounds. SAMME and other analytic solvers don't apply because the formulas don't extend to costs, but can modify numerical solvers such as RMBoost. Instead of bounding binary loss by logistic, bound cost-adjusted loss by cost-adjusted logistic. E.g., for RMBoost, the adjusted gradient = the logistic gradient $\times C(\text{the actual class, the best wrong class})$.

```
Matrix<double> getEqualCostMatrix(int nClasses)
{
```

```

Matrix<double> result(nClasses, nClasses);
for(int i = 0; i < nClasses; ++i)
    for(int j = 0; j < nClasses; ++j) if(i != j) result(i, j) = 1;
return result;
}
template<typename LEARNER = NoParamsLearner<DecisionTree, int>,
         typename PARAMS = EMPTY, typename X = NUMERIC_X> class RMBoost
{
    Vector<LEARNER> classifiers;
    int nClasses;
    struct BinomialLoss
    {
        Vector<Vector<double>> F;
        BinomialLoss(int n, int nClasses): F(n, Vector<double>(nClasses, 0))
        {}
        int findBestFalse(int i, int label)
        {
            double temp = F[i][label];
            F[i][label] = -numeric_limits<double>::infinity();
            double result = argMax(F[i].getArray(), F[i].getSize());
            F[i][label] = temp;
            return result;
        }
        double getNegGrad(int i, int label, Matrix<double> const& costMatrix)
        {
            int bestFalseLabel = findBestFalse(i, label);
            double margin = F[i][label] - F[i][bestFalseLabel];
            return costMatrix(label, bestFalseLabel)/(exp(margin) + 1);
        }
    };
public:
    template<typename DATA> RMBoost(DATA const& data, Matrix<double>
        costMatrix = Matrix<double>(1, 1), PARAMS const& p = PARAMS(),
        int nClassifiers = 100): nClasses(findNClasses(data))
    {//initial weights are based on ave cost
        if(costMatrix.getRows() != nClasses)
            costMatrix = getEqualCostMatrix(nClasses);
        int n = data.getSize();
        assert(n > 0 && nClassifiers > 0);
        BinomialLoss l(n, nClasses);
        Vector<double> dataWeights(n), classWeights(nClasses);
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j)
                classWeights[i] += costMatrix(i, j);
        for(int i = 0; i < n; ++i)
            dataWeights[i] = classWeights[data.getY(i)];
        for(int i = 0; i < nClassifiers; ++i)
        {
            normalizeProbs(dataWeights);
            AliasMethod sampler(dataWeights);
            PermutedData<DATA> resample(data);
            for(int j = 0; j < n; ++j) resample.addIndex(sampler.next());
            classifiers.append(LEARNER(resample, p));
            for(int j = 0; j < n; ++j)
            {
                l.F[j][classifiers.lastItem().predict(data.getX(j))] +=
                    RMRate(i);
                dataWeights[j] = l.getNegGrad(j, data.getY(j), costMatrix);
            }
        }
    }
    int predict(X const& x) const
    {

```

```

        Vector<double> counts(nClasses, 0);
        for(int i = 0; i < classifiers.getSize(); ++i)
            counts[classifiers[i].predict(x)] += RMRate(i);
        return argMax(counts.getArray(), counts.getSize());
    }
};

```

Since costs $\in [0, 1]$, can define **cost accuracy** = 1 - cost risk. These are curved and averaged across several data sets as regular accuracies.

RF	RPMIC	RMBDT	SVM	SVM_Ave15	SVM_RMB15
0.963	0.999	0.969	0.964	0.981	0.990

Figure 26.23: Performance comparison on C generated deterministically by making every nondiagonal entry either 0.01 or 1

So for vector X , random forest with probability output seems to be the method of choice. Otherwise, boosted SVM ensemble of small size 15 does well. The implementation is for vector X for simplicity.

```

class BoostedCostSVM
{
    RMBoost<MulticlassSVM>, pair<GaussianKernel, double> > model;
public:
    template<typename DATA> BoostedCostSVM(DATA const& data,
        Matrix<double> const& cost = Matrix<double>(1, 1)):
        model(data, cost, NoParamsSVM::gaussianMultiClassSVM(data), 15) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};

typedef ScaledLearner<BoostedCostSVM, int, Matrix<double> > SBoostedCostSVM;

```

Boosted SVM can't be expected to do better than SVM for noncost learning though. For $k = 2$, an alternative to cost boosting is the average cost method, which performs well with resampling:

```

class AveCostSVM
{
    typedef pair<GaussianKernel, double> P;
    AveCostLearner<WeightedBaggedLearner<MulticlassSVM>, P>, P> model;
public:
    template<typename DATA> AveCostSVM(DATA const& data,
        Matrix<double> const & cost = Matrix<double>(1, 1)):
        model(data, cost, NoParamsSVM::gaussianMultiClassSVM(data)) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};

typedef ScaledLearner<AveCostSVM, int, Matrix<double> > SAveCostSVM;

```

Despite appeal, generic methods change the data in some way, which can be problematic. E.g., re-sampling duplicates some examples, potentially forcing A to pay too much attention to nonrepresentative ones. Using probabilities as in random forest can be problematic if they are inaccurate. So A -specific modifications will have the advantage of least estimation error, but changing each A to handle costs is time-consuming.

26.19 Imbalanced Learning

In a given data set, some classes may appear more often than others. This can be by natural data distribution or bias in selection because rare class instances are more costly to get. E.g., for weather prediction, on most days it doesn't rain, so always predicting no rain is quite accurate.

Can have two main types of imbalance:

- Global—the iid example assumption is violated, causing unnatural imbalance.
- Local— X has many patches of few examples, which are surrounded by more numerous examples of different classes. This can happen even if examples are iid and for the majority class; just need a complex enough distribution.

In either case, classification will have estimation error by pushing the margins toward the smaller group, increasing generalization error. The best solution to both problems is getting more iid data if possible.

Can remedy global imbalance to some degree. Want to recover as much as possible of the risk due to imbalance (Prati et al. 2014). Ignoring imbalance during training is reasonable for mildly imbalanced data sets, i.e., when the example ratio is on the order of 2-to-1. It's usually hard to tell exactly when imbalance is mild. Also, it's the difference between S and the actual data distribution that counts, not between S and all-

equal-class-distribution. E.g., postal envelop digits come from zip codes, where some digits are less likely than others due to population and zip code distribution, even though the data set is reasonably balanced. Global imbalance is usually a problem only if it causes local imbalance. E.g., if have 1000 or so examples in the minority class, even 1000-to-1 imbalance might not affect good A because for a simple enough distribution all patches will have enough support. For intuition about this, consider k -NN consistency requirements. It may also be helpful to compare class counts before learning. Some A such as boosting and SVM seem naturally resistant to local imbalance by trying to impose margins around all small-example groups.

A simple solution is reducing to cost-sensitive learning. By the cost folk theorem, the risk on samples from a balanced distribution = the risk learning from the original one with $\text{cost}(\text{predicted}, \text{actual}) = (\text{the number of instances of the predicted class}/\text{the number of instances of the actual class})$. But, though costs are known without estimation error, costs from class proportions are likely to not be reflective of the distribution from which they were sampled, unless n is large. Reducing tends to do poorly (Prati et al. 2014). By the same reasoning, changing A individually to take class distribution into account can put too much emphasis on it, though this should improve performance on balanced accuracy if the test data has similar imbalance proportions.

Direct resampling is more robust in a sense that various A don't blindly trust the resample proportions but learn as much as needed from them. Want to have each class is equally represented, i.e., with equal total weight \forall class.

```
template<typename DATA>
Vector<double> findImbalanceWeights(DATA const& data)
{
    int n = data.getSize(), properK = 0, nClasses = findNClasses(data);
    Vector<double> counts(nClasses);
    for(int i = 0; i < n; ++i) ++counts[data.getY(i)];
    for(int i = 0; i < nClasses; ++i) if(counts[i] > 0) ++properK;
    Vector<double> dataWeights(n, 0);
    for(int i = 0; i < data.getSize(); ++i)
        dataWeights[i] = 1.0/properK/counts[data.getY(i)];
    return dataWeights;
}
```

Weighted bagging allows this for other base learners such as SVM:

```
class ImbalanceSVM
{
    WeightedBaggedLearner<MulticlassSVM>,
    pair<GaussianKernel, double> model;
public:
    template<typename DATA> ImbalanceSVM(DATA const& data): model(data,
        findImbalanceWeights(data), NoParamsSVM::gaussianMultiClassSVM(data)) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};
typedef ScaledLearner<NoParamsLearner<ImbalanceSVM, int>, int> SImbSVM;
```

Random forest extends directly—change its bootstrap to use the alias method, and sample as in boosting:

```
class WeightedRF
{
    Vector<DecisionTree> forest;
    int nClasses;
public:
    template<typename DATA> WeightedRF(DATA const& data, Vector<double> const
        & weights, int nTrees = 300): nClasses(findNClasses(data))
    {
        assert(data.getSize() > 1);
        AliasMethod sampler(weights);
        for(int i = 0; i < nTrees; ++i)
        {
            PermutedData<DATA> resample(data);
            for(int j = 0; j < data.getSize(); ++j)
                resample.addIndex(sampler.next());
            forest.append(DecisionTree(resample, 0, true));
        }
    }
};
```

```

    }
    int predict(NUMERIC_X const& x) const
        {return RandomForest::classifyWork(x, forest, nClasses);}
};

class ImbalanceRF
{
    WeightedRF model;
public:
    template<typename DATA> ImbalanceRF(DATA const& data, int nTrees = 300):
        model(data, findImbalanceWeights(data), nTrees) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};

```

Resampling can introduce estimation error through duplication of examples, so it's an imperfect but arguably the best available solution for strong imbalance.

RF_05_300	SSVM_10n10k_001_seps_c10	ImbRF_300	ImbSVM
0.964	0.940	0.971	0.946

Figure 26.24: Some performance comparisons

But resampling is not a substitute for good iid data—learning is as good as the data, and garbage in = garbage out.

For online A , resampling for imbalance is problematic because can't even reasonably estimate class proportions.

Solving local imbalance is essentially hopeless—at best, a clever A can put reasonable margins around small groups, but need enough examples to do so, otherwise it's best to consider small groups as noise.

26.20 Feature Selection

For the first attempt, a convenient strategy is using wrapper search. Random forest is probably the best base A due its high accuracy and speed. It has some variance in performance, but this shouldn't be a problem.

```

template<typename SUBSET_LEARNER = RandomForest> struct SmartFSLearner
{
    typedef FeatureSubsetLearner<SUBSET_LEARNER> MODEL;
    MODEL model;
public:
    template<typename DATA> SmartFSLearner(DATA const& data, int limit = 20):
        model(data, selectFeaturesSmart(SCVRiskFunctor<MODEL, Bitset<>, DATA>(
            data), getD(data), limit)) {}
    int predict(NUMERIC_X const& x) const{return model.predict(x);}
};

```

Based on my experiments with several data sets, 44% of features are selected on average, with 0.938 balanced accuracy, which is reasonable. For the iris data, only feature 3 is selected.

For small D and n , SVM can also be useful because is too gets high accuracy and doesn't have variance, though is slower. Can try to speed it up by selecting parameters once \forall runs, but this can spoil search if different subsets need different parameters.

Among embedded methods, decision tree and linear SVM are the most useful. E.g., linear SVM weights, perhaps multiplied by the average feature value, decide relative influence of features on the result of the linear combination; so features with larger influence should be more important. Also, because adding features to a linearly separable problem doesn't improve accuracy, for data perfectly classified by linear SVM, can take out features until no longer linearly separable.

26.21 Comparing Classifiers

The main experimental studies are Caruana & Niculescu-Mizil (2006), Caruana et al. (2008), and Fernández-Delgado et al. (2014; particularly comprehensive). In the last two, random forest is best, and in the first one it's second best. SVM and neural network also do well, but not as much in all three. Decision tree, naive Bayes, and k -NN aren't competitive. Others, which also have no desirable special properties, have no use cases.

Below is a simpler comparison on several UCI data sets. Some are already partitioned into training and testing. Those that aren't have been partitioned using stratified 20% holdout.

sMeanNN	BRF_SVM	SKNN_ oddlg	SSVM_10n10k_001_seps_c10
0.842	0.973	0.910	0.940
SLSVMsgd100kcd100	mqk2nn	DT-z1 50	RF 05 300
0.917	0.927	0.871	0.964
NumericalBayesE			
0.820			

The results here also suggest to expect best performance from random forest and SVM, with neural network a little behind. Choosing the best of them gives even better performance:

```

template<typename X, typename Y> struct LearnerInterface
{
    virtual Y predict(X const& x) const = 0;
    virtual LearnerInterface* clone() const = 0;
};

template<typename LEARNER, typename X, typename Y>
struct TypeFreeLearner: public LearnerInterface<X, Y>
{
    LEARNER model;
    template<typename DATA> TypeFreeLearner(DATA const& data): model(data) {}
    Y predict(X const& x) const{return model.predict(x);}
    LearnerInterface<X, Y>* clone() const{return new TypeFreeLearner(*this);}
};

template<typename Y, typename X = NUMERIC_X>
class BestCombiner
{
    LearnerInterface<X, Y>* model;
    double risk;
public:
    BestCombiner(): model(0) {}
    BestCombiner(BestCombiner const& rhs): model(rhs.model->clone()) {}
    BestCombiner& operator=(BestCombiner const& rhs)
        {return genericAssign(*this, rhs);}
    template<typename LEARNER, typename DATA, typename RISK_FUNCTOR>
        void addNoParamsClassifier(DATA const& data, RISK_FUNCTOR const& r)
    {
        double riskNew = r EMPTY();
        if(!model || riskNew < risk)
        {
            delete model;
            model = new TypeFreeLearner<LEARNER, X, Y>(data);
            risk = riskNew;
        }
    }
    Y predict(X const& x) const{assert(model); return model->predict(x);}
    ~BestCombiner(){delete model;}
};
class SimpleBestCombiner
{
    BestCombiner<int> c;
public:
    template<typename DATA> SimpleBestCombiner(DATA const& data)
    {
        c.addNoParamsClassifier<RandomForest>(data, SCVRiskFunctor<
            NoParamsLearner<RandomForest, int>, EMPTY, DATA>(data));
        c.addNoParamsClassifier<SSVM>(data, SCVRiskFunctor<
            NoParamsLearner<SSVM, int>, EMPTY, DATA>(data));
    }
    int predict(NUMERIC_X const& x) const{return c.predict(x);}
};

```

An interesting result is that combining only random forest and SVM gave better average curved balanced accuracy of 0.971, but it seems more prudent to compare more models for a new task. E.g., could include LSVM, but for general use the best selection policy needs an extensive study with many data sets.

The above comparisons aren't the best possible—note that compare algorithms, so using cross-validation

tion instead of hold-out would reduce variance. But it would take several times longer and so wasn't done here.

Other A are generally useful only for special cases, even though \exists data sets for which they have top performance:

- Nearest neighbor—for unusual distance functions where SVM doesn't scale or want quick exploration
- Naive Bayes—for large-scale categorical data where features are reasonably independent
- Linear SVM—for large-scale data

One dilemma is whether to try to make sense of the data by learning understandable patterns, or deploy classifiers that "magically" do well. Performance can be acceptable/unacceptable and understanding good/bad. A problematic outcome is bad understanding of unacceptable performance. Usually, understandable models such as small decision trees or linear SVM are so only for very small D , which is the case for many applications where need understandably, such as deciding to grant a loan. A reasonable strategy for a new data set:

1. If X isn't vector, use SVM with a specialized kernel or k -NN with a specialized distance function. Prefer SVM due to its sparser model and better performance, but k -NN is often the first quick-to-setup A to run.
2. If n is too large, use LSVM for numeric data and naive Bayes for categorical.
3. Try a highly pruned decision tree, then LSVM. Don't need others if the performance of either is good enough because the results give insight into the data. Decision tree is essentially the only fast A with $O(n^2)$ resource use, so for exploratory analysis it's the best method.
4. Try effective black-box learners: the best among random forest, SVM, and neural network, or just random forest due to its speed. The latter is preferable due to efficiency and may work better when n is too small (<100 maybe) to pick the best out of several A . Trying a few simpler things first on the same data set is technically multiple testing, but with a reduced effect because predictor clarity is the main rejection criteria.
5. If performance isn't good enough, look at the data and try to find out why learning fails. Probably need to help A by doing feature engineering, or augment it with domain-specific logic such as a known good neural network structure.
6. If the above still isn't good enough, research domain-specific methods for the considered domain and related ones. \exists many other specialized A , useful for specific domains but not in general, that aren't discussed in this chapter. But it's likely that the data doesn't have any useful information about the labels.

26.22 Implementation Notes

Every implementation needed substantial research and parameter setting, despite being based on a simple algorithms discussed in many textbooks. Each has something new:

- Confusion matrix—I choose not to calculate confidence intervals on class-specific metric
- Decision tree—the pruning criteria and the depth restriction are original, and I enforce binary nodes
- k -NN—the default k calculation is original
- Naive Bayes—allowing everything to be online is original
- SVM—the optimization algorithm based on discrete compass search is original and seems to work well because of a good starting point
- Random forest—the parameter selection is original
- Boosting—the numerical boosting implementation's use for cost-sensitive learning is original

The combination of random forest and SVM is perhaps most questionable. I would probably only use random forest and a single algorithm on any vector data.

26.23 Comments

For performance evaluation in information retrieval recall and precision have been used for $k = 2$ and computed only for examples with $y = 1$, which presumably is the class of interest. So these metrics may be thought of as losing too much information if both classes are important.

Other somewhat popular performance metrics are application-specific (Witten et al. 2016):

- **F-measure**—defined for a class as the harmonic mean of precision and recall, out of which it attempts to produce a single combined metric. But it makes sense only for problems with $k = 2$

where one class is important and the other one isn't. E.g., for online search, only the relevant results matter and want to neither miss some nor get irrelevant ones.

- **Area under an ROC curve (AUC)**—applies to binary classifiers. Tries to capture performance over all possible cost matrices (Witten et al. 2016) and is sometimes popular among researchers. But for a single cost matrix reduces to something like F -measure, just less intuitive. Calculating it directly makes sense only for probabilistic binary classifiers. For others can use simulation to learn with costs with randomly picked cost matrices, which works well for $k = 2$. Toolkits such as Weka compute AUC automatically, but its advantage, if any, over balanced accuracy in unclear, and the calculation complexity is substantial. Also unclear how to extend to $k > 2$.
- **Kappa** is technically a measure of agreement giving values $\in [-1, 1]$ but in this context measures adjusted accuracy by not counting accuracy on examples expected to be correctly classified by chance. E.g., any f with preference for more represented classes will have artificially high accuracy by chance, and kappa can detect this. Comparison on kappa is similar to that on accuracy, except kappa penalizes preference for more represented classes. Generally, performance over a baseline method is useful because it determines if a model is useful; e.g., any weather prediction model should outperform the historical average and yesterday's weather. Kappa isn't performance over an uninformed A that always returns the majority class, but over an uninformed specific A , making it similar to balanced accuracy, but much less intuitive. It's also hard to extend to learning with costs, but balanced accuracy is easy.

For kappa and other metrics that don't come with confidence, the only way to get it is by resampling test set results, creating the corresponding confidence matrices, and using bootstrap on the resulting metrics.

Many methods can give heuristic probabilities, but with issues. E.g., naive Bayes needs to normalize, but the model itself is usually simplistic. Logistic regression (discussed later) also naturally outputs probabilities, but those are accurate only assuming its model is correct. For a decision tree, a common solution is to report leaf counts, but the resulting probabilities aren't accurate if leaves are small. Looks like random forest's probability output is the only reasonably accurate one.

The analog of VC dimension for $k > 2$ is **Natarajan dimension** (Shalev-Schwartz & Ben-David 2014). But concentration bounds based on Rademacher complexity haven't yet been developed for it, and doing so will bring little improvement in conceptual understanding.

For naive Bayes, instead of binning, can use the prior-posterior methods of Bayesian statistics to model numeric data. But simple distributions such as normal are specified by few parameters and can't model multipeak distributions. Counts of categorical features with enough categories can, so binning tends to work better (Witten et al. 2016).

For k -NN, other natural extensions aren't useful:

- Weighing found neighbors by a function of distance—intuitively, should weigh closer neighbors more, maybe using $w_i = \frac{1}{1+d(x, neighbor_i)}$, but in practice this partially cancels out the smoothing effect of $k > 1$. Asymptotically, not weighing is optimal (Devroye et al. 1996). Related **Kriging** for regression takes the average of all neighbors weighed by distance, and is successful, but apparently only for regression in small D , e.g., for spatial statistics.
- Removing some instances—this seems to potentially cure the memory problem. But ! \exists a good way to do this because only instances well inside the margins and noisy mislabeled ones aren't useful. Asymptotically, instance selection can't lower risk (Devroye et al. 1996). One reasonable method, **IB3**, does poorly on tests (Fernández-Delgado et al. 2014). Other heuristics are better in accuracy, memory saving, or performance (Garcia et al. 2014), but not enough to justify use, mostly due to inefficiency—usually $O(n^2)$ runtime. Theoretically useful results (Gottlieb et al. 2014) aren't practical. It appears that only SVM does instance selection properly when selecting support vectors.

Locality-sensitive hashing (LSH) (for details see Andoni & Indyk 2008) allows finding nearest neighbors within a fixed distance R with high probability and more efficiently than by using exact search data structures by calculating hashes that are likely to map close x to the same bucket. Though very successful in some applications (mostly not related to classification, and using specialized hash functions), a generally useful implementation with Euclidean distance is problematic. A basic attempt is saving memory by storing only y and hashes, and returning the majority of found labels or defaulting to nearest mean if none is found. R and parameter " k " are found by cross-validation, and " l " set to 10. In my tests, this is only slightly more accurate than nearest mean. Any improvement will need much more memory and isn't much better than the usual k -NN query, for which VP or k -d tree are very fast on typical data despite the poor worst case.

For decision trees, **Gini index** (Aggarwal 2014) is an alternative to entropy. Intuitively, it represents probability that an example will be misclassified and has similar performance.

Can cut a $\lg(n)$ factor in decision tree construction runtime by presorting the examples by each attribute (Witten et al. 2016), but the extra memory use and code complexity aren't worth it. Some decision tree variants allow multiway trees, where a feature is split only once and not considered again. But split points may be hard to determine because selection based on information gain becomes biased in favor of features with a larger number of splits. Corrections such as **gain ratio** (Aggarwal 2014) address this somewhat, but not completely. Also, the implementation is more complicated.

Many decision tree pruning methods have been proposed. Experimentally, none of several major ones is best in all cases (Aggarwal 2014). Most need a separate validation set, which can't be reused in creating the final tree, resulting in a weaker overall model. The best-known is **cost complexity pruning**—generate a number of increasingly simpler trees by pruning the “least useful” node at a time, and pick the simplest tree whose performance is at least within one standard deviation of the most accurate tree's performance. Because VC dimension of a tree is the number of leaves, this makes sense. The main method that uses the same data set is **pessimistic error pruning**. Implemented in C4.5, it uses binomial confidence limits from table interpolation at $a = 0.25$ to reject if the tree's performance at that level doesn't improve that of the node. Newer Weka implementation uses Wilson score interval (Witten et al. 2016). Based on my tests, using the sign test is simple and works better than pessimistic pruning by loosing less accuracy with more pruning, but \exists published extensive studies.

A possible extension (called a **model tree**) is replacing majority leafs by smarter models such as logistic regression (Rusch 2012). Despite intuitive appeal, model trees don't seem to be extensively experimentally investigated. Some apparent design flaws:

- Lost interpretability
- Leaves need to have a lot of data to avoid estimation error, which isn't much of a problem for majority
- Splitting is much less efficient because need to build leaf models \forall split, instead of incrementally updating entropy

Another possibility (called an **oblique tree**) is splitting on a linear combination of several variables. This is generally difficult, and tested approaches don't outperform a regular decision tree (Fernández-Delgado et al. 2014). The main goal is producing trees with fewer nodes, which presumably makes the tree more interpretable, but this is subjective because need more estimation for complex splits. Useful algorithms for multivariate splits rely on heuristics and can be slow for large k (Truong 2009).

A logical attempt is having oblique tree nodes separate space using LSVM. The intuition is to mimic something like a BSP tree (google it), so that when linear separation is problematic, further separation should improve performance. But from my experiments this fails because, due to linear SVM's own usually high accuracy, the split data will have strong class imbalance; so 2nd- and higher-level LSVMs are likely to learn to pick the majority class, mostly because SGD is very sensitive to class imbalance, and fixing it by oversampling doesn't help.

A somewhat silly extension of trees is **rules** (Witten et al. 2016). The idea is that rules are easier to understand, so try to flatten a tree into a list of rules. But a compact tree can bloat into many rules, trees are easy to understand when small, and rules are never smaller than trees.

Due to simplicity of linear separators like linear SVM, \exists many varieties:

- **Logistic regression**—equivalent to both neural network without hidden layers and linear regression which attempts to predict $\Pr(y_i = 1|x_i)$ based on the logit function. After a few decades of use, it's popular in statistics and very similar to linear SVM. The optimization problem of both more generally minimizes $\text{penalty}(w) + \sum L(y_i, f(x_i))$. Using $L = \ln(1 + e^{y_i f(x_i)})$ gives logistic regression. Both usually perform similarly in terms of generalization error. SVM is preferred theoretically because it doesn't estimates probabilities, ignoring examples outside the margins and so being more robust. But solving logistic regression to high precision is easier because its objective = convex function + differentiable function, for which currently \exists more efficient solvers (Lee et al. 2014). Also, coordinate descent for logistic regression is guaranteed to converge, unlike for SVM (Hastie et al. 2015).
- **Linear discriminant analysis (LDA)**—assumes that the examples with different labels come from multivariate normal distributions with same covariance. It works well for many problems, is very efficient, online, and not affected by the order of examples, unlike SGD. But normality and same covariance assumptions are rarely satisfied in practice, and logistic regression is more robust by making strictly fewer assumptions because linearity of log odds is implied by normality (Hastie et al. 2009). Also, need $n \geq D$ for LDA to work as is. For large D , even with modifications to the algorithm, covariance matrix estimation is difficult. Nevertheless, LDA may have an advantage for online learning.

- Linear SVM/linear SVM with squared hinge loss/logistic regression, all based on L_2 regularization. This leads to easier optimization problems because of extra differentiability, but no advantages otherwise. Despite much more efficient algorithms for solving to high precision in some cases (Yuan et al. 2012), in terms of generalization error SGD is hard to improve enough to prefer a more complex approach.

Beware that some sources use the formulation “penalty + sum/ n ”. This and a few other variants are a somewhat different problem with different properties, and don't prefer them over the standard SVM.

Some sources mention the outdated **perceptron algorithm**. It's designed for training neural networks without hidden layers, uses user-specified small constant learning rate and, unlike SGD, doesn't converge if the examples aren't linearly separable.

For SMO, values of the termination precision, the maximum number of iterations, and the precision to select support vectors are heuristic but currently the best known and seem robust for properly scaled data and kernels. Termination precision 0.001 is particularly suspicious because it's absolute and fails to take into account the relative C and b . Still, for efficiency the value should be the largest one such that decreasing it won't noticeably improve generalization for vast majority of problems, and it's hard to improve the tried-and-true LIBSVM strategy. Using $b = 0$ reportedly leads to smaller runtime and better termination criteria (Steinwart et al. 2011) but needs more research to be favored.

Another optimization, used by LIBSVM, is selecting i and j based on 2nd order information (Bottou et al. 2007). This is faster than the maximal violating pair, but not substantially. Several other optimizations are useful (Chang & Lin 2011).

Kernel SVM can't work online properly because $O(n)$ memory use is unacceptable for large n . Need to maintain a set of examples and eventually discard those that allegedly can't be support vectors. If high memory use isn't an issue, among many algorithms (Shawe-Taylor & Sun 2011; Du & Swamy 2019) **LASVM** (Bordes et al. 2005) is currently the most efficient approach. Also, can solve SVM in the primal. In particular, the representer theorem directly applies to SVM with $b = 0$ (also indirectly to $b \neq 0$), because minimizing inverse margin + hinge risk has exactly this form. Then can use SGD or **kernel PEGASOS**, but unlike LASVM and SMO, because the primal isn't differentiable, these can't use gradient information to select variables to optimize. By leaving many support vectors at 0, LASVM finds a sparse solution unlike the others.

Because larger C correspond to larger margins, it would be interesting to change the parameter selection objective to give up some accuracy to favor larger C or at least resolve ties this way. This appears unexplored.

For both kernel and linear SVM and variants, when picking C or l , it seems better to start with the solution to a previously solved problem for efficiency. But starting with all 0's is better for regularization for many solvers because making a value 0 tends to be easier than setting it back to 0.

Grid search and my heuristic tweak of it are just some of the methods for parameter optimization. Many more are reviewed in Hutter et al. (2019). Automated parameter selection, for which SVM is a important use case, is currently a very active research area.

Though always using a good kernel such as Gaussian works well, can try to automatically pick the kernel from data (Gönen & Alpaydin 2011). But the benefits of doing so in practice are unclear, mostly due to the inefficiency of the currently best known approaches.

Random forest is similar to bagging merged with **random subspace method**, which trains ensembles of classifiers on random subsets of features. The latter applies to any base A , and the former only to trees but gets much less correlation between them. Because low bias and high variance of trees makes them ideal for random forests, there is little interest in extending to other A .

✗ several attempted improvements of random forest, but all have flaws. **Rotation forest** (Kuncheva 2014) uses a sparse matrix to rotate x and creates a decision tree of the result, making a forest of these. The idea is appealing, but rotating is slow, needs scaling of data, and generally doesn't improve on random forests (Fernández-Delgado et al. 2014). Another possibility is, when constructing a tree, to sample the next feature based on some function of how useful that feature is, but the result is similar to bagging.

More is known about the approximation power of neural networks (Scarselli & Tsoi 1998), but little of it is useful. Beware that some such results appear to not depend on D , but the constants in various bounds do, and neural networks don't overcome the curse of dimensionality. ✗ asymptotic lower bounds on function approximation that depend on n and D (Ripley 1996).

SGD isn't the only way to train neural networks. Batch methods compute gradients based on all examples and use general optimization methods such as L-BFGS, which converges superlinearly (see the “Numerical Optimization” chapter), enabling high precision solutions. But for large n calculating gradients is expensive, and both high- and low-precision solutions have similar generalization, so prefer SGD. Contrary to intuition, many hidden neurons can work better than a few, maybe because due to random initialization have a higher

chance of discovering a good pattern in some neurons, while the rest harmlessly remain ineffective without introducing extra estimation error.

Another approach to regularizing a network is early stopping. But risk can decrease even more after increasing (Montavon et al. 2012), so don't have a clear way to do this. Weight decay is safer.

Feed-forward network is the most used but not the only type. **Recurrent network** allows neurons to feed themselves with own outputs (i.e., self-criticism attempt), but suffer more from vanishing gradients. Still, they have recently emerged as part of deep learning, particularly for speech recognition. **RBF network** uses kernels in input layer, but doesn't perform well (Fernández-Delgado et al. 2014), and kernel SVM is better at nonlinearity. **Bayesian neural network** got top performance on several high-dimensional data sets (Hastie et al. 2009). The idea is to have $\text{normal}(0, \sigma)$ prior weights for some σ such as 1. Assume independent outputs, and calculate $\Pr(y_i|x_i, \text{weights})$. This allows inference by MCMC sampling from the posterior. The method is very slow but seems suitable to the problems that were solved (Hastie et al. 2009). Complexity and slowness prevent its general use. Intuitively, should be able to get the same results from averaging several networks optimized using global search with weight decay, but this too would be very slow. Because MCMC is optimizing and averaging at the same time, maybe iterated local search can do the same, i.e., average all of its locally optimized results across the jumps, but unclear how to find a good jump step.

Another, perhaps more popular way to do deep learning is a **restricted Boltzmann machine (RBM)**. But an auto-encoder is simpler, and currently \exists experimental comparisons preferring one over the other in general. See Montavon et al. (2012), Du & Swamy (2019), and Bengio et al. (2015) for further discussion of many complexities of neural networks. It's also interesting to use unsupervised output from a deep network as features to another A .

When it comes to popular, new algorithms such as deep learning, it's important to ignore the hype from popular media. It's typical to hear that some company invested \$ few billion into applying deep learning into lets say their self-driving car prototype. More likely than not this is no more than an advertisement of how "cool" they are. I would be surprised if several decades from now deep learning will be any more significant than, e.g., optimization by linear programming that saved millions to early adapter companies, but nowadays is hardly a hot topic and needs a lot of engineering to work efficiently. One advice about deep learning I have read is "don't try this at home." Still, it seems that the classical "shallow" methods such as SVM have been thoroughly explored, and "deep" models have the potential for improvement.

