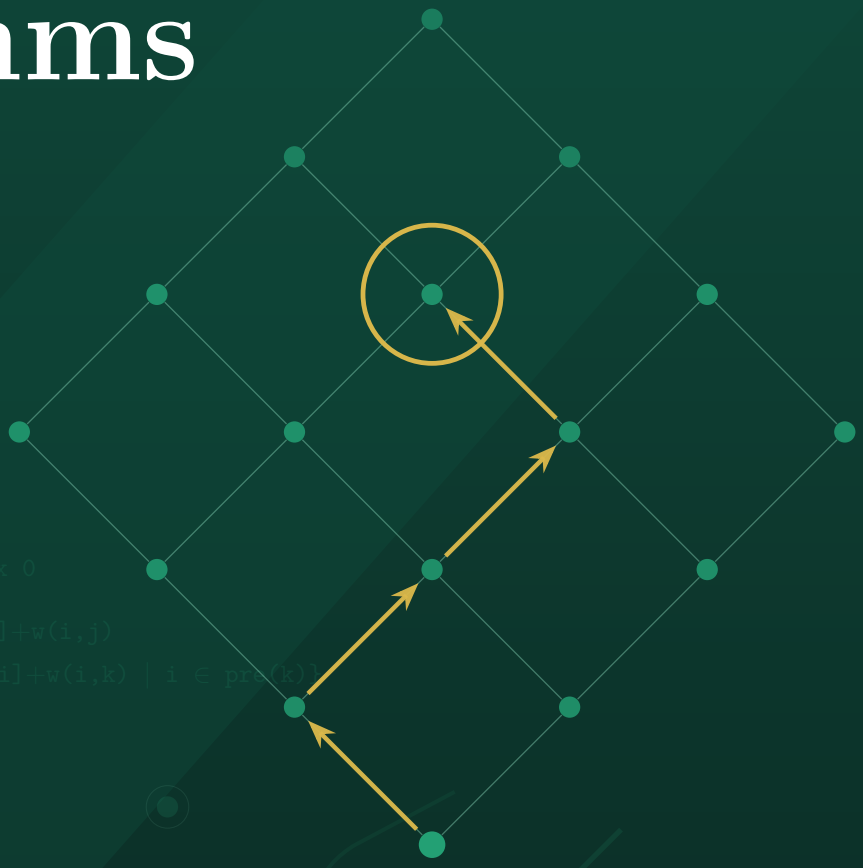
A Systematic Approach to

Sequential Algorithms

Full-Color Edition



LLP-BellmanFord

G[0..n−1]: int // distance from vertex 0

forbidden(j): ∃i ∈ pre(j): G[j] > G[i]+w(i,j)

advance (j): ∀ k ≠ 0: G[k] := min{G[i]+w(i,k) | i ∈ pre(k)}

Vijay K. Garg

Department of Electrical and Computer Engineering
The University of Texas at Austin

A Systematic Approach to Sequential Algorithms: Full-Color Edition

Published by Garg Algorithmics Press

© 2026 Vijay K. Garg

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Limits of Liability/Disclaimer of Warranty: While the author has used good faith efforts to ensure that the information contained in this book is accurate, the author makes no representations or warranties with respect to the accuracy or completeness of the contents of this book and disclaims any implied warranties of merchantability or fitness for a particular purpose. The advice and strategies contained herein may not be suitable for every situation. The author shall not be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages, arising out of the use of or reliance on the information contained in this book.

ISBN: 979-8-9984323-1-6

First Edition, 2026

Vijay K. Garg
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084

To my family

Contents

Preface **xxv**

1 Introduction to Algorithms **3**
1.1 Introduction . . . 3
1.2 What is an Algorithm? . . . 4
1.3 Asymptotic Notation (Big O, Big Omega, Big Theta) . . . 5
1.4 Analyzing Algorithm Efficiency . . . 7
1.5 Common Data Structures . . . 9
1.6 Heaps . . . 11
1.7 Organization of the Book . . . 14
1.8 Summary . . . 15
1.9 Problems . . . 17
1.10 Bibliographic Remarks . . . 18

2 Lattice Linear Predicate Detection **21**
2.1 Introduction . . . 21
2.2 Lattice-Linear Predicates . . . 23
2.3 LLP Notation . . . 28
2.4 A Sequential Implementation of the LLP Algorithm . . . 31
2.5 Properties of the LLP Algorithm . . . 32
2.6 Additional Examples of LLP Algorithms . . . 33
2.7 The *priority* Tag . . . 34
2.8 Dual of Lattice-Linearity . . . 35
2.9 Summary . . . 36
2.10 Problems . . . 38
2.11 Bibliographic Remarks . . . 39

3 The Stable Marriage Problem **41**
3.1 Introduction . . . 41
3.2 Proposal Vector Lattice . . . 42
3.3 Gale–Shapley Algorithm . . . 43
3.4 LLP Stable Matching Algorithm . . . 45
3.5 Extending Algorithms . . . 48
3.6 Summary . . . 49
3.7 Problems . . . 51

3.8 Bibliographic Remarks . . . 52

4 Sorting Algorithms 53
4.1 Introduction . . . 53
4.2 Algorithms Based on Swapping Consecutive Entries . . . 53
4.3 Quicksort . . . 56
4.4 Merge Sort . . . 58
4.5 Lower Bound for Comparison-Based Sorting . . . 59
4.6 Non-comparison Based Sorting . . . 61
4.7 LLP Perspective . . . 64
4.8 Summary . . . 66
4.9 Problems . . . 68
4.10 Bibliographic Remarks . . . 68

5 Graphs 71
5.1 Introduction . . . 71
5.2 Graph Representations . . . 71
5.3 Breadth-First Search (BFS) . . . 73
5.4 Depth-First Search (DFS) . . . 76
5.5 Layering of a Directed Acyclic Graph . . . 78
5.6 Topological Sort . . . 81
5.7 Strongly Connected Components . . . 82
5.8 LLP Perspective . . . 84
5.9 Summary . . . 87
5.10 Problems . . . 88
5.11 Bibliographic Remarks . . . 89

6 Greedy Algorithms 91
6.1 Introduction . . . 91
6.2 Interval Scheduling Problem . . . 92
6.3 Interval Partition Problem . . . 94
6.4 Minimizing Maximum Lateness of Jobs . . . 96
6.5 Huffman Tree . . . 98
6.6 Fractional Knapsack . . . 101
6.7 LLP Perspective . . . 103
6.8 When Greedy Fails . . . 106
6.9 Summary . . . 108
6.10 Problems . . . 109
6.11 Bibliographic Remarks . . . 110

7 The Shortest Path Problem 111
7.1 Introduction . . . 111
7.2 Dijkstra's Algorithm . . . 112
7.3 BellmanFord Algorithm . . . 115
7.4 FloydWarshall Algorithm . . . 118
7.5 Johnson's Algorithm . . . 120