Many boosting variants have been proposed, but no extensive studies have yet established their merits. **AdaBoost.M1** uses AdaBoost for $k > 2$ as is, except for disallowing answer flipping, but needs base A accuracy > 0.5 . **AdaBoost.MH** and others based on output codes (Schapire & Freund 2012) allow weaker learners but are more complex than SAMME and haven't been shown to perform better. **AdaBoost.L** (Schapire & Freund 2012) uses binomial loss with analytically derived optimal weights, improving earlier **LogitBoost**, but according to published (Schapire & Freund 2012) and my tests it performs similarly to AdaBoost.M1, and \exists a reason to prefer it over SAMME. Intuitively, logistic loss isn't very different from exponential because in both correct-example weights drop off exponentially fast.

Even though gradient boosting isn't suitable for classification, **gradient tree boosting** is claimed to perform well (Hastie et al. 2009). The idea is to do OVA and train k low-depth regression trees. Then \forall example, their outputs are combined using multinomial loss into a single score = 1 – the estimated correct class probability. Multinomial loss also gives probabilities directly, but those have numeric problems without proper normalization. But:

- Inefficient—need k trees, training each to minimize multinomial loss gradient can't use increment computation, and finding multinomial loss minimizers needs numerical minimization. Using low depth doesn't help.
- Per Fernández-Delgado et al. (2014) random forest still wins.
- Small constant tree weights are recommended because the optimal weights can lead to substantial overfitting, which is very suspicious.
- \exists studies comparing it with other boosting variants on many data sets.

In practice, \exists a reason to generally prefer boosting over random forest, which is robust to noise. More complex algorithms such as **BrownBoost** (Schapire & Freund 2012; Cheamanunkul et al. 2014) resist noise by eventually giving up on some examples (which loses convexity), but \exists experimental evidence in their favor.

Can combine unrelated A using **weighted majority voting**, where base A with accuracy a have relative weight $\ln\left(\frac{a}{1-a}\right)$ (Kuncheva 2014). This is less efficient than picking the best due to having to train all and use the results of all. Also, it's not better. The reason for not improving performance probably is that, unlike in homogeneous combinations such as random forest, base learners aren't intended to have low correla-

tion. In fact ! \exists an optimal combination method due to NFL effects (Hu & Damper 2008).

Can adapt decision trees for online learning. A typical method is **VLDT** (Gama 2010; see also Bifet et al. 2018 for some recent work). A simplified version of it, which also applies to continuous labels, recursively buffers examples in each leaf until have m (maybe 100), splits the leaf, and discards its examples. Discarding makes pruning unnecessary because don't have estimation error due to reusing examples for further splits. One problem is that can have a sorted random stream with m examples of the same class, in which case must either increase m or drop the majority ones until meet some balance criteria. Memory can become an issue, and keeping leaf buffers on disk can incur 1 I/O per update in the worst case, though using a large LRU cache will help. This also works for regression trees (see the "Machine Learning—Regression" chapter).

Can also make bagging (and so random forest) online by drawing the multiplicity of each example from an appropriate Poisson distribution (Gama 2010). But because decision tree buffers don't like multiplicity, this is questionable.

Cost-sensitive learning needs more experimental research and research for methods appropriate for various cases and supporting theory, all of which are currently lacking. An interesting possibility is to introduce a *don't know* class. The idea is that not knowing is usually much cheaper than making a wrong decision. This can help reduce $E[\text{the cost risk}]$, but need A -specific modifications to handle a *don't know*. A generic approach is to modify RMBoost to return *don't know* if the overall margin is small enough, using maybe some cross-validated constant.

An interesting idea is to relabel examples on which mistakes are costly. This is implemented by **Metacost** (Witten et al. 2016). Relabeling data may seem problematic because it introduces extra label noise, but nothing wrong with this. A more fundamental problem is using bagging to estimate probabilities. The estimates are accurate only for unstable A , which are few, and for decision trees random forest is much better. If the goal is to produce an understandable decision tree, using Metacost with random forest instead of bagging is a good solution.

For imbalance correction, a classic solution is **oversampling**, i.e., randomly increasing the number of minority classes' examples by resampling. This can cause too much duplication and overfit and, unlike ensemble resampling, commits to a single resample. So it's not recommended, despite being faster. **SMOTE** (Lopez et al. 2013) attempts to solve overfitting by extrapolating known examples, which seems silly regardless of its good performance and works only for vector x . Experimentally, studying the extra risk due to imbalance is difficult because have many possibilities to consider, such as n , imbalance ratio, specific local clusters, etc.

A class of algorithms generalizing naive Bayes is **graphical models** such as **Bayesian network**. They try to estimate probabilities more accurately by assuming dependencies among groups of features (Russell & Norvig 2020). E.g., in an image, neighboring pixels are dependent and distant pixels usually aren't. Of course given what the image represents, even distant pixels are related, so the technique doesn't seem suitable for machine learning. Also finding out which features depend on which others is difficult. Though some graphical models are successful in certain domains such as natural language processing, where substantial prior knowledge suggests structure, they aren't black-box because automatically and efficiently finding good structure from data is problematic. A typical approach is iteratively grouping correlated variables using some correlation measure such as mutual information, or doing global optimization using MDL as the objective function. These and others have numerous flaws, with both estimation and approximation errors; having to discretize continuous variables contributes to both. Also, estimating probabilities accurately solves a more general problem, increasing estimation error. Bayesian networks seem better adapted for domain-expert-driven probability modeling where have clear sequences of events in logical order.

26.24 Projects

- Improve the balanced accuracy calculation to test for missing labels
- Change k -NN to use a hash table to find the majority class
- Do more experiments on decision-tree pruning methods. Particularly interesting would be some MDL-type logic. Any solution should extend to a regression tree.
- Switch SVM to LRU cache of a user-specified size
- Create the generic neural network using ReLU for hidden neurons and 3–5 layers. Does this improve performance?
- Random forest has a natural feature-ranking mechanism—features selected in top nodes of trees are the most useful. Research details the specific algorithm, and compare the result with feature selection based on wrapper methods.

26.25 References

- Andoni, A. & Indyk, P. (2008). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1), 117–122.
- Anthony, M., & Bartlett, P. L. (2009). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.
- Aggarwal, C. C. (2014). *Data Classification: Algorithms and Applications*. CRC Press.
- Bache, K. & Lichman, M. (2013). *UCI Machine Learning Repository* [<http://archive.ics.uci.edu/ml>]. University of California. Accessed 10/19/2014.
- Bengio, Y., Goodfellow, I. J., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Biau, G., Devroye, L., & Lugosi, G. (2008). Consistency of random forests and other averaging classifiers. *The Journal of Machine Learning Research*, 9, 2015–2033.
- Bifet, A., Gavaldà, R., Holmes, G., & Pfahringer, B. (2018). *Machine Learning for Data Streams: With Practical Examples in MOA*. MIT Press.
- Bordes, A., Ertekin, S., Weston, J., & Bottou, L. (2005). Fast kernel classifiers with online and active learning. *The Journal of Machine Learning Research*, 6, 1579–1619.
- Bottou, L. (Ed.). (2007). *Large-scale Kernel Machines*. MIT Press.
- Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010* (pp. 177–186). Physica-Verlag HD.
- Breiman, L. (1996). Heuristics of instability and stabilization in model selection. *The annals of statistics*, 24(6), 2350–2383.
- Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5–32.
- Caruana, R., Karampatziakis, N., & Yessenalina, A. (2008). An empirical evaluation of supervised learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning* (pp. 96–103). ACM.
- Caruana, R., & Niculescu-Mizil, A. (2006). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning* (pp. 161–168). ACM.
- Cérou, F., & Guyader, A. (2006). Nearest neighbor classification in infinite dimension. *ESAIM: Probability and Statistics*, 10, 340–355.
- Chang, C. C., & Lin, C. J. (2011). LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3), 27.
- Cheamanunkul, S., Ettinger, E., & Freund, Y. (2014). Non-convex boosting overcomes random label noise. *arXiv preprint arXiv:1409.2905*.
- Daniely, A., Sabato, S., & Shwartz, S. S. (2012). Multiclass learning approaches: a theoretical comparison with implications. In *Advances in Neural Information Processing Systems* (pp. 485–493).
- Devroye, L., Gyorfi, L., & Lugosi, G. (1996). *A Probabilistic Theory of Pattern Recognition*. Springer.
- Domingos, P. (2000). A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning*. Stanford CA Morgan Kaufmann (pp. 231–238).
- Du, K. L., & Swamy, M. N. S. (2019). *Neural Networks and Statistical Learning*. Springer.
- Elliott, D. L. (1993). A better activation function for artificial neural networks.
- Fernández-Delgado, M., Cernadas, E., Barro, S., & Amorim, D. (2014). Do we need hundreds of classifiers to solve real world classification problems?. *The Journal of Machine Learning Research*, 15(1), 3133–3181.
- Gama, J. (2010). *Knowledge Discovery from Data Streams*. CRC Press.
- García, S., Luengo, J., & Herrera, F. (2014). *Data Preprocessing in Data Mining*. Springer.
- Gönen, M., & Alpaydin, E. (2011). Multiple kernel learning algorithms. *The Journal of Machine Learning Research*, 12, 2211–2268.
- Gottlieb, L. A., Kontorovitch, A., & Nisnevitch, P. (2014). Near-optimal sample compression for nearest neighbors. In *Advances in Neural Information Processing Systems* (pp. 370–378).
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
- Hastie, T., Tibshirani, R., & Wainwright, M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2), 251–257.
- Hsu, C. W., Chang, C. C., & Lin, C. J. (2010). A practical guide to support vector classification.
- Hu, R., & Damper, R. I. (2008). A ‘no panacea theorem’ for classifier combination. *Pattern Recognition*, 41(8), 2665–2673.
- Hutter, F., Kotthoff, L., & Vanschoren, J. (2019). Automated Machine Learning-Methods, Systems, Challenges. *Automated Machine Learning*.
- Hyafil, L., & Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-complete. *Information*

- Processing Letters*, 5(1), 15–17.
- Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI* (Vol. 14, No. 2, pp. 1137–1145).
- Kuncheva, L. I. (2014). *Combining Pattern Classifiers: Methods and Algorithms*. Wiley.
- Lee, J. D., Sun, Y., & Saunders, M. A. (2014). Proximal Newton-type methods for minimizing composite functions. *SIAM Journal on Optimization*, 24(3), 1420–1443.
- Leshno, M., Lin, V. Y., Pinkus, A., & Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6), 861–867.
- Long, P. M., & Servedio, R. A. (2010). Random classification noise defeats all convex potential boosters. *Machine Learning*, 78(3), 287–304.
- López, V., Fernández, A., García, S., Palade, V., & Herrera, F. (2013). An insight into classification with imbalanced data: Empirical results and current trends on using data intrinsic characteristics. *Information Sciences*, 250, 113–141.
- Mandziuk, J. (2010). *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*. Springer.
- Margineantu, D. D. (2001). Methods for cost-sensitive learning. (Doctoral dissertation, Oregon State University).
- Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning*. MIT Press.
- Montavon, G., Orr, G. B., & Müller, K. R. (2012). *Neural Networks: Tricks of the Trade*. Springer.
- Orriols-Puig, A., Macia, N., & Ho, T. K. (2010). Documentation for the data complexity library in C++. *Universitat Ramon Llull, La Salle*, 196.
- Prati, R. C., Batista, G. E., & Silva, D. F. (2014). Class imbalance revisited: a new experimental setup to assess the performance of treatment methods. *Knowledge and Information Systems*, 1–24.
- Ripley, Brian D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.
- Russell, S. J., Norvig, P. (2020). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- Rusch, T. (2012). *Recursive Partitioning of Models of a Generalized Linear Model Type*. WU Vienna University of Economics and Business.
- Scarselli, F., & Tsoi, A. C. (1998). Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural networks*, 11(1), 15–37.
- Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- Shawe-Taylor, J., & Sun, S. (2011). A review of optimization methodologies in support vector machines. *Neurocomputing*, 74(17), 3609–3618.
- Steinwart, I., Hush, D., & Scovel, C. (2011). Training SVMs without offset. *The Journal of Machine Learning Research*, 12, 141–202.
- Truong, A. K. Y. (2009). *Fast Growing and Interpretable Oblique Trees via Logistic Regression Models*. University of Oxford.
- Witten, I. H., Frank, E., & Hall, M.A. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann.
- Yuan, G. X., Ho, C. H., & Lin, C. J. (2012). Recent advances of large-scale linear classification. *Proceedings of the IEEE*, 100(9), 2584–2603.
- Zadrozny, B., Langford, J., & Abe, N. (2003). Cost-sensitive learning by cost-proportionate example weighting. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on* (pp. 435–442). IEEE.
- Zhu, J., Zou, H., Rosset, S. & Hastie, T. (2009) Multi-class AdaBoost. *Statistics and Its Interface*. (Vol 2, pp. 349–360).

27 Machine Learning—Regression

27.1 Introduction

Regression is strictly more difficult than classification but even more useful. The reader should be familiar with linear regression from a statistics class.

Theoretically, regression is stochastic multidimensional function approximation (stochastic because of possible y noise), except the latter is easier because can query a function at any point. Extrapolation is particularly difficult for regression because for regions far enough from those with available data, at best can assume that the unknown function has value = weighted average of known points. In this chapter the view is that of machine learning, but can also take a statistics view. See Berk (2020) for a good discussion of the latter.

Due to the curse of dimensionality a local basis set of something like linear patches doesn't scale in D . So all methods are "semilocal" in that they make global assumptions and try to model local behavior in subregions with enough data support, i.e., a get global basis with local support.

27.2 Risk Estimation

The most natural error metric is by how much a prediction is expected to be wrong. This is L_1 loss. But many algorithms have difficulty minimizing it, and want to have larger penalty on larger errors. So instead use differentiable L_2 loss, called **root-mean-squared error (RMSE)** as the main optimization criteria. To compare, L_∞ loss is the worst-case error, at least on the test set, but the true value is estimable only if use a fixed quantile.

RMSE is more difficult to interpret than the other two. Think of it as the standard deviation of y given f . This leads to **expStd**—the % of explained standard deviation = $1 - \frac{\text{RMSE}}{\text{stdev}(y_{\text{test data}})}$. Think of it as how much f explains more over the simple average of test data labels. It seems to be the best criteria for comparing various A . Because $-\text{expStd}$ is a monotonic function of RMSE, using either for comparison gives the same result, but the former is easy to turn into a curved grade for comparing many A on many data sets.

Cross-validation uses RMSE directly. Though it has no problem with minimizing the more intuitive and direct L_1 loss, it seems more appropriate to use RMSE because various A use it, and it puts larger penalty on larger errors. Not all applications have such preference, but it's hard to define a perfect error metric. In regression what you need, want, and get can be different, and L_2 loss tends to give what you need more than the others. Special applications must use customized error metrics that take error costs into account.

```
struct RegressionStats{double expStd, rmse, l1Err, l1InfErr;};
RegressionStats evaluateRegressor(
    Vector<pair<double, double> > const* testResult)
{
    IncrementalStatistics yStats, l2Stats, l1Stats;
    for(int i = 0; i < testResult.getSize(); ++i)
    {
        yStats.addValue(testResult[i].first);
        double diff = testResult[i].second - testResult[i].first;
        l1Stats.addValue(abs(diff));
        l2Stats.addValue(diff * diff);
    }
    RegressionStats result;
    result.l1InfErr = l1Stats.maximum();
    result.l1Err = l1Stats.getMean();
    result.rmse = sqrt(l2Stats.getMean());
    result.expStd = 1 - result.rmse/yStats.stdev();
    return result;
}
template<typename LEARNER, typename DATA, typename PARAMS> double
crossValidateReg(PARAMS const* p, DATA const* data, int nFolds = 5)
{
    return evaluateRegressor(crossValidateGeneral<LEARNER,
        typename DATA::Y_TYPE>(p, data, nFolds)).rmse;
```

```

}

template<typename LEARNER, typename PARAM, typename DATA>
struct RRiskFunctor
{
    DATA const& data;
    RRiskFunctor(DATA const& theData) : data(theData) {}
    double operator()(PARAM const& p) const
        { return crossValidateReg<LEARNER>(p, data); }
};

```

Stratification isn't available unlike for classification, but repeated cross-validation is:

```

template<typename LEARNER, typename DATA, typename PARAMS> double
    repeatedCVReg(PARAMS const& p, DATA const& data, int nFolds = 5,
    int nRepeats = 5)
{
    return evaluateRegressor(repeatedCVGeneral<double>(
        LEARNER(data, p), data, nFolds, nRepeats)).rmse;
}

template<typename LEARNER, typename PARAM, typename DATA>
struct RRCVRiskFunctor
{
    DATA const& data;
    RRCVRiskFunctor(DATA const& theData) : data(theData) {}
    double operator()(PARAM const& p) const
        { return repeatedCVReg<LEARNER>(p, data); }
};

```

Sometimes it's useful to have confidence intervals on individual parameters. Bootstrap methods are a general way to do this, though some models such as linear regression have specialized logic and tests. But parameters are usually dependent on each other, so, e.g., if one goes up, some others must go down to compensate, etc. Usually want to know if the parameters $\neq 0$ instead of their actual values. And the intervals are only as good as the model because, e.g., if compute intervals of a biased penalized model, such as lasso, the intervals will be affected by the bias. A confidence interval is only good for telling how good is a parameter estimate, not how good is the parameter.

Another question is deciding whether the data is modeled correctly using a particular, usually parametric model such as linear regression. Despite some specialized tests for specific models, checking accuracy using cross-validation and variants is the only general method. Also, specialized methods become invalid under any minor tweaks to the model. Any model will have some estimation and approximation errors.

Finally, given a specific prediction for a given x , often want to get some confidence interval on it. Bagging is a general way to do so—check the predictions of models trained on resampled data. A question is how much to trust in such intervals—at best they capture the variance in the model, but not the bias. So a large interval is warning sign, but a small one isn't a guarantee that all is well. These are more tolerance than confidence intervals (see the “Computational Statistics” chapter).

27.3 Complexity Control

For a useful analysis of regression, must assume loss \leq some large constant M . This isn't the case for L_2 loss, but intuitively can consider L_2 loss trimmed at M . If don't have a bound on loss, a single very bad loss value's can skew the risk by an arbitrary amount, so can't give a probabilistic guarantee.

Similarly to VC dimension in classification, a complexity measure of G is **pseudo-dimension** (Mohri et al. 2018):

- For n points, create a set of classifiers IG , given by picking constants $t_i \forall$ example and defining $ig(x_i) = \text{sign}(g(x_i) - t_i)$. I.e., g are discretized around how much they oscillate around a particular set of control points.
- $\text{PDim}(G) = \text{VC-dim}(IG)$ under the worse-case choice of t_i .

Theorem (Mohri et al. 2018): For G consisting of linear predictors, $\text{PDim}(G) = D + 1$.

As for classification, can bound generalization error by a function of training error, n , and the complexity of G . Theorem (Mohri et al. 2018): Let G have pseudo-dimension d . Then, with probability $\geq 1 - p$,

$$R_f \leq R_{f,n} + M \left(\sqrt{\frac{2d \ln(en/d)}{n}} + \sqrt{\frac{-\ln(p)}{2n}} \right).$$

27.4 Linear Regression

Want the best-fit linear combination of features $y = wx + b$, where the weight vector w and the bias term b minimize $\sum (y_i - f(x_i))^2$ for the training data. For simplicity, make b part of w by adding a constant feature to x with value 1. Then for X = the $n \times (D+1)$ matrix of feature values for all data and Y the vector of outputs, $w = (X^T X)^{-1} X^T Y$. Assuming $y_i - f(x_i) \sim \text{normal}$, \exists confidence intervals for w and b , but these are useful only if the model is a very good approximation and not considered further.

The computation fails when $D > n$ or X is singular. Need additional logic such as computing a pseudo-inverse, or, in the latter case, removing redundant features or examples.

Computing the inverse needs $O(n^3)$ time and $O(n^2)$ space, which doesn't scale beyond medium n . A more efficient way to compute the solution is using an iterative process such as the one for lasso regression.

Two main assumptions are made:

- The model is linear. This usually leads to large approximation error. But when have little data or too many variables, assuming a linear model may be the only way to keep estimation error in check.
- The errors are iid and $\sim \text{normal}$. This is reasonable for measurement errors, i.e., the assumed model is correct, but the data is subject to some randomness due to being measured incorrectly. But systematic difference in errors is typical because, e.g., their could be more variance in large values than in small ones.

Linear model allows much analysis such as estimates on the effect of individual variables on the solution. These are only relevant if the linear model has a low approximation error. Even adding more variables can result in a different model where confidence intervals on the variables in the smaller model change substantially. This is **Simpson's paradox** (Wikipedia 2018). E.g., in a graduate school admissions bias study, gender was determined a significant predictor in favor of males, but it turned out that department-by-department females actually did better on acceptance rate, and the bias effect was due to the fact that they applied to more competitive departments (here the gender predictor even flipped sign). So structure of a model seems restricted to that particular model and not absolute truth.

But the study was observational. With iid data Simpson's paradox is less likely due to lack of a global bias but possible if don't have additivity of factors as assumed by the linear model.

27.5 Lasso Regression

A good solution to the classical regression's problems is to use L_1 penalty on weights, getting minimization objective $l|w| + \sum (y_i - f(x_i))^2$, for small constant l . This is called the **lasso**. As for LSVM, the L_1 penalty leads to sparse solutions.

The problem is convex and of the form convex function + differentiable function. So coordinate descent (see the "Numerical Optimization" chapter) is globally convergent (Hastie et al. 2015). Also, the 1D subproblems have analytic solutions, and penalties on all other components remain constant, allowing incremental evaluation.

For simplicity, add a $\frac{1}{2}$ factor, and minimize $l|w| + \frac{1}{2} \sum (y_i - f(x_i))^2$. When working with weight j , aggregate the other components of $f(x_i)$ into $s_i = y_i - (b + \sum_{k \neq j} w_k x_{ik})$. It's at the optimum when $0 \in$ subgradient set, i.e., $0 = IS(w) + \sum (s_i - w_j x_{ij})(-x_{ij})$, where S is the sign function. Solving, $w_j = \frac{a + IS(w_j)}{c}$ for $a = \sum s_i x_{ij}$, and $c = \sum x_{ij}^2$. Because $c \geq 0$, $a \geq -l$ if $w_j \geq 0$ and $\leq l$ if $w_j \leq 0$. So considering intervals of a , $w_j < 0$ if $a < -l$, > 0 if $a > l$, and 0 otherwise. So

$$w_j = \begin{cases} \frac{a-l}{c}, & \text{if } a < -l \\ \frac{a+l}{c}, & \text{if } a > l \\ 0 & \text{otherwise} \end{cases}$$

The formula is numerically stable because very small c cause $w_j = 0$.

$b = \frac{1}{n} \sum s_i$ because then $\frac{1}{2} \sum (s_i - b)^2$ is smallest. l is selected using cross-validation, from the same range as for linear SVM.

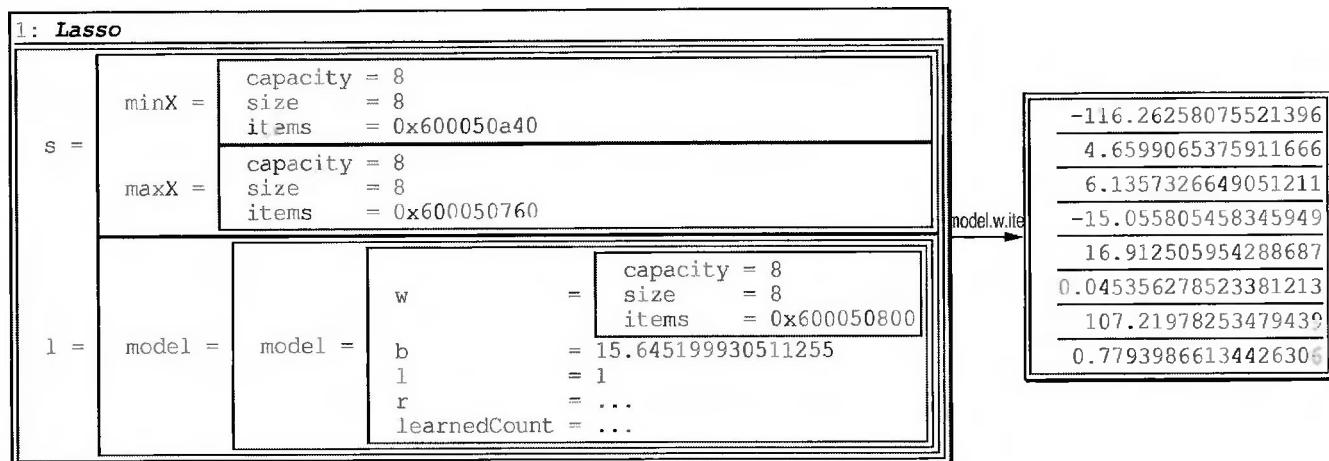


Figure 27.1: Memory layout of lasso regression result for the energy data

```

class L1LinearReg
{
    Vector<double> w;
    double b, l, r;
    int learnedCount; //this and r are only for online learning with SGD
    double f(NUMERIC_X const& x) const{return dotProduct(w, x) + b;}
    template<typename DATA> void coordinateDescent(DATA const& data,
        int maxIterations, double eps)
    {
        assert(data.getSize() > 0);
        int D = getD(data);
        Vector<double> sums(data.getSize());
        for(int i = 0; i < data.getSize(); ++i) sums[i] = data.getY(i);
        bool done = false;
        while(!done && maxIterations--)
        {
            done = true;
            for(int j = -1; j < D; ++j)
            {
                double oldVar = j == -1 ? b : w[j];
                //remove current var from sum
                for(int i = 0; i < data.getSize(); ++i)
                    sums[i] += j == -1 ? b : w[j] * data.getX(i, j);
                //solve for opt current var
                if(j == -1)
                {//update bias
                    IncrementalStatistics s;
                    for(int i = 0; i < data.getSize(); ++i)
                        s.addValue(sums[i]);
                    b = s.getMean();
                }
                else
                {//update weight
                    double a = 0, c = 0;
                    for(int i = 0; i < data.getSize(); ++i)
                    {
                        double xij = data.getX(i, j);
                        a += sums[i] * xij;
                        c += l * xij * xij;
                    }
                    if(a < -l) w[j] = (a - l)/c;
                    else if(a > l) w[j] = (a + l)/c;
                    else w[j] = 0;
                }
                //add back current var to up
                for(int i = 0; i < data.getSize(); ++i)
                    sums[i] -= j == -1 ? b : w[j] * data.getX(i, j);
            }
        }
    }
}

```

```

        if(abs((j == -1 ? b : w[j]) - oldVar) > eps) done = false;
    }
}
public:
    template<typename DATA> L1LinearReg(DATA const& data, double theL,
        int nCoord = 1000): l(theL/2), b(0), w(getD(data)), learnedCount(-1)
    {coordinateDescent(data, nCoord, pow(10, -6));}
    typedef pair<int, pair<double, double>> PARAM; //D/1/r
    L1LinearReg(PARAM const& p): l(p.second.first/2), r(p.second.second),
        b(0), w(p.first), learnedCount(0) {}
    void learn(NUMERIC_X const& x, double y)
    {
        assert(learnedCount != -1); //can't mix batch and offline
        double rate = r * RMRate(learnedCount++), err = y - f(x);
        //1/n*|w| + (y - wx + b)^2
        //dw = 1/n*sign(w) - x(y - (wx + b));
        //db = - (y - (wx + b))
        for(int i = 0; i < w.getSize(); ++i) w[i] +=
            rate * (x[i] * err - (w[i] > 0 ? 1 : -1) * 1/learnedCount);
        b += rate * err;
    }
    double predict(NUMERIC_X const& x) const{return f(x);}
    template<typename MODEL, typename DATA>
    static double findL(DATA const& data)
    {
        int lLow = -15, lHigh = 5;
        Vector<double> regs;
        for(double j = lHigh; j > lLow; j -= 2) regs.append(pow(2, j));
        return valMinFunc(regs.getArray(), regs.getSize(),
            RRiskFunctor<MODEL, double, DATA>(data));
    }
};
struct NoParamsL1LinearReg
{
    L1LinearReg model;
    template<typename DATA> NoParamsL1LinearReg(DATA const& data):
        model(data, L1LinearReg::findL<L1LinearReg>(data)) {}
    double predict(NUMERIC_X const& x) const{return model.predict(x);}
};
typedef ScaledLearner<NoParamsLearner<NoParamsL1LinearReg, double>, double>
    SLasso;

```

Each iteration takes $O(n)$ time. See the “Machine Learning—Classification” chapter for when the data is on disk.

To train online, use SGD. Given rate r , differentiating the objective gives the update equations $w_i \leftarrow w_i + r \left(x_i \epsilon - S(w_i) \frac{1}{n} \right)$ and $b \leftarrow b + r \epsilon$, for $\epsilon = y - f(x)$. As for online LSVM, use the number of seen examples as n . Unlike for LSVM, need to race-tune the initial r because SGD steps can become too large. SGD converges very slowly, so can't get sparsity because many eventual 0 components of w may not get close enough to 0. For racing, the initial r use the same range as for l , which is the same as for classification.

```

    typedef pair<int, pair<double, double>> PARAM; //D/1/r
    L1LinearReg(PARAM const& p): l(p.second.first/2), r(p.second.second),
        b(0), w(p.first), learnedCount(0) {}
    void learn(NUMERIC_X const& x, double y)
    {
        assert(learnedCount != -1); //can't mix batch and offline
        double rate = r * RMRate(learnedCount++), err = y - f(x);
        for(int i = 0; i < w.getSize(); ++i) w[i] +=
            rate * (x[i] * err - (w[i] > 0 ? 1 : -1) * 1/learnedCount);
        b += rate * err;
    }
}

```

```

class SRaceLasso
{
    ScalerMQ s;
    RaceLearner<L1LinearReg, L1LinearReg::PARAM> model;
    static Vector<L1LinearReg::PARAM> makeParams(int D)
    {
        Vector<L1LinearReg::PARAM> result;
        int lLow = -15, lHigh = 5, rLow = -15, rHigh = 5;
        for(int j = lHigh; j > lLow; j -= 2)
        {
            double l = pow(2, j);
            for(int i = rHigh; i > rLow; i -= 2) result.append(
                L1LinearReg::PARAM(D, pair<double, doublereturn result;
    }
public:
    template<typename DATA> SRaceLasso(DATA const& data):
        model(makeParams(getD(data))), s(getD(data))
    {
        for(int j = 0; j < 1000000; ++j)
        {
            int i = GlobalRNG().mod(data.getSize());
            learn(data.getX(i), data.getY(i));
        }
    }
    SRaceLasso(int D): model(makeParams(D)), s(D) {}
    void learn(NUMERIC_X const& x, double y)
    {
        s.addSample(x);
        model.learn(s.scale(x), y);
    }
    double predict(NUMERIC_X const& x)const
        { return model.predict(s.scale(x)); }
};
```

Lasso is the default approach, replacing linear regression as a black box due to favorable properties of the L_1 penalty. Using $l=0$ and not scaling reduces to linear regression, and the only conceivable benefit is to avoid scaling to get a slightly easier-to-explain model. But can reverse-scale w and b (which correspondingly absorb the multiplicative and the additive term of the scaling) of lasso to get the same result.

27.6 Nearest Neighbor Regression

Compared to classification, use the average instead of the plurality. The algorithm is consistent for L_2 loss under the same conditions as for classification (Györfi et al. 2002). The same theoretical logic applies to selecting k .

```

template<typename X = NUMERIC_X, typename INDEX = VpTree<X, double, typename
    EuclideanDistance<X>::Distance> class KNNReg
{
    mutable INDEX instances;
    int k;
public:
    template<typename DATA> KNNReg(DATA const& data, int theK = -1): k(theK)
    {
        assert(data.getSize() > 0);
        if(k == -1) k = 2 * int(log(data.getSize()) / 2) + 1;
        for(int i = 0; i < data.getSize(); ++i)
            learn(data.getY(i), data.getX(i));
    }
    void learn(double label, X const& x){instances.insert(x, label);}
    double predict(X const& x)const
    {
        Vector<typename INDEX::NodeType*> neighbors = instances.kNN(x, k);
```

```

        IncrementalStatistics s;
        for(int i = 0; i < neighbors.getSize(); ++i)
            s.addValue(neighbors[i]->value);
        return s.getMean();
    }
};

typedef ScaledLearner<NoParamsLearner<KNNReg>, double>, double> SKNNReg;

```

As for classification, search can take $\approx O(n)$ time for large D .

27.7 Regression Tree

Compared to a decision tree, predict real y . This assumes that a piecewise-constant f is a good approximation to the real one. A natural loss is the **sum of squared errors (SSE)**, the minimizer of which is the average of the y values. It's updatable incrementally, allowing to calculate splits efficiently.

As for classification, every subtree must be better than its root to not be pruned. This is checked by comparing squared errors using the sign test with the default z-score 0.25, which seems to be best experimentally.

```

struct RegressionTree
{
    struct Node
    {
        union
        {
            double split; //for internal nodes
            double label; //for leaf nodes
        };
        int feature; //for internal nodes
        Node *left, *right;
        bool isLeaf() { return !left; }
        Node(int theFeature, double theSplit) : feature(theFeature),
            split(theSplit), left(0), right(0) {}
    } *root;
    Freelist<Node> f;
    double SSE(double sum, double sum2, int n) const
    {
        return sum2 - sum * sum/n;
    }
    template<typename DATA> struct Comparator
    {
        int feature;
        DATA const& data;
        double v(int i) const { return data.data.getX(i, feature); }
        bool operator()(int lhs, int rhs) const { return v(lhs) < v(rhs); }
        bool isEqual(int lhs, int rhs) const { return v(lhs) == v(rhs); }
    };
    void rDelete(Node* node)
    {
        if(node)
        {
            rDelete(node->left);
            f.remove(node->left);
            rDelete(node->right);
            f.remove(node->right);
        }
    }
    double classifyHelper(NUMERIC_X const& x, Node* current) const
    {
        while (!current->isLeaf()) current = x[current->feature] <
            current->split ? current->left : current->right;
        return current->label;
    }
    template<typename DATA> Node* rHelper(DATA& data, int left, int right,
        double pruneZ, int depth, bool rfMode)
    {

```

```

int D = data.getX(left).getSize(), bestFeature = -1,
    n = right - left + 1;
double bestSplit, bestScore, sumY = 0, sumY2 = 0;
Comparator<DATA> co = {-1, data};
for(int j = left; j <= right; ++j)
{
    double y = data.getY(j);
    sumY += y;
    sumY2 += y * y;
}
double ave = sumY/n, sse = max(0.0, SSE(sumY, sumY2, n));
Bitset<> allowedFeatures;
if(rfMode)
{//sample features for random forest
    allowedFeatures = Bitset<>(D);
    allowedFeatures.setAll(0);
    Vector<int> p = GlobalRNG().sortedSample(sqrt(D), D);
    for(int j = 0; j < p.getSize(); ++j) allowedFeatures.set(p[j], 1);
}
if(sse > 0) for(int i = 0; i < D; ++i)//find best feature and split
{
    if(allowedFeatures.getSize() == 0 || allowedFeatures[i])
    {
        co.feature = i;
        quickSort(data.permutation.getArray(), left, right, co);
        double sumYLeft = 0, sumYRight = sumY, sumY2Left = 0,
               sumY2Right = sumY2;
        int nRight = n, nLeft = 0;
        for(int j = left; j < right; ++j)
        {//incrementally roll counts
            int y = data.getY(j);
            ++nLeft;
            sumYLeft += y;
            sumY2Left += y * y;
            --nRight;
            sumYRight -= y;
            sumY2Right -= y * y;
        }
        double fLeft = data.getX(j, i), score =
            SSE(sumYLeft, sumY2Left, nLeft) +
            SSE(sumYRight, sumY2Right, nRight),
            fRight = data.getX(j + 1, i);
        if(fLeft != fRight && //don't split equal values
           (bestFeature == -1 || score < bestScore))
        {
            bestScore = score;
            bestSplit = (fLeft + fRight)/2;
            bestFeature = i;
        }
    }
}
if(n < 3 || depth <= 1 || sse <= 0 || bestFeature == -1)
    return new(f.allocate())Node(-1, ave);
//split examples into left and right
int i = left - 1;
for(int j = left; j <= right; ++j)
{
    if(data.getX(j, bestFeature) < bestSplit)
        swap(data.permutation[j], data.permutation[++i]);
}
if(i < left || i > right) return new(f.allocate())Node(-1, ave);
Node* node = new(f.allocate())Node(bestFeature, bestSplit);
//recursively compute children
node->left = rHelper(data, left, i, pruneZ, depth - 1, rfMode);
node->right = rHelper(data, i + 1, right, pruneZ, depth - 1, rfMode);
//try to prune
double nodeWins = 0, treeWins = 0;

```

```

    for(int j = left; j <= right; ++j)
    {
        double y = data.getY(j), eNode = ave - y, eTree =
            classifyHelper(data.getX(j), node) - y;
        if(eNode * eNode == eTree * eTree)
        {
            nodeWins += 0.5;
            treeWins += 0.5;
        }
        else if(eNode * eNode < eTree * eTree) ++nodeWins;
        else ++treeWins;
    }
    if(!rfMode && signTestAreEqual(nodeWins, treeWins, pruneZ))
    {
        rDelete(node);
        node->left = node->right = 0;
        node->label = ave;
        node->feature = -1;
    }
    return node;
}
Node* constructFrom(Node* node)
{
    Node* tree = 0;
    if(node)
    {
        tree = new(f.allocate()) Node(*node);
        tree->left = constructFrom(node->left);
        tree->right = constructFrom(node->right);
    }
    return tree;
}
public:
    template<typename DATA> RegressionTree(DATA const& data, double pruneZ =
        0.25, int maxDepth = 50, bool rfMode = false): root(0)
    {
        assert(data.getSize() > 0);
        int left = 0, right = data.getSize() - 1;
        PermutedData<DATA> pData(data);
        for(int i = 0; i < data.getSize(); ++i) pData.addIndex(i);
        root = rHelper(pData, left, right, pruneZ, maxDepth, rfMode);
    }
    RegressionTree(RegressionTree const& other)
    {
        root = constructFrom(other.root);
    }
    RegressionTree& operator=(RegressionTree const& rhs)
    {
        return genericAssign(*this, rhs);
    }
    double predict(NUMERIC_X const& x) const
    {
        return root ? classifyHelper(x, root) : 0;
    }
};

```

The runtime = that of decision tree.

27.8 Random Forest Regression

Compared to classification, use the average instead of the plurality to combine:

```

class RandomForestReg
{
    Vector<RegressionTree> forest;
public:
    template<typename DATA> RandomForestReg(DATA const& data,
        int nTrees = 300){addTrees(data, nTrees);}
    template<typename DATA> void addTrees(DATA const& data, int nTrees)
    {

```

```

assert(data.getSize() > 1);
for(int i = 0, D = getD(data); i < nTrees; ++i)
{
    PermutatedData<DATA> resample(data);
    for(int j = 0; j < data.getSize(); ++j)
        resample.addIndex(GlobalRNG().mod(data.getSize()));
    forest.append(RegressionTree(resample, 0, 50, true));
}
double predict(NUMERIC_X const& x) const
{
    IncrementalStatistics s;
    for(int i = 0; i < forest.getSize(); ++i)
        s.addValue(forest[i].predict(x));
    return s.getMean();
}
};

```

27.9 Neural Network

The structure is similar to the one for classification, but with some changes:

- Cross-validate the number of hidden neurons. Quadrupling from 1 until 64 seems reasonable.
- Cross-validate the initial learning rate for SGD to not diverge. Use the same range as for online lasso.
- Don't use an activation function for the output unit.

```

class HiddenLayerNNReg
{
    Vector<NeuralNetwork> nns;
public:
    template<typename DATA> HiddenLayerNNReg(DATA const& data,
                                                Vector<double>const& p, int nGoal = 100000, int nNns = 5):
        nns(nNns, NeuralNetwork(getD(data), true, p[0]))
    { //structure
        int nHidden = p[1], D = getD(data),
            nRepeats = ceiling(nGoal, data.getSize());
        double a = sqrt(3.0/D);
        for(int l = 0; l < nns.getSize(); ++l)
        {
            NeuralNetwork& nn = nns[l];
            nn.addLayer(nHidden);
            for(int j = 0; j < nHidden; ++j)
                for(int k = -1; k < D; ++k)
                    nn.addConnection(0, j, k, k == -1 ? 0 :
                        GlobalRNG().uniform(-a, a));
            nn.addLayer(1);
            for(int k = -1; k < nHidden; ++k)
                nn.addConnection(1, 0, k, 0);
        }
    //training
        for(int j = 0; j < nRepeats; ++j)
            for(int i = 0; i < data.getSize(); ++i)
                learn(data.getX(i), data.getY(i));
    }
    void learn(NUMERIC_X const& x, double label)
    {
        for(int l = 0; l < nns.getSize(); ++l)
            nns[l].learn(x, Vector<double>(1, label));
    }
    double evaluate(NUMERIC_X const& x) const
    {
        double result = 0;
        for(int l = 0; l < nns.getSize(); ++l)
            result += nns[l].evaluate(x)[0];
    }
};

```

```

        return result/nns.getSize();
    }
    int predict(NUMERIC_X const& x) const{return evaluate(x);}
};

struct NoParamsNNReg
{
    HiddenLayerNNReg model;
    template<typename DATA> static Vector<double> findParams(DATA const&
        data, int rLow = -15, int rHigh = 5, int hLow = 0, int hHigh = 6)
    {
        Vector<Vector<double>> sets(2);
        for(int i = rLow; i <= rHigh; i += 2) sets[0].append(pow(2, i));
        for(int i = hLow; i <= hHigh; i += 2) sets[1].append(pow(2, i));
        return gridMinimize(sets,
            RRiskFunctor<HiddenLayerNNReg, Vector<double>, DATA>(data));
    }
    template<typename DATA> NoParamsNNReg(DATA const& data):
        model(data, findParams(data)) {}
    double predict(NUMERIC_X const& x) const{return model.predict(x);}
};
typedef ScaledLearner<NoParamsLearner<NoParamsNNReg, double>, double, EMPTY,
    ScalerMQ> SNNReg;

```

As with classification, the runtime per example is $O(W)$. With global optimization and properly chosen number of hidden units, a neural network is consistent for L_2 loss (Györfi et al. 2002).

27.10 Feature Selection

As for classification, wrapper search with random forest is useful for selecting features:

```

template<typename SUBSET_LEARNER = RandomForestReg> struct SmartFSLearnerReg
{
    typedef FeatureSubsetLearner<SUBSET_LEARNER> MODEL;
    MODEL model;
public:
    template<typename DATA> SmartFSLearnerReg(DATA const& data,
        int subsampleLimit = 20): model(data, selectFeaturesSmart(
            RRiskFunctor<MODEL, Bitset<>, DATA>(data), getD(data),
            subsampleLimit)) {}
    double predict(NUMERIC_X const& x) const{return model.predict(x);}
};

```

27.11 Comparing Performance

When using average curved expStd, random forest seems to be the best A.

mqLasso	mqOnlineLasso	RegTree	RFRegTree	B RF Lasso	sKNNReg	mqNNcv
0.62	0.41	0.5	0.84	0.9	0.73	0.74

Figure 27.2: Some performance comparisons

Combining it with lasso gives even better results at the cost of an extra cross-validation:

```

class SimpleBestCombinerReg
{
    BestCombiner<double> c;
public:
    template<typename DATA> SimpleBestCombinerReg(DATA const& data)
    {
        c.addNoParamsClassifier<RandomForestReg>(data, RRiskFunctor<
            NoParamsLearner<RandomForestReg, double>, EMPTY, DATA>(data));
        c.addNoParamsClassifier<SLasso>(data, RRiskFunctor<
            NoParamsLearner<SLasso, double>, EMPTY, DATA>(data));
    }
    double predict(NUMERIC_X const& x) const{return c.predict(x);}
};

```

In the literature, currently !comprehensive comparisons of modern regression methods based on many

data sets. Meyer et al. (2003) compared some algorithms on some data sets.

	LinReg	RegTree	NN	MARS	AdditiveModel	ProjectionPursuit	RF	Boosting
	2.83	6.82	6.83	5.5	4.33	3.08	2.83	4.67

Figure 27.3: Converted to Friedman ranks, the mean L_2 errors from Meyer et al. (2003) give the following average ranks (SVR, boosting, and MARS are discussed in the comments; for boosting, projection pursuit, and additive models see Hastie et al. 2009)

Random forest and neural networks appear to be the algorithms of choice based on this study; both are much more efficient than SVR. Combined with my experiments, random forest seems to be the best single A.

In a simulation study, LOESS (discussed in the comments) and MARS performed poorly on high-dimensional data (Clarke et al. 2009).

But simple models are often good enough—e.g., for housing prices linear regression makes logical sense due to natural additivity of features such as school quality and transportation convenience. Also the estimated coefficients logically can't run into Simpson's paradox due to natural additivity.

27.12 Implementation Notes

Looking at RMSE as explained standard deviation percentage is original and makes sense as a scale measure.

The lasso implementation is the most complicated due to needing to work out the mathematics. A mix of coordinate descent and the usual SGD seems to be a good way to optimize it.

The regression tree implementation closely mirrors that of the classification tree because using the sign test for pruning working in the same way in both cases. Other algorithms are also mostly an extension of the classification ones, though with potentially different parameters.

27.13 Comments

For regression, costs matter too because an overestimate can be less or more expensive than an underestimate, but modeling this in a way that leads to efficient A is difficult.

Another interesting complexity measure is **fat-shattering dimension**, which, however, is more complicated and doesn't lead to a more useful upper bound (Mohri et al. 2018). It leads to a lower bound though, similar to the VC dimension one (Anthony & Bartlett 1999). That bound is also to be taken with a grain of salt because imposing extra conditions on G such as restricting number sizes usually lowers any complexity measure.

Sparsity is an active research field (Hastie et al. 2015). Additional tricks for lasso include using a simpler formula for coordinate descent when use the mean-variance scale, and starting with the previous l solution when selecting l . But doing this with cross-validation is clumsy.

Linear regression isn't robust in many ways, but robust solutions also have problems. See Wilcox (2016) for a good overview. This also applies to sparse regression. A popular option called **least median of squares (LMS)** has a high breakdown but is exponentially inefficient in D to compute. So there seem to be no good options for robust regression for all occasions (Huber & Ronchetti 2009). But for simple computational investigations, such as studying asymptotics of algorithms, deal with few variables (usually one or two), and can generate data on demand, so LMS seems to be a good technique.

Ridge regression uses the L_2 penalty, which also fixes linear regression problems and has an analytic solution but doesn't lead to sparsity. Also, the analytic solution takes $O(\min(n, D)^3)$ time, which doesn't scale. Perhaps the biggest problem with lasso is that when have correlated variables it can select any of them. An interesting proposal to address this is **elastic net**, i.e., using a combination of L_1 and L_2 penalties. It also has an analytic solution, but have an extra combination weight parameter, and know less about it theoretically.

Kernel support vector regression (also called **SVR**) fits an ε -tube for some number ε in the enhanced feature space, instead of a line (Mohri et al. 2018). This gives sparseness in the number of support vectors because examples inside a tube can't be support vectors. But SMO-like solution is more complicated than the classification one (Liao et al. 2002), and cross-validation also needs to pick ε . Still, essentially use the same algorithm, i.e., iteratively solve sequences of 2D problems; the difference is mostly in the calculation of gradients and solving the 2D problems. For parameters, all else equal, want small C , large y , and large ε .

Kernel ridge regression is a direct extension of ridge regression (Mohri et al. 2018). It also has an analytic solution but $O(n^3)$ runtime. Also, unlike for SVR, solutions aren't sparse in the number of support vectors.

Nadaraya-Watson kernel regression is an older method, which, despite a trivial generalization to $D > 1$ behaves exponentially poorly with increasing D (Clarke et al. 2009). Essentially the local kernel bandwidth

is fixed, but, e.g., distances between nearest neighbors are adaptive.

For a regression tree, many pruning criteria have been proposed, though fewer than for a decision tree. Cost complexity pruning also applies and is the main alternative.

Can alternatively consider L_1 loss for the tree, whose minimizer is the median. It may be more robust than the average but isn't suitable for all cases, and its incremental calculation is a bit complicated. One way to do it efficiently is using a balanced search tree with subtree size augmentation, where the key is a combination of absolute error and the number of the example (to allow nonunique items in the tree).

An interesting alternative is a model tree with linear regression at the leaves that uses smoothing (Witten et al. 2016). But despite claims of outperforming a regular regression tree, it doesn't seem to have a use case:

- It's much less interpretable
- Random forest does smoothing better
- Averaging at a leaf is less likely to overfit with little data
- \exists some design flaws such as needing a smoothing degree parameter and to fit a linear model to leaves with few examples

For random forest, the original-paper recommendation for the number of used features is $D/3$. But there seems to be no reason to prefer this over the more efficient \sqrt{D} of classification.

B-splines (see the comments in the “Numerical Algorithms—Working with Functions” chapter) fitted with least squares approximation can be used for regression in 1D. But even here have difficulty with noisy data; penalization methods help to some degree, but $D > 1$ is still problematic (Clarke et al. 2009). Extension of regression splines to $D > 1$ requires tensor products of splines (Hastie et al. 2009), which is essentially imposing a grid on the range of data and fitting a spline product to each cell. Despite universal consistency (Györfi et al. 2002), this doesn't scale.

A reasonable solution is **MARS** (Wikipedia 2016b; Hastie et al. 2009). Linear splines can be equivalently represented by a sum of SVM-like hinge functions (nonlinear generalize similarity), and MARS greedy approach idea is to add a hinge function at a time so that the addition is a hinge or a product of a hinge with a function already in the sum. Despite several computational tricks (Hastie et al. 2009), the runtime is slower and performance worse than those of random forest.

Local regression methods generalize k -NN in several ways. A typical solution is to use a kernel to model local region of several nearest neighbors instead of the average. Methods such as **LOESS** (Wikipedia 2016a; Hastie et al. 2009) do this, but SVR retains only support vectors and so seems much better. Also, LOESS needs more data than other methods for good estimation even for $D = 1$ (Clarke et al. 2009).

Many methods such as wavelets (Hastie et al. 2009) are available in 1D that don't scale to larger D . They have use cases for specialized domains, particularly for producing visualizations.

Any localized basis set is crushed by the curse of dimensionality. All well-performing methods make some assumptions where data effects carry over far away in the feature space—e.g., a tree split splits all of the feature space in two pieces, and not only areas for which have data. This dependence between local regions requires assumptions, and can't avoid these.

Another special model is **generalized linear model (GLM)** (Hastie et al. 2009). Logistic regression is a special case of it. The idea is to use a **link function** such as the logit transformation because the result is presumably easier to estimate. Can fit such models using a number of methods, including SGD. Many special algorithms have been developed, but simple modifications like the lasso penalty invalidate many of these. E.g., consider the **runs test**—for a good fit about the same number of examples will appear below and above the regression hyperplane. But with lasso that's no longer true. The flaw with a more general **general additive model (GAM)** is that the individual variables are still treated independently even if use some kernel regression transformations for the individual variables. E.g., for the digit data transformations won't make a difference.

In general, **semiparametric regression** models like GAM make limited parametric assumptions. E.g., domain logic can suggest that the response value is additive in the component values, but so that have diminishing returns. Still need to make some subjective choices about whatever parametric form remains, and don't want to try out too many forms or pick a wrong one.

An interesting application of regression is making very expensive simulations more efficient. E.g., consider designing a better car crash system, where a simulation means driving a car with doll passengers into a wall. To save cars, create a regression model (called **metamodel** or a **surrogate model**), and optimize it to know which design parameter configuration to try next. Essentially anything that can be done with a real model can be with the surrogate model, and the latter is updated after every evaluation of the real model. Random forest makes a good surrogate model, but can consider SVR if need differentiability for some analysis.

Using a metamodel contradicts the idea of solving a problem directly, but, e.g., applying an optimization method directly results in many more real model evaluations. The usual optimization methods aren't designed for very expensive simulations and strive to make the runtime O-scalable. Kriging/Gaussing processes (google these) are often used as metamodels in the published literature, but random forest and SVR are much faster and known to perform well for regression.

Sometimes even have metamodels. E.g., the first real model is maybe a car engine, the metamodel is the high-accuracy computational specification of it, and the metamodel is a random forest based on some key variables. It's also useful to distinguish regression models, which are designed for potentially noisy data, and interpolation models, and assume absence of noise.

Another typical application is regression for log-odds, which is essentially classification with confidence. Though not doing classification directly leads to a higher estimation error, and log-odds can be not estimated very well, this is useful in some applications, particularly when RMSE test error in log-odds is small enough. For numerical reasons, code 0/1 classes as 0.001/0.999.

27.14 Projects

- As an evaluation metric for final performance measurement, instead of RMSE experiment with trimmed mean of |error|. Same for 90th percent quantile of |error|. Do these give different results?
- Implement elastic net regression and experiment with it—does it do better than lasso for feature selection with correlated features? Extension—apply it to classification with logistic transformation.
- Implement SVR and compare its performance and efficiency to random forest.
- Investigate using numerical boosting with regression tree for learning with asymmetric loss functions such as linex.
- As boosting strategy, experiment on a linear model followed by random forest.
- Research and implement LMS. For beyond small D especially use global optimization or a greedy algorithm if find a good one. Extend the model to use sparsity as for regular regression. Use the result for algorithm performance analysis (start with some sorting algorithms) with doubling input sizes.
- Research and implement elastic net.

27.15 References

- Anthony, M., & Bartlett, P. L. (2009). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.
- Berk, R. A. (2020). *Statistical Learning from a Regression Perspective*. Springer.
- Clarke, B., Fokoue, E., & Zhang, H. H. (2009). *Principles and Theory for Data Mining and Machine Learning*. Springer.
- Györfi, L., Kohler, M., Krzyzak, A., & Walk, H. (2002). *A Distribution-Free Theory of Nonparametric Regression*. Springer.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
- Hastie, T., Tibshirani, R., & Wainwright, M. (2015). *Statistical Learning with Sparsity: The Lasso and Generalizations*. CRC Press.
- Huber, P. J., & Ronchetti, E. M. (2009). *Robust Statistics*. Wiley.
- Liao, S. P., Lin, H. T., & Lin, C. B. (2002). A note on the decomposition methods for support vector regression. *Neural Computation*, 14(6), 1267–1281.
- Meyer, D., Leisch, F., & Hornik, K. (2003). The support vector machine under test. *Neurocomputing*, 55(1), 169–186.
- Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). *Foundations of Machine Learning*. MIT Press.
- Wikipedia (2013). Regression analysis. http://en.wikipedia.org/wiki/Regression_analysis. Accessed May 18, 2013.
- Wikipedia (2016a). Local regression. https://en.wikipedia.org/wiki/Local_regression. Accessed August 16, 2016.
- Wikipedia (2016b). Multivariate adaptive regression splines. https://en.wikipedia.org/wiki/Multivariate_adaptive_regression_splines. Accessed August 16, 2016.
- Wikipedia (2018). Simpson's paradox. https://en.wikipedia.org/wiki/Simpson%27s_paradox. Accessed February 18, 2018.
- Wilcox, R. R. (2016). *Introduction to Robust Estimation and Hypothesis Testing*. Academic Press.
- Witten, I. H., Frank, E., & Hall, M.A. (2016). *Data Mining: Practical Machine Learning Tools and Techniques*.

Morgan Kaufmann.

28 Machine Learning—Clustering

28.1 Introduction

This chapter describes implementations of major clustering algorithms, with several original algorithmic choices. Some of the methods are more theoretical, but still need to be discussed due to popularity in machine learning literature. The most interesting result is the k -medoids algorithm with k selection by the simplified silhouette metric as a general-purpose tool.

28.2 Setup

To humans, clustering in $D \leq 3$ tends to be easy visually by picking distinct shapes and concluding that some samples are similar based on the result. To an algorithm, even deciding if a set of examples forms one group or several is difficult because, unlike a human, it can't use prior knowledge to decide the nature of the data and guess the pattern.

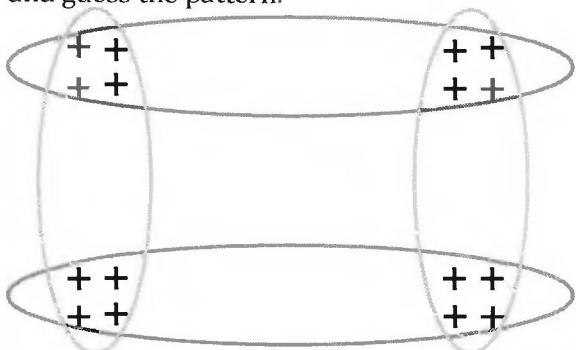


Figure 28.1: Don't know which of the two clusterings to prefer

So mathematically clustering is an ill-posed problem with no clear:

- Choice between alternative groupings suggested by different features—e.g., by height or by weight. Choose based on domain knowledge. Various A assume specific models where the problem is well-posed, but such models needn't be a good fit for the data. In particular, many A assume a domain-specific distance function d .
- Guidance for selecting the number of classes k . Usually maximize some performance measure, but the true k is rarely discovered and might not exist, so some clusters are split or joined.
- Way to select features. Similar heuristics as for picking k can be used with wrapper search, but the result has much more variance. Also, usually scale features to not give some an undue prior influence. Unlike for supervised tasks, need much domain knowledge to select good features.

Though enough model assumptions are made to make clustering with a chosen model well-posed, still have instability in the found clusters in that sometimes one promising solution is found and sometimes the other one (as in the above picture). Also, no A can discover clusters formed by separated subclusters such that there are other clusters in-between. In contrast, for supervised tasks separated data usually isn't a problem with strong models.

Assuming all information about a data set is available through d , no algorithm can satisfy all of **Kleinberg's axioms** (Shalev-Shwartz & Ben-David 2014):

- Scaling d leads to the same clustering
- Every partition is possible
- Changing d such that examples in the same cluster are closer and in the different clusters further leads to the same clustering

The last one, however, makes sense only for well-separated clusters (in the below picture this isn't the case). These are desiderata and not must-haves, though mildly related to approximation error.

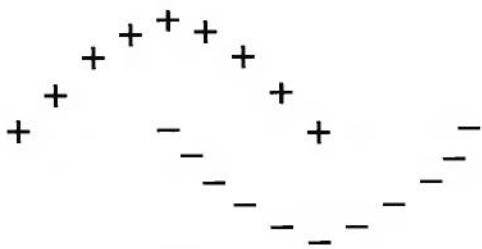


Figure 28.2: Clusters that aren't well-separated

Despite the problems, some A produce useful clusters from some data. A lot of unlabeled data is available, and discovering clusters in easily separable data automatically is very valuable. Clustering results have exploratory nature; one view is that they are simple or not interesting.

Clustering is very different from auto-encoding. Both are unsupervised, but the goal of clustering is not to reconstruct X but to group related x using a reconstruction distance.

Theoretically, seeing all data is equivalent to knowing its density. This still doesn't help because even if consider the modes of the PDF as clusters, it's still unclear what k is or how exactly to partition into k modes. All A need additional assumptions.

28.3 External Evaluation

Want to compare a clustering of labeled data to the labels. As for classification, all measures derive from a contingency matrix. The number of rows = k , and the number of columns = the number of data classes. Cell(row, column) = the number of examples with label = column, and cluster partition = row.

```
template<typename DATA> Matrix<int> clusterContingencyMatrix(
    Vector<int> const& assignments, DATA const& data)
{//row is assignment, column is label
    int n = assignments.getSize();
    assert(n == data.getSize());
    int k = valMax(assignments.getArray(), n) + 1;
    Matrix<int> counts(k, findNClasses(data));
    for(int i = 0; i < n; ++i)
        ++counts(assignments[i], data.getY(i));
    return counts;
}
```

An intuitive measure is accuracy that would result if the clustering would be used to predict the labels. A simple but not a great measure of this is **purity** = $\frac{\sum_r \max_c n_{rc}}{n}$. It assumes that the label of each cluster would be the majority label and weighs clusters by size.

```
double clusterPurity(Matrix<int> const& counts)
{
    int sum = 0, total = 0;
    for(int i = 0; i < counts.rows; ++i)
    {
        int maxi = 0;
        for(int j = 0; j < counts.columns; ++j)
        {
            total += counts(i, j);
            maxi = max(maxi, counts(i, j));
        }
        sum += maxi;
    }
    return sum * 1.0/total;
}
```

Purity doesn't pay attention to cluster identification, i.e., it doesn't explicitly penalize breaking up a label into multiple clusters. This is penalized implicitly only if k = the number of labels. For A that pick k automatically, purity is inappropriate because larger k usually have better purity by chance.

So purity isn't a good comparative measure, but a potentially good descriptive one because in some applications overestimating k costs much less than mixing clusters. In particular, if want data reduction, high purity signals success even if k is much overestimated; this may be important if the classes aren't easily

separable.

Can calculate classification error directly by a reduction to an assignment problem (see the “Graph Algorithms” chapter). This prevents needing to enumerate all assignments of classes to labels to find the best one. To allow the assignment solver with Djikstra shortest path search, the costs are adjusted to be nonnegative using $\text{cost}_{rc} = n - n_{rc}$.

```
double clusterClassificationAccuracy(Matrix<int> const& counts)
{
    int n = 0, sum = 0;
    for(int i = 0; i < counts.rows; ++i)
        for(int j = 0; j < counts.columns; ++j) n += counts(i, j);
    Vector<pair<pair<int, int>, double> allowedMatches;
    for(int i = 0; i < counts.rows; ++i)
        for(int j = 0; j < counts.columns; ++j) allowedMatches.append(
            make_pair(make_pair(i, counts.rows + j), n - counts(i, j)));
    Vector<pair<int, int>> matches = assignmentProblem(counts.rows,
        counts.columns, allowedMatches);
    assert(matches.getSize() == min(counts.rows, counts.columns));
    for(int i = 0; i < matches.getSize(); ++i)
        sum += counts(matches[i].first, matches[i].second - counts.rows);
    return sum * 1.0/n;
}
```

Classification error has some flaws (Aggarwal & Reddy 2014), so it's best to not use it as the primary comparison metric. In particular, its intuitive meaning becomes unclear if the estimated k doesn't match the number of labels. But it's also a good descriptive measure.

A different approach is the **Rand index** = the fraction of all possible example pairs that is the same in both clusterings; a pair is the same if both examples belong to either the same or different clusters. It can be

computed from the contingency matrix (Aggarwal & Reddy 2014) as $1 - \frac{m_1 + m_2 - \sum_{rc} \binom{n_{rc}}{2}}{M}$, where $m_1 = \sum_r \binom{\sum_c n_{rc}}{2}$, $m_2 = \sum_c \binom{\sum_r n_{rc}}{2}$, and $M = \binom{n}{2}$. The value $\in [0, 1]$.

Though Rand is a useful measure, the pairs that are different tend to dominate the value—in particular with large k the expected value is high. So normalize, creating the **adjusted Rand index** = $\frac{R - E[R]}{1 - E[R]}$. $E[R]$ is computed assuming the row and the column sums are constant, and that the cell counts \sim a hypergeometric such that $E\left[\sum_{rc} \binom{n_{rc}}{2}\right] = \frac{m_1 m_2}{M}$. Other components of the formula for R remain constant, so, after simplifying the math, adjusted Rand = $\frac{\sum \sum \binom{n_{rc}}{2} - \frac{m_1 m_2}{M}}{(m_1 + m_2)/2 - m_1 m_2/M}$. The upper bound is still 1, but the value can be

below 0 (random clustering) or even -1.

```
double nChoose2(int n){return n * (n - 1)/2;}
double AdjustedRandIndex(Matrix<int> const& counts)
{
    int total = 0, SumRC2 = 0, SumR2 = 0, SumC2 = 0;
    for(int i = 0; i < counts.rows; ++i)
    {
        int sumI = 0;
        for(int j = 0; j < counts.columns; ++j)
        {
            int c = counts(i, j);
            total += c;
            sumI += c;
            SumRC2 += nChoose2(c);
        }
        SumR2 += nChoose2(sumI);
    }
    for(int j = 0; j < counts.columns; ++j)
```

```

    {
        int sumJ = 0;
        for(int i = 0; i < counts.rows; ++i) sumJ += counts(i, j);
        SumC2 += nChoose2(sumJ);
    }
    double EV = SumR2 * SumC2 * 1.0/nChoose2(total);
    return (SumRC2 - EV)/(0.5 * (SumR2 + SumC2) - EV);
}

```

$E[\text{adjusted Rand}]$ is asymptotically invariant in k (Hennig et al. 2016), which, despite wanting nonasymptotic invariance, enables comparison of clusterings with different k . Intuitively, for a fixed k the adjustment is a linear transformation because the expected value is constant, but \forall particular k it's a different linear transformation.

Adjusted Rand is good a comparative and a good descriptive metric. For interpretation, the following rule of thumb seems useful: a value ≥ 0.8 corresponds to grade A, $0.6 \leq \text{value} < 0.8$ to B, etc.

Another interesting measure is accuracy of a classifier, trained from a clustering, on another test set. This is probably most useful with random forest to minimize the bias of introducing a classifier. It may also be useful to train a decision tree, and manually inspect it to try to get an idea of the logic, or can just count its number of nodes.

```

template<typename CLASSIFIER, typename DATA> double findTestCAcc(
    DATA const& train, Vector<int> assignments, DATA const& test)
{
    assert(train.getSize() == assignments.getSize());
    RelabeledData<DATA> rd(train);
    rd.labels = assignments;
    CLASSIFIER c(rd);
    Vector<int> assignmentsTest(test.getSize());
    for(int i = 0; i < test.getSize(); ++i)
        assignmentsTest[i] = c.predict(test.getX(i));
    return clusterClassificationAccuracy(
        clusterContingencyMatrix(assignmentsTest, test));
}

```

A clustering can be arbitrary but learnable, such as split along a single variable, so this may be a good measure to detect such situation.