7.6 LLP Perspective . . . 124
7.7 Summary . . . 128
7.8 Problems . . . 130
7.9 Bibliographic Remarks . . . 131

8 The Minimum Spanning Tree Problem 133
8.1 Introduction . . . 133
8.2 Key Properties of a Minimum Spanning Tree . . . 134
8.3 The Find-Union Data Structure . . . 136
8.4 Kruskal's Algorithm . . . 138
8.5 Prim's Algorithm . . . 141
8.6 Boruvka's Algorithm . . . 144
8.7 Comparison of Algorithms . . . 146
8.8 LLP Perspective . . . 147
8.9 Constrained Minimum Spanning Tree . . . 153
8.10 Summary . . . 154
8.11 Problems . . . 156
8.12 Bibliographic Remarks . . . 157

9 Divide and Conquer 159
9.1 Introduction . . . 159
9.2 The Master Theorem . . . 160
9.3 Closest Pair of Points in the Euclidean Space . . . 161
9.4 Counting Inversions in an Array Using Divide and Conquer . . . 164
9.5 Planar Convex Hull . . . 166
9.6 Karatsuba's Multiplication Algorithm . . . 167
9.7 Strassen's Matrix Multiplication . . . 169
9.8 Fast Fourier Transform . . . 171
9.9 LLP Perspective . . . 174
9.10 Summary . . . 177
9.11 Problems . . . 178
9.12 Bibliographic Remarks . . . 178

10 Dynamic Programming 179
10.1 Introduction . . . 179
10.2 Recursion vs Dynamic Programming . . . 180
10.3 Weighted Interval Scheduling . . . 181
10.4 Longest Increasing Subsequences . . . 183
10.5 Optimal Binary Search Tree . . . 185
10.6 Chain Matrix Multiplication . . . 186
10.7 Knapsack Problem . . . 187
10.8 LLP Perspective . . . 188
10.9 Summary . . . 191
10.10 Problems . . . 193
10.11 Bibliographic Remarks . . . 194

11 Network Flow **195**
11.1 Introduction . . . 195
11.2 Definitions . . . 196
11.3 The FordFulkerson Algorithm . . . 197
11.4 Min-Cut and the Max-Flow Min-Cut Theorem . . . 202
11.5 EdmondsKarp Algorithm . . . 204
11.6 Applications of Max-Flow . . . 206
11.7 LLP Perspective . . . 208
11.8 Summary . . . 209
11.9 Problems . . . 210
11.10 Bibliographic Remarks . . . 211

12 Bipartite Matching **213**
12.1 Introduction . . . 213
12.2 A Bipartite Matching Algorithm . . . 214
12.3 Chain Partition of a Poset . . . 217
12.4 Vertex Cover in a Bipartite Graph . . . 219
12.5 Dilworth's Theorem and Maximum Antichain . . . 221
12.6 Reductions Between Structures . . . 222
12.7 Menger's Theorem . . . 223
12.8 Summary . . . 225
12.9 Problems . . . 226
12.10 Bibliographic Remarks . . . 228

13 Intractability **229**
13.1 Introduction . . . 229
13.2 Class P . . . 230
13.3 Polytime Reductions . . . 230
13.4 More NP-Complete Problems . . . 238
13.5 co-NP Class of Problems . . . 245
13.6 Coping with NP-hardness . . . 248
13.7 Summary . . . 248
13.8 Problems . . . 250
13.9 Bibliographic Remarks . . . 251

14 Approximation Algorithms **253**
14.1 Introduction . . . 253
14.2 The Approximation Ratio . . . 253
14.3 Vertex Cover . . . 254
14.4 Set Cover . . . 262
14.5 Approximation Schemes: PTAS and FPTAS . . . 264
14.6 Hardness of Approximation . . . 268
14.7 Summary . . . 269
14.8 Problems . . . 270
14.9 Bibliographic Remarks . . . 270

15 The Housing Allocation Problem **273**
15.1 Introduction . . . 273
15.2 Gale's Top Trading Cycle Algorithm . . . 273
15.3 An LLP Algorithm for the Housing Market Problem . . . 276
15.4 Summary . . . 281
15.5 Problems . . . 282
15.6 Bibliographic Remarks . . . 284

16 Linear Programming **285**
16.1 Introduction . . . 285
16.2 A Worked Example of Linear Programming . . . 286
16.3 Duality in Linear Programming . . . 288
16.4 LP Relaxation and Total Unimodularity . . . 292
16.5 LP Formulations of Combinatorial Problems . . . 294
16.6 Comparison with LLP . . . 299
16.7 Summary . . . 300
16.8 Problems . . . 302
16.9 Bibliographic Remarks . . . 304

17 The Assignment Problem **305**
17.1 Introduction . . . 305
17.2 Problem Formulation . . . 306
17.3 Market Clearing Price . . . 306
17.4 Constrained Market Clearing Price Problem . . . 311
17.5 Certificates of Optimality . . . 313
17.6 Summary . . . 314
17.7 Problems . . . 315
17.8 Bibliographic Remarks . . . 316

18 Horn and 2-SAT Satisfiability **317**
18.1 Introduction . . . 317
18.2 Horn Satisfiability . . . 318
18.3 Arithmetization of Horn Clauses . . . 322
18.4 2-SAT . . . 323
18.5 Summary . . . 327
18.6 Problems . . . 329
18.7 Bibliographic Remarks . . . 330

19 Algorithms in Number Theory **333**
19.1 Introduction . . . 333
19.2 Greatest Common Divisor (GCD) Algorithm . . . 334
19.3 Extended GCD Algorithm . . . 336
19.4 Chinese Remainder Theorem . . . 338
19.5 Viewing Numbers as a Distributive Lattice . . . 339
19.6 Summary . . . 343
19.7 Problems . . . 344

19.8 Bibliographic Remarks . . . 345

20 The Predicate Detection Problem 347
20.1 Introduction . . . 347
20.2 Preliminaries: Computations, Cuts, and Their Lattice . . . 348
20.3 Complexity of Predicate Detection Problem . . . 349
20.4 Detecting Conjunctive Predicates . . . 351
20.5 Succinct Level-k Detection . . . 353
20.6 Recognizing Closure Properties . . . 353
20.7 Detection without Efficient Advancement . . . 355
20.8 LLP Perspective . . . 358
20.9 Summary . . . 358
20.10 Problems . . . 360
20.11 Bibliographic Remarks . . . 360

A Lattice Theory 361
A.1 Introduction . . . 361
A.2 Relations . . . 361
A.3 Partial Orders . . . 362
A.4 Lattices . . . 362
A.5 Distributive Lattices and Birkhoff's Theorem . . . 364
A.6 Problems . . . 365
A.7 Bibliographic Remarks . . . 365

Bibliography 365

Algorithm Index 379

General Index 381

List of Figures

1.1 Comparison of growth rates for n up to 20: linear, quadratic, cubic, and exponential (log scale) . . . 7
1.2 Min-heap as a binary tree . . . 12
1.3 Min-heap as an array . . . 12