28.4 Internal Evaluation and Selecting the Number of Clusters

On new data labeled examples for external evaluation are unavailable. So compute statistics that give hints about clustering quality. The most obvious one is generic MDL. For probabilistic models, $\text{MDL} = \text{the number of bits to encode the data} + \text{the number of bits to encode the parameters} = \text{constant} + \log(\text{the likelihood of data}) + \text{the number of estimated parameters} \times \text{the average number of bits per parameter}$ (constant because likelihood is only proportional to the probability of the data given parameters). Because in the best case of averaging, parameters converge at the rate $O(1/\sqrt{n})$ by the CLT, need about $\lg(n)/2$ bits per parameter. Expressing bit size in nats, $2(\text{MDL} - \text{the constant}) = \text{the BIC statistic} = 2LL + p\ln(n)$, where LL = the log likelihood, and p = the number of estimated parameters. But BIC is only available for probabilistic models and not necessarily the best-performing measure.

Though often know the number of clusters for many domains (e.g., \exists ten digits), want to be able to discover it automatically. Given a quality-measure statistic, such as BIC, that properly penalizes larger estimated k , a generic approach is to start with $k = 2$ and increase while the statistics gets better, up to a safety limit such as \sqrt{n} .

The **silhouette index** (Aggarwal & Reddy 2014) is a general measure = the average silhouette over all examples. Here, silhouette = $\frac{b-a}{\max(b,a)}$, where a = the average distance of the example to other examples in its cluster, and b = \min_c the average distance of the example to the examples in class c that doesn't contain the example.

```

template<typename DATA, typename DISTANCE> double clusterSilhouette(
    DATA const& data, Vector<int> const& assignments, DISTANCE const& d)
{
    int n = assignments.getSize();
    assert(n > 0);
    int k = valMax(assignments.getArray(), n) + 1;

```

```

double sum = 0;
for(int i = 0; i < n; ++i)
{
    int c = assignments[i];
    Vector<double> ds(k);
    Vector<int> sizes(k);
    for(int j = 0; j < n; ++j) if(i != j)
    {
        int c2 = assignments[j];
        ++sizes[c2];
        ds[c2] += d(data.getX(i), data.getX(j));
    }
    for(int j = 0; j < k; ++j) if(sizes[j]) ds[j] /= sizes[j];
    double ai = ds[c], bi = numeric_limits<double>::infinity();
    for(int j = 0; j < k; ++j) if(j != c) bi = min(bi, ds[j]);
    sum += (bi - ai)/max(bi, ai);
}
return sum/n;
}

```

But it's expensive to compute, needing $O(n^2k)$ time. So usually use the **simplified silhouette**, which is based on distances to selected representative points of clusters instead of the averages. Though considered not general enough (Kauffman & Rousseeuw 1990), it's useful for representative-based algorithms. If d is Euclidean distance, $\forall x \notin$ cluster with centroid c formed by x_i , $\sum d(x, x_i)^2 = n_c(d(x, c) + \sum d(c, x_i)^2)$. Proof: For 1D, $\sum d(x - x_i)^2 = nc \left(x_i^2 + \frac{\sum x_i^2}{nc} - 2 xc \right) = nc \left((x - c)^2 + \frac{\sum x_i^2}{nc} - c^2 \right) = nc((x - c)^2 + \text{Var}(x_i))$; generalizes to $D > 1$ because squared Euclidean distance is additive \square . So the simplification effectively ignores the variance of the cluster; the same interpretation should apply for other d where such decomposition doesn't hold. Experiments in Vendramin et al. (2010) show that the performance of the simplified version is essentially equivalent.

```

template<typename DATA, typename DISTANCE, typename REPS>
double clusterSimplifiedSilhouette(DATA const& data,
    Vector<int> const& assignments, REPS const& r, DISTANCE const & d)
{
    int n = assignments.getSize();
    assert(n > 0);
    int k = valMax(assignments.getArray(), n) + 1;
    double sum = 0;
    for(int i = 0; i < n; ++i)
    {
        int c = assignments[i];
        double ai = d(data.getX(i), r[c]),
            bi = numeric_limits<double>::infinity();
        for(int j = 0; j < k; ++j) if(j != c)
            bi = min(bi, d(data.getX(i), r[j]));
        sum += (bi - ai)/max(bi, ai);
    }
    return sum/n;
}
template<typename DATA> double clusterSimplifiedSilhouetteL2(DATA const& data,
    Vector<int> const& assignments)
{
    int k = valMax(assignments.getArray(), assignments.getSize()) + 1;
    return clusterSimplifiedSilhouette(data, assignments, findCentroids(data,
        k, assignments), EuclideanDistance<NUMERIC_X>::Distance());
}

```

For the result of a clustering it's useful to have both the assignments and the value of an internal index. Computing the latter is usually much less expensive if A and the index are paired properly. If not, the index value is set to ∞ and computed later.

```

struct ClusterResult
{

```

```

    Vector<int> assignments;
    double comparableInternalIndex;
    ClusterResult(Vector<int> const& theAssignments, double theCIP =
        numeric_limits<double>::infinity()): assignments(theAssignments),
        comparableInternalIndex(theCIP) {}
};

This allows estimating  $k$ :
```

1. Start with $k = 2$
2. Until reach max $k = \sqrt{n}$
3. Cluster with current k , and compute the value of the index
4. If got a worse value, return the previous clustering

```

template<typename CLUSTERER, typename DATA, typename PARAMS> ClusterResult
    findClustersAndK(DATA const& data, CLUSTERER const& c, PARAMS const& p,
    int maxK = -1)
{
    if(maxK == -1) maxK = sqrt(data.getSize());
    Vector<int> dummy;
    ClusterResult best(dummy);
    for(int k = 2; k <= maxK; ++k)
    {
        ClusterResult result = c(data, k, p);
        if(isfinite(result.comparableInternalIndex) &&
            result.comparableInternalIndex < best.comparableInternalIndex)
            best = result;
        else break;
    }
    return best;
}
template<typename CLUSTERER, typename PARAMS = EMPTY> struct FindKClusterer
{
    CLUSTERER c;
    PARAMS p;
    FindKClusterer(PARAMS const& theP = PARAMS()): p(theP) {}
    template<typename DATA> ClusterResult operator()(DATA const& data, int k)
        const{return c(data, k, p);}
    template<typename DATA> ClusterResult operator()(DATA const& data) const
        {return findClustersAndK(data, c, p);}
};

```

Though parameters aren't usually passed to clustering A , it's useful to be able to do so in general. As for supervised tasks, the following wrapper allows dummy parameters:

```

template<typename CLUSTERER> struct NoParamsClusterer
{
    CLUSTERER c;
    template<typename DATA> ClusterResult operator()(DATA const& data, int k,
        EMPTY const& p) const{return c(data, k);}
    template<typename DATA> ClusterResult operator()(DATA const& data,
        EMPTY const& p) const{return c(data);}
};

```

Technically can use internal indices to pick values of other parameters and perhaps even do wrapper search for feature selection, but this risks overfitting due to doing too much work with the same data. A criticism of silhouette is that its expected value increases in D (Tomašev & Radovanović 2016), so wrapper search using simplified silhouette is questionable.

28.5 Calculating Stability

The general stability algorithm based on loss functions doesn't apply here because even for reasonable losses, such as the k -means one, it's unclear how to form the main predictor.

The basic method using clustering algorithm A (Von Luxburg 2010):

1. B times for maybe $B = 100$
2. Cluster two bootstrap resamples of the data

3. Calculate external index using samples common to both

4. Stability = the average of the indices

Beware that **instability** ($= 1 - \text{stability}$) naturally increases with k , so its interpretation may be difficult; some normalization methods have been proposed but are questionable. So adjusted Rand seems to be a good external index because of its adjustment.

```
Matrix<int> clusterOnlyContingencyMatrix(Vector<int> const& assignments1,
                                         Vector<int> const& assignments2)
{
    int n = assignments1.getSize();
    assert(n == assignments2.getSize());
    int k1 = valMax(assignments1.getArray(), n) + 1,
        k2 = valMax(assignments2.getArray(), n) + 1;
    Matrix<int> counts(k1, k2);
    for(int i = 0; i < n; ++i) ++counts(assignments1[i], assignments2[i]);
    return counts;
}

template<typename CLUSTERER, typename PARAMS, typename DATA> double
findStability(CLUSTERER const &c, PARAMS const& p, DATA const& data,
              int k = -1, int B = 100)
{
    double sum = 0;
    int n = data.getSize();
    for(int j = 0; j < B; ++j)
    {
        //draw and cluster bootstraps
        Vector<int> assignments[2] = {Vector<int>(n, -1), Vector<int>(n, -1)};
        for(int l = 0; l < 2; ++l)
        {
            PermutatedData<DATA> dataP(data);
            for(int i = 0; i < n; ++i) dataP.addIndex(GlobalRNG().mod(n));
            Vector<int> pAssignments = k == -1 ? c(dataP, p).assignments :
                c(dataP, k, p).assignments;
            for(int i = 0; i < n; ++i)
                assignments[l][dataP.permutation[i]] = pAssignments[i];
        }
        //compute and score intersection
        for(int i = n - 1; i >= 0; --i)
            if(assignments[0][i] == -1 || assignments[1][i] == -1)
                for(int l = 0; l < 2; ++l)
                {
                    assignments[l][i] = assignments[l].lastItem();
                    assignments[l].removeLast();
                }
        //need > 1 examples to avoid NaN; should have many more
        if(assignments[0].getSize() > 1) sum += AdjustedRandIndex(
            clusterOnlyContingencyMatrix(assignments[0], assignments[1]));
    }
    return sum/B;
}
```

Perhaps a must-have theoretical requirement \forall good A is convergence in stability—i.e., \forall data set and a reasonable definition of stability, as $n \rightarrow \infty$, stability of A converges to a fixed value. This is probably the best that can hope for because, e.g., an algorithm can be forced to randomly alternate between two equally good clusterings as more data is added and be unstable, but the stability itself should converge. Stability is a property of both A and the data set.

This definition of stability is in some sense more correct than the general one because it takes structure into account, and the main predictor doesn't—e.g., for classification very different partitions can have the same accuracy.

Using stability to pick k and possibly other parameters is interesting, but potentially very slow. Also (Von Luxburg 2010):

- \exists theoretical guarantees or extensive experimental evidence of good performance
- If A is unstable, e.g., when its assumptions are substantially wrong, the results are unpredictable

So stability seems to be just a useful quality metric.

28.6 Clustering in Euclidean Space

Given k and vector space X , k -means is simple and efficient:

1. Pick the initial clusters
2. Until converge or reach some iteration limit
3. Calculate centroids of the current clusters
4. Assign each point to the cluster of the closest centroid

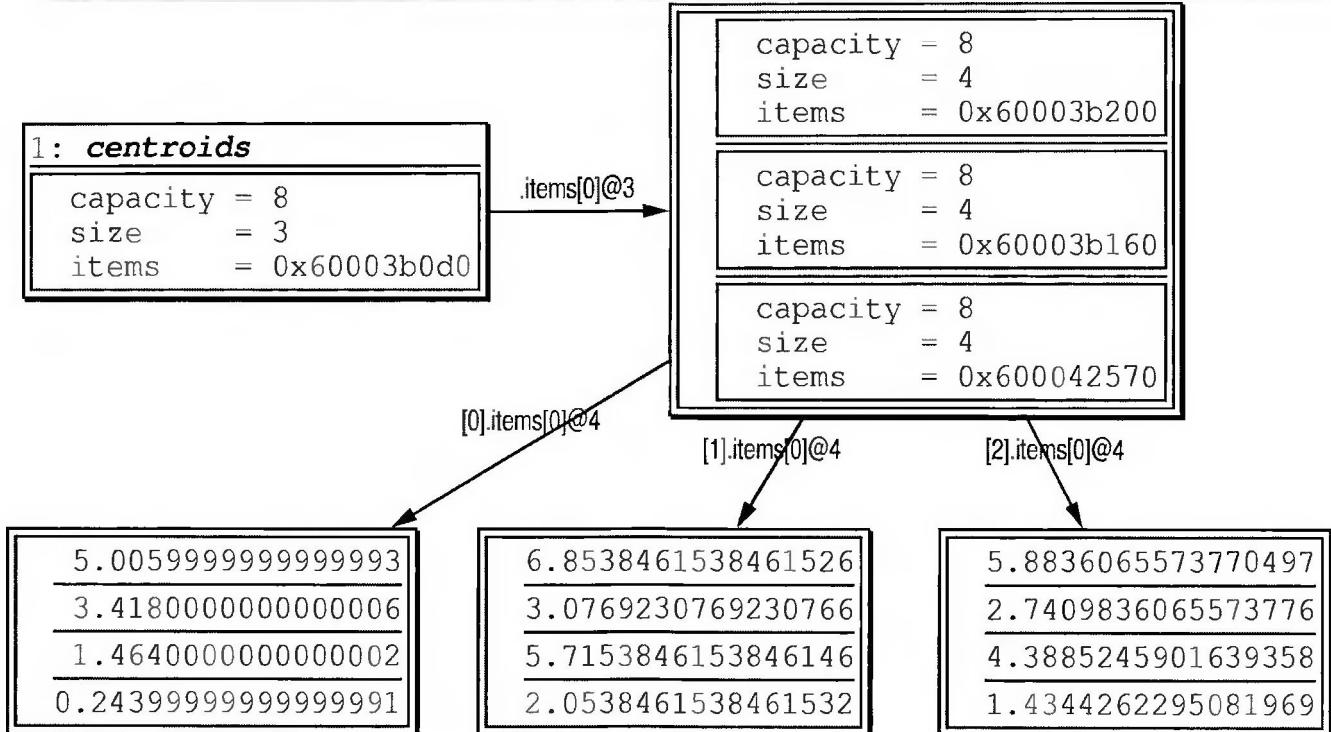


Figure 28.3: Memory layout of a k -means clustering on the iris data

A good initialization is particularly important because the optimization done by clustering algorithms is local. Finding optimal centroids is equivalent to NP-complete facility location, for which k -means is a good local search—the objective never increases (but technically k -means may not find a local minimum if have equal distances; Shalev-Shwartz & Ben-David 2014). An approximation algorithm with $E[O(\ln(k))]$ approximation ratio is effective for initialization (Arthur & Vassilvitskii 2007). The idea is to pick centroids that are different from each other:

1. Pick a random sample as the first centroid
2. $k - 1$ times
3. Pick a sample with relative probability $\min_i d(x_i, c_i)^2$ as the next centroid; c_i is any already computed centroid

It applies to metric d and guarantees the same expected approximation ratio (up to a constant). The only change in computation is that don't need to square distances.

Use VP tree nearest neighbor search to assign each example to the nearest centroid. Based on my experiments, it's slightly faster than k -d tree despite not needing to compute the square roots in the latter. Model selection uses simplified silhouette.

```
template<typename DATA, typename DISTANCE> Vector<int>
findKMeansPPCentroids(DATA const& data, int k, DISTANCE const& d,
bool isMetric = false)
{//approximation algorithm to initialize centroids
    int n = data.getSize();
    assert(n > 0 && k <= n);
    Vector<double> closestDistances(n, numeric_limits<double>::infinity());
    Vector<int> centroids(1, GlobalRNG().mod(n));
    for(int i = 1; i < k; ++i)
        {//recompute closest center distances
            for(int j = 0; j < n; ++j)
                closestDistances[j] = min(closestDistances[j],
                    d(data.getX(j), data.getX(centroids.lastItem())));
        }
    //sample next center in proportion to squared closest distance
}
```

```

Vector<double> probs(n);
for(int j = 0; j < n; ++j)
{
    probs[j] = closestDistances[j];
    if(!isMetric) probs[j] *= closestDistances[j];
}
normalizeProbs(probs);
AliasMethod a(probs);
centroids.append(a.next());
}
return centroids;
}

template<typename DATA> Vector<typename DATA::X_TYPE> assemblePrototypes(
    DATA const& data, Vector<int> const& medoids)
//helper to get cluster centers from data and indices
{
    Vector<typename DATA::X_TYPE> result(medoids.getSize());
    for(int i = 0; i < medoids.getSize(); ++i)
        result[i] = data.getX(medoids[i]);
    return result;
}

template<typename DATA> Vector<NUMERIC_X> findCentroids(DATA const& data,
    int k, Vector<int> const& assignments)
{
    Vector<int> counts(k);
    Vector<NUMERIC_X> centroids(k, data.getX(0) * 0);
    for(int i = 0; i < data.getSize(); ++i)
    {
        ++counts[assignments[i]];
        centroids[assignments[i]] += data.getX(i);
    }
    for(int i = 0; i < k; ++i) centroids[i] *= 1.0/counts[i];
    return centroids;
}

struct KMeans
{
    typedef EuclideanDistance<NUMERIC_X>::Distance EUC_D;
    template<typename DATA> static bool findAssignments(DATA const& data,
        Vector<NUMERIC_X> const& centroids, Vector<int>& assignments)
    //assign all examples to their nearest centroids
    {
        VpTree<NUMERIC_X, int, EUC_D> t;
        bool converged = true;
        for(int i = 0; i < centroids.getSize(); ++i) t.insert(centroids[i], i);
        //assign each point to the closest centroid
        for(int i = 0; i < data.getSize(); ++i)
        {
            int best = t.nearestNeighbor(data.getX(i))>value;
            if(best != assignments[i])
                //done if no assignment changed
                converged = false;
            assignments[i] = best;
        }
    }
    return converged;
}

template<typename DATA> ClusterResult operator()(DATA const& data,
    int k, int maxIterations = 1000) const
{
    assert(k > 0 && k <= data.getSize() && data.getSize() > 0);
    Vector<int> assignments(data.getSize());
    findAssignments(data, assemblePrototypes(data, findKMeansPPCentroids(
        data, k, EUC_D())), assignments);
    for(int m = 0; m < maxIterations; ++m) if(findAssignments(data,
        findCentroids(data, k, assignments), assignments)) break;
}

```

```

        return ClusterResult(assignments, -clusterSimplifiedSilhouetteL2(data,
            assignments));
    }
};

typedef FindKClusterer<NoParamsClusterer<KMeans>> KMeansGeneral;

Repeating several times (10 seems reasonable) until the objective function improves can be useful.

template<typename DATA> double kMeansSimpSil(DATA const& data,
    Vector<int> const& assignments)

{
    int n = assignments.getSize();
    assert(n > 0);
    int k = valMax(assignments.getArray(), n) + 1;
    Vector<NUMERIC_X> centroids = findCentroids(data, k, assignments);
    typename EuclideanDistance<NUMERIC_X>::DistanceIncremental d;
    double sum = 0;
    for(int i = 0; i < n; ++i)
        sum += d(data.getX(i), centroids[assignments[i]]);
    return sum;
}

struct RepeatedKMeans
{
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k, int maxIterations = 1000, int nRep = 10) const
    {
        KMeans km;
        ClusterResult best = km(data, k, maxIterations);
        double ss = kMeansSimpSil(data, best.assignments);
        while(--nRep)
        {
            ClusterResult result = km(data, k, maxIterations);
            double ssNew = kMeansSimpSil(data, result.assignments);
            if(ssNew < ss)
            {
                ss = ssNew;
                best = result;
            }
        }
        return best;
    }
};
typedef FindKClusterer<NoParamsClusterer<RepeatedKMeans>> RKMeansGeneral;

```

The most expensive operation in an iteration is finding the closest centroids. It takes expected $O(nD\lg(k))$ time assuming efficient nearest neighbor search, but could be $O(nDk)$. May need exponentially many iterations to converge (Russell & Norvig 2020), which is why limit the number of iterations to a large constant, which seems effective in practice. Smoothed complexity is polynomial (Arthur et al. 2011).

k -means suffers from high approximation error, in the same way as the nearest mean classifier. In particular, it can only distinguish well-separated shapes, but still not necessarily well because a small shape can take away support from a larger one:

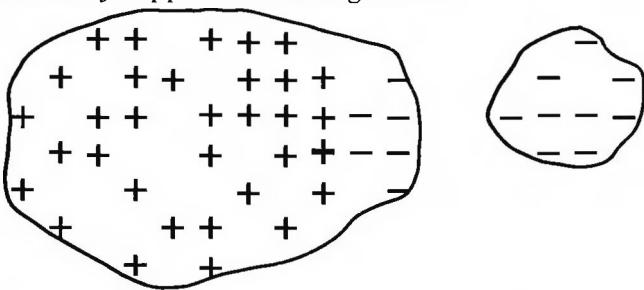


Figure 28.4: Shape effect on k -means—the larger group takes away some instances from the smaller one

28.7 Clustering in a Metric Space

Computing centroids needs a vector space. Extension to metric X needs modifications such as **k -medoids** (Hennig et al. 2016), where the representative points (called **medoids**) are picked from the data. Look for one of $\binom{n}{k}$ combinations of points such that the sum of distances of examples to their closest medoid is minimal.

Do local search for the best combination:

1. Start with some initial solution, here use **k -means++ initialization**
2. Until satisfy some termination criteria
3. Pick a medoid and a nonmedoid and exchange them if this decreases the objective

Picking pairs to exchange has been done in several ways:

- The original **PAM algorithm** (Kauffman & Rousseeuw 1990) considers all possible pairs, picks the best, and exchanges until convergence, which is expensive
- A more scalable method (Aggarwal & Reddy 2014) is to pick random pairs and exchange for some number of iterations

A better strategy is something in-between—consider all pairs formed by a random nonmedoid and all medoids. With proper caching of distances, this has about the same iteration cost as the random pair because the bottleneck operation is computing all distances from the picked nonmedoid to other samples, and the result is reused in considering all medoids. In particular, \forall example store its distance to every current medoid. This needs $O(nk)$ memory, which usually isn't much more than the minimum $O(n)$ for specifying the assignments.

When considering a pair, \forall example have two cases, depending on whether it's closer to the:

- Medoid—check all other medoids and the candidate to find the closest
- Candidate—the candidate is the closest

Use a random permutation to control the order in which nonmedoids are considered:

- Small n —converge if medoids didn't change after a complete pass through the permutation. In every pass, consider a different random permutation to diversify the local search.
- Large n —by default, 1000 iterations is the limit. By the coupon collector's problem, need $E[k \log(k)]$ examples to get all classes represented assuming balanced data, so for reasonable k 1000 should be enough, and the found medoids should be close enough to the true ones.

Use simplified silhouette to select k .

```
template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct KMedoids
{
    static bool isIMedoid(int i, Vector<int> const& medoids)
    {
        for(int j = 0; j < medoids.getSize(); ++j)
            if(medoids[j] == i) return true;
        return false;
    }
    template<typename DATA> ClusterResult operator()(DATA const& data, int k,
        int maxRounds = 1000) const
        {return findClusters(data, k, maxRounds).first;}
    template<typename DATA> static pair<ClusterResult, Vector<int>>
        findClusters(DATA const& data, int k, int maxRounds = 1000)
    //initialize current medoids
        int n = data.getSize();
        assert(k > 0 && k <= n && n > 0);
        DISTANCE d;
        Vector<int> perm(n), medoids(findKMeansPPCentroids(data, k, d, true)),
            assignments(n);
        Vector<Vector<double>> dCache(n, Vector<double>(k));
        for(int i = 0; i < n; ++i)
        //compute current assignments and cache the distances
            for(int j = 0; j < k; ++j)
                dCache[i][j] = d(data.getX(i), data.getX(medoids[j]));
        int best = argMin(dCache[i].getArray(), k);
        assignments[i] = best;
```

```

    }
    for(int i = 0; i < n; ++i) perm[i] = i;//initialize the permutation
    bool converged = false;
    while(!converged)
    {
        converged = true;
        GlobalRNG().randomPermutation(perm.getArray(), n);
        for(int i = 0; i < n && maxRounds > 0; ++i)
        {
            if(isIMedoid(perm[i], medoids)) continue;
            Vector<double> tempDs(n);
            for(int l = 0; l < n; ++l)
                tempDs[l] = d(data.getX(perm[i]), data.getX(l));
            int bestJ = -1;
            double bestDiff;
            for(int j = 0; j < k; ++j)
            {
                double DSumDiff = 0;
                for(int l = 0; l < n; ++l)
                {
                    double dOld = dCache[l][assignments[l]];
                    if(assignments[l] == j)
                    {
                        dCache[l][j] = tempDs[l];
                        DSumDiff += valMin(dCache[l].getArray(), k) - dOld;
                        dCache[l][j] = dOld;
                    }
                    else if(tempDs[l] < dOld) DSumDiff += tempDs[l] - dOld;
                }
                if(bestJ == -1 || DSumDiff < bestDiff)
                {
                    bestDiff = DSumDiff;
                    bestJ = j;
                }
            }
            if(bestDiff < 0)
            {
                converged = false;
                for(int l = 0; l < n; ++l)
                {
                    dCache[l][bestJ] = tempDs[l];
                    if(assignments[l] == bestJ) assignments[l] =
                        argMin(dCache[l].getArray(), k);
                    else if(tempDs[l] < dCache[l][assignments[l]])
                        assignments[l] = bestJ;
                }
                medoids[bestJ] = perm[i];
            }
            --maxRounds;
        }
    }
    return make_pair(ClusterResult(assignments,
        -clusterSimplifiedSilhouette(data, assignments,
        assemblePrototypes(data, medoids), d)), medoids);
};

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
using KMedGeneral = FindKClusterer<NoParamsClusterer<KMedoids<DISTANCE> > >;

```

As for k -means, repeating can be useful:

```

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct RepeatedKMedoids
{

```

```

template<typename DATA> static double kMedS(DATA const& data,
    Vector<int> const& assignments, Vector<int> const& medoids)
{
    int n = assignments.getSize();
    assert(n > 0);
    int k = valMax(assignments.getArray(), n) + 1;
    Vector<typename DATA::X_TYPE> m = assemblePrototypes(data, medoids);
    DISTANCE d;
    double sum = 0;
    for(int i = 0; i < n; ++i) sum += d(data.getX(i), m[assignments[i]]);
    return sum;
}

template<typename DATA> ClusterResult operator()(DATA const& data,
    int k, int maxRounds = 1000, int nRep = 10) const
{
    KMedoids<DISTANCE> km;
    pair<ClusterResult, Vector<int>> best =
        KMedoids<DISTANCE>::findClusters(data, k, maxRounds);
    double s = kMedS(data, best.first.assignments, best.second);
    while(--nRep)
    {
        pair<ClusterResult, Vector<int>> result =
            KMedoids<DISTANCE>::findClusters(data, k, maxRounds);
        double sNew = kMedS(data, result.first.assignments, result.second);
        if(sNew < s)
        {
            s = sNew;
            best.first = result.first;
        }
    }
    return best.first;
}
};

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
using RKMedGeneral =
    FindKClusterer<NoParamsClusterer<RepeatedKMedoids<DISTANCE> >>;

```

For very large n , where $O(nk)$ space to store the cache is unavailable, can store it on disk, and use an incremental algorithm such as SGD. I.e., read an IO page of examples with their associated distances, and try them all out one by one.

k -medoids gives a better understanding of the data compared to other A because the clusters are defined from particular examples, which usually can be easily inspected by a human. Arguably, this and the ability to work with general data makes k -medoids the first A to try, like decision tree for classification.

28.8 Spectral Clustering

Let W be a similarity matrix created from the training data, using kernels or some other method. Think of it as a weighted adjacency matrix of some graph. Its **degree matrix** D is the diagonal matrix where $D[i,i] = \sum_j W[i,j]$. The **graph Laplacian** $L = D - W$. This transformation is useful because if L has k connected components, each corresponding to a particular cluster, its k smallest eigenvalues are 0, and the corresponding eigenvectors are the indicator vectors for the rows that belong to each cluster (Aggarwal & Reddy 2014). So in this case they form a feature set that is better than the original one.

It's rare for clusters to be cleanly separated and the similarity-generating method to be exact for L to have k connected components. But if the components are interconnected by connections of small weights, taking these "first" k eigenvectors still forms a good feature set.

Though kernels are a familiar method to generate W , this generates a dense matrix and usually needs a parameter for the kernel. Can make W sparse by cutting off entries $< \max(W[i,i], W[j,j])\epsilon$ for $\epsilon = 0.001$ or so, but still need to select a parameter, and can have scale sensitivity. A simpler option is using a symmetric nearest neighbor generator where $W[i,j] = W[j,i]$ if i is one of j 's several neighbors or vice versa (both aren't necessarily true). The number of neighbors to check can be set to the default of the k -NN classifier (see the "Machine Learning—Classification" chapter).

The resulting algorithm also has certain tweaks:

1. Create W using nearest neighbors or some other method
2. Create normalized $L = I - D^{-0.5}WD^{-0.5}$ (note that $D^{-0.5}WD^{-0.5} \neq D^{-1}W$)
3. Find its k first eigenpairs
4. Create a new data set where the feature vector x for instance $i = \{\text{eigenvector}[j][i] \text{ for } 0 \leq j < k\}$, normalized to have $\|x\| = 1$
5. Use any clustering algorithm such as k -means on it

The symmetrization allows a more efficient eigenvalue solver (see the "Numerical Methods—Introduction and Matrix Algebra" chapter), and preserves the connected component property up to a linear transformation. The normalization of x protects against possible lack of scale of eigenvector components in the same coordinate (von Luxburg 2007).

When need to pick k , need a suitable index that is based on the final assignments. Among the presented indices, silhouette seems to be the best choice (simplified silhouette doesn't gain efficiency), so use it as a default. Its computation is much faster than the eigenvalue calculation. Note that using the index of the final clustering algorithm (k -means here) instead is wrong because it's working in a different feature space and deciding based on the latter; also selecting k eigenvectors solves a slightly different problem with each k , so can't use it for selecting k . Use sparse matrix algebra to compute L , though must convert to dense to get the eigenvectors.

```

template<typename DISTANCE> EuclideanDistance<NUMERIC_X>::Distance>
struct SpectralClusterer
{//eigenpairs and permutations for sorting
    typedef pair<pair<Vector<double>, Matrix<double> >, Vector<int> > EIGS;
    template<typename DATA> SparseMatrix<double> createLaplacian(
        DATA const& data) const
    {//setup kNN Laplacian
        int n = data.getSize(), nNeighbors = lgFloor(n)/2 + 1;//kNN default
        SparseMatrix<double> W(n, n);
        typedef VpTree<typename DATA::X_TYPE, int, DISTANCE> TREE;
        TREE tree;
        for(int i = 0; i < n; ++i) tree.insert(data.getX(i), i);
        for(int i = 0; i < n; ++i)
        {
            Vector<typename TREE::NodeType*> neighbors =
                tree.kNN(data.getX(i), nNeighbors + 1);
            for(int j = 0; j < neighbors.getSize(); ++j)
            {//first nn is usually self
                int l = neighbors[j]->value;
                if(l != i)
                {
                    W.set(i, l, 1);
                    W.set(l, i, 1);
                }
            }
        }
        return W;
    }
    EIGS findLaplacianEigs(SparseMatrix<double> const& W) const
    {//normalize
        int n = W.getRows();
        SparseMatrix<double> Dm05(n, n);
        for(int i = 0; i < n; ++i)
        {
            double di = 0;
            for(int j = 0; j < n; ++j) di += W(i, j);
            Dm05.set(i, i, 1/sqrt(di));
        }//find eigs
        EIGS eigs(QREigenSymmetric(toDense<double>(SparseMatrix<double>::
            identity(n) - Dm05 * W * Dm05)), Vector<int>(n));
        //sort the permutation
        for(int i = 0; i < n; ++i) eigs.second[i] = i;
        quickSort(eigs.second.getArray(), 0, n - 1,
            IndexComparator<double>(eigs.first.first.getArray()));
    }
}

```

```

    return eigs;
}
template<typename DATA> ClusterResult operator()(DATA const& data, int k,
EIGS const& eigs) const //to be called by k search
{//make new features
    int n = data.getSize();
    assert(k > 0 && k < n);
    InMemoryData<NUMERIC_X, int> data2;
    for(int i = 0; i < n; ++i)
    {//eigenvectors are rows
        Vector<double> x(k);
        for(int j = 0; j < k; ++j)
            x[j] = eigs.first.second(eigs.second[j], i);
        double xNorm = norm(x); //normalize
        if(xNorm > 0) x *= 1/xNorm;
        data2.addZ(x, 0);
    }//cluster
    RepeatedKMeans km;
    ClusterResult result = km(data2, k);
    result.comparableInternalIndex =
        -clusterSilhouette(data, result.assignments, DISTANCE());
    return result;
}
template<typename DATA> ClusterResult operator()(DATA const& data, int k)
const //if know k
{return operator()(data, k, findLaplacianEigs(createLaplacian(data)));}
template<typename DATA> ClusterResult operator()(DATA const& data) const
{//if don't know k
    return findClustersAndK(data, *this,
        findLaplacianEigs(createLaplacian(data)));
}
}
};

```

The algorithm is statistically consistent under some conditions when k is known (von Luxburg 2007). It can also be viewed as trying to cut the graph into k balanced connected components in an optimal way. Even approximating the optimal balanced cut up to a constant factor is NP-hard (von Luxburg 2007), so such heuristic makes sense. Because for large n (for my computer not too far from >5000) will run out of memory, mix the algorithm with repeated k -medoids to give good results for arbitrary distances.

```

template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct SpectralSmart
{
    RKMedGeneral<DISTANCE> km;
    SpectralClusterer<DISTANCE> s;
    template<typename DATA> bool useKMed(DATA const& data) const
        (return data.getSize() > 5000;)//for memory
        //feasibility + efficiency
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k) const
    {
        if(useKMed(data)) return km(data, k);
        else return s(data, k);
    }
    template<typename DATA> ClusterResult operator()(DATA const& data) const
    {
        if(useKMed(data)) return km(data);
        return s(data);
    }
};

```

28.9 Experiments

The comparisons are based on the training sets of 15 different data sets (the same ones as for classification). Due the exploratory nature and the difficulty of clustering, the results are more of a proof-of-concept. Average Friedman ranks are reported. Randomized algorithms have been run 5 times.

	kMeansPP01	kMed01	kMed01R10	
A_Rand	3	3.07	3.13	2.87
Stability	2.93	1.27	3.07	2.73 N/A

Figure 28.5: Performance comparison with known k

For guessed k adjusted Rand seems to be better a measure than the relative error in found k because, e.g., an overestimate is usually better than an underestimate.

	kMeansPP01Gap	kMeansPPSimpSil	kMeansPPR10SimpSil	kMed01SimpSil	kMed01R10SimpSil	Spectral
A_Rand	3.67	3.4	3.6	3.2	2.87	3.2
Stability	4.47	2.73	1.4	3.4	3 N/A	
AbsKDiff	2.6	2.67	2.33	2.73	2.47	4.3

Figure 28.6: Performance comparison with guessed k (the gap is discussed in the comments)

When k is known, spectral with k -median switch-over for large n is a clear winner, and works with arbitrary distances. When k isn't known, this is also the first option to try—though k -median does better from the table, spectral wins for certain clean data sets such as the digits.