2.1 LLP program for the job scheduling problem: (a) using the **forbidden/advance** notation, (b) using the compact **ensure** clause. Both are equivalent; the **ensure** form is used when the advance expression is a monotone function of G. 29
2.2 LLP traversal projected onto $(G[3], G[4])$ after settling $G[1] = 3$ and $G[2] = 5$. The feasible states $\{(7,8), (7,9), (8,9)\}$ are shaded; the least is $(7,8)$. (a) basic formulation (bad evaluation order): advance job 4 to $(4,6)$, then job 3 to $(7,6)$, then job 4 *again* to $(7,8)$. Job 4 is advanced twice. (b) with fixed indices: fix job 3 first (reaching $(7,1)$), then fix job 4 which advances directly to $(7,8)$. Each job advances once. . . . 31
2.3 Finding the GCD of an array via LLP. . . . 33

3.1 Stable matching problem with men's preference list ($mpref$) and women's preference list ($wpref$). The first column of each table (in bold) indexes by man / woman; the remaining cells give that person's ordered preference list, most preferred first. . . . 42

4.1 A decision tree for sorting three elements. Internal nodes are comparisons $i{:}j$ (is $a_i < a_j$?); left branches mean "yes," right branches mean "no." The six leaves correspond to the six permutations. . . . 61
4.2 Lattice of inversion vectors for arrays of size three, applied to $A = [45, 12, 15]$. Each node shows an inversion vector $(a, b, 0)$ and the array obtained by applying the corresponding permutation to A. The sorted array $[12, 15, 45]$ sits at $(2, 0, 0)$ (shaded); sorting reduces to advancing upward in the lattice to that node. . . . 65

5.1 An undirected graph (left), its adjacency matrix (middle), and its adjacency list representation (right). . . . 72
5.2 A directed graph (left), its adjacency matrix (middle), and its adjacency list representation (right). . . . 72

5.3 A weighted directed graph (left), its adjacency matrix with edge weights in place of 1s and ∞ in place of 0 for pairs with no edge (middle), and its adjacency list with each entry annotated by the weight of the corresponding edge (right). 73
5.4 A directed graph used to illustrate BFS and DFS traversals. 74
5.5 BFS tree rooted at v_0 for example 5.1. Every vertex appears at its shortest distance from v_0. 76
5.6 DFS tree rooted at v_0 for example 5.2. The edge $v_2 \to v_3$ of the original graph is a *cross edge* (not part of the DFS tree, since v_3 is discovered earlier via v_4). 78
5.7 A directed acyclic graph used to illustrate layering. 80
5.8 The DAG of figure 5.7 redrawn with each vertex placed on its computed layer. Every directed edge goes from a strictly smaller layer to a strictly greater one. 81
5.9 (a) example directed graph $\mathcal{G}$ with three strongly connected components $\{1, 2, 3\}$, $\{4\}$, and $\{5\}$. (b) the condensation $\mathcal{G}_{\text{scc}}$, with each node labeled by the maximum DFS finishing time $f(S)$ of its component. Edges in the condensation point from larger-f components to smaller, illustrating lemma 5.5. 84

6.1 Huffman tree for six symbols. 101

7.1 A weighted directed graph 113
7.2 A directed graph used as an example for the LLP-FloydWarshall algorithm. . 126

8.1 An undirected weighted graph and its minimum spanning tree, the edges of weights $\{2, 3, 4, 7\}$ 134
8.2 The Cut Property. Of the edges crossing the cut $(S, V \setminus S)$, the one of minimum weight, e, belongs to an MST. 135
8.3 The Cycle Property. The strictly heaviest edge e of a cycle belongs to no MST: deleting it from any tree that contains it splits the tree, and some other edge of the cycle reconnects the two parts more cheaply. 135
8.4 Forest after each step of the Find-Union trace. The rank of root 1 is $0, 1, 1, 2, 2, 2$ across the six panels; ranks of all other roots stay at their initial values. Step 5 shows the effect of path compression: FIND(4) walks $4 \to 3 \to 1$ and then rewires every node on the find path directly to the root. 138
8.5 Prim's algorithm from vertex a: the shaded fragment grows by one vertex per step, each time across its lightest outgoing edge. 143
8.6 Boruvka's algorithm. Every component contributes its minimum weight outgoing edge at once, so the five singleton components become $\{a, b, c\}$ and $\{d, e\}$ in one round, and a single tree in the next. 146
8.7 Sorted adjacency lists for each non-root vertex of the graph in fig. 8.1, with root a. 149

9.1 Recursion tree for $T(n) = 2T(n/2) + O(n)$ with $n = 8$. Every level costs $n = 8$, and there are $1 + \log_2 n = 4$ levels, giving $T(n) = O(n \log n)$. 161

9.2 The δ-strip around the dividing line L, partitioned into $\delta/2 \times \delta/2$ boxes. Each box contains at most one point from its side. Numbers show the sorted-by-y positions. Points p (position 1) and q (position 16) span four rows with two complete rows between them, giving a vertical distance of at least δ. 162
9.3 An example of planar convex hull 166
9.4 Divide-and-conquer for convex hull on the points of Fig. 9.3. The left subset $P_L = \{G, A, F, B\}$ and right subset $P_R = \{C, E, D\}$ each have their hull drawn. The dashed lines show the upper tangent (G–C) and lower tangent (F–E) used to merge the two hulls. 167

10.1 Weighted intervals (weights in parentheses) with optimal selection (thick lines). 183
10.2 Longest increasing subsequence for $A = [3, 10, 2, 1, 20]$. The highlighted $G[4] = 3$ indicates the LIS length. Arrows show one possible LIS: $[3, 10, 20]$. 185

11.1 A flow network, labelled *flow/capacity* on each edge. The flow shown has value $3 + 7 = 10$. The dashed line marks the cut with $S = \{s, a\}$ and $T = \{b, t\}$, whose capacity is also $7 + 3 = 10$: both of its edges are saturated, so no more flow can cross. Max-Flow Min-Cut (Theorem 11.9) says this coincidence is no accident. 195
11.2 An example that shows that the FordFulkerson algorithm may take time proportional to the maxflow in the graph. 203
11.3 Reduction from bipartite matching to max-flow. Every edge in (b) has capacity 1. A maximum flow corresponds to a maximum matching in (a). 207
11.4 Edge-disjoint paths from s to t in a graph with all unit-capacity edges. The solid top path ($s \to a \to c \to t$) and dashed bottom path ($s \to b \to d \to t$) are edge-disjoint; the two remaining thin cross edges ($a \to d$, $b \to c$) are unit-capacity edges that are not used. The minimum s-t edge cut has size 2 (e.g., the two edges out of s), matching the maximum number of edge-disjoint paths. 208
11.5 Flow network with capacities. 210

12.1 Various structures and transformations between them 214
12.2 A bipartite graph 215
12.3 A matching M shown with dashed edges in the bipartite graph 216
12.4 A poset 218
12.5 A strict split of the poset in fig. 12.4 218

13.1 Maximum independent set and minimum vertex cover example. 232
13.2 Relationship between P, NP-complete, and NP, assuming P $\neq$ NP. 235
13.3 Reduction of the 3-SAT instance $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ to independent set. Each clause becomes a triangle (clause gadget). Dashed edges connect inconsistent literals across clauses: $x_1 \leftrightarrow \neg x_1$ and $\neg x_3 \leftrightarrow x_3$. The shaded vertices $\{x_1$ from C_1, x_2 from $C_2\}$ form an independent set of size $2 = m$, corresponding to the satisfying assignment $x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{anything}$. 238
13.4 The same graph as in Figure 13.1, with the clique $Q = \{1, 2, 3\}$ shaded. . . . 239

13.5 Reduction of the 3-SAT instance $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$ to clique. No edges exist within the same clause gadget; an edge joins two vertices from different clauses iff their literals are non-contradictory (gray). The shaded vertices $\{x_2$ in $C_1,\ x_2$ in $C_2,\ \neg x_3$ in $C_3\}$ together with the bold edges form a clique of size $m = 3$, witnessing that ϕ is satisfiable. . . 240
13.6 NP-complete problems and reductions 244
13.7 Relationship between NP, co-NP, and P. NP and co-NP are distinguished by opposite diagonal hatching (not by shade alone); P is drawn within NP $\cap$ co-NP, which also includes problems like integer factorization. SAT is the canonical NP-complete problem; its complement UNSAT is co-NP-complete. It is unknown whether NP = co-NP. 246
13.8 Reductions established or cited in this chapter, where an arrow $A \to B$ means $A \leq_P B$ (NP-hardness transfers from A to B). Cook–Levin (bold box) seeds the chain by giving SAT's NP-hardness from first principles; every other problem inherits NP-hardness through one of these reductions. 249

14.1 Graph for vertex cover examples. 255
14.2 Graph $\mathcal{G}$: triangle $\{1, 2, 3\}$ with pendant edge $\{1, 4\}$. Edges: e_1=$\{1, 2\}$, e_2=$\{1, 3\}$, e_3=$\{2, 3\}$, e_4=$\{1, 4\}$. 256
14.3 From the vertex lattice to the edge lattice. (a) the vertex lattice 2^V of the triangle-with-pendant graph of Figure 14.2; the vertex covers (shaded) form an up-set but are *not* meet-closed — the meet of the minimum covers $\{1, 2\}$ and $\{1, 3\}$ is $\{1\}$, which is not a cover. (b) the edge lattice 2^E of a *separate* two-path instance, with the *explored sublattice* $\mathcal{L}_{\text{vc}} = \{0, 1\}^{M^*}$ shaded ($M^* = \{e_1, e_3\}$ the lex-first maximal matching): a principal ideal of 2^E, closed under meet, whose top is the matching the algorithm returns. The deterministic lex rule never leaves this sublattice, and that is where B_{vc} is lattice-linear. 257
14.4 Connected graph for Example 14.6. Vertex 5 is the hub linking the two branches. 259
14.5 Weighted star $K_{1,3}$ for example 14.9. 262

15.1 Housing market and the matching returned by the top trading cycle algorithm 274
15.2 The top choice graph at the first stage. 276
15.3 LLP-Housing-Market on the running instance 279

16.1 Feasible region of the primal LP, with constraints $x_1 + x_2 \geq 4$ and $x_1 + 3x_2 \geq 6$ (and $x_1, x_2 \geq 0$). The shaded region extends unboundedly to the upper right. The objective $3x_1 + 5x_2$ is minimized at the vertex $x^* = (3, 1)$, where both constraints are tight; the dashed line is the corresponding objective level set $3x_1 + 5x_2 = 14$. 287
16.2 Dual feasible region for the worked example. The vertex $(2, 1)$ maximizes $4y_1 + 6y_2$ at value 14, attained where both constraints $y_1 + y_2 = 3$ and $y_1 + 3y_2 = 5$ are tight. 288
16.3 Flow network for problem 3. 302

17.1 The computation graph for a market with three items and three bidders. The valuation and price of any item is a number between 0 and 4. Solid arrows are the per-item price chain; dashed arrows are the precedence constraints $B \equiv (p[2] \geq 2 \Rightarrow p[1] \geq 3) \wedge (p[1] \geq 2 \Rightarrow p[3] \geq 1)$ 312

18.1 Lattice of Boolean assignments on (x_1, x_2, x_3) for the Horn formula $B = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$. The four satisfying assignments form a meet-closed subset $\{000, 001, 011, 111\}$. 319
18.2 Implication graph for $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee x_3)$. 324
18.3 The two SCCs of the implication graph for the formula in Example 18.13. There are no edges between S_+ and S_- in the SCC-contracted graph, so either order is a valid topological order. Choosing S_- before S_+ makes the rule in Algorithm 2-SAT set all three variables to true. 327

20.1 Transformation from SAT to global predicate detection 350
20.2 State-based model of a distributed computation. 352
20.3 Transformation for Theorems 20.8 and 20.9. 354

A.1 Hasse diagram of a poset 362
A.2 Only the first two posets are lattices. In (iii), $\{b, c\}$ has no least upper bound. 363
A.3 The two smallest nondistributive lattices. 364
A.4 A lattice L, its join-irreducibles $J(L)$, and the lattice of ideals $L_I(J(L))$. . . . 365

List of Algorithms

GCD . . . 5
Factorial . . . 6
MaxElement . . . 6
LinearSearch . . . 7
NestedSum . . . 8
PairSum . . . 8
BinarySearch . . . 9
Hanoi . . . 10
MergeSort-Intro . . . 10
Heapify . . . 12
HeapInsert . . . 13
HeapExtractMin . . . 13
BuildHeap . . . 14

LLP . . . 27
LLP-JobScheduling-Forbidden . . . 29
LLP-JobScheduling-Ensure . . . 29
LLP-JobScheduling-Fixed . . . 30
LLP-Sequential . . . 32
LLP-ArrayGCD . . . 33
Regular . . . 36

Gale–Shapley . . . 44
LLP-StableMarriage-Spec . . . 46
LLP-StableMarriage . . . 47

BubbleSort . . . 54
InsertionSort . . . 55
QuickSort . . . 56
ThreeWayPartition . . . 57
MergeSort . . . 59
MergeTwo . . . 60
CountSort . . . 62
RadixSort . . . 64
LLP-Sort-Adjacent . . . 65
LLP-Sort-Pairwise . . . 66

Traversal . . . 74
BFS-Traversal . . . 75
DFS-Traversal . . . 77

Layering . . . 79
Kosaraju-SCC . . . 83
LLP-Traversal . . . 85
LLP-BFS . . . 85
LLP-Traversal-BFS-priority . . . 86
LLP-Layering . . . 86

Interval . . . 92
IntervalPartition . . . 95
MinMaxLateness . . . 96
HuffmanCoding . . . 99
FractionalKnapsack . . . 102
LLP-IntervalScheduling . . . 104
LLP-MinMaxLateness . . . 105
LLP-FractionalKnapsack . . . 107