Considering all attributes, it seems that currently k -means, k -medians, and their repeated versions are safe to use on all problems. Others need stability evaluation, which may be too expensive, unless already have domain knowledge of good performance. An interesting idea is to consider other A only when evaluating their stability is fast enough. Repeated k -means is almost always the most stable one and far less often the best one, so this strategy needs more research.

28.10 Implementation Notes

The textbooks Aggarwal & Reddy (2014) and Hennig et al. (2016) cover most of the algorithms I implemented but not all. Some were only available from the primary literature. Algorithm selection was the main difficulty, in both evaluation and the actual clustering.

The implementation of k -median is original, though a somewhat obvious golden middle between more extreme alternatives of more or less work per iteration. It also uses memory very well and it cache-efficient.

Many algorithms I implemented and discarded when first starting to research the topic because many seem promising at first, but don't do reasonably well on the easy digits data which I used as a passing benchmark.

28.11 Comments

Clustering is sometimes suggested to gather data for training a classifier. This seems appealing, but may be misleading because a classifier can pick up any pattern that is enough to separate the data, however silly it is.

Adjusted Rand tends to outperform many other metrics, particularly those that also look at pairs (Aggarwal & Reddy 2014). It has been heavily tested and used since being introduced in 1985, and its behavior is reasonably understood. Perhaps information metrics such as **normalized mutual information** are the only potential competitor (Aggarwal & Reddy 2014; Hennig et al. 2016; Vinh et al. 2010), but need further research due to some issues:

- Unnormalized mutual information is flawed (Aggarwal & Reddy 2014). But the sum normalization that fixes it makes it equivalent to a particular normalization of **variation of information**, which is claimed to perform well without normalization. It's suspicious for a measure to be fixed by a normalization.
- The sum normalization has a less tight denominator bound than the minimum one (Vinh et al. 2010), but it's unclear if the latter has the same good properties.
- Both of the above normalizations use the minimum value instead of the expected value, unlike adjusted Rand, because presumably the latter is hard to calculate (Aggarwal & Reddy 2014). But the calculations under the same hypergeometric model have been done (Vinh et al. 2010), and it's unclear which is better.

A recommended measure (Aggarwal & Reddy 2014) is **Van Dongen distance** = 1 – the average of purity and purity(contingency^T). Despite satisfying some good properties when normalized, it loses information by not paying attention to nonmaximum contingency entries (Aggarwal & Reddy 2014; Hennig et al. 2016).

Purity(contingency^T) may be a useful descriptive metric, but it doesn't appear to be explored. For descriptive measures like purity and classification error, it's interesting to consider balancing by labeled class sizes, though, unlike for classification, this doesn't seem to have been explored. Balancing by discovered cluster size instead makes no sense because accidentally discovered small clusters will have too much impact.

\exists several methods for stability evaluation (Von Luxburg 2010; Hennig et al. 2016). A slightly simpler and more efficient bootstrap strategy is comparing the clustering of the whole data to clusterings of bootstraps, but this can have higher variance because the clustering with all the data can too.

A simple but flawed way to measure stability is doing several runs and calculating variance of some quality measure. This only captures variance due to randomization in A .

Using classification accuracy instead of adjusted Rand for stability is claimed to have some theoretical advantages (Von Luxburg 2010), but adjusting for chance seems more important.

For k -means the VP tree implementation is slightly slower than **Hamerly's algorithm** (the simplest among several triangle-inequality methods; Hamerly & Drake 2015; Drake 2013), but not enough to justify the complication of implementing the latter (unless don't have a VP tree implementation).

k -medoids, designed with robustness in mind (Kauffman & Rousseeuw 1990), actually isn't robust (Hennig et al. 2016). But need experimental validation to decide if robust variants are practical, i.e., sufficiently improve performance with little extra computational cost. k -means based on medians (called k -medians; sometimes mistakenly refers to k -medoids) is one option, but \exists some others (Hennig et al. 2016).

For k -means many other initialization methods have been proposed and compared (Celebi et al. 2013). k -means++ initialization beats random initialization; some more complicated methods perform slightly better but don't have the expected approximation ratio guarantee. Also, k -means++ initialization works well with repetition.

One academically popular algorithm that I find to be not useful is **EM clustering**. It tries to reduce the approximation error of k -means by fitting a multidimensional Gaussian to every cluster. Then express the likelihood of the data as their mixture according to k mixture probabilities. To compute the parameters use the **expectation maximization (EM) algorithm**. It iteratively computes the likelihood and the mixture/Gaussian parameters until convergence, similarly to numerical fixed point iteration (Gupta & Chen 2010; Aggarwal & Reddy 2014). Some concerns:

- A single iteration costs $O(k(n+D)D^2)$ time, which is factor $O(D)$ more expensive than for k -means. So even $D > 100$ might get infeasible, and also estimating large- D covariance matrices is known to be problematic.
- Performance studies in the literature aren't promising. EM performed similarly to k -means on the data on simulated data (Steinley & Brusco 2011a), and speech data (Kinnunen et al. 2011). On some specific data EM is much better (Vermunt 2011), but such cases tend to be unusual (Steinley & Brusco 2011b). Poor performance of guessing k via BIC has been observed in Steinley & Brusco (2011a). Perhaps the multidimensional Gaussian model itself is questionable for practical data sets.

Soft/fuzzy **c -means** (Aggarwal & Reddy 2014) is an extension of k -means where cluster membership is probabilistic, like for EM. It's not considered further because:

- c -means is factor $O(k)$ slower than k -means. Also, detecting convergence is harder because have continuous parameters.
- Some studies comparing c -means and k -means report mixed results and are flawed by using few data sets, improper comparison metrics, a single run, simulated data, or random k -means initialization.
- Fuzzy clustering has specialized internal evaluation metrics, and k -means++ technically doesn't apply to it.

For spectral clustering, with large sparse matrices **implicitly restarted Lanczos iteration** (see the "Numerical Methods—Introduction and Matrix Algebra" chapter) is a good solution. Von Luxburg (2007) suggests that this may be a suitable approximation, but this isn't implemented here.

The normalized symmetric Laplacian algorithm isn't the only option. The other main algorithm has some theoretical advantages (von Luxburg 2007), but is much less popular and not considered further.

An interesting statistic for picking k for vector space data is the **gap** (Tibshirani 2001; Hastie et al. 2009). Consider deciding whether a clustering is better than random. Need a null hypothesis that describes what exactly is random. Here use random position data—assuming some distribution on the vector space, such as multidimensional uniform in the range of the data, generate and cluster many random samples to derive the null distribution.

So with k classes, $\text{gap}_k = E[\text{statistic of clustering on random data}] - \text{statistic of clustering on the data}$. The

standard deviation of the random data statistic is also computed as $s_k \sqrt{1 + \frac{1}{B}}$, where did B simulations to compute the expectation. The adjustment factor seems to be purely heuristic. $B = 20$ is used by the authors in their experiments (Hastie et al. 2009). Select k with the one standard deviation rule—pick the max k such that $\text{gap}_k > \text{gap}_{k+1} + s_{k+1}$.

The statistic used by the gap is $\log(W)$, where, given centroids c_j , $W = \sum_i n_i \sum_{\text{cluster}(i)=j} d(x_i, c_j)$ (Hastie et al. 2009). A more general noncentroid definition needs $O(n^2)$ time to compute. Use $\log(W)$ instead of W because of its better normality, so it seems that any statistic \sim reasonably normal can be used instead of $\log(W)$. This allows using the gap for other cases like EM clustering. It doesn't extend to metric space though due to needing the ability to sample.

A nice feature of the gap is being able to compare the no-clustering $k = 1$ against $k > 1$. But the multidimensional uniform is too random if the data features are highly correlated, and the underlying dimension is much smaller, which is likely for multidimensional data with many features. This is perhaps why in my tests simplified silhouette does better. It's also B times faster.

For picking k for spectral clustering an interesting alternative is the **eigengap heuristic** (von Luxburg 2007) where stop when the eigenvalues suddenly become larger. Unfortunately this doesn't define an algorithm. Some possibilities that I tested don't perform well (in all cases enforce $k \leq \sqrt{n}$):

1. The largest eigengap—suggested in several online sources, but even according to the examples in von Luxburg (2007) needn't pick the first eigengap
2. The first eigengap $>$ the next one
3. The first eigengap \geq the 95% eigengap
4. Find k using (1) but then find it again from $[1, k]$ using the index of the final clustering algorithm on k features

So need more research. Also the eigengap is a relative index, i.e., useful for only for selecting k whereas silhouette measures the quality of the assignment regardless of A or purpose.

Another way to implement EM is to enforce positive eigenvalues of the covariance matrix. Could perhaps bound their range to limit the condition number, or just require certain minimum small values. But the presented heuristics seems to work well.

Using kernels for clustering is generally problematic because without cross-validation don't have a good way to select parameters such as Gaussian width. Also, straightforward use of kernels, as in **kernel k-means**, leads to problems due to high nonlinearity—optimization by k -means local search is much less effective here (Sugiyama 2016). Spectral methods remedy this by simultaneously reducing dimension; they are also faster for sparse matrices. See Chitta (2015) for attempts to scale the optimization.

An interesting method is **hierarchical clustering**. Merging two groups with closest distance to each other is **single linkage**. The idea is to repeatedly merge two closest examples (or example groups) until have a single group. This can be implemented efficiently with an index heap, but still need nonscalable

$O(n^2 \lg(n))$ time and $O(n^2)$ memory due to needing to consider all $\binom{n}{2}$ example pairs in the first pass.

Though similar to nearest neighbor and able to discover clusters of arbitrarily shape, it suffers from **chaining**—noisy samples are likely to cause merging of unrelated clusters early (by the same logic it seems to not converge in stability). So, out of several possibilities, the best seems to be **average linkage**, i.e. average distance between groups. On document data average linkage performed better than other hierarchical methods but worse than k -means (Karypis et al. 2000). Hierarchical clustering tends to be most useful for visualizing clusterings of small data sets (by drawing the hierarchy; Hennig et al. 2016).

Another different approach is **density-based clustering**. Find connected dense patches of points, and declare them clusters, which is how humans cluster. The most representative and simplest A is **DBSCAN** (Ester et al. 1996). But my brief experiments with it gave poor results. Perhaps because it uses a single global ϵ to measure density, DBSCAN seems to make sense only when density doesn't vary much. Given that even with random sampling density tends to vary, it doesn't seem to be a robust approach.

Neural-network based **self-organizing map (SOM)** seems to have lost popularity; it also needs parameter tuning (Kinnunen et al. 2011).

See Aggarwal & Reddy (2014) for specialized methods for specific data types. Also, there and in Hennig et al. (2016) many other, less useful clustering methods are discussed. In particular:

- **Grid-based methods** are similar to density-based methods and partition X into a grid and try to find connected cells. But they have runtime exponential in D , are complicated to implement, and don't work in a metric X .
- Stream and “big data” methods tend to be overengineered to work exactly for the specific models

they follow. In practice, parallelizing well-established A with tools like Hadoop and Spark tends to be more popular. E.g., k -means is easily adapted, though the initialization needs some care (Bahmani et al. 2012).

- Ensemble-based methods don't work in the same way as for classification because can't form the main predictor. Even using classifiers on the clustering results doesn't help because it's unclear how to combine the latter. Instead, find a **consensus solution**, usually such that it has maximum stability with respect to other solutions and in some sense is their "centroid". For stability calculation find average similarity of every base clustering with respect to every other, which works because all base A cluster the same data. Can combine diverse algorithms this way in something similar to the majority vote for classification, but the advantages aren't particularly clear, and need further experimentation.

Extensive experimental comparison of clustering methods hasn't been done yet.

28.12 Projects

- For spectral check if there is any performance difference between regular and simplified silhouette.
- For spectral try implicitly restarted Lanczos to calculate the eigenvectors.

28.13 References

- Aggarwal, C. C., & Reddy, C. K. (Eds.). (2014). *Data Clustering: Algorithms and Applications*. CRC.
- Arthur, D., Manthey, B., & Röglin, H. (2011). Smoothed analysis of the k-means method. *Journal of the ACM (JACM)*, 58(5), 19.
- Arthur, D., & Vassilvitskii, S. (2007). k -means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 1027–1035). SIAM.
- Bahmani, B., Moseley, B., Vattani, A., Kumar, R., & Vassilvitskii, S. (2012). Scalable k -means++. *Proceedings of the VLDB Endowment*, 5(7), 622-633.
- Bouveyron, C., & Brunet-Saumard, C. (2014). Model-based clustering of high-dimensional data: A review. *Computational Statistics & Data Analysis*, 71, 52-78.
- Celebi, M. E., Kingravi, H. A., & Vela, P. A. (2013). A comparative study of efficient initialization methods for the k -means clustering algorithm. *Expert Systems with Applications*, 40(1), 200-210.
- Chitta, R. (2015). *Kernel-based Clustering of Big Data* (Doctoral dissertation, Michigan State University).
- Drake, J. (2013). *Faster k-means Clustering*. Baylor University.
- Ester, M., Kriegel, H. P., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd* (Vol. 96, No. 34, pp. 226-231).
- Gupta, M. R., & Chen, Y. (2010). *Theory and Use of the EM Algorithm*. Now Publishers.
- Hamerly, G., & Drake, J. (2015). Accelerating Lloyd's algorithm for k -means clustering. In *Partitional Clustering Algorithms* (pp. 41-78). Springer.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
- Hennig, C., Meila, M., Murtagh, F., & Rocci, R. (Eds.). (2016). *Handbook of Cluster Analysis*. CRC.
- Karypis, M. S. G., Kumar, V., & Steinbach, M. (2000). A comparison of document clustering techniques. In *TextMining Workshop at KDD2000*.
- Kauffman, L., & Rousseeuw, P. J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley.
- Kinnunen, T., Sidoroff, I., Tuononen, M., & Fräntti, P. (2011). Comparison of clustering methods: A case study of text-independent speaker modeling. *Pattern Recognition Letters*, 32(13), 1604-1617.
- Russell, S. J., Norvig, P. (2020). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- Sugiyama, M. (2016). *Introduction to Statistical Machine Learning*. Morgan Kaufmann.
- Steinley, D., & Brusco, M. J. (2011a). Evaluating mixture modeling for clustering: recommendations and cautions. *Psychological Methods*, 16(1), 63-78.
- Steinley, D., & Brusco, M. J. (2011b). K-Means Clustering and Mixture Model Clustering: Reply to McLachlan (2011) and Vermunt (2011). *Psychological Methods*, 16(1), 89-92.
- Tibshirani, R., Walther, G., & Hastie, T. (2001). Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2), 411-423.
- Tomašev, N., & Radovanović, M. (2016). Clustering Evaluation in High-Dimensional Data. In *Unsupervised Learning Algorithms* (pp. 71-107). Springer.
- Vendramin, L., Campello, R. J., & Hruschka, E. R. (2010). Relative clustering validity criteria: A comparative overview. *Statistical Analysis and Data Mining*, 3(4), 209-235.

- Vermunt, J. K. (2011). K-means may perform as well as mixture model clustering but may also be much worse: Comment on Steinley and Brusco (2011).
- Vinh, N. X., Epps, J., & Bailey, J. (2010). Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research*, 11(Oct), 2837-2854.
- von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and computing*, 17(4), 395-416.
- von Luxburg, U. (2010). *Clustering Stability*. Now Publishers.

29 Machine Learning—Other Tasks

29.1 Introduction

This chapter briefly reviews machine learning tasks that weren't discussed in the previous chapters. The main one is reinforcement learning, which has many books written about it, but useful practical applications are limited.

29.2 Reinforcement Learning

Want to find a **value-maximizing move picking policy** for an agent that moves in some **state space** and receives **reward(state)** on entering a state. The **value** of a state = $E[\text{the reward} + \text{the value of the state into which will move next}] = E[\sum \text{rewards over the picked sequence of states starting from the state}]$. An **episode** is sequence that leads to a terminal state. Rewards are **discounted**, so that reward r k states away is worth ry^k for $0 < y \leq 1$. For nonepisodic state spaces, $y < 1$ prevents $\sum \text{rewards} \rightarrow \infty$. This models many problems:

- Games like chess—any move sequence leads to a checkmate or absence of material exchange for 50 moves. Also blackjack—take another card or not, given knowledge of played cards. A common r is 1 for a win, $\frac{1}{2}$ for a draw, and -1 for a loss. The optimal reward-maximizing policy defines a perfect player.
- Control of complex systems such as autopilot. Possible settings of navigation and other controls define moves. ! \exists terminal states, and rewards are danger and performance quality signals.
- The **multi-armed bandit problem**. A bandit is a casino slot machine with many levers, each with unknown $E[\text{payout}]$. r is a payout after pulling a lever.
- Resource allocation. E.g., a bond investor perpetually chooses between cash and bonds with different interest rates, maturities, and default risks after every income event.

Simulation is the simplest and most general solution strategy. Given a state, from every next state generate many random episodes, and pick the state maximizing the average \sum sequence rewards. For nonepisodic tasks, run for a fixed number of moves or until r becomes too small.

For efficiency, give more simulations to promising moves based on the initial simulations. The **UCB1 criteria**, after n simulations in a state, simulates move $i = \arg\max_i \left(\text{ave}(i) + \sqrt{\frac{2 \log(n)}{n(i)}} \right)$ for rewards $\in [0, 1]$,

where move i was simulated $n(i)$ times. For differently scaled rewards, the constant $\neq 2$, so scale first. Any choice guarantees asymptotically optimal $E[\text{the loss of not making the best move discovered after } n \text{ trials}] \forall n$ (Auer et al. 2002).

```
double UCB1(double averageValue, int nTries, int totalTries)
{ return averageValue + sqrt(2 * log(totalTries) / nTries); }
```

Applying UCB1 to more than one step doesn't change optimality (Kocsis & Szepesvari 2006), so after an episode, if have enough memory, store and update the averages and counts \forall visited state. Can discard any saved information after committing a move. Keeping track of max/min values allows dynamic scaling, but this is only an approximation because given a current scale prior decision need not be optimal if very differently scaled.

If can't scale to $[0, 1]$, use the OCBA (see the "Computational Statistics" chapter)—unlike the CLT, it doesn't need scaling.

29.3 Value Function Temporal Difference Learning

Another scalable approach is to create a function that assigns a value to every state and move to the maximum-value state. $\text{Value}(\text{current state}) = \text{reward}(\text{current state}) + \text{the average of the observed value(next state)} V_{\text{next}}$. The incremental average update $\text{ave}_n = \text{ave}_{n-1} + \frac{1}{n}(x_n - \text{ave}_{n-1})$ gives

$V_{\text{current}} += \frac{1}{n}(r + V_{\text{next}} - V_{\text{current}})$. The algorithm sets all $V = 0$ and repeatedly starts an episode from the initial state, picking the next state using current V and applying the update rule. It provably converges to values satisfying $V_{\text{current}} = r + E[V_{\text{next}}]$ for the explored states (Russell & Norvig 2020). Changing the $1/n$ learning rate to slower-growing $n^{-0.501}$ can converge faster. Making the initial values optimistic/pessimistic encourages exploration/exploitation and converges faster if the hint is useful. Converged values needn't be opti-

mal if the next state selection is greedy and doesn't explore. Using UCB1 or starting episodes from random states ensures convergence to optimal values.

```
template<typename PROBLEM> void TDLearning(PROBLEM& p)
{
    while(p.hasMoreEpisodes())
    {
        double valueCurrent = p.startEpisode();
        while(!p.isInFinalState())
        {
            double valueNext = p.pickNextState();
            p.updateCurrentStateValue(p.learningRate() * (p.reward() +
                p.discountRate() * valueNext - valueCurrent));
            p.goToNextState();
            valueCurrent = valueNext;
        }
        p.updateCurrentStateValue(p.learningRate() *
            (p.reward() - valueCurrent));
    }
}
```

For small state spaces, V is discrete and remembers the value of each state:

```
struct DiscreteValueFunction
{
    Vector<pair<double, int>> values;
    double learningRate(int state){return 1.0/values[state].second;}
    void updateValue(int state, double delta)
    {
        ++values[state].second;
        values[state].first += delta;
    }
    DiscreteValueFunction(int n): values(n, n, make_pair(0.0, 1)){}
};
```

For huge state spaces such as chess, V is an approximation. A linear combination of state features provably converges to the best possible linear approximation for a Robbins-Munro rate (Russell & Norvig 2020). Other representations such as a neural network can give better results but may not converge.

```
struct LinearCombinationValueFunction
{
    Vector<double> weights;
    int n;
    double learningRate(){return 1.0/n;}
    void updateWeights(Vector<double> const& stateFeatures, double delta)
    //set one of the state features to 1 to have a bias weight
    assert(stateFeatures.getSize() == weights.getSize());
    for(int i = 0; i < weights.getSize(); ++i)
        weights[i] += delta * stateFeatures[i];
    ++n;
}
LinearCombinationValueFunction(int theN): weights(theN, theN, 0), n(1) {}
```

Learning with a value function is similar to doing regression, but need a policy to assign the end-goal value to intermediate states, and learn online for efficiency. For further reading see Sutton & Barto (2018). For application to games see Mandziuk (2010)—for an unknown reason some games like backgammon are more suited to it than some others like chess. In recent years dramatic successes of game programs AlphaGo and AlphaZero renewed interest in reinforcement learning, particularly when use a deep neural network for state value assignment.

29.4 Finding Frequent Item Combinations

Recommendation systems such as Amazon show “customers who bought this also bought” items. A database records every purchase as a subset of **item ids**. Let $n = \max$ size of any **basket**. Any $\binom{n}{k}$ combination is **frequent** if it occurs in other baskets enough times.

The **Apriori algorithm** avoids the combinatorial explosion by sorting items in a basket by id and including a combination if $k = 1$, or the last item and the combination of the $k - 1$ first items are frequent enough. It processes baskets in rounds, with round k considering k -combinations. Sorting ensures that combinations of $k - 1$ items are processed before combinations of k .

```
struct APriori
{
    LCPTreap<Vector<int>, int> counts;
    void noCutProcess (Vector<Vector<int>> const& baskets, int nRounds)
    {
        for(int k = 1; k <= nRounds; ++k)
            for(int i = 0; i < baskets.getsize(); ++i)
                processBasket(baskets[i], k);
    }
};
```

Basket processing returns the number of added items so that the calling code can stop if nothing is added in the current round or adjust cutoffs.

```
int processBasket (Vector<int> const& basket, int round,
                    int rPrevMinCount = 0, int rlMinCount = 0)
{
    int addedCount = 0;
    if(basket.getsize() > round)
    {
        Combinator c(round, basket.getsize());
        do //prepare current combination, needn't sort if each
            //basket is already sorted
        {
            Vector<int> key, single;
            for(int i = 0; i < round; ++i) key.append(basket[c.c[i]]);
            quickSort(key.getArray(), key.getsize());
            int* count = counts.find(key);
            if(count) ++*count; //combination is frequent if already
            else if(round == 1) //frequent or round is 1
            {
                counts.insert(key, 1);
                ++addedCount;
            }
            else //combination is frequent if the last item and
                //combination without the last item are both frequent
                single.append(key.lastItem());
                if(*counts.find(single) >= rlMinCount)
                {
                    key.removeLast();
                    if(*counts.find(key) >= rPrevMinCount)
                    {
                        key.append(single[0]);
                        counts.insert(key, 1);
                        ++addedCount;
                    }
                }
            }
        } while(!c.next());
    }
    return addedCount;
}
```

29.5 Semi-supervised Learning

Want to classify using both labeled and unlabeled samples because labeling usually needs human judgment. Labeled data sets are almost never too large, unless some data is from labeling by an automatic source. But unlabeled data is plentiful.

Unlabeled samples allow better estimation of the properties of the distribution on X . E.g., LDA estimates feature covariance matrix without using labels, so unlabeled samples help. But it's generally a poor classifier even with a good covariance matrix estimate. PCA dimension reduction also works without needing

labels.

Also can cluster all samples, and use a classifier trained on the labeled data to classify them, and train the final classifier on the result. This may work well but not when the clusters don't correspond to the classes.

Ǝ several unsolved problems:

- Model selection is hard because it can base its decision mostly on the labeled samples.
- Experimentally, !Ǝ good black-box methods (Chapelle et al. 2006), unlike for supervised learning. Also algorithm selection needs understanding the data.

29.6 Density Estimation

Want to estimate a probability distribution, given samples from it. It's ill-posed in that $Pr(x) = (x \in \text{data} ? 1/n : 0)$, so must give probability to $x \notin \text{data}$, and at best can use something like ORM to find a simple way to do so.

Several algorithms have been useful for small D (usually ≤ 3 ; Scott 2015):

- **Histogram**—partition X into a grid, and ∀ cell estimate its probability by the proportion of contained examples. Use $o(n)$ cells to balance approximation and estimation errors, and start the counts at 1 for smoothing. It's interesting to consider using a k -d or a VP tree to extend to large D by doing adaptive space partitioning. Then to estimate $Pr(x)$, can walk the tree, and average the current node's example proportions.
- **Kernel density estimator (KDE)**—express the PDF as a sum of kernel functions, as for SVM. Typically use Gaussian kernel.

Without knowing the true density to be estimated, !Ǝ a good L for evaluation. So it's hard to compare estimates, even for picking parameters.

Taking into account how will use a density estimate, and solving that problem directly is better due to less estimation error and being less ill-posed. E.g., oB reduces classification to density estimation, but in practice solve the former directly. Theoretical aspects of density estimation are discussed by Devroye & Lugosi (2001).

Empirical distribution of bootstrap resampling is usually better for whatever task is needed to be performed with the density estimate, so consider bootstrap first.

An interesting practical problem is estimating the maximum rate of user requests for let's say a website or some other service. E.g., for stock trading the opening time is one well-known case. Need to both estimate and maximize; also may want confidence intervals. Density estimation with equal-frequency bins for a histogram is one solution.

29.7 Outlier Detection

An **outlier** is a sample generated by a "very different" mechanism than the rest of the data; how different is enough is subjective. Detection is very useful economically—e.g., any fraud activity such as with credit cards is an outlier relative to the usual transactions. But the problem is ill-posed because it means the iid assumption is violated, and want to detect violations. Such problems are usually better solved as classification, possibly with imbalanced data.

A classic rule of thumb is that observations at ≥ 3 standard deviations from the mean are outliers (Aggarwal 2017). This works heuristically only for data \sim normal, and using 3 is somewhat arbitrary. It appears to come from early methods for industrial process control such as a control chart. A more assumption-free approach is to consider the distribution of the corresponding order statistic from the underlying model distribution and conclude that the value is an outlier if it's more extreme than maybe 99.9% of the values that can be generated. But this is also arbitrary, and using the rule on many samples is subject to multiple testing. So modern methods treat outlier detection as classification into normal and outlier.

In general, without domain knowledge can't detect an outlier—it's only so in a particular assumed model. Another view is that any "small" group of examples with too much influence on the model is outlier. So at best can only identify potential outliers. Some more complicated methods:

- Use a Gaussian mixture model for clustering the data, then declare unlikely examples outliers
- Use DBSCAN to find outliers
- Estimate density, and declare samples in low density regions outliers
- Use reverse nearest neighbor—for every point, mark k nearest neighbors, and deem as outliers any unmarked or low-marked points.

If the goal of outlier detection is making estimates more robust against bad data, it's best to use robust

methods instead.

Outlier analysis is best used as a preliminary task. Eventually, particularly when try to evaluate the effectiveness of detection, will get labeled data and turn the problem into classification, perhaps with imbalanced data. This will allow more effective inference for the domain.

29.8 Implementation Notes

All the implementations follow the textbooks very closely and are very complete in details without any substantial effort. The algorithms are mostly simple, with few decisions unspecified. But not that many algorithms have been presented, and many others are much more complicated.

29.9 Comments

The methods briefly discussed here don't have numerous publicly available data sets, unlike for the main machine learning tasks. This is slowly improving though, particularly for reinforcement learning.

Reinforcement learning is large field with several textbooks, but simulation and value-function-based learning seems to be the only useful algorithms. Other types of algorithms are policy-based, i.e., they value an action by more than just the state where it leads—see Sutton & Barto (2018) if curious. Given my experience in chess, this is at best silly and leads to extra unnecessary combinatorial explosion. The basic idea of sharing the end-goal reward among all the actions is the main ingredient of reinforcement learning, and is well-captured by value functions. The key to making thing work well is whether a good value function exists.

The iid assumption is very important for theoretical understanding of learning. Methods such as **active learning** (A learns online and picks the next example) violate it and so weaken their theoretical guarantees. Active learning must use a collection of unlabeled examples and ask for them to be labeled one at a time, and finding the next best one may be inefficient. Simply generating examples has the problem that they needn't correspond to anything valid, e.g., an image that isn't a digit. Active learning ≠ design of experiments because in the latter all parameter choices are valid or have clear range constraints.

For much more on frequent item combination search see Aggarwal et al. (2014). This task is commercially valuable but not as popular as the others. For the available data sets due to the unsupervised nature and lack of evaluation theory it's hard to verify quality of algorithms.

Transfer learning is another problematic model. The idea is that examples from one problem can help learning for another. This works for humans, but hard to do usefully for a computer.

29.10 References

- Aggarwal, C. C. (2017). *Outlier Analysis*. Springer.
- Aggarwal, C. C., & Han, J. (Eds.). (2014). *Frequent Pattern Mining*. Springer.
- Chapelle, O., Schölkopf, B., Zien, A. (2006). *Semi-supervised Learning*. MIT Press.
- Devroye, L. & Lugosi, G. (2001). *Combinatorial Methods in Density Estimation*. Springer.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
- Russell, S. J., Norvig, P. (2020). *Artificial Intelligence: a Modern Approach*. Prentice Hall.
- Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3), 235–256.
- Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte Carlo planning. In *Machine Learning: ECML 2006* (pp. 282–293). Springer.
- Mandziuk, J. (2010). *Knowledge-Free and Learning-Based Methods in Intelligent Game Playing*. Springer.
- Scott, D. W. (2015). *Multivariate Density Estimation: Theory, Practice, and Visualization*. Wiley.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

30 Scrap—Not Useful Algorithms and Data Structures

30.1 Introduction

Some algorithms and data structures that don't seem useful are too good to discard, despite the fact that the corresponding chapters give good enough reasons to do so. They are either well-established theoretically, are prominent in the standard curriculum, or just satisfy a certain curiosity. Having created a large number of such experimental implementations, I think it might be useful to share them for the benefit of readers who might consider their own experiments. For every implementation, see the comments in the corresponding chapter for references and the rejection reason—I chose to not replicate the information here.

With that said, these implementations are not maintained—i.e., they likely don't have good test sets, descriptions of functionality (even in terms of further references), well-thought-out structure and robustness, or references. Particularly so for the topics in earlier chapters because I never polished the material enough to publish it in the first draft edition. Everything is salvaged from my backup files. Not everything I have is presented though—limited at least to implementations that pass basic tests and are interesting enough. For those without decent descriptions the code is only on the website—expect to do a lot of research with the provided (and not provided) references for more information. Finally, only the implementations that are useful given the references have been presented—e.g., a few implementations not passing basic tests have been dropped permanently.

30.2 Sorting a Linked List

A basic linked list:

```
template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM>>
struct LinkedList
{
    struct Node
    {
        ITEM item;
        Node* next;
        Node(ITEM const& theItem, Node* theNext)
            : item(theItem), next(theNext) {}
    }*root;
    int size;
    Freelist<Node> freelist;
    COMPARATOR c;
    LinkedList(COMPARATOR theC = COMPARATOR()): root(0), size(0), c(theC) {}
    void prepend(ITEM const& item)
    {
        root = new(freelist.allocate())Node(item, root);
        ++size;
    }
};
```

Mergesort is the best algorithm for lists because it doesn't need random access and can manipulate list pointers without a temporary array. A useful optimization is using the size of the list instead of calculating the middle (by having one pointer running twice faster than the other, guaranteeing that the slow pointer is at the middle when the fast one reaches the end), which with a bit of clever coding gives a short, efficient algorithm. The below functions are members of the list.