Dijkstra . . . 112
BellmanFord . . . 116
FloydWarshall . . . 119
Johnson . . . 122
LLP-Edge-Relaxation . . . 124
LLP-BellmanFord . . . 125
LLP-Dijkstra . . . 126
LLP-FloydWarshall . . . 127
LLP-Johnson . . . 128

Find . . . 137
Union . . . 137
Kruskal . . . 139
Prim . . . 142
Boruvka . . . 145
LLP-Kruskal . . . 148
LLP-Prim . . . 150
LLP-Boruvka . . . 151
Mandatory-MST . . . 153

ClosestPair . . . 163
CountingInversions . . . 165
Karatsuba . . . 168
Strassen . . . 169
FFTMultiply . . . 172
FFT . . . 173
LLP-NearestNeighbor . . . 176
LLP-CountInversions . . . 177

RecFib . . . 180

RecFibMemo . . . 181
WeightedIntervalScheduling . . . 182
LIS . . . 184
OptimalBinarySearchTree . . . 186
Knapsack . . . 188
LLP-WeightedIntervalScheduling . . . 189
LLP-LIS . . . 189
LLP-OptimalBinarySearchTree . . . 190
LLP-Knapsack . . . 191

FordFulkerson . . . 198
EdmondsKarp . . . 205

Seq-AugmentingPath . . . 216
BipartiteMatching . . . 217
LLP-Antichain . . . 219
VertexCoverFromMatching . . . 220

ApproxVertexCover . . . 254
LLP-LexMatchVertexCover . . . 257
LLP-WeightedVertexCover . . . 260
ApproxSetCover . . . 263
FPTAS-Knapsack . . . 265
LLP-ApproxKnapsack . . . 267

TTC . . . 275
LLP-Housing-Market . . . 278

LLP-Assignment . . . 309

LLP-HornSAT . . . 321
2-SAT . . . 326

LLP-GCD . . . 335
LLP-ExtGCD . . . 337
CRT-via-ExtGCD . . . 339

LLP-ConjunctiveDetection . . . 352

List of Tables

2.1 Problems unified by the LLP framework. Each classical algorithm is a specific schedule of the generic LLP algorithm on the corresponding lattice. 22

3.1 Algorithms for the stable marriage problem 50

4.1 Sequential sorting algorithms. For counting sort, r is the range of input values. For radix sort, k is the number of digits and the radix is treated as a constant. 66

5.1 Graph algorithms discussed in this chapter. 87

6.1 Summary of greedy algorithms. 108

7.1 Overview of shortest-path algorithms in this chapter, where n is the number of vertices and m is the number of edges. 111

7.2 Classical and LLP shortest-path algorithms discussed in this chapter. 129

8.1 Minimum spanning tree algorithms discussed in this chapter, where n is the number of vertices and m is the number of edges. 155

9.1 Divide-and-Conquer algorithms 177

10.1 Dynamic programming algorithms discussed in this chapter. 192

11.1 Network-flow algorithms and their applications discussed in this chapter. . . . 209

15.1 Preferences for the agents. 282

16.1 Primal–dual correspondence in standard form. 289

19.1 Summary of the LLP-based number-theory algorithms of this chapter. Here n is the number of GCD inputs and r is the number of CRT congruences. In the LLP-GCD row, $M = \max_i A[i]$; in the CRT row, $M = \prod_j m_j$. 343

Preface

Prerequisites

This book assumes the background of a first course in discrete mathematics (sets, relations, and basic proof technique) and some programming experience, but no prior exposure to algorithm design. Asymptotic (big-O) notation and the handful of data structures used throughout—linked lists, stacks, queues, binary search trees, and heaps—are reviewed in Chapter 1 for readers who want a refresher. No background in lattice theory is assumed: the lattice concepts needed for the LLP perspective are developed from scratch in Chapter 2 and Appendix A.

Approach

This book presents algorithms from two complementary perspectives. The *traditional* perspective develops each algorithm in its classical form: greedy choices, divide-and-conquer recursions, dynamic-programming table fills, network-flow augmenting paths, and so on. The *lattice-linear predicate* (LLP) perspective recasts each problem as a search for an extremal element—typically the least element—of an appropriate distributive lattice that satisfies a Boolean predicate B. An algorithm in the second form maintains a single state vector G, identifies any component that is *forbidden* under B, and *advances* it according to a rule derived from the problem's structure.

The lattice-search perspective brings several advantages:

- **Simpler statements.** Almost every algorithm in this book uses a single state vector G. Any auxiliary state, when present, is syntactic sugar over the underlying lattice machinery.
- **Nondeterministic schedules.** The LLP loop fixes only that *some* forbidden index advances next; the choice of *which* is left open. A proof that holds for the nondeterministic schema applies uniformly to every concrete schedule, including the textbook sequential ones.
- **Vector-based reasoning.** Reasoning about G as a vector in a lattice replaces case analysis with monotonicity arguments.
- **Natural parallelism.** Because the schedule is open, every forbidden index is a candidate for advance in the same round. Many of the algorithms in this book are parallel by inheritance, requiring no separate derivation.

- **Easier correctness proofs.** The standard recipe—prove lattice-linearity of B, exhibit the advance rule, and apply the LLP fixed-point theorem—replaces the algorithm-specific invariant work found in classical presentations.

Guiding principles

The book is organized around three principles.

Derive, do not present. The goal of each chapter is to *derive* the algorithm from the feasibility and optimality structure of the underlying problem, rather than to present a finished algorithm and then verify that it works. The lattice formulation makes this derivation systematic: identify the lattice, write down B, check lattice-linearity, and read off the advance rule.

Prefer nondeterminism. Whenever possible, algorithms are stated nondeterministically. A nondeterministic statement is typically simpler than any of its deterministic refinements, and—crucially—a single proof applies to the entire class of refinements. The reader who wants a sequential schedule, a parallel schedule, or a distributed schedule can specialize the same algorithm to her needs.

Parallel by default. Many algorithms in this book admit a simple parallel form: start from a feasible point and improve it monotonically toward the optimum. The LLP framework exposes this structure explicitly—every forbidden index can advance in the same round—so parallelism need not be retrofitted onto a sequential description.

A standard one-semester course can follow the chapters in order; a suggested cutoff and notes on which later chapters are independent of each other appear in Section 1.7, "Organization of the Book," in Chapter 1.

Companion material

Every algorithm in the book ships as runnable code on the companion website, available at `http://www.ece.utexas.edu/~garg`. The associated GitHub repository containing implementations of all the algorithms is available at `https://github.com/vijaygarg1/systematic-algorithms`. Each algorithm is written once in a lattice language and compiled to Java, Python, C++, and JavaScript. The website also hosts interactive demos that step through the LLP loop on small concrete instances. Both the source code and the demos are intended to help the reader develop intuition for what an LLP algorithm *does*, beyond what the printed page alone can convey.

A set of lecture slides, one deck per chapter, and the TikZ source for the book's figures are also available at my website.

A solution manual covering the end-of-chapter problems is available to instructors on request.

The list of known errors and supplementary material for the book is maintained at `http://www.ece.utexas.edu/~garg`.

Acknowledgments

I thank my co-authors on papers whose results appear in this book: David Alves, Bharath Balasubramanian, John Bridgman, Craig Chase, Himanshu Chauhan, Rohan Garg, Milos Gligoric, Changyong Hu, Sujatha Kashyap, Madan Krishnakumar, Neeraj Mittal, Vinit Ogale, Abdullah Rasheed, Alper Sen, Robert Streit, Brian Waldecker, and Xiong Zheng.

I am especially thankful to Rohan Garg for his thorough review of the manuscript and for many useful comments that have improved the book.

I thank the Department of Electrical and Computer Engineering at The University of Texas at Austin, where I have had the opportunity to develop and teach a course on algorithms. The students in that course have, over many years, sharpened the presentation of nearly every chapter.

This work has been supported in part by many grants from the National Science Foundation; this book would not have been possible without that support.

I am thankful to LaTeX for typesetting this book, and to Claude and ChatGPT, which assisted in editing and figures in the book.

Finally, I thank my family. Without their love and support, this book would not have been conceived.

Vijay K. Garg

Austin, Texas
October 2026

Chapter 1

Introduction to Algorithms

1.1 Introduction

Computers are fast, but algorithms are what make them useful. The same problem can admit algorithms whose running times differ by factors of millions on inputs that already fit comfortably in memory: sorting a billion records takes minutes with a good algorithm but centuries with a bad one. Understanding what makes one algorithm faster than another, and being able to predict how an algorithm scales before running it, is the central skill of algorithm design.

A single perspective runs through the chapters that follow, and it is worth stating up front. Most of the algorithms in this book can be cast as a *search*: the candidate solutions form a structured space, the desired answer is the smallest (or largest) element of that space satisfying a property, and the algorithm walks through the space toward that element. The novelty is in the structure we use to organize the search and the discipline it imposes on what counts as a step toward the answer.

The algorithms in this book all work with a vector of values — a vector of distances from a source vertex, a vector of prices on items, an assignment of one set of agents to another. Two such vectors are compared *componentwise*: one vector is smaller than another if it is smaller (or equal) at every coordinate. Two vectors can also be combined componentwise, either by taking the componentwise minimum or the componentwise maximum. These two combining operations — minimum and maximum coordinate by coordinate — are at the heart of how the framework reasons about progress toward the answer. We will refer to this vector as the algorithm's *state vector*.

Denote the state vector by G. What we ask of G is captured by a Boolean predicate B, and we want the smallest G such that $B(G)$ holds. The framework's central observation is that for many problems B has a closure property called *lattice-linearity*: whenever $B(G)$ is false, some component j of G is provably **forbidden** — every vector that agrees with G at coordinate j and is otherwise componentwise larger still fails B. The algorithm then writes itself: maintain a current state G, find any forbidden coordinate j, advance G at j by the minimum amount that removes the violation, and repeat. The loop terminates with the unique smallest G satisfying

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

B.

This is the *lattice-linear predicate* (LLP) framework, developed formally in Chapter 2. Its appeal is that a single short proof — that B is lattice-linear — replaces the case-by-case correctness arguments traditionally given for individual algorithms. As we will show in later chapters, many classical algorithms turn out to be instances of the LLP schema under appropriate choices of the lattice and the predicate: the Gale–Shapley algorithm for stable marriage, Dijkstra's algorithm for shortest paths, Kruskal's and Prim's algorithms for minimum spanning trees, Bellman–Ford for shortest paths with negative edges, and Kuhn's Hungarian method for the assignment problem all fit the same template. The lattice differs from problem to problem — proposal vectors in stable marriage, distance vectors in shortest paths, edge-inclusion booleans in MST — but the **forbidden** / **advance** loop is the same.

Adopting this perspective brings three practical benefits. First, a single correctness argument carries across a whole family of algorithms — once you have shown that an instance of the framework is lattice-linear, the rest of the analysis is bookkeeping. Second, the same algorithm reused on different lattices solves problems that look superficially unrelated: the same loop computes shortest paths, market-clearing prices, and stable matchings. Third, the LLP recasting often exposes a natural parallel version of a sequential algorithm: any number of independently forbidden coordinates can advance in the same round, so a sequential LLP algorithm typically has a free parallel sibling without any new algorithmic insight required. We make use of all three observations throughout the book. The reader is encouraged to ask, on encountering each new algorithm: *what is the lattice, what is the predicate, and what is the advance rule?*

This chapter is organized as follows. Section 1.2 defines what an algorithm is, using Euclid's GCD as a running example. Section 1.3 introduces asymptotic notation (Big-O, Big-Omega, Big-Theta) for describing growth rates. Section 1.4 analyzes the running time of several familiar algorithms (linear search, binary search, the Tower of Hanoi) to make the asymptotic vocabulary concrete. Section 1.5 reviews the basic data structures (linked lists, stacks, queues, binary search trees) that later chapters take as primitives. Section 1.6 describes the heap data structure and the linear-time Algorithm BuildHeap. Section 1.7 previews the organization of the chapters that follow.

1.2 What is an Algorithm?

An algorithm is a finite, unambiguous sequence of instructions that solves a problem or computes a function, transforming an input into an output in a finite number of steps. It must exhibit *finiteness* (terminates), *definiteness* (each step is precisely specified), and *effectiveness* (each operation is basic and executable). We start with one of the earliest algorithms due to Euclid. Suppose that we are required to find the greatest common divisor (GCD) of two natural numbers a and b (integers greater than or equal to 1). If they are both equal, then we have the answer: a. If a is greater than b, then we claim that it is safe to reduce a to a mod b — the remainder when a is divided by b. If b divides a, the remainder is 0; in that case we set a to b, and the two values are now equal. Similarly, if b is greater than a, then we reduce b to b mod a, or to a if a divides b. The algorithm terminates with both numbers identical and equal to the GCD.

Algorithm GCD: Euclidean algorithm for greatest common divisor

input: a, b: positive integers
output: greatest common divisor of a and b

```
1  while a ≠ b do
2  |  if a > b then
3  |  |  if a mod b = 0 then a := b;
4  |  |  else a := a mod b;
5  |  else
6  |  |  if b mod a = 0 then b := a;
7  |  |  else b := b mod a;
8  |  end
9  end
10 return a
```

Example 1.1 For $a = 48$, $b = 18$: since $a > b$, we compute $48 \bmod 18 = 12$, so a becomes 12. In the next step, $b > a$, so $18 \bmod 12 = 6$, and b becomes 6. In the following step, $a > b$ and $12 \bmod 6 = 0$, so a is set to $b = 6$. Now $a = b = 6$, and the algorithm returns 6.