Merging lists is similar to merging arrays:

```
Node* advanceSmaller(Node*& a, Node*& b)
{
    Node*& smaller = c(b->item, a->item) ? b : a;
    Node* nodeToAppend = smaller;
    smaller = smaller->next;
    return nodeToAppend;
```

```

    }
    Node* merge(Node* a, Node* b)
    { //pick head
        Node* head = advanceSmaller(a, b), *tail = head;
        //append from smaller until one runs out
        while(a && b) tail = tail->next = advanceSmaller(a, b);
        //append the rest of the remaining list
        tail->next = a ? a : b;
        return head;
    }
}

```

To calculate where the middle is without having random access, the idea is to remember the pointer to the start of the right array and update it during the recursive calls.

```

Node* mergesort(Node* list, int n, Node*& nextAfterLast)
{
    if(n==1)
    {
        nextAfterLast = list->next;
        list->next = 0;
        return list;
    }
    int middle = n/2;
    Node *secondHalf, *m1 = mergesort(list, middle, secondHalf);
    return merge(m1, mergesort(secondHalf, n - middle, nextAfterLast));
}
void sort()
{
    if(size > 1)
    {
        Node* dummy;
        root = mergesort(root, size, dummy);
    }
}

```

30.3 Partial Sort

To sort incrementally, sorting another item only if needed, select it but reuse the right bound values from the previous calls, stored on a stack. This uses the optimal expected $O(n + k \lg(k))$ time after selecting k items (Paredes & Navarro 2006) and works because selection is from left to right, so quickselect always executes `right = j`. Before the first call, the stack needs to contain the array's rightmost index, and after each call pop it to ensure $\text{left} + 1 \leq \text{right}$ for the next call. The array is sorted when the stack is empty, and $\text{left} \geq \text{right}$. Can't modify the stack and the vector between calls.

```

template<typename ITEM, typename COMPARATOR> ITEM incrementalQuickSelect(
    ITEM* vector, int left, Stack<int>& s, COMPARATOR comparator)
{
    for(int right, i, j; left < (right = s.getTop()); s.push(j))
        partition3(vector, left, right, i, j, comparator);
    s.pop();
    return vector[left];
}

```

The unlikely worst case is $O(n^2)$. To sort completely using partial sort:

```

template<typename ITEM, typename COMPARATOR> void incrementalSort(
    ITEM* vector, int n, COMPARATOR const& c)
{
    Stack<int> s;
    s.push(n - 1);
    for(int i = 0; i < n; ++i) incrementalQuickSelect(vector, i, s, c);
}

```

It's hard to think of a good use case for this because dynamic sorted sequences offer better functionality and regular sorting is more efficient for static data. But conceptually it's an interesting idea that took relatively long to be discovered.

30.4 Patricia Trie

Some tries only work with keys that are sequences of bits and not generic comparable objects. They are useful conceptually, but not in practice except in hardware because bit extraction is relatively slow, and map operations use w of them where w is the word size. Also use requires interpreting objects as bit sequences, which is clumsy for nonword types and usually not portable due to endianness. So conceptually nonessential operations such as copy constructors and iterators (whatever they mean here) aren't provided.

Keys can be interpreted as byte sequences easily but nonportably:

```
template<typename KEY> struct DefaultRank
{
    unsigned char* array;
    int size;
    DefaultRank(KEY const& key)
        //works with pod types only
        array = (unsigned char*)&key;
        size = sizeof(key);
    }
};
```

On a little-endian architecture the least significant byte will go into position 0 of the array, which isn't what you might have expected. For map operations this results in a speedup if the lower bytes vary more than the higher ones, but the resulting bit string isn't in lexicographic order, which removes support for prefix operations. A portable ranking (i.e., conversion to a bit sequence) would use divisions to extract the bytes in big-endian way, but each key type needs own implementation.

Each node has an index of the next bit block to check. The implementation is a bit complicated, particularly for deletions. No item must be a prefix of another item, as is the case all items have the same size w . So incremental search doesn't make sense.

All nodes branching at a node have the same bit path up to that node. This follows from the insertion algorithm. Think of items as not related to internal nodes. During tasks like deletion, though they have an important property that back pointers to them are located in their subtrees, and the lcp they have with each descendant node is at least the index of the node at which they are stored.

Think of the case when keys are integers and each of 2^{32} of them is present. There will be two of them that match in 31 bits and differ in 32nd one. The nodes to differ between them will branch on 32nd bit as will the child node.

The extra space is three words per key, as for a treap, and two words less than for a LCPTreap (both without augmentations).

```
template<typename KEY, typename ITEM, typename RANK = DefaultRank<KEY>>
class PatriciaTrie
{
    enum{B = 8, TABLE_SIZE = 2, S = B-1};
    int extract(unsigned char* key, int i)
        {return (unsigned char)(key[i/B] << (i%B)) >> S;}
    struct Node
    {
        KEY key;
        ITEM item;
        int index;
        Node *next[2];
        Node(KEY const& theKey, ITEM const& theItem, int theTo, bool bit,
            Node* other): key(theKey), item(theItem), index(theTo)
        {
            next[!bit] = other;
            next[bit] = this;
        }
    }*root;
    Freelist<Node> freelist;
    template<typename ACTION>
    void forEachHelper(Node* t, ACTION& action)
    {
        for(int i = 0; t && i < 2; ++i)
        {
            Node* child = t->next[i];
```

```

        if(child)
        {
            if(t->index >= child->index) action(child);
            else forEachHelper(child, action);
        }
    }
struct AppendAction
{
    Vector<Node*>& result;
    AppendAction(Vector<Node*>& theResult) : result(theResult) {}
    void operator() (Node* node) {result.append(node);}
};

Node** findForwardPointer(Node* query, Node** tree)
{
    RANK rank(query->key);
    Node* node;
    while((node = *tree) != query)
    {
        tree = &node->next[extract(rank.array, node->index)];
    }
    return tree;
}
Node** findBackwardPointer(Node* query, Node** tree)
{
    RANK rank(query->key);
    int prevIndex = -1;
    Node** pointer = tree;
    Node* node;
    while((node = *pointer) && prevIndex < node->index)
    {
        prevIndex = node->index;
        pointer = &node->next[extract(rank.array, prevIndex)];
    }
    return pointer;
}
void removeSingleChildNode(Node** forwardPointer)
{
    Node* node = *forwardPointer;
    *forwardPointer = node->next[!node->next[0]];
    freelist.remove(node);
}

public:
PatriciaTrie(): root(0){}
//acts on nodes in lexicographic bit order
template<typename ACTION> void forEach(ACTION& action)
{
    forEachHelper(root, action);
}
Node* findLongestMatch(KEY const key)
{//will crash if key is prefix of another
    if(!root) return 0;
    RANK rank(key);
    int prevIndex = -1;
    Node* node = root;
    while(prevIndex < node->index)
    {
        prevIndex = node->index;
        Node* next = node->next[extract(rank.array, prevIndex)];
        if(!next) break;
        node = next;
    }
    return node;
/*
    Longest match because all nodes of untested bits on the path have the
*/
}

```

```

same bits and this is the one that can match furtherst. Other nodes
can't be eliminated by untested bits because they are the same their
because the bits are not tested or by tested bits because if so search
would take to different path
*/
}

ITEM* find(KEY const& key)
{
    Node* node = findLongestMatch(key);
    return node && key == node->key ? &node->item : 0;
}

void insert(KEY const& key, ITEM const& item)
{//1. Find index of key to be inserted
    Node* lcpNode = findLongestMatch(key);
    RANK rank(key);
    int theIndex = 0;
    if(lcpNode)
    {
        if(key == lcpNode->key){lcpNode->item = item; return;}
        RANK rank2(lcpNode->key);
        while(extract(rank.array, theIndex) ==
              extract(rank2.array, theIndex)) ++theIndex;
        if(theIndex == lcpNode->index && theIndex < rank.size * B)
            ++theIndex;
    }
    //2. Create and insert the node
    Node **pointer = &root, *node;
    int prevIndex = -1;
    //new node goes before an existing one if node->index >= theIndex or
    //after a self pointing node if prevIndex >= node->index
    while((node = *pointer) && node->index < theIndex &&
          prevIndex < node->index)
    {
        prevIndex = node->index;
        pointer = &node->next[extract(rank.array, prevIndex)];
    }
    *pointer = new(freelist.allocate())
        Node(key, item, theIndex, extract(rank.array, theIndex), node);
}

Vector<Node*> prefixFind(KEY const& key, int minLCP)
{
    RANK rank(key);
    int prevIndex = -1;
    Node* node = root;
    while(node && node->index < minLCP && prevIndex < node->index)
    {
        prevIndex = node->index;
        node = node->next[extract(rank.array, prevIndex)];
    }
    Vector<Node*> result;
    if(node && node->index >= minLCP - 1)//&& findlcp is good
    {
        if(prevIndex >= node->index) result.append(node);
        else
        {
            AppendAction action(result);
            forEachHelper(node, action);
        }
    }
    return result;
}

void remove(KEY const& key)
{
    RANK rank(key);
}

```

```

int prevIndex = -1;
Node** pointer = &root, **parentForwardPointer = &root;
Node* node, *parent = 0;
while((node = *pointer) && prevIndex < node->index)
{
    prevIndex = node->index;
    parentForwardPointer = pointer;
    pointer = &node->next[extract(rank.array, prevIndex)];
}
if(node && key == node->key)
{
    Node* parent = *parentForwardPointer;
    *pointer = 0;
    //if self pointer then remove and link forward pointer to the other
    //child
    if(node == parent)
        removeSingleChildNode(findForwardPointer(node, &root));
    else
        {//if not replace item by that of the parent, redirect parent's
         //backward pointer to it and remove parent as single child node
        node->key = parent->key;
        node->item = parent->item;
        *findBackwardPointer(parent, parentForwardPointer) = node;
        removeSingleChildNode(parentForwardPointer);
    }
}
}
};


```

I experimented with extending Patricia trie to branch on several bits at a time for efficiency. Then a special null link allows to remove the prefix requirement. But this will take more space for nodes and lose deletions because replacing the deleted node by the node that points to it can force a leaf node to have two backward pointers to nodes other than itself, which means that this leaf cannot be deleted. This happens when a node is deleted and replaced by another node with a forward pointer turning into a backward pointer.

Only weak deletions (see the “Miscellaneous Algorithms” chapter) are supported by making nodes as deleted and periodically rebuilding. Alternatively, can store items in external nodes, which will allow deletions but makes the implementation clumsier and less efficient.

	LLRBT	Treap	CHT	Patricia	LCPTreap
int	18.4(37/36)	16(38/36)	8.0(33/32)	34(37/36)	20(43/42)
double					23(61/60)
Struct10	54(90/90)	72(90/90)	49(86/85)	45(90/90)	26(96/96)
Struct10_2	19(90/90)	21(90/90)	49(86/85)		26(96/96)
Struct10_4				54(90/90)	44(96/96)
Struct10_5					145(96/96)

Figure 30.1: Some performance comparisons with LLRBT, Patricia trie, and better data structures (without augmentations). The numbers are (time, memory₁, memory₂) based on the Windows task manager.

30.5 Cuckoo Hashing

Have worst-case O(1) find and remove and expected O(1) insert. Unfortunately the expected O(1) insert might be an infinite loop with a bad hash function. Even universal h can lead to a problematic and to be theoretically safe $\lg(n)$ -independent hashes are sufficient (Ditzfelbinger and Schellbach 2009). Because of this cuckoo is good for testing h but not for general use.

A cuckoo hash table consists of two item arrays, hashes, and status arrays. Insert puts item x into cell $h_1(x)$ of table₁, and if that cell is occupied, item y inside it is kicked out into cell $h_2(x)$ of table₂, possibly kicking out item z into cell $h_1(z)$ of table₁, etc., until an item is placed into an empty cell or this has happened k times (50 is often used, theoretically need $O(\lg(n))$), at which point all items are rehashed using another pair of randomly picked hashes. Need $a < 0.5$ for this to have expected O(1) performance. Find and remove need to check only cells $h_1(x)$ of table₁ and cell $h_2(x)$ of table₂ to find x or conclude that it's not there.

Optimizations are to use a single table and to check if the other cell is empty before the first kickout during insertion. Kutzelnigg (2009) describes and analyses several variations. Cuckoo hashing has the only practical advantage that it's adaptive to the data in a sense that it will be forced to pick a good h and will do so in expected $O(1)$ time. Because of needing a small a need more memory than for chaining or linear probing.

```

template<typename KEY, typename VALUE, typename HASHER = EHash<BUHash>>
class CuckooHashTable
{
    enum{NOT_FOUND = -1};
    int capacity, size;
    typedef KVPair<KEY, VALUE> Node;
    Node* table;
    bool* isOccupied;
    HASHER h1, h2;
    int hash2(KEY const& key, int hash1)
    {
        //to remember which hash function was used to hash the key
        //the trick is to use h1 = hash1(key) and
        //h2 = (hash2(key) - h1) & capacity because this way
        //given cell, the other cell is (hash2(key) - cell) & capacity
        int result = h2(key) - hash1;
        if(result < 0) result += capacity;
        return result;
    }
    int findNode(KEY const& key)
    {
        for(int i = 0, cell = h1(key); i < 2; ++i)
        {
            if(isOccupied[cell] && table[cell].key == key) return cell;
            if(i == 0) cell = hash2(key, cell);
        }
        return NOT_FOUND;
    }
    void allocateTable()
    {
        table = rawMemory<Node>(capacity);
        isOccupied = new bool[capacity];
        setGoodState();
    }
    void setGoodState()
    {
        h1 = HASHER(capacity);
        h2 = HASHER(capacity);
        size = 0;
        for(int i = 0; i < capacity; ++i) isOccupied[i] = false;
    }
    static void cleanUp(Node* theTable, int theCapacity, bool* isOccupied)
    {
        for(int i = 0; i < theCapacity; ++i)
            if(isOccupied[i]) theTable[i].~Node();
        rawDelete(theTable);
        delete[] isOccupied;
    }
    bool insertHelper(KEY const& key, VALUE const& value, bool duringResize)
    {
        //load factor must be < 0.5, which is phase transition for a universal
        //hash function, for kickout phase to take
        if(size > capacity * 0.4) resize(true);
        //can try to first check if both locations are empty rather then just
        //the first one. This gives about 17% insertion speedup at expense of
        //more code, so not worth it since Cuckoo Hashing is not the
        //method of choice
    }
}

```

```

for(Node node(key, value);;resize(false))
{
    int cell = h1(node.key);
    //best max limit choice is not clear, but 50 works fine
    for(int limit = 0; limit < 50; ++limit)
    {
        if (!isOccupied[cell])
        {
            ++size;
            new(&table[cell]) Node(node);
            isOccupied[cell] = true;
            return true;
        }
        if(limit < 2 && table[cell].key == node.key)
        {
            table[cell].value = node.value;
            return true;
        }
        swap(node, table[cell]);
        cell = hash2(node.key, cell);
    }
    if(duringResize) return false;
}
}

void resize(bool increaseSize)
{
    Node* oldTable = table;
    int oldCapacity = capacity;
    bool* oldIsOccupied = isOccupied;
    if(increaseSize) capacity = nextPowerOfTwo(max(4 * size, 8));
    allocateTable();
    for(int i = 0; i < oldCapacity; ++i)
    {
        if(oldIsOccupied[i] && !insertHelper(oldTable[i].key,
            oldTable[i].value, true))
        {
            setGoodState();
            i = -1;
        }
    }
    cleanUp(oldTable, oldCapacity, oldIsOccupied);
}
}

public:
CuckooHashTable(int initialSize = 8): capacity(nextPowerOfTwo(max(
    initialSize, 8))), h1(capacity), h2(capacity) {allocateTable();}
VALUE* find(KEY const& key)
{
    int result = findNode(key);
    return result == NOT_FOUND ? 0 : &table[result].value;
}
~CuckooHashTable(){cleanUp(table, capacity, isOccupied);}
void insert(KEY const& key, VALUE const& value)
{
    insertHelper(key, value, false);
}
void remove(KEY const& key)
{
    int result = findNode(key);
    if(result != NOT_FOUND)
    {
        table[result].~Node();
        isOccupied[result] = false;
        if(--size < capacity * 0.1) resize(true);
    }
}

```

```

typedef Node NodeType;
int getSize() {return size;}
template<typename ACTION> void forEach(ACTION& action)
{
    for(int i = 0; i < capacity; ++i)
        if(table[i].isOccupied) action(&table[i]);
}
class Iterator
{
    int i;
    CuckooHashTable& hashTable;
    void advance()
        while(i < hashTable.capacity && !hashTable.isOccupied[i])++i;
public:
    Iterator(CuckooHashTable& theHashTable): i(0), hashTable(theHashTable)
        {advance();}
    bool hasNext()
        {return i < hashTable.capacity && hashTable.isOccupied[i];}
    NodeType* next()
    {
        NodeType* result = hasNext() ? &hashTable.table[i++] : 0;
        advance();
        return result;
    }
};
}

```

30.6 A Few Priority Queues

The first alternative to a reporter-based indexed heap is to use pointers. It loses efficiency from cache misses but is simpler.

```

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
class IndexedPointerHeap
{
    COMPARATOR c;
    struct Item
    {
        ITEM item;
        int index;
        Item(ITEM const theItem, int theIndex):item(theItem),index(theIndex){}
    };
    Freelist<Item> freelist;
    Vector<Item*> items;
    int getParent(int i){return (i-1)/2;}
    int getLeftChild(int i){return 2*i+1;}
    void report(Item* item, int i){item->index = i;}
    void moveUp(int i)
    {
        Handle temp = items[i];
        for(int parent; i > 0 && c(temp->item,
            items[parent = getParent(i)]->item); i = parent)
            report(items[i] = items[parent], i);
        report(items[i] = temp, i);
    }
    void moveDown(int i)
    {
        Handle temp = items[i];
        for(int child; (child = getLeftChild(i)) <
            items.getSize(); i = child)
        {
            int rightChild = child + 1;
            if(rightChild < items.getSize() && c(items

```

```

        [rightChild->item, items[child]->item)) child = rightChild;
    if(!c(items[child]->item, temp->item)) break;
    report(items[i] = items[child], i);
}
report(items[i] = temp, i);
}

void heapify()
{
    for(int i = getParent(items.getSize()-1); i >= 0; --i) moveDown(i);
void remove(int i)
{
    freelist.remove(items[i]);
    if(items.getSize() > 1)
    {
        items[i] = items.lastItem();
        moveDown(i);
    }
    items.removeLast();
}
public:
typedef Item* Handle;
bool isEmpty(){return items.getSize() <= 0;}
ITEM const& getMin(){assert(!isEmpty());return items[0]->item;}
Handle insert(ITEM const& item)
{
    Handle result = new(freelist.allocate())Item(item, items.getSize());
    items.append(result);
    moveUp(items.getSize()-1);
    return result;
}
ITEM deleteMin(){ITEM result = getMin();remove(0);return result;}
void changeKey(Handle pointer, ITEM const& item)
//assert(pointer && pointer is not garbage);
{
    bool decrease = c(item, pointer->item);
    pointer->item = item;
    decrease ? moveUp(pointer->index) : moveDown(pointer->index);
}
void decreaseKey(Handle index, ITEM const& item){changeKey(index, item);}
void deleteKey(Handle pointer){remove(pointer->index);}
};

```

The main method, at least in theoretical discussion, is a **pairing heap**. It also needs a map from lets say graph node numbers to pointer handles, which is given in a wrapper.

It's represented by a collection of trees and a pointer to the one whose root has the smallest priority. The other trees are its children. An item is inserted by creating a new tree for it and, depending on its priority, making it the root or adding it as the root's child. Delete min removes the root, linearly scans over the children to find the new one, makes it the parent. Then it pairs the remaining children by partitioning them into pairs and making the smaller member of each pair the parent of the other member. Decrease key cuts off the subtree rooted by the node of the handle and makes it a child of the root. Merge makes the larger root a child of the smaller one. If have pointers to the right sibling, oldest child, and youngest elder, delete min takes amortized $\lg(n)$ time, decrease key between $O(\lg\lg(n))$ and $2^{O(\sqrt{\log\log(n)}))}$, which in practice is $O(1)$, and other operations $O(1)$. See Wikipedia (2022) for more analysis.

```

template<typename ITEM, typename COMPARATOR = DefaultComparator<ITEM> >
class PairingHeap
{
    /*Delete in pairing heap is unlink tree with the element, delete min on it,
     *then if the deleted element was not the root merge with the root. To
     *increase key need to cut children and put in siblings list. These
     *additional operations are not as useful and have not been implemented.*/
    COMPARATOR c;
    int size;
    struct Node
    {

```

```

ITEM element;
Node* elder, *oldestChild, *youngerSibling;
Node(ITEM const& theItem)
:element(theItem), elder(0), oldestChild(0), youngerSibling(0){}
} *root;
Freelist<Node> freeList;
PairingHeap(PairingHeap<ITEM> const&);
PairingHeap& operator=(PairingHeap<ITEM> const&);
void insertNode(Node* node)
{
    if(c(root->element, node->element)) linkRoots(root, node);
    else{linkRoots(node, root);root = node;}
}
void linkRoots(Node* parent, Node* child)
{
    //current youngerSiblings are ignored
    Node* oldestChild = parent->oldestChild;
    child->youngerSibling = oldestChild;
    if(oldestChild) oldestChild->elder = child;
    parent->oldestChild = child;
    child->elder = parent;
}
void pairUp()
{
    for(Node* currentRoot = root->youngerSibling; currentRoot; )
    {
        Node* nextRoot = currentRoot->youngerSibling;
        if(!nextRoot)
        {
            insertNode(currentRoot);
            break;
        }
        Node* youngerSibling = nextRoot->youngerSibling;
        if(c(currentRoot->element, nextRoot->element))
            swap(nextRoot, currentRoot);
        linkRoots(nextRoot, currentRoot);
        insertNode(nextRoot);
        currentRoot = youngerSibling;
    }
    correctRoot();
}
void correctRoot(){root->elder = root->youngerSibling = 0;}
public:
typedef Node* Handle;
PairingHeap(): root(0), size(0){}
Handle insert(ITEM const& theItem)
{
    ++size;
    Node* node = new(freeList.allocate())Node(theItem);
    if(isEmpty()) root = node;
    else insertNode(node);
    return node;
    //assert(client will not corrupt the heap through newRoot)
}
Handle replaceMin(ITEM const& theItem)
{
    deleteMin();
    return insert(theItem);
}
void decreaseKey(Handle node, ITEM const& newItem)
{
    assert(node && !c(node->element, newItem) && !isEmpty());
    node->element = newItem;
}

```

```

if(node != root)
{
    Node *elder = node->elder, *youngerSibling = node->youngerSibling;
    if(youngerSibling) youngerSibling->elder = elder;
    (elder->oldestChild == node ?
        elder->oldestChild : elder->youngerSibling) = youngerSibling;
    insertNode(node);
    correctRoot();
}
bool isEmpty(){return !root;}
ITEM const& getMin(){assert(!isEmpty());return root->element;}
ITEM deleteMin()
{
    --size;
    ITEM result = getMin();
    Node* oldRoot = root;
    root = root->oldestChild;
    freeList.remove(oldRoot);
    if(root) pairUp();
    return result;
}
int getSize(){return size;}
//make sure external freelist is used when merging is needed!
void merge(Node* otherRoot){insertNode(otherRoot);}
};

template typename ITEM> struct IndexedPaHeap
{
    typedef typename PairingHeap<ITEM>::Handle HANDLE;
    LinearProbingHashTable<int, HANDLE> map;
    PairingHeap<ITEM> heap;
public:
    bool isEmpty(){return heap.isEmpty();}
    void insert(ITEM const& item, int handle)
    {
        map.insert(handle, heap.insert(item));
    }
    ITEM const& getMin(){return heap.getMin();}
    ITEM deleteMin(){return heap.deleteMin();}
    void changeKey(ITEM const& item, int handle)
    {
        HANDLE h = map.find(handle);
        if(h) heap.changeKey(h->item, Item(item, h));
        else insert(item, handle);
    }
    void deleteKey(int handle)
    {
        HANDLE h = map.find(handle);
        assert(h);
        heap.remove(*h);
    }
};

```

Finally, an interesting but never generally competitive option is a **bucket queue** for items with priorities $\in [0, N - 1]$. An item is inserted by placing it into the list corresponding to its priority, updating the index to it if it's the new smallest. Delete min removes the item from the list of the index and increments the index until it hits a nonempty list. Delete and change key of an item pointed to by a handle need doubly-linked lists (not implemented here). Delete min is $O(N)$, and all other operations are $O(1)$ with the worst case when items with priority 0 and $N - 1$ are repeatedly inserted and deleted. Merging takes $O(N)$ time if keep the pointers to the list ends.

```

template typename ITEM> class BucketQueue
{
    int capacity, minIndex;
    struct Node
    {

```

```

ITEM item;
Node* next;
Node(ITEM const& theItem, Node* theNext):item(theItem), next(theNext){}
}** buckets;
Freelist<Node> freelist;
public:
BucketQueue(int maxN): capacity(maxN + 1), buckets(new Node*[capacity]),
minIndex(capacity){for(int i = 0; i < capacity; ++i)buckets[i] = 0;}
void insert(int priority, ITEM const& item)
{
    assert(priority < capacity);
    buckets[priority] =
        new(freelist.allocate())Node(item, buckets[priority]);
    if(priority < minIndex) minIndex = priority;
}
bool isEmpty(){return minIndex == capacity;}
ITEM findMin(){assert(!isEmpty()); return buckets[minIndex]->item;}
void deleteMin()
{
    assert(!isEmpty());
    Node* temp = buckets[minIndex];
    buckets[minIndex] = temp->next;
    freelist.remove(temp);
    while(minIndex < capacity && !buckets[minIndex]) ++minIndex;
}
~BucketQueue(){delete[] buckets;}
};

```

	Time to sort 1.5×10^7 integers % 10 (time, memory ₁ , memory ₂)
Heap	1.4(62/67)
IndexedHeap	7.2(487/501)
IndexedArrayHeap	2.6(179/198)
IndexedPointerHeap	3.1(238/244)
PairingHeap	1.4(297/296)
BucketQueue	0.6(179/178)
PairingMapLP	5.2(592/592)
PointerMapLP	8.2(534/540)

Figure 30.2: Some performance comparisons

30.7 Lowest Common Ancestor and Range-minimum Query

Both assume union-find-like parent-based tree representation based on an array of items, which is easily changed to work with parent pointers and even some succinct tree representations.

LCA is essentially the same algorithm as that for the interview question of finding out when two linked lists merge: first adjust for the height difference, then step up both until merge or get to a root with no merge:

```

template<typename ITEM> int LCA(ITEM* array, int i, int j)
{//oth performance
    int hDiff = 0;
    for(int k = i; array[k] != -1; k = array[k]) --hDiff;
    for(int k = j; array[k] != -1; k = array[k]) ++hDiff;
    if (hDiff < 0) {swap(i, j); hDiff = -hDiff;}
    while(hDiff--) j = array[j];
    while(i != j) {i = array[i]; j = array[j];}
    return i;
}

```

RMQ is most easily done with a **left range minimum (LRM)** tree. Given an array of item, it computes the left smallest value for each. But use a tree structure so that search left only while the items to the left are decreasing.

```

template<typename ITEM> struct LRMTree

```

```

//previous smaller value tree
Vector<int> parents;
LRMTree(ITEM* array, int size): parents(size, -1)
{
    for(int i = 1; i < size; ++i)
        //expand the rightmost branch unless a smaller value causes another
        //rightmost branch to be created
        parents[i] = i - 1;
        while(parents[i] != -1 && array[parents[i]] >= array[i])
            parents[i] = parents[parents[i]];
    }
}
int RMQ(int left, int right)
{
    int l = LCA(parents.getArray(), left, right);
    if(l == left) return l;
    while(parents[right] != l) right = parents[right];
    return right;
}
};

```

Some suffix index implementations use RMQ functionality, with the implementations customized to the index representation.

30.8 Wilcoxon Signed Rank Test for Two Samples

It assumes that the samples:

- Are symmetrically distributed around the median (essentially like normal, but fat tails OK)
- Come from a continuous distribution

Then, when the observations are joined and converted to ranks, calculate the mean and the variance of the **signed rank** sum distribution, assume it's normal, and use the z-score test. A signed rank = the rank of |the paired observation difference| × the sign, where the sign = 1 if the corresponding first group observation > the second group one, and -1 otherwise.

The ranks of tied observations are averaged. A good way to handle 0 differences is to distribute them evenly and drop one if the number is odd (Demšar 2006). In theory don't have ties due to the continuity assumption, but as for the sign test ties happens, and this method works well. Given n remaining observations, under the null, the mean = 0, and the variance = $\sum \text{rank}_i^2$ (Gibbons & Chakraborti 2011). With some equal ranks, the variance is actually a bit less, so still control type I error. Adjusting the variance calculation to gain a bit of power is slightly complicated, and not done here.

```

struct SignedRankComparator
{
    typedef pair<double, double> P;
    int sign(P const& p) const {return p.first - p.second > 0 ? 1 : -1; }
    double diff(P const& p) const {return abs(p.first - p.second); }
    bool isLess(P const& lhs, P const& rhs) const
        {return diff(lhs) < diff(rhs); }
    bool isEqual(P const& lhs, P const& rhs) const
        {return diff(lhs) == diff(rhs); }
};
double signedRankZ(Vector<pair<double, double> > a)
//same test, use Conover version!
SignedRankComparator c;
quickSort(a.getArray(), 0, a.getSize() - 1, c);
int nP = a.getSize(), i = 0;
//if odd number of 0's, drop first, distribute rest evenly
while(i < a.getSize() && c.diff(a[i]) == 0) ++i;
if(i % 2) --nP;
double signedRankSum = 0, rank2Sum = 0;
for(i = i % 2; i < a.getSize(); ++i)
//rank lookahead to scan for ties, then sum computation
    int j = i;
    while(i + 1 < a.getSize() && c.isEqual(a[i], a[i + 1])) ++i;

```

```

        double rank = (i + j)/2.0 + 1 + nP - a.getSize();
        while(j <= i)
        {
            signedRankSum += c.sign(a[j++]) * rank;
            rank2Sum += rank * rank;
        }
    }
    return rank2Sum == 0 ? 0 : abs(signedRankSum) / sqrt(rank2Sum);
}

```

If the observations are normally distributed, asymptotically the signed rank test has 95% of the power of the t -test, and the sign test 50% of the power of the signed rank test (Gibbons & Chakraborti 2011).

Wilcoxon normal approximation is asymptotic and inaccurate for small samples. Here can use a permutation test with exact and not rank values instead. Wilcoxon is better for small samples though if outliers are suspected, and its tie-reduced variance is accounted for. Permutation tests work well with ranks too and account for ties automatically but are less computationally efficient.

A technical problem with Wilcoxon is that never know if a distribution is symmetric around the median, so only the sign test is safe in general. E.g., for Bernoulli distribution the sign test is the only valid option, despite the fact that the median is a poor estimator of location here. Wilcoxon tends to be preferred over the sign test in applications though due to much better power.

If don't have symmetry, the sign test considers the median of the differences (its Hodges-Lehmann estimator—see the Comments), which \neq the mean. So given two distributions with the same mean, one skewed left and the other right, with $n \rightarrow \infty$ the sign test will report a significant difference in the medians, though there isn't one in the means.

30.9 Friedman Test for Matched Samples

It's a generalization of the sign test, assumes observations come from a continuous distribution (Wikipedia 2015, Gibbons & Chakraborti 2011). First convert observations to ranks in each domain i . Ranks are unique with a continuous distribution, but in practice have tied. Handle ties by averaging tied ranks. Calculate:

- $\forall 0 \leq j < k, r_j = \sum_i r_{ij}$ = rank sum for domain j
- $r_{ave} = (k + 1)/2$ = the expected rank
- $S = \sum_j (r_j - nr_{ave})^2$ = the sum of squared differences between rank sums of alternatives and their expected value
- $st = \sum_i (r_{ij} - r_{ave})^2$ = the sum of squared differences between individual ranks and their expected value

Asymptotically $(k - 1)S/st \sim \chi^2$ with $k - 1$ degrees of freedom under the null equality of all alternatives. Also return the r_i to allow post hoc pairwise analysis.

```

pair<double, Vector<double>> FriedmanPValue(Vector<Vector<double>> const& a)
{ // a[i] is vector of responses on domain i
    assert(a.getSize() > 0 && a[0].getSize() > 1);
    int n = a.getSize(), k = a[0].getSize();
    double aveRank = (k + 1)/2.0, SSAalternative = 0, SSTotal = 0;
    Vector<double> alternativeRankSums(k);
    for(int i = 0; i < n; ++i)
    {
        assert(a[i].getSize() == k);
        Vector<double> ri = convertToRanks(a[i]);
        for(int j = 0; j < k; ++j)
        {
            alternativeRankSums[j] += ri[j];
            SSTotal += (ri[j] - aveRank) * (ri[j] - aveRank);
        }
    }
    for(int j = 0; j < k; ++j)
    {
        double temp = alternativeRankSums[j] - n * aveRank;
        SSAalternative += temp * temp;
    }
    double p =
        1 - evaluateChiSquaredCdf(SSAalternative * (k - 1)/SSTotal, k - 1);
}

```

```

    return make_pair(p, alternativeRankSums);
}

```

The asymptotic distribution appears good enough even for small samples (Gibbons & Chakraborti 2011), but can use permutation testing with S statistic if want an exact test.