It is easy to see that the algorithm always terminates. After each iteration, $\max(a, b)$ strictly decreases and both values remain positive integers, so a must equal b after finitely many steps.

We use the *mod* function to update the larger value. This is what makes the algorithm fast: every two iterations shrink $\max(a, b)$ by at least a factor of two, giving a running time of $O(\log(\max(a, b)))$. In fact, after the very first iteration both values are already at most the original $\min(a, b)$ (the value being replaced becomes $a \bmod b < b$, and the other value b was already equal to the original $\min(a, b)$), so the same shrinking argument applied from that point on bounds the number of remaining iterations by $O(\log(\min(a, b)))$; the running time is therefore $O(\log(\min(a, b)))$.

To compute the factorial of a non-negative integer n, Algorithm Factorial uses *recursion*. The base case is $n = 0$; for $n > 0$, the algorithm uses Factorial$(n-1)$ to compute Factorial(n).

As a final simple example of an algorithm, suppose that we are required to find the largest element in an array A. Algorithm MaxElement uses a *for* loop to find this element.

1.3 Asymptotic Notation (Big O, Big Omega, Big Theta)

Asymptotic notation quantifies runtime growth, abstracting constants and lower-order terms to focus on scalability.

Algorithm Factorial: Recursive factorial computation

input: a non-negative integer n
output: $n!$
1 **if** $n = 0$ **then return** *1* ;
2 **return** $n \cdot \mathit{Factorial}(n-1)$

Algorithm MaxElement: Finding the maximum element in an array

input: A: array$[0..n-1]$ of int
output: maximum element in A
1 $max := A[0]$;
2 **for** $i = 1$ **to** $n-1$ **do**
3 | **if** $A[i] > max$ **then** $max := A[i]$;
4 **end**
5 **return** max

- $T(n) = O(f(n))$ if $\exists c, n_0 > 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$. For example, suppose that $T(n) = 2n + 5$. Since $2n + 5 \leq 3n$ for $n \geq 5$, we conclude that $T(n)$ equals $O(n)$.
- $T(n) = \Omega(f(n))$ if $\exists c, n_0 > 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$. For example, suppose that $T(n) = n^2 - n$. Since $n^2 - n \geq 0.5n^2$ for $n \geq 2$, we conclude that $T(n)$ equals $\Omega(n^2)$.
- $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$. For example, suppose that $T(n) = 4n^2 + 3n$, $3n^2 \leq 4n^2 + 3n \leq 5n^2$ for $n \geq 3$. We conclude that $T(n)$ equals $\Theta(n^2)$.

Example 1.2 We first give an example in which $T(n)$ is a linearly growing function of n. Let $T(n) = 7n + 10$. Then, it is easy to see that $T(n)$ equals $O(n)$: $7n + 10 \leq 8n$ for $n \geq 10$. In addition, $T(n)$ equals $\Omega(n)$: $7n + 10 \geq 7n$. Thus, $T(n)$ equals $\Theta(n)$.

Example 1.3 As another example, suppose $T(n) = 2n^3 + 5n^2 + n$. Then $T(n)$ equals $O(n^3)$ because $2n^3 + 5n^2 + n \leq 3n^3$ for $n \geq 6$. Similarly, $T(n)$ equals $\Omega(n^3)$ because $2n^3 + 5n^2 + n \geq 2n^3$. Hence, $T(n)$ equals $\Theta(n^3)$.

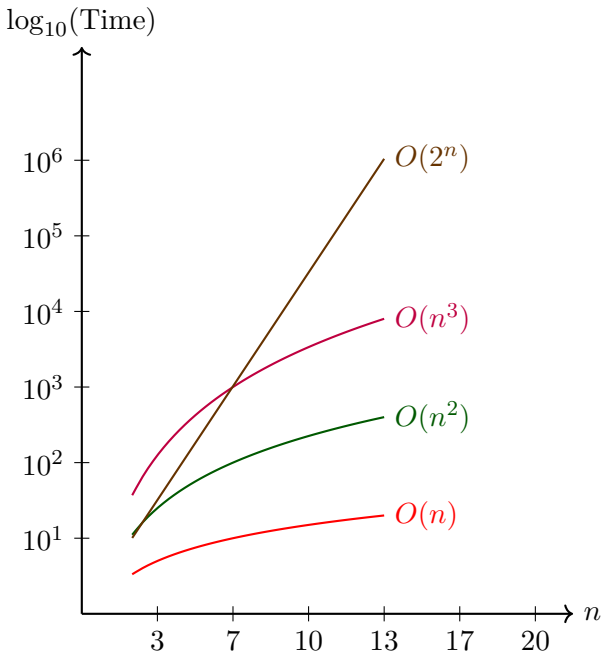


Figure 1.1: Comparison of growth rates for n up to 20: linear, quadratic, cubic, and exponential (log scale)

1.4 Analyzing Algorithm Efficiency

Efficiency analysis evaluates the time and space complexity across the best, worst, and average cases, providing a holistic view of performance. We start with the Algorithm LinearSearch.

Algorithm LinearSearch: Linear search for a key in an array

input: A: array$[0..n-1]$ of int; *key*: int
output: index of *key* in A, or -1 if not found

```
1 for i = 0 to n − 1 do
2 |  if A[i] = key then return i ;
3 end
4 return −1
```

The time complexity of the algorithm is as follows.

- *Best*: $O(1)$, key at $A[0]$.
- *Worst*: $O(n)$, key absent.

- *Average*: $\frac{n+1}{2} = O(n)$, assuming the key is present and equally likely to occur at any position.

We now consider the example of a *nested loop* shown in Algorithm NestedSum. The time

Algorithm NestedSum: Sum of products via nested loops

```
   input: integer n
   output: sum of products
1  total := 0 ;
2  for i = 0 to n − 1 do
3  |  for j = 0 to n − 1 do
4  |  |  total := total + i · j ;
5  |  end
6  end
7  return total
```

complexity of the algorithm is $O(n^2)$. The space complexity of the algorithm is $O(1)$. From now on, the time and space complexity will always refer to the *worst case* unless otherwise specified.

Let us consider another example (Algorithm PairSum), in which our goal is to find the number of pairs in an array A that sum to the provided target sum. The reader should verify that the time complexity of the algorithm is $O(n^2)$.

Algorithm PairSum: Counting pairs that sum to a target

```
   input: A: array[0..n − 1] of int; target: int
   output: number of pairs summing to target
1  count := 0 ;
2  for i = 0 to n − 1 do
3  |  for j = i + 1 to n − 1 do
4  |  |  if A[i] + A[j] = target then  count := count + 1 ;
5  |  end
6  end
7  return count
```

We now consider Algorithm BinarySearch, which speeds up Algorithm LinearSearch. Note that linear search works on any array, whether sorted or not. Binary search, however, requires

the array to be sorted. Given a *key*, the algorithm compares it to the middle element of the array A. If *key* is equal to the middle element, then we have found the index in the array, *mid*, such that $A[mid]$ equals *key*. If *key* is less than the middle element, then we know that it cannot be on the right side of the array A because the array is sorted. We can then restrict our attention to the left side of the array. Finally, if *key* is greater than the middle element, then we can restrict our attention to the right side of the array. If $T(n)$ equals the time complexity of searching in a sorted array of size n, then the recurrence $T(n) = T(n/2) + 1$ is satisfied. Later in this book, we show that $T(n) = O(\log n)$.