30.10 A MADS-like Optimization Algorithm

A simpler **decreasing step random search (DSRS)** can be effective with $O(D)$ evaluations per iteration:

1. Start from some initial step size s , usually 1
2. Until convergence when s becomes too small based on the specified precision
3. Generate a random unit direction
4. Move into $s \times$ direction if doing so decreases the function value
5. If successful, double s
6. Else shrink by some factor $\in [0.5, 1]$, usually 0.8

Effect of s is estimated from Taylor series of f , as $s(s + dd)$, where dd = directional derivative estimated from the last move. This termination strategy is useful because if a bad direction is picked, have many chances to recover before s is too small.

```

template<typename FUNCTION> pair<Vector<double>, double>
DSRS(Vector<double> const& x0, FUNCTION const& f,
double step = 1, double factor = 0.8, int maxFEvals = 10000000,
double yPrecision = numeric_limits<double>::epsilon())
{
    pair<Vector<double>, double> xy(x0, f(x0));
    for(double dd = 0; --maxFEvals && step * (dd + step) > yPrecision;)
    {
        Vector<double> direction = x0;//ensure non-zero direction
        for(int j = 0; j < direction.getSize(); ++j) direction[j] =
            GlobalRNG().uniform01() * GlobalRNG().sign();
        direction *= 1/norm(direction);
        double yNew = f(xy.first + direction * step);
        if(isELess(yNew, xy.second, yPrecision))
        {
            dd = (xy.second - yNew)/step;
            xy.first += direction * step;
            xy.second = yNew;
            step *= 2;
        }
        else step *= factor;
    }
    return xy;
}

```

30.11 SAMME Boosting Classification Algorithm

Please review the boosting section in the “Machine Learning—Classification” chapter first.

```

template<typename LEARNER = NoParamsLearner<DecisionTree, int>,
        typename PARAMS = EMPTY, typename X = NUMERIC_X> class AdaBoostSamme
{
    Vector<LEARNER> classifiers;
    Vector<double> weights;
    int nClasses;
public:
    template<typename DATA> AdaBoostSamme(DATA const& data, PARAMS const&
        p = PARAMS(), int nClassifiers = 300): nClasses(findNClasses(data))
    {
        int n = data.getSize();
        assert(n > 0 && nClassifiers > 0 && nClasses > 0);
        Vector<double> dataWeights(n, 1.0/n);
        for(int i = 0; i < nClassifiers; ++i)
        {
            AliasMethod sampler(dataWeights);

```

```

PermutatedData<DATA> resample(data);
for(int j = 0; j < n; ++j) resample.addIndex(sampler.next());
classifiers.append(LEARNER(resample, p));
double error = 0;
Bitset<> isWrong(n);
for(int j = 0; j < n; ++j) if(classifiers.lastItem().predict(
    data.getX(j)) != data.getY(j))
{
    isWrong.set(j);
    error += dataWeights[j];
}
if(error >= 1 - 1.0/nClasses) classifiers.removeLast();
else if(error == 0)
{//replace ensemble by classifier
    Vector<LEARNER> temp;
    temp.append(classifiers.lastItem());
    classifiers = temp;
    weights = Vector<double>(1, 1);
    break;
}
else
{
    double expWeight = (nClasses - 1) * (1 - error)/error;
    weights.append(log(expWeight));
    for(int j = 0; j < n; ++j)
        if(isWrong[j]) dataWeights[j] *= expWeight;
    normalizeProbs(dataWeights);
}
}
}

int predict(X const& x)const
{
    Vector<double> counts(nClasses, 0);
    for(int i = 0; i < classifiers.getSize(); ++i)
        counts[classifiers[i].predict(x)] += weights[i];
    return argMax(counts.getArray(), counts.getSize());
}
};

```

30.12 Boosting for Regression

As for classification, can use boosting to improve results, particularly of a regression tree. Use the gradient boosting version (see the “Machine Learning—Classification” chapter), so only change y , and don’t need to resample. Given current F , want to find a for the last h such that $\sum(F(x_i) + ah(x_i)) - y_i)^2$ is minimal. The solution is $a = \sum(y_i - F(x_i))h(x_i)/\sum h(x_i)$ (Schapire & Freund 2012). For regression, boosting is intuitively similar to Tukey twicing, i.e., first do regression for y , then again for the errors.

```

template<typename LEARNER = NoParamsLearner<RegressionTree, double>,
typename PARAMS = EMPTY, typename X = NUMERIC_X> class L2Boost
{
    Vector<LEARNER> classifiers;
    Vector<double> weights;
    double getWeight(int i)const {return 1/pow(i + 1, 0.501);}
    struct L2Loss
    {
        Vector<double> F;
        L2Loss(int n): F(n, 0) {}
        double getNegGrad(int i, double y){return 2 * (y - F[i]);}
        double loss(int i, double y){return (F[i] - y) * (F[i] - y);}
    };
    public:
        template<typename DATA> L2Boost(DATA const& data,
            PARAMS const& p = PARAMS(), int nClassifiers = 300)

```

```

{
    int n = data.getSize();
    assert(n > 0 && nClassifiers > 0);
    L2Loss l(n);
    RelabeledData<DATA> regData(data);
    for(int j = 0; j < n; ++j) regData.addLabel(data.getY(j));
    for(int i = 0; i < nClassifiers; ++i)
        //find gradient, relabel data, fit learner, and update p
        for(int j = 0; j < n; ++j)
            regData.labels[j] = l.getNegGrad(j, data.getY(j));
        classifiers.append(LEARNER(regData, p));
    Vector<double> h;
    for(int j = 0; j < n; ++j)
        h.append(classifiers.lastItem().predict(data.getX(j)));
    double sumH2 = 0, weight = 0;
    for(int j = 0; j < n; ++j)
    {
        sumH2 += h[j] * h[j];
        weight += (data.getY(j) - l.F[j]) * h[j];
    }
    if(weight > 0 && isfinite(weight/sumH2))
    {
        weights.append(weight/sumH2);
        for(int j = 0; j < n; ++j)
            l.F[j] += weights.lastItem() * h[j];
    }
    else
    {
        classifiers.removeLast();
        break;
    }
}
}

double predict(X const& x) const
{
    double sum = 0;
    for(int i = 0; i < classifiers.getSize(); ++i)
        sum += classifiers[i].predict(x) * weights[i];
    return sum;
}
};

```

Can try RM weights instead of the optimal ones or RM fractions of optimal ones (instead of the implicit 1) if overfitting is a problem. Due to not resampling, an early termination due to 0 training error is likely and the default 300 trees may not be reached, which can give an efficiency advantage over random forest.

30.13 Probabilistic Clustering

Please review the comments in the “Machine Learning—Clustering” chapter first. Use the EM algorithm to train a mixture of k Gaussians:

1. Pick the initial values for the parameters—mixture weights w , means m , and covariances Σ
2. Until convergence
3. Calculate class membership probabilities of samples using the parameters
4. Calculate the parameters using the membership probabilities
5. Assign all samples to their maximum likelihood clusters

(3) is the E-step and (4) the M-step, but the correspondence isn't intuitive. Estimating class membership is an expectation of a special function, and (4) is using the maximum likelihood to estimate the parameters. For multivariate normal with mean m and covariance matrix Σ , for a single x log-likelihood $LL(x) = -\frac{1}{2}(D \ln(2\pi) + \ln(\det(\Sigma)) + (x - \mu)^T \Sigma^{-1} (x - \mu))$. For a mixture with cluster weights w :

- “Log-weighted-likelihood” $lwl(w, x) = \ln(w) + LL(x)$
- The total log-likelihood for all data = $\sum_i \ln(\sum_j e^{lwl(w, x)})$

The update equations:

- The memberships probabilities g_{ij} , for example i and class j , are proportional to $lwl(w_j, x_i)$.
- The total support for class j , $n_j = \sum_i g_{ij}$
- $w_j = \frac{n_j}{n}$
- $\bar{x}_j = \frac{\sum_i g_{ij} x_i}{n_j}$
- $\Sigma_j = \frac{\sum_i g_{ij} (x_i - \bar{x}_j) \otimes (x_i - \bar{x}_j)}{n_j}$

The Σ_j aren't adjusted by degrees of freedom because use maximum likelihood estimates. As for k -means, the log-likelihood never decreases (Gupta & Chen 2010), so the algorithm is a proper local search (but in rare cases numerical error or various robustness heuristics can invalidate this).

For implementation, the initial x are found by repeated k -means, and use BIC to find k because it's naturally supported. For convergence, check if relative change in log likelihood $\leq 10^{-5}$ (Fraley et al. 2012). For safety bound the number of iterations by a constant, 1000 by default.

I never managed to resolve numerical issues to get sufficient robustness (though the below implementation worked well on all tested data sets):

- Getting singularities in Σ under conditions such as 0-variance features. This can be boosted by averaging in some pooled variance, but the best method is yet to be determined.
- Even with pooled variance get can clusters with < 1 -example-support collapse $n_j < 1$. E.g., a large component can completely subsume a smaller one, leaving the latter with near-0 support. I don't see a good solution to this other than giving up the tried number of clusters.
- Inverting Σ using Cholesky decomposition (the most stable one for symmetric and positive definite matrices; see the "Numerical Algorithms—Introduction and Matrix Algebra" chapter) would sometimes fail numerically. This is despite computing lwl in safe ways.

The initial m are found by repeated k -means, w are set to $1/k$, and Σ to identity \times pooled variance, where the latter is based on initial m and assignments and $= \frac{\sum_i (x_i - m(i))^2}{(n-k)D}$. The pooled variance assumes diagonal Σ with the same variance for every component. Because estimated k initial means, have $n - k$ degrees of freedom. This makes the implementation easy—proper w and Σ are calculated in the next iteration. In case of numerical issues the initial clustering is the default, and its log-likelihood is returned.

Some numerical issues still need correcting, though how is rarely discussed in the literature, and the best way is unclear. Use the following heuristics:

- Average $I \times$ pooled variance with support 1 into Σ (with support n_j) to prevent singularities under conditions such as 0-variance features. Pooled variance evolves with w and m , but updating it defeats its function as a "prior". Removing features with 0 variance may seem more correct, but some clusters can still have 0 variance in a particular feature.
- If Cholesky decomposition fails, terminate. This signals a problem that can't be corrected.
- Use the fact that $\sum e^{x_i} = e^b \sum e^{x_i - b}$, where $b = \max(x_i)$; this makes using lwl numerically more stable because the first part usually carries most of the value of the sum and can be processed separately.
- Check for < 1 -example support collapse $n_j < 1$, and terminate if detected. Though the pooled variance discourages collapses, it can't prevent them, and a large component can completely subsume a smaller one, leaving the latter with near-0 support.
- Check the log likelihood against infinities and NaN. This is a catch-all.

```
struct EMClustering
{
    static double normalLL(Vector<double> x, Vector<double> const& m,
                           Cholesky<double> const& l)
    {
        x -= m;
        return -(x.getSize() * log(2 * PI()) + l.logDet() +
                  dotProduct(l.solve(x), x))/2;
    }
    static double llim(){return log(numeric_limits<double>::min())/2;}
    static pair<Vector<double>, double> findILL(NUMERIC_X const& x,
                                                 Vector<double> const& w, Vector<Vector<double>> const& m,
```

```

Vector<Cholesky<double>> const &ls)
{
    int k = w.getSize();
    Vector<double> temp(k);
    for(int j = 0; j < k; ++j)
        temp[j] = w[j] > 0 ? log(w[j]) + normalLL(x, m[j], ls[j]) : 0;
    double b = argMax(temp.getArray(), k);
    for(int j = 0; j < k; ++j) temp[j] -= b;
    return make_pair(temp, b);
}
template<typename DATA> static double findLL(DATA const& data,
    Vector<double> const& w, Vector<Vector<double>> const& m,
    Vector<Cholesky<double>> const &ls)
{
    double ll = 0;
    for(int i = 0; i < data.getSize(); ++i)
    {
        pair<Vector<double>, double> temp =
            findILL(data.getX(i), w, m, ls);
        double kSum = temp.second;
        for(int j = 0; j < w.getSize(); ++j) kSum += exp(temp.first[j]);
        ll += log(kSum);
    }
    return ll;
}
template<typename DATA> ClusterResult operator()(DATA const& data, int k,
    int maxIterations = 1000) const
{
    int n = data.getSize(), D = getD(data);
    assert(k > 0 && k <= n && n > 0);
    //Initial values
    Vector<int> assignments =
        RepeatedKMeans()(data, k, maxIterations).assignments;
    Vector<Vector<double>> m = findCentroids(data, k, assignments),
        g(n, Vector<double>(k));
    Vector<double> w(k, 1.0/k);
    double pooledVar = 0;
    for(int i = 0; i < n; ++i)
    {
        Vector<double> diff = (data.getX(i) - m[assignments[i]]);
        pooledVar += dotProduct(diff, diff);
    }
    pooledVar /= (n - k) * D;
    Vector<Cholesky<double>> ls(k,
        Cholesky<double>(Matrix<double>::identity(D) * pooledVar));
    double ll = findLL(data, w, m, ls);
    while(maxIterations--)
    { //E step
        for(int i = 0; i < n; ++i)
        {
            pair<Vector<double>, double> temp =
                findILL(data.getX(i), w, m, ls);
            for(int j = 0; j < k; ++j) g[i][j] = exp(temp.first[j]);
            normalizeProbs(g[i]);
        }
        //M step
        bool isNumericIssue = false;
        for(int j = 0; j < k; ++j)
        { //nj
            double nj = 0;
            for(int i = 0; i < n; ++i) nj += g[i][j];
            if(nj < 1){isNumericIssue = true; break;}
        }
        //w
    }
}

```

```

        w[j] = nj/n;
        //...
        for(int i = 0; i < n; ++i) m[j] += data.getX(i) * g[i][j];
        m[j] *= 1/nj;
        //average in pooled variance
        Matrix<double> covar = Matrix<double>::identity(D) *
            pooledVar;
        for(int i = 0; i < n; ++i)
        {
            Vector<double> xm = data.getX(i) - m[j];
            covar += outerProduct(xm, xm) * g[i][j];
        }
        covar *= 1/(1 + nj);
        ls[j] = Cholesky<double>(covar);
        if(ls[j].failed) {isNumericIssue = true; break;}
    }
    if(isNumericIssue) break;
    double newLL = findLL(data, w, m, ls);
    if(!isfinite(newLL) || !isELess(ll, newLL, 0.00001)) break;
    ll = newLL;
}
double BIC = -2 * ll + (k * (1 + D + D * (D + 1)/2) - 1) * log(n);
for(int i = 0; i < n; ++i)
    assignments[i] = argMax(g[i].getArray(), k);
return ClusterResult(assignments, BIC);
}
}

```

For $D > 100$ use k -means because EM gets inefficient.

```

typedef FindKClusterer<NoParamsClusterer<EMClustering>> EMBIC;
struct EMSmart
{
    KMeansGeneral km;
    EMBIC em;
    template<typename DATA> bool useKMeans(DATA const& data) const
        {return getD(data) > 100;} //for efficiency and numerics
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k) const
    {
        if(useKMeans(data)) return km(data, k);
        else return em(data, k);
    }
    template<typename DATA> ClusterResult operator()(DATA const& data) const
    {
        if(useKMeans(data)) return km(data);
        return em(data);
    }
};

```

30.14 Hierarchical Clustering

Please review the comments in the “Machine Learning—Clustering” chapter first. Compute average linkage efficiently using **Lance-Williams formula** (Aggarwal & Reddy 2014): when groups i and j are merged, for any other group k , $d(k, ij) = \frac{n_i d(k, i) + n_j d(k, j)}{n_i + n_j}$.

Hierarchical clustering has several advantages of other methods (Kauffman & Rousseeuw 1990)—in particular, it estimates expected group distance \forall two groups and in that sense doesn't it vary with the number of examples in each cluster; few other measures have this property. It also works with any dissimilarity measure (unlike a distance, a dissimilarity doesn't need to satisfy triangle inequality; averaged distances usually stop being distances and become dissimilarities).

Beware that the code below introduced a bug that causes a test failure (see the projects).

```
template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
```

```

struct HierarchicalClustering
{
    struct Node
    {
        union{int i, sequenceNumber;};
        int size;
        Node *left, *right;
        bool isLeaf(){return !left;}
        Node(int theI): i(theI), left(0), right(0), size(1){}
        bool operator<(Node const& rhs)const //larger wins
            {return sequenceNumber > rhs.sequenceNumber;}
    } *root;
    Freelist<Node> f;
    void createAssignmentsHelper(Node* node, Vector<int>& assignments,
        int nextC)const
    {
        if(node->isLeaf()) assignments[node->i] = nextC;
        else
        {
            createAssignmentsHelper(node->left, assignments, nextC);
            createAssignmentsHelper(node->right, assignments, nextC);
        }
    }
    Vector<int> createAssignments(int k, int n)const
    {
        Vector<int> assignments(n);
        Heap<Node*, PointerComparator<Node> > h;
        h.insert(root);
        int nextC = 0;
        while(!h.isEmpty())
        {
            Node* node = h.deleteMin();
            if(node->isLeaf() || k < 2)
                createAssignmentsHelper(node, assignments, nextC++);
            else
            {
                --k;
                h.insert(node->left);
                h.insert(node->right);
            }
        }
        return assignments;
    }
    template<typename DATA> HierarchicalClustering(
        DATA const& data, DISTANCE const& d = DISTANCE()): root(0)
    {
        int n = data.getSize(), sequenceNumber = n;
        Vector<Node*> nodeIndex(n);
        for(int i = 0; i < n; ++i) nodeIndex[i] = new(f.allocate())Node(i);
        IndexedArrayHeap<double> iHeap;
        for(int i = 1; i < n; ++i) for(int j = 0; j < i; ++j)
            iHeap.insert(d(data.getX(i), data.getX(j)), i * n + j);
        while(!iHeap.isEmpty())
        {
            IndexedArrayHeap<double>::ITEM_TYPE minP = iHeap.deleteMin();
            int index = minP.value, j = index % n,
                i = index/n;//j < i by construction
            //update smaller index pair distance, put node in small index map
            Node* winnerNode = new(f.allocate())Node(sequenceNumber++);
            int sizeI = nodeIndex[i]->size, sizeJ = nodeIndex[j]->size;
            winnerNode->left = nodeIndex[j];
            winnerNode->right = nodeIndex[i];
            winnerNode->size = sizeI + sizeJ;
        }
    }
}

```

```

        nodeIndex[j] = winnerNode;
        nodeIndex[i] = 0;
        for(int k = 0; k < n; ++k) //update all i and j distances
            if(i != k && j != k && nodeIndex[k])
            {
                int indexIK = max(i, k) * n + min(i, k),
                    indexJK = max(j, k) * n + min(j, k);
                double dijk = (sizeI * *iHeap.find(indexIK) +
                    sizeJ * *iHeap.find(indexJK))/(sizeI + sizeJ);
                iHeap.changeKey(dijk, indexJK);
                iHeap.deleteKey(indexIK);
            }
        }
        root = nodeIndex[0];
    }
struct Functor
{
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k, HierarchicalClustering const& h) const
    {
        Vector<int> assignments = h.createAssignments(k, data.getSize());
        return ClusterResult(assignments, -clusterSilhouette(data,
            assignments, DISTANCE()));
    }
    template<typename DATA> ClusterResult operator()(DATA const& data,
        int k) const
    { return operator()(data, k, HierarchicalClustering(data)); }
    template<typename DATA> ClusterResult operator()(DATA const& data) const
    {//Wrong! - needs to recompute -- need to fix!
        return findClustersAndK(data, Functor(),
            HierarchicalClustering(data));
    }
};

```

30.15 Density-based Clustering

Please review the comments in the “Machine Learning—Clustering” chapter first. For DBSCAN, given parameters ϵ and $nMin$, a:

- **Core point** has $\geq nMin$ within radius ϵ
- **Border point** has a core point within radius ϵ but isn't a core point
- **Noise point** is neither

Noise points are considered to be outliers that don't belong to any cluster. Nearby core points form clusters, and border points go to clusters of their core point. The difficult part is deciding the parameters. Use $nMin = \lceil \log(n) \rceil$; this makes more sense than the original paper suggestion because of k -NN arguments. The latter suggests $nMin = 4$, and min ϵ such that the proportion of noise points is acceptable, without giving more explicit instructions.

In the original DBSCAN core points must be within radius ϵ to connect. But border points neighboring two core points can belong to either one and are assigned depending on the example order. A simple solution is to link core points with a common border point. This also allows simple implementation with union-find, and is a minor change (essentially ϵ doubles):

1. **\forall point**
2. **If core, join its ϵ -neighbors to it**
3. **\forall union-find root**
4. **Check if it contains core points**
5. **If yes, it forms a cluster, else noise**
6. **Assign all points to clusters represented by their roots**

The heuristic implemented here for parameter setting:

1. **Calculate the average and standard deviation of distance to the $nMin^{\text{th}}$ nearest neighbor, based on all points**

2. Starting from $z = -3$
3. Use $\epsilon = \text{ave} + z \times \text{stdev}$
4. Stop if noise proportion < 0.05 or $z = 3$, else $z \leftarrow z + 0.5$

The logic is relying on Chebyshev's inequality that \forall distribution most data is within few standard deviation of the mean (see the "Computational Statistics" chapter; here this is just an approximation because the parameters are estimated), so the parameter search seems reasonably exhaustive and efficient. For efficiency, use an index data structure that allows radius and k -NN queries. VP tree is particularly suitable due to working with metric distances.

```
template<typename DISTANCE = EuclideanDistance<NUMERIC_X>::Distance>
struct DBSCAN
{
    template<typename DATA, typename TREE> static pair<Vector<int>, double>
    findClusters(TREE const& t, DATA const& data, int nMin, double eps)
    {//determine core points and their border points
        int n = data.getSize();
        UnionFind uf(n);
        Vector<bool> isCore(n);
        Vector<int> assignments(n, -1);
        for(int i = 0; i < n; ++i)
        {
            Vector<typename TREE::NodeType*> epsNeighbors =
                t.distanceQuery(data.getX(i), eps);
            if(epsNeighbors.getSize() - 1 >= nMin)
            {
                isCore[i] = true;
                for(int j = 0; j < epsNeighbors.getSize(); ++j)
                {
                    int v = epsNeighbors[j]->value;
                    uf.join(i, v);
                    assignments[v] = i;//reuse array for membership
                }
            }
        }
        //determine core/noise clusters by whether their roots are core
        for(int i = 0; i < n; ++i) if(isCore[i]) isCore[uf.isRoot(i)] = true;
        int k = 0;
        for(int i = 0; i < n; ++i)
            if(uf.isRoot(i) && isCore[i]) assignments[i] = k++;
        //find classes of border points +
        //code -1 as k + 1 for compatibility with analysis code
        int noise = 0;
        for(int i = 0; i < n; ++i)
        {
            if(assignments[i] == -1)
            {
                assignments[i] = k;
                ++noise;
            }
            else if(!uf.isRoot(i))
                assignments[i] = assignments[uf.find(assignments[i])];
        }
        return make_pair(assignments, noise * 1.0/n);
    }
    template<typename DATA> ClusterResult operator()(  

        DATA const& data, double noisePercentage = 0.05) const  

    {//estimate params
        int n = data.getSize(), nMin = log(n) + 1;  

        assert(n > 0);
        typedef VpTree<typename DATA::X_TYPE, int, DISTANCE> TREE;  

        TREE t;
        for(int i = 0; i < n; ++i) t.insert(data.getX(i), i);
        IncrementalStatistics s;
        DISTANCE d;
```

```

for(int i = 0; i < n; ++i)
{
    Vector<typename TREE::NodeType*> nMinNNs = t.kNN(data.getX(i),
        nMin + 1); //+1 for self
    s.addValue(d(data.getX(i), nMinNNs.lastItem()->key));
}
double stdev = sqrt(s.getVariance()), z = -3;
pair<Vector<int>, double> result;
do
{
    result = findClusters(t, data, nMin, s.getMean() + z * stdev);
    z += 0.5;
} while(result.second >= noisePercentage && z <= 3);
return ClusterResult(result.first);
}
};

```

Need $O(n)$ k -NN queries for parameter selection and $O(n)$ radius queries to find status of points. For low-dimensional data, usually the runtime is $O(\lg(n) + n\min)$ time for each, but can be close to $O(n)$ for high-dimensional data without exploitable structure.

30.16 Implementations not Presented

Many implementations were rejected before the need to produce detailed descriptions of them. I didn't want this chapter to become a code dump, so they are only briefly mentioned here, and the code is on the book's website.

- Left-leaning red-black tree—a basic version without iterators and augmentations.
- A double-sided heap—actually not bad in terms of efficiency for what it does. I simply never had the time to describe it properly or a reason to use it.
- External memory hashing—cryptographic hash functions now became feasible because they take less time than an IO. Linear probing using an EM vector as the table is a simple choice, but can do better mainly because resizing is clumsy:
 - Extendible hashing is a well-known solution to resizing problem. It's essentially an internal-memory trie that groups its tails into buckets. The buckets stored in external memory. Unfortunately this structure takes too much internal memory for the trie which has at least n/B pointers to bucket pages. Can store the trie structure in external memory, but a single insertion can cause much recomputing.
 - Linear hashing also allows resizing and does not require substantial internal memory. Unfortunately the code salvage broke something, and neither linear hashing nor the EM linear probing works anymore, so see the projects.
- Adaptive arithmetic coding—unfortunately the code runs only in some but not all cases. It's independently interesting because of a Fenwick tree data structure for adaptive updates.
- Interval tree—to find all intervals containing a point. If n intervals are stored in a static structure, can compute it in $O(\lg(n))$ time. See for details see de Berg et al. (2008). Lack of use cases and dynamic updating are the reasons for rejection. Same for many other similar data structures such as segment tree, priority search tree, etc.
- Locality-sensitive hashing—for Euclidean data. Additional references are Leskovec et al. (2020), Andoni & Indyk (2008), and Slaney & Casey (2008). For using it for classification with nearest neighbors use grid search to find a good parameter range and fall back to the nearest mean classifier if none are found.
- A version of BFGS which starts with a finite-difference Hessian estimate, and adjusts or recomputes it when non positive definite.
- Optimization with conjugate gradient—I used the PRP+ formula (Nocedal & Wright 2006) and the same initial step, line search, restart policy, and terminating conditions as L-BFGS (though step scaling improved for L-BFGS later, without affecting test performance). This initial step seems more robust than the first-order change matching of Nocedal & Wright (2006). Near a minimum where a quadratic is a good approximation, in exact arithmetic, and with exact line search D iterations suffice. Based on this fact it was believed that it's best to restart CG every D iterations, but converge results for PRP+ formula don't need this (Nocedal & Wright 2006); essentially it has automatic restart capability. Still, need restart because directions can spoil due to an evolving numerical state,

like for L-BFGS. **Hager-Zhang** formula did worse than PRP+, despite claims in (Hager & Zhang 2006) for the latter method. Conjugate gradient with PRP+ needs a bit less from the problem to guarantee convergence though (e.g., no bound on $\kappa(H_0)$; for details see Nocedal & Wright 2006), but this doesn't seem to make a difference in practice.

- MRMR feature selection algorithm for supervised learning and its mutual information estimator.

30.17 Projects

- Experiment with one of these. Create good test files. Do you agree with my decision for discarding? If so, would you ever use and maintain the algorithm or data structure in production code?
- Fix the code for EM hashing. Add persistence to allow initializing from a storage file.
- Fix the code for adaptive arithmetic coding.
- Improve termination precision and scaling of DSRS (which was implemented before I gave this much thought).
- Try EM clustering with simplified silhouette instead of BIC. Compare to using repeated k -means to select k with subsequent EM.
- Fix the code for hierarchical clustering.

30.18 References

- Aggarwal, C. C., & Reddy, C. K. (Eds.). (2014). *Data Clustering: Algorithms and Applications*. CRC.
- Andoni, A., & Indyk, P. (2008). Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Communication of the ACM*, 51(1), 117.
- de Berg, M., Cheong, O., & Van Kreveld, M., Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer.
- Dahlquist, G., & Björck, Å. (2008). *Numerical Methods in Scientific Computing*. SIAM.
- Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine learning research*, 7(Jan), 1-30.
- Dietzfelbinger, M., & Schellbach, U. (2009). On risks of using cuckoo hashing with simple universal hash classes. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms* (pp. 795-804). Society for Industrial and Applied Mathematics.
- Gibbons, J. D., & Chakraborti, S. (2011). *Nonparametric Statistical Inference*. Springer.
- Gupta, M. R., & Chen, Y. (2010). *Theory and Use of the EM Algorithm*. Now Publishers.
- Fraley, C., Raftery, A. E., Murphy, T. B., & Scrucca, L. (2012). mclust version 4 for R: Normal mixture modeling for model-based clustering, classification, and density estimation. *Department of Statistics, University of Washington*, 23, 2012.
- Hennig, C., Meila, M., Murtagh, F., & Rocci, R. (Eds.). (2016). *Handbook of Cluster Analysis*. CRC.
- Kutzelnigg, R. (2009). *Random Graphs and Cuckoo Hashing: A Precise Average Case Analysis of Cuckoo Hashing and Some Parameters of Sparse Random Graphs*. Suedwestdeutscher Verlag fuer Hochschulschriften.
- Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). *Mining of Massive Data Sets*. Cambridge University Press.
- Nocedal, J., Wright, S. (2006). *Numerical Optimization*. Springer.
- Paredes, R., & Navarro, G. (2006). Optimal incremental sorting. In *Proc. 8th Workshop on Algorithm Engineering and Experiments and 3rd Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO'06)* (pp. 171–182). SIAM.
- Slaney, M., & Casey, M. (2008). Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 25(2), 128-131.
- Wikipedia (2015). Friedman test. https://en.wikipedia.org/wiki/Friedman_test. Accessed November 18, 2015.
- Wikipedia (2022). Pairing heap. https://en.wikipedia.org/wiki/Pairing_heap. Accessed February 5, 2022.

31 Appendix—C++ Notes

31.1 Introduction

I often get asked about how to get better at C++, knowing which well is a prerequisite for this book. So I decided to recommend the resources I found useful.

31.2 A Guide to C++ Literature

My current guides are websites en.cppreference.com and cplusplus.com. They are up-to-date, easy to navigate, and have concise example for using the functionality. Googling something also often leads to stackoverflow.com, which is helpful for specific questions and debugging compiler errors. But this only work for those already familiar with the language.

A beginner should start with an introductory book. I can't recommend a specific one, but a good one will discuss compilation basics and working with and an IDE.

After that should go through the main references:

- Stroustrup (2013)—for core language
- Josuttis (2012)—for the library
- Vandevoorde et al. (2017)—for templates

Next want to improve conceptual language mastery:

- Start with books of Meyers (2005, 1996, 2001, 2014), in this order
- Dewhurst (2005), Lippman (1996), Coplien (1992), and Sutter (2005, and maybe its predecessors)
- van der Linden (1994)—need to know some C peculiarities
- Alexandrescu (2001)—the origin of template metaprogramming, much of which the latest C++ standards made better

31.3 Projects

- Look at some newer promising books that I never read, such as Josuttis (2019) and Lakos (2019, and upcoming volumes). More are coming out every year. Is it worth do add some of these to the above list?

31.4 References

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley.

Coplien, J. O. (1992). *Advanced C++ Programming Styles and Idioms*. Addison-Wesley.

Dewhurst, S. C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Pearson.

Josuttis, N. M. (2012). *The C++ Standard library: A Tutorial and Reference*. Addison-Wesley.

Josuttis, N. M. (2019). *C++17: Complete Guide*. Independently Published.

Lakos, J. (2019). *Large-Scale C++ Volume I: Process and Architecture*. Addison-Wesley.

Lippman, S. B. (1996). *Inside the C++ Object Model*. Addison-Wesley.

Meyers, S. (1996). *Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley.

Meyers, S. (2001). *Effective STL: 50 Specific Ways to Improve Your Standard Template Library*. Addison-Wesley.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley.

Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. Addison-Wesley.

Stroustrup, B. (2013). *The C++ programming language*. Addison-Wesley.

Sutter, H. (2005). *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*.

Boston, MA: Addison-Wesley.

van der Linden, P. (1994). *Expert C programming: Deep C secrets*. Pearson.

Vandevoorde, D., Josuttis, N. M. & Gregor, D. (2017). *C++ Templates: The Complete Guide*. Addison-Wesley.

Index

2-3 tree.....	117	autoregression.....	370
2-3-4 tree.....	117	avalanche hash function property.....	125
A-stable ODE solver.....	476	average linkage clustering.....	647
A* algorithm.....	234	AVL tree.....	117
absolute condition number.....	392	B-splines.....	481
absolute precision.....	393	B-tree.....	190
accept/reject method for sampling.....	70	B+ tree.....	184
activation function in neural network.....	587	back-propagation algorithm.....	589
active learning.....	654	backtracking.....	462
acyclic graph.....	145	backward difference.....	457
AdaBoost.....	598	backward edge of DFS.....	147
AdaBoost.L.....	611	backward error.....	392
AdaBoost.M1.....	611	backward Euler.....	476
AdaBoost.MH.....	611	backward stable algorithm.....	392
Adams-Bashford method.....	483	bag-of-words model.....	552
Adams-Moulton method.....	483	bagged decision tree.....	594
adaptive integration.....	450	bagging.....	542
address persistence.....	6	balanced accuracy.....	560p.
adjacency array.....	146	balanced error rate.....	560
AES.....	308	balanced tree.....	99
agile software development.....	14	band matrix.....	404
agnostic PAC learnability.....	554	barycentric interpolation formula.....	434
algorithm.....	1	Bayesian network.....	612
algorithm engineering.....	2	Bayesian neural network.....	611
algorithm mixing.....	9	Bayesian optimization.....	524
algorithm's contract.....	3	BCH codes.....	305
alias method.....	70	belief logic.....	3
AllDifferent constraint.....	258	Bellman-Ford algorithm.....	153
alternative hypothesis.....	340	Benjamini-Hochberg procedure.....	348
ambiguous data type.....	6	bet on sparsity principle.....	543
amortized resource use.....	5	BFGS algorithm.....	495
ANOVA.....	383	bias of predictor.....	541
ant colony optimization.....	262	BIC statistic.....	633
antithetic variates.....	83	bidirectional search.....	556
AnyBoost.....	597	big data.....	553
anytime algorithm.....	229	bin packing problem.....	226
append operation.....	41	binary constraint.....	257
approval voting.....	354	binary heap.....	138
approximation algorithm.....	227	binary search.....	93
approximation error.....	390, 533	binary symmetric channel.....	293
approximation intervals.....	381	binary tree.....	53
Apriori algorithm.....	652	bipartite graph.....	145
architectural patterns.....	16	bipartite matching.....	156
ARIMA.....	376	bisection method.....	461
arithmetic codes.....	222	bit algorithms.....	54
Arnoldi iteration.....	423	bit mask.....	54
array doubling.....	41, 162	bit-parallelism.....	54
Arrow's impossibility theorem.....	354	bitset.....	56
artificial variable.....	520	black swan problem.....	355
assignment problem.....	158	black-box testing.....	19
AUC.....	608	blackboard pattern.....	16
augmenting path algorithm.....	155	block array.....	44
authentication protocol.....	309	Bloom filter.....	135
authorization pattern.....	17	Bluestein's algorithm.....	430
automated regression testing.....	19	Bonferroni correction.....	347

Bonferroni inequality.....	347	code.....	210
Boole inequality.....	347	codebook.....	215
boosting in machine learning.....	597	coding idioms.....	17
bootstrap BC interval.....	334	colleague matrix.....	438
bootstrap BCa interval.....	334	collision.....	120
bootstrap calibration.....	334	command pattern.....	17
bootstrap method.....	330	commitment protocol.....	309
bootstrap percentile interval.....	334	common random numbers.....	81
bootstrap pivotal interval.....	332	companion matrix.....	471
bootstrap-t interval.....	332	compass search.....	501
bounded alphabet.....	191	competitive ratio.....	5
bracketing search.....	468	complete cubic spline.....	447
branch and bound.....	228	complete logic.....	2
breadth-first search.....	149	complete problem.....	223
Brent method.....	482	complete reorthogonalization.....	426
Brent search.....	521	complexity theory.....	223
Brodal queue.....	144	composite pattern.....	17
broker pattern.....	17	composition method method for sampling.....	70
BrownBoost.....	611	computational basis.....	6
Broyden's method.....	463	concentration inequality.....	555
bubble sort.....	93	concept.....	6
bucket queue.....	143, 666	concurrency patterns.....	17
builder pattern.....	16	condition number.....	403
bundle methods.....	503	Condorcet criterion.....	354
Burrows-Wheeler transform.....	220	confidence interval.....	321
byte code.....	211	confidence region.....	321
BZIP2 compressor.....	221	confidence set.....	321
c-means.....	646	confusion matrix.....	561
cache-oblivious model.....	179	conjugate gradient for optimization.....	679
calibrated Chebyshev interval.....	334	conjugate gradient minimization.....	521
capacity to overfit.....	534	connected components.....	150
Carmichel numbers.....	272	connected graph.....	145
categorical data.....	544	consistent learner.....	536
Cauchy precision.....	393	consistent ODE solver.....	473
CCW orientation test.....	287	constraint processing.....	257
central difference.....	457	constraint, logical.....	257
central limit catastrophe.....	250	context object pattern.....	16
central limit theorem.....	78	contract.....	33
CGNR.....	421	contraction hierarchies.....	160
chaining hashing.....	127	control variates.....	83
change key operation in heap.....	140	convex hull.....	287
chaos engineering.....	19	convexification.....	543
characterization testing.....	20	convolution.....	430
Chebyshev extrema nodes.....	433	coordinate descent.....	489
Chebyshev loss for confidence intervals.....	323	copyable data type.....	6
Chebyshev polynomials.....	436	copyright.....	31
Chebyshev's inequality.....	316	correctly rounded in floating point.....	389
chi-squared test.....	356	cost complexity pruning.....	609
Cholesky Decomposition.....	403	cost folk theorem.....	600
circular linked list.....	44	cost risk in learning.....	600
class imbalance.....	560	cost-sensitive learning.....	600
class label.....	528	counterfactual.....	364
classification.....	528	counting sort.....	88
Clenshaw-Curtis quadrature.....	438, 450	CRC.....	291
Clenshaw's algorithm.....	437	cross edge of DFS.....	148
cluster purity.....	631	cross-entropy method.....	262
clustering.....	528	cross-validation.....	537
CMA-ES algorithm.....	523	cryptographically secure hash function.....	309

cryptographically secure random output.....	68
cubic spline.....	445
cubic splines.....	481
cuckoo hashing.....	137
curse of dimensionality.....	535
Cuthill-McKee algorithm.....	426
Dahlquist's second barrier theorem.....	484
data normalization pattern.....	16
data structure.....	1
data transfer of disk.....	171
data type.....	6
DBSCAN algorithm.....	647, 677
deadlock prevention.....	10
deceptive function.....	523
decision problem.....	223
declarative configuration.....	16
decreasing step random search.....	670
dedicated components pattern.....	18
deduction.....	531
deep learning.....	593
defective matrix.....	417
delta method.....	328
dense graph.....	145
density estimation.....	653
depth-first search.....	147
descent direction.....	462, 491
design patterns.....	16
detailed balance of Markov chain.....	364
diagonal dominance.....	406
dictionary compression.....	216
difference between two strings.....	197
differential evolution.....	510
digital search tree.....	118
Dijkstra's algorithm.....	152
direct methods for sparse matrices.....	426
directed graph.....	145
discrete cosine transform.....	431
discrete Fourier transform.....	429
disjoint set problem.....	60
disk drive.....	171
distance query.....	277
divide and conquer.....	9
DKW test.....	357
Dormand-Prince method.....	474
double-ended heap.....	144
double-ended queue.....	51
doubling error.....	450
doubling lemma.....	200
doubly-linked list.....	44
dual problem.....	520
dual-pivot quicksort.....	94
Dutch national flag problem.....	94
Dvoretzky-Kiefer-Wolfowitz inequality.....	330
dynamic data structure.....	162
dynamic programming.....	9
dynamic regression model.....	370
dynamic sorted sequence.....	95
eager acquisition pattern.....	17
early stopping strategy.....	543
easy problem.....	223
economic utility.....	223
edge of graph.....	145
efficient exponentiation.....	271
eigenvalue.....	409
eigenvector.....	409
elastic net.....	626
elementary functions.....	457
EM clustering.....	646
empirical Bernstein inequality.....	381
empirical distribution function.....	313
empirical likelihood method.....	382
empirical risk.....	531
empirical risk minimization.....	534
encapsulated implementation pattern.....	17
entropy.....	208
Epicurus principle.....	554
equal frequency binning.....	555
equal width binning.....	544
equidistribution of pseudorandom generator.....	65
equivalence testing.....	382
Erdos-Vertesi theorem.....	432
ergodic Markov chain.....	364
estimation error.....	390, 533
estimation of distribution algorithms.....	262
Euclidean algorithm.....	272
Euclidean distance.....	276
event point in computation geometry.....	288
example support in learning.....	535
exceptional shifts.....	414
exchangeability assumption.....	350
execution trace pattern.....	18
expectation maximization.....	646
expected value of a variable.....	64
explicit Rung-Kutta.....	473
exploratory data analysis.....	383
exponent in floating point.....	388
exponential search.....	93, 468
expressive computation basis.....	6
expStd.....	615
extended pattern.....	195
external memory sorting.....	181
external memory vector.....	179
extrapolation methods.....	483
F-measure.....	607
facade pattern.....	17
factor of string.....	191
factory pattern.....	16
failure injection pattern.....	19
fair use doctrine.....	32
faithful rounding.....	457
false coverage rate.....	383
false discovery rate.....	347
family-wise error rate.....	347
fast Fourier transform.....	429
fat node method.....	162
fat-shattering dimension.....	626

feature.....	527	generating all permutations.....	167
feature extraction.....	557	generating all subsets.....	168
feature informativeness error.....	533	generating order statistics.....	77
feature selection.....	546	generating ordered samples.....	77
feature space.....	527	generating random objects.....	76
Fenwick tree.....	679	generating samples from distributions.....	
Fermat's theorem.....	272	Bernoulli distribution.....	75
Fibonacci code.....	210, 222	binomial distribution.....	75
Fibonacci heap.....	144	Cauchy distribution.....	74
file.....	171	exponential distribution.....	73
filter methods.....	556	gamma distribution.....	74
find operation.....	95	geometric distribution.....	75
finite difference methods.....	480	normal distribution.....	74
finite element methods.....	480	Poisson distribution.....	75
finite field.....	291	uniform distribution.....	73
firewall pattern.....	17	genetic algorithms.....	251
first-in-first-out.....	51	genetic local search.....	511
first-in-last-out.....	50	geometric distribution.....	63
first-order entropy.....	208	Gershgorin circle theorem.....	414
first-order logic.....	2	Gibbs sampler.....	383
first-sale doctrine.....	32	Gini index.....	608
five-point stencil.....	457	Givens rotation.....	426
fixed point iteration.....	461	global optimum.....	243
flux design pattern.....	16	Glushkov algorithm.....	194
FNV hash function.....	125	GMRES.....	426
forward difference.....	457	golden section minimization.....	488
forward edge of DFS.....	148	golf hole optimization landscape.....	243
forward error.....	392	Golomb code.....	222
forward Euler algorithm.....	472	Golub-Kahan algorithm.....	411
forward search feature selection.....	549	gradient boosting.....	598
forward stable algorithm.....	392	gradient descent.....	494
forward substitution for matrix equations.....	402	gradient tree boosting.....	611
free list.....	46	Gram-Schmidt orthogonalization.....	393
frequent item combinations.....	651	graph.....	145
Friedman ranks.....	348	graphical models.....	612
Friedman test.....	669	GRASP.....	263
Frobenius norm.....	400	greatest common divisor.....	272
Fubini's theorem.....	454	greedy algorithm.....	9
full adder.....	266	grid search.....	504, 514
functionality reduction pattern.....	18	GZIP compressor.....	222
fuzzing.....	20	halting problem.....	223
Gale-Shapley algorithm.....	158	Hamerly's algorithm.....	646
Galois field.....	294	Hamming distance.....	197
gamma code.....	210	hard problem.....	223
gap statistic for clustering.....	646	hash function.....	120
garbage-collecting freelist.....	46	hash table.....	127
Gauss-Lobatto-Kronrod rule.....	451	HashQ algorithm.....	191
Gauss-Thomas algorithm.....	406	heap order.....	99
Gaussian kernel.....	551	heapify operation.....	140
Gaussian process metamodel.....	524	heapsort.....	140
Gaussian quadrature.....	482	Hermite interpolation.....	481
general additive model.....	627	Hessenberg form of matrix.....	407
general alphabet.....	191	hierarchical clustering.....	647
general position assumption.....	346, 389	hierarchy distance.....	276
generalization in machine learning.....	527	hint information in machine learning.....	527
generalized linear model.....	627	hint noise.....	535
generating all combinations.....	168	histogram density estimator.....	653
generating all partitions.....	168	Hodges-Lehmann estimate.....	343

Hoeffding inequality.....	323	jury theorem.....	594
holdout method.....	532	k-d tree.....	281
Holm procedure.....	347	k-means algorithm.....	571, 637
Horspool algorithm.....	191	k-medoids algorithm.....	640
Householder reduction.....	406	kappa metric.....	608
Huffman codes.....	213	Karush-Kuhn-Tucker conditions.....	520
Hyrum's law.....	21	Kendall's correlation.....	330
I/O model.....	4, 176	kernel.....	550
IB3 algorithm.....	608	kernel density estimator.....	653
ill-conditioned problem.....	392	kernel k-means.....	647
ill-posed problem.....	366	kernel PCA.....	557
imbalanced learning.....	603	kernel PEGASOS algorithm.....	610
implicit graph.....	145	kernel ridge regression.....	626
implicit Rung-Kutta.....	473	kernel support vector regression.....	626
implicit shift algorithm.....	413	kernel trick.....	550
importance sampling.....	83	key in cryptography.....	307
improper means in trade secret.....	31	key-indexed counting sort.....	89
in-order iteration.....	95	Kleinberg's axioms.....	630
include guards.....	36	knapsack problem.....	225
incomplete Cholesky factorization.....	426	known good state pattern.....	18
incremental evaluation.....	240	Koksma-Hlawka inequality.....	360
incremental search operation.....	111	Kolmogorov complexity.....	208
incrementally computable distance.....	276	Kolmogorov test.....	357
inDel string distance.....	197	Kolmogorov-Smirnov statistic.....	357
independence of irrelevant alternatives.....	354	Kraft's inequality.....	210
indexed alphabet.....	191	Kriging.....	524, 608
indexed heap.....	140	Kruskal-Wallis test.....	383
induction.....	531	Kruskal's algorithm.....	160
information query.....	396	Krylov subspace.....	421
insert operation.....	95	L-BFGS algorithm.....	495
insertion sort.....	85	L-stability.....	476
integer programming.....	227	Lagrange interpolation error theorem.....	432
integer programming problem.....	227	Lagrange's theorem.....	123
integer sorting.....	88	Lagrangian of optimization problem.....	520
integration testing.....	19	Lanczos iteration.....	422
interior point methods.....	524	large number.....	265
intermittent search.....	509	large rational.....	273
interpolation.....	432	Las Vegas algorithm.....	5
interpolation search.....	94	lasso regularization.....	578
interval arithmetic.....	423	LASVM algorithm.....	610
interval query.....	284	latin hypercube sampling.....	363
interval tree.....	679	law of large numbers.....	78
intractable problem.....	223	layers pattern.....	16
invariant.....	3	lazy acquisition pattern.....	17
inverse iteration.....	415	lcp.....	89
inverse method for sampling.....	69	lcp array.....	201
inversion in sorting.....	85	LCP augmentation.....	106
inverted index.....	200	LDPC codes.....	302
IO model.....	4	leasing pattern.....	17
irreducible polynomial.....	291	least median of squares.....	626
irrelevant feature.....	547	least recently used cache policy.....	163
iterated local search.....	250	leave-one-out cross-validation.....	555
iterative deepening.....	236	Lebesgue constant.....	432
iterator pattern.....	17	left-learning red-black tree.....	117
Jackson theorem.....	433	length indication.....	210
Jacobi algorithm.....	426	Lewenstein string distance.....	197
join operation.....	60, 95	lexicographic order.....	38
joker character.....	195	lexicographic successor.....	167

license.....	30, 33	median absolute deviation.....	327
likelihood ratio tests.....	341	mergesort.....	87
likelihood, statistical.....	567	Mersenne Twister.....	66
line search.....	491	meta-analysis.....	382
linear code.....	294	metaalgorithm.....	395
linear congruential generator.....	65	Metacost.....	612
linear convergence.....	394	metaheuristic.....	243
linear discriminant analysis.....	609	metaheuristics.....	247
linear hashing.....	190	metamodel.....	627
linear interpolation.....	440	method of lines.....	480
linear kernel.....	551	Mill's inequality.....	81
linear probing hashing.....	131	Miller-Rabin algorithm.....	272
linear programming.....	518	Milne error.....	483
linked list.....	44	minimal computation basis.....	6
Lipschitz function.....	392	minimax polynomial.....	482
load factor.....	127	minimum description length.....	541
local optimum.....	239	minimum spanning tree.....	151
local search.....	239	minimum-cost flow.....	154
locality-sensitive hashing.....	290, 608	missing at random.....	546
location measure.....	324	mixed Chebyshev interval.....	332
LOESS regression.....	627	model.....	540
log likelihood.....	567	model class.....	533, 540
logic of knowledge.....	2	model tree.....	609
logistic regression.....	609	model-of-computation error.....	18
LogitBoost.....	611	model-view-controller pattern.....	16
longest match operation.....	106	modular inverse.....	272
lookup pattern.....	17	modulus of continuity.....	392
loss function.....	528	Monte Carlo algorithm.....	5
lossy compression.....	222	Monte Carlo integration.....	455
LSD sort.....	91	Monte Carlo method.....	78
LUP decomposition.....	400	move down operation in heap.....	139
Lyness rule.....	482	move up operation in heap.....	139
LZW compression.....	216	move-to-front transform.....	219
machine epsilon.....	389	MRG32k3a generator.....	67
machine learning.....	527	mRMR algorithm.....	556
machine model.....	4	multi-armed bandit problem.....	650
machine-independent algorithms.....	425	multi-objective optimization.....	256
MADS algorithm.....	522	multidimensional heap.....	144
main predictor.....	541	multikey quickselect.....	92
manifold hypothesis.....	535	multikey quicksort.....	89
mantissa in floating point.....	388	multimap.....	106
map operations.....	95	multiple comparison.....	347
margin in machine learning.....	566	multiple components pattern.....	18
Markov brothers' inequality.....	435	multiple recursive generator.....	67
Markov chain Monte Carlo.....	364	multiple selection.....	92
Markowitz pivoting.....	426	multiple testing.....	347
MARS regression.....	627	multiset.....	106
massive data model.....	5	Nadaraya-Watson kernel regression.....	626
master theorem.....	9	naive Bayes.....	567
matched pairs experiment.....	346	Natarajan dimension.....	608
material terms in contract.....	33	natural cubic spline.....	445
matrix ∞ -norm.....	400	nearest mean classifier.....	571
matrix inverse.....	403	nearest neighbor classifier.....	569
matrix representation.....	397	nearest neighbor query.....	276
maximum a posteriori.....	554	negative cycle.....	153
maximum flow.....	154	neighborhood of local search solution.....	243
maximum satisfiability.....	226	Nelder-Mead algorithm.....	498
MD5 algorithm.....	310	Nemenyi test.....	348

neural network.....	587	partial match query.....	284
neuron in neural network.....	587	partial rebuilding.....	162
newsworthy results.....	7	partial sort.....	656
Newton-Krylov method.....	482	partially formable data type.....	6
Newton's method.....	462, 494	partially persistent data structure.....	162
no free lunch theorem.....	261, 536, 566	particle swarm optimization.....	523
noncompete agreement.....	34	partition, combinatorial.....	168
nondisclosure agreement.....	34	partitioning of an array.....	86
nonessential behavior.....	3	password.....	307
noninferiority testing.....	382	patent.....	30
nonlinear programming.....	520	path in graph.....	145
nonsolicitation agreement.....	34	patricia trie.....	118
normal distribution.....	63	Patricia trie.....	657
normal in floating point.....	388	Pearson correlation.....	328
normal loss for confidence intervals.....	323	perceptron algorithm.....	610
normalized mutual information.....	645	perfect hashing.....	137
not-a-knot spline.....	445	perfect matching.....	156
NP complexity class.....	223	permutation sort.....	91
null distribution.....	340	permutation tests.....	350
null hypotheses.....	340	persistence.....	173
numeric data.....	544	persistent data structure.....	162
numerical integration.....	449	pessimistic error pruning.....	609
oblique tree.....	609	phase transition.....	261
observational studies.....	364	Picard's theorem.....	472
OCBA heuristic.....	367	pipeline pattern.....	16
Occam razor.....	535	plain old data.....	6
Occam risk minimization.....	534	plane sweep.....	288
offer in contract.....	33	plateau optimization landscape.....	243
one standard error rule.....	354	plug-in estimator.....	314
one vs one reduction.....	564	POD.....	6
one vs rest reduction.....	563	pointer to implementation idiom.....	17
one-time pad.....	307	polar method.....	74
online algorithm.....	5	pop count operation.....	54
open addressing hashing.....	136	pop operation.....	50
optimal Bayes learner.....	533	postcondition.....	3
optimization error.....	390, 533	power of a test.....	342
order of a method.....	394	PPM compressor.....	222
order-decomposable problem.....	162	PRAM.....	4
ordered alphabet.....	191	precision.....	561
ordered tree.....	53	precondition.....	3
ordinal data.....	544	preconditioner.....	421
original Hoeffding inequality.....	324	predecessor operation.....	95
orthogonal matrix.....	406	prediction.....	528
oscillatory integrals.....	454	prediction interval.....	381
out-of-bag risk estimate.....	595	predictor.....	540
out-of-bag samples.....	542	predictor-corrector methods.....	483
outlier.....	535	prefix.....	191
outlier detection.....	653	prefix search operation.....	106
overfitting.....	532	prefix successor operation.....	110
overflow in floating point arithmetic.....	389	prefix-free code.....	210
oversampling for imbalanced learning.....	612	premature termination.....	393
p-value.....	340	prepend operation.....	45
PAC learnability.....	554	prequential error.....	551
pairing heap.....	144, 664	Prim's algorithm.....	151
parent pointer.....	104	primality testing.....	272
Pareto-optimal front.....	257	primitive polynomial.....	291
partial acquisition pattern.....	17	principal component analysis.....	557
partial different equations.....	479	prior likelihood.....	554

priority change operation.....	138	reference counting pattern.....	17
priority queue.....	138	refined MDL.....	555
proactor pattern.....	17	reflection pattern.....	16
probably approximately correct.....	531	refreshing error correction strategy.....	294
property.....	30	regression.....	528
propositional logic.....	2	regular expression.....	194
prototype pattern.....	16	regular function.....	6
proximal algorithms.....	504	regularization.....	541
proxy pattern.....	17	reinforcement learning.....	650
pseudo-dimension.....	616	relative condition number.....	392
pseudorandom number generation.....	64	relative precision.....	390, 393
PSPACE complexity class.....	223	relative underflow in floating point arithmetic.....	389
push operation.....	50	relief algorithm.....	556
push-relabel technique for flow.....	160	Remez algorithm.....	482
q-gram.....	191	remove operation.....	95
QR decomposition.....	407	repeated holdout.....	555
quadratic convergence.....	394	replicated components pattern.....	16, 18
quadratic probing hashing.....	137	representative operations model.....	4
QualityXorshift64.....	66	representer theorem.....	551
quasi-experimental.....	364	reservoir sampling.....	77
quasi-random sequence.....	360	residual of equation.....	403
queue.....	51	resource acquisition is initialization pattern.....	17
quickselect.....	92	resource management patterns.....	17
quicksort.....	85	resource pool pattern.....	17
race condition.....	10	restricted Boltzmann machine.....	611
racing.....	551	ridge regression.....	626
Radau IIA method.....	477	risk of learner.....	528
Rademacher complexity.....	555	RMBoost.....	599
RAID0 disk configuration.....	179	Robbins-Munro algorithm.....	516
Rand index for clustering.....	632	robust component.....	19
random coordinate descent.....	506	role-based access pattern.....	17
random forest algorithm.....	594	rolling hash function.....	120
random hash function.....	120	root-mean-squared error.....	615
random permutation.....	76	rotation forest.....	610
random sequence.....	64	rotations.....	101
random subspace method.....	610	round-off accumulation in floating point arithmetic.....	389
random walk Metropolis algorithm.....	364	RSA algorithm.....	309
randomization ensemble.....	542	rule of three in statistics.....	355
randomization tests.....	350	run length encoding.....	218
randomized Algorithm.....	5	Runge-Kutta methods.....	472
range query.....	281	safe interface pattern.....	16
range search operation.....	95	Saltelli's algorithm.....	359
rank-select data structure.....	203	SAMME.....	598
ranking/unranking technique.....	168	sample average approximation.....	516
rate of convergence.....	394	sample complexity.....	534
rational fraction interpolation.....	481	sample mean.....	78
RBF network.....	611	sample path approximation.....	516
RC4 generator.....	68	sample trimmed mean.....	325
reactor pattern.....	17	sample variance.....	78
real RAM model.....	4	sampling distribution.....	314
real-time model.....	4	sanity checks pattern.....	18
realtime A*	233	satisfiability problem.....	226
recall.....	561	saturation in neural network.....	590
recurrent network.....	611	search space.....	533
recursive best-first search.....	237	secant method.....	462
red-black tree.....	117	second-order logic.....	2
redundant data pattern.....	18	secondary key.....	85
Reed-Solomon codes.....	297		

secret sharing protocol.....	310	star discrepancy.....	360
security patterns.....	17	STAR questions.....	24
seek in disk.....	171	state space.....	234
selection in array.....	92	static code.....	210
selection sort.....	93	static data structure.....	162
self-organizing map.....	647	statistical quality control.....	383
semi-dynamic data structure.....	162	stiff ODE.....	476
semi-supervised learning.....	528	stochastic dominance.....	328
sensitivity analysis.....	397	stochastic gradient descent.....	516
sequential search.....	93	stochastic perturbation stochastic approximation algorithm.....	517
serialization.....	173	strategy pattern.....	17
set.....	106	stratified sampling.....	383
SHA1 algorithm.....	310	stream cipher.....	307
SHA2 algorithm.....	310	stream model.....	4
SHA3 algorithm.....	309	strong duality.....	520
Shapiro-Wilks test.....	357	strong Wolfe conditions.....	492
Shellsort.....	94	strongly connected graph.....	145
shift-and algorithm.....	195	strongly relevant feature.....	546
shifted QR iteration.....	409	structural risk minimization.....	534
shooting method for BVP.....	479	subdifferential set.....	503
shortest paths.....	152	subgradient.....	502
sieve of Eratosthenes.....	166	subgradient descent.....	503
silhouette index.....	633	subnormal in floating point.....	388
simplex method.....	518	successor operation.....	95
simplified AC3 algorithm.....	258	succinct data structure.....	203
simplified silhouette for clustering.....	634	sufficient descent.....	462
Simpson's rule.....	450	suffix.....	191
simulated annealing.....	247	suffix array.....	200
single linkage clustering.....	647	suffix index.....	200
singleton bound.....	294	suffix tree.....	207
singleton pattern.....	16	sum heap.....	71
singular value decomposition.....	411	supersymbol.....	208
singular values of matrix.....	411	support vector.....	583
six sigma.....	383	surrogate loss function.....	543, 597
skip list.....	95	surrogate model.....	627
slack variable.....	520	symmetric and positive definite kernel.....	550
SMO algorithm.....	583	syntax tree.....	203
smoothed resource use.....	5	systematic code.....	294
SMOTE algorithm.....	612	table hash function.....	126
Sobol sequence.....	360	table maker's dilemma.....	457
Sobol's sensitivity index.....	358	tableau system.....	2
social engineering.....	309	tabu search.....	262
soft SVM.....	577	tail recursion.....	9
software alchemy method.....	552	Taylor's theorem.....	392
sorting.....	85	temporal difference learning.....	650
sorting stability.....	85	temporal logic.....	3
sparse graph.....	145	terminating character.....	210
sparse matrix.....	417	ternary treap trie.....	112
sparse solution.....	578	threshold accepting.....	261
Spearman correlation.....	329	tolerance interval.....	381
spectral methods.....	480	tombstone.....	162
spectral clustering.....	642	topological sort.....	150
spectral decomposition.....	409	total rebuilding.....	162
splay tree.....	117	tractable problem.....	223
split operation.....	95	trade secret.....	31
stable algorithm.....	392	trademark.....	32
stable matching.....	157	transfer learning.....	654
stack.....	50		

transformation invariance.....	328	value function.....	650
transformation pipeline pattern.....	16	value function reinforcement learning.....	528
transpose string distance.....	197	value of an object.....	6
trapezoid rule.....	449	Van Dongen distance.....	645
trapezoid rule for ODE.....	476	vanishing gradient in neural network.....	593
traveling salesman problem.....	224	Vapnik's principle.....	534
treap.....	99	variable-neighborhood search.....	262
tree.....	53, 145	variance of a variable.....	64
tree edge of DFS.....	147	variance of predictor.....	541
tree iterators.....	104	VC dimension.....	565
tridiagonal matrix.....	404	vector.....	40
trie.....	111	vector key.....	106
trust region methods.....	522	vector sorting.....	89
truth table.....	2	vertex of graph.....	145
Tukey twicing.....	597, 671	Vigna code.....	222
Tukey's HSD test.....	383	VLDT algorithm.....	612
Turbo codes.....	305	VP tree.....	276
Turnstall code.....	222	Wald interval.....	321
type I error.....	341	weak duality.....	520
UCB1 criteria.....	650	weak learner.....	598
ulp.....	389	weak order relation.....	85
unary code.....	210	weak update.....	162
undecidable problem	223	weakly relevant feature.....	547
undefined behavior.....	3	Weierstrass theorem.....	432
underflow in floating point arithmetic.....	389	weight-balanced tree.....	117
undirected graph.....	145	weighted majority voting.....	611
uniform convergence.....	534	well-conditioned problem.....	392
uniformly b-stable learner.....	537	well-formable data type.....	6
uniformly most accurate.....	321	well-posed problem.....	392
uniformly most powerful test.....	342	white-box testing.....	19
union bound.....	347	Wilcoxon signed rank test.....	343, 668
union-by-height with compression.....	61	Wilson score interval.....	355
union-by-size with compression.....	61	Wilcoxon-Mann-Whitney test.....	343
union-find.....	60	WMMM algorithm.....	197
unique completion token.....	17	Wolfe conditions.....	491
uniquely represented data type.....	6	word RAM model.....	4
unit testing.....	19	Wu-Manber algorithm.....	192
universal hash function.....	120	Xorshift generator.....	65
universally consistent learner.....	536	Xorshift hash function.....	125
unsatisfiable constraint problem.....	257	zero-stability.....	484
use case.....	15	Zobrist hashing.....	126
validation.....	540	Zoutendijk's theorem.....	492
validation of numerical software.....	397		

Made in United States
Orlando, FL
05 June 2022



18504043R00385



Dmytro Kedyk develops financial software at Citadel, having previously worked at Meta and Bloomberg. He has a Masters in Applied Mathematics and a Bachelors in Computer Science from Macaulay Honors at Hunter College. Dmytro is a chess FIDE Master, and curiosity about programming chess AI made him interested in algorithms and finding optimal solutions. He spends much of his spare time researching various useful algorithms and data structures. Dmytro's study of software engineering, particularly software economics and developer productivity, influenced this book's emphasis on simplicity and preference for solution methods applicable to a variety of problems. Please send comments and suggestions to

```
template<typename FUNCTION> pair<double, double> minimizeGS(
    FUNCTION const& f, double xLeft, double xRight,
    double relAbsXPrecision = numeric_limits<double>::epsilon());

assert isfinite(xLeft) && isfinite(xRight) && xLeft <= xRight &&
    relAbsXPrecision >= numeric_limits<double>::epsilon();

double GR = ...;
double xMiddle = xLeft + GR * xRight * (1 - GR),
    yMiddle = f(xMiddle);

while (isLess(xLeft, xRight, relAbsXPrecision)) {
    bool chooseR = xRight - xMiddle > xMiddle - xLeft;
    double prevIfiff = xRight - xLeft, xNew = GR * xMiddle + (1 - GR) * ...
        chooseR ? xRight : xLeft, yNew = f(xNew);
    if (yNew < yMiddle) {
        if (chooseR) xLeft = xMiddle;
        xMiddle = xNew;
        yMiddle = yNew;
    } else chooseR ? xRight = xLeft : xNew;
}

return make_pair(xMiddle, yMiddle);
```

- ✓ Clear explanations of useful algorithms and data structures for many tasks
- ✓ Working, concise, and understandable code
- ✓ Learn something new in every chapter
- ✓ Many new ideas



```
template<typename OBJECT, typename INDEX = Vptree<OBJECT, int,
        typename EuclideanDistance<OBJECT>, Distance>> class kNNClassifier
{
public:
    INDEX instances;
    void learn(int classNumber, OBJECT const* instance,
              INDEX const* instance, classNumber);
    int classify(OBJECT const* instance);
    INDEX nearestNeighor(INSTANCE const* value);
}
```

→ 11475