Algorithm BinarySearch: Binary search for a key in a sorted array

```
   input: A: array[low..high] of int // sorted
   key: int; low, high: int
   output: index of key in A, or −1 if not found
1  if low > high then return −1 ;
2  mid := ⌊(low + high)/2⌋ ;
3  if A[mid] = key then return mid ;
4  else if A[mid] > key then return BinarySearch(A, key, low, mid − 1) ;
5  else return BinarySearch(A, key, mid + 1, high) ;
```

As another example, we describe the problem called Towers of Hanoi. The Towers of Hanoi problem involves three pegs and n disks of increasing size, initially stacked on one peg in decreasing size order. The goal is to move the entire stack to another peg, obeying two rules: only one disk can be moved at a time, and a larger disk cannot be placed on a smaller disk. The solution requires $2^n - 1$ moves, achieved recursively by moving $n-1$ disks to a spare peg, transferring the largest disk, then moving the $n-1$ disks onto the largest. Algorithm Hanoi shows a recursive solution to this problem. Here, the recurrence is: $T(n) = 2T(n-1) + 1$, $T(1) = 1$. On solving this recurrence, we get $T(n) = 2^n - 1$.

Finally, we consider the problem of sorting an array A using a method called *MergeSort*. The algorithm uses recursion to sort an array using a method called divide-and-conquer. It sorts the left half of the array and the right half of the array, recursively. Now, given that the left side and the right side of the array are sorted, it merges these two sides to get a fully sorted array. The Merge procedure is described later in Section 4.4. In this example, we have the recurrence $T(n) = 2T(n/2) + n$. We later show that the recurrence can be solved to determine that $T(n) = O(n \log n)$.

1.5 Common Data Structures

We briefly review four fundamental data structures: linked lists, stacks, simple queues, and binary search trees. We assume the reader has seen these before; our goal here is to establish the time complexity of each operation for use in later chapters.

Algorithm Hanoi: Moving a tower of n disks between three pegs

input: n: number of disks; $source, aux, target$: pegs
output: the sequence of moves transferring all n disks from $source$ to $target$

```
1 if n = 1 then  Move disk from source to target ;
2 else
3 |   Hanoi(n − 1, source, target, aux) ;
4 |   Move disk from source to target ;
5 |   Hanoi(n − 1, aux, source, target) ;
6 end
```

Algorithm MergeSort-Intro: Merge sort as an example of recursion

input: A: array$[low..high]$ of int; $low, high$: int
output: A: array$[low..high]$ of int `// sorted in non-decreasing order`

```
1 if low < high then
2 |   mid := ⌊(low + high)/2⌋ ;
3 |   MergeSort(A, low, mid) ;
4 |   MergeSort(A, mid + 1, high) ;
5 |   Merge(A, low, mid, high) ;
6 end
```

Linked Lists

A linked list is a linear sequence of nodes, where each node contains data and a reference (pointer) to the next node. A singly linked list supports traversal in one direction; a doubly linked list adds a pointer to the previous node, enabling traversal in both directions.

- **Insert at head**: Create a new node and set its next pointer to the current head. $O(1)$.
- **Insert at tail**: If a tail pointer is maintained, append a new node and update the tail pointer. $O(1)$ with a tail pointer; $O(n)$ otherwise, since we must traverse the entire list.
- **Delete**: Given a pointer to the node, redirect the predecessor's next pointer to skip it. $O(1)$ with a direct pointer (in a doubly linked list or when the predecessor is known); $O(n)$ in a singly linked list if the node must first be found by traversal.
- **Search**: Walk the list from head to tail, comparing each element. $O(n)$ in the worst case.

Stacks

A stack is a Last-In-First-Out (LIFO) structure, typically implemented with an array or a linked list. Elements are added and removed at one end, called the *top*.

- **Push**: Add an element to the top. $O(1)$.
- **Pop**: Remove and return the element at the top. $O(1)$.
- **Top** (or **Peek**): Return the top element without removing it. $O(1)$.

Simple Queues

A simple queue is a First-In-First-Out (FIFO) structure. Elements are added at the rear and removed from the front. A linked-list implementation maintains pointers to both ends.

- **Enqueue**: Add an element at the rear. $O(1)$.
- **Dequeue**: Remove and return the element at the front. $O(1)$ with a linked list or a circular array; $O(n)$ with a naive array implementation that shifts all elements.
- **Front** (or **Peek**): Return the front element without removing it. $O(1)$.

Binary Search Trees

A binary search tree (BST) is a binary tree in which every node's left subtree contains only smaller values and its right subtree contains only larger values. This ordering invariant enables efficient search by eliminating half the remaining candidates at each step.

- **Search**: Starting at the root, compare the target with the current node and recurse into the left or right subtree accordingly. $O(h)$, where h is the height of the tree.
- **Insert**: Search for the correct position (a null child pointer) and place the new node there. $O(h)$.
- **Delete**: Locate the node, then handle three cases: a leaf is simply removed; a node with one child is replaced by that child; a node with two children is replaced by its in-order successor (or predecessor), which is then deleted from its original position. $O(h)$.

In the worst case (a degenerate tree shaped like a linked list), $h = n$. A balanced BST guarantees $h = O(\log n)$, giving $O(\log n)$ time for all three operations.

1.6 Heaps

We now describe a data structure *heap* that is useful for many classical algorithms, such as Dijkstra's shortest path algorithm. A heap is a complete binary tree satisfying the heap property: In a minimum heap, the value of each parent is less than or equal to the values of its children. We focus on min-heaps here, implemented as arrays.

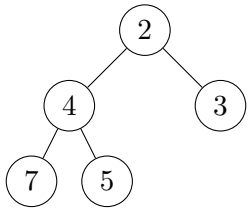


Figure 1.2: Min-heap as a binary tree

2	4	3	7	5
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$

Figure 1.3: Min-heap as an array

Structure and Representation

Figures 1.2 and 1.3 show a min-heap with values 2, 4, 3, 7, 5.

1.6.0.1 Heapify

The Heapify algorithm ensures the min-heap property at a given node by comparing it with its children and swapping with the smallest if necessary, then recursively applying the process downward. It takes an array A, an index i, and a heap size n, running in $O(\log n)$ time due to the height of the tree.

Algorithm Heapify: Restoring the min-heap property at a node

input: array A, index i, heap size n
output: A with the min-heap property restored at node i

$smallest := i$;
$left := 2i + 1$;
$right := 2i + 2$;
if $left < n$ ***and*** $A[left] < A[smallest]$ **then** $smallest := left$;
if $right < n$ ***and*** $A[right] < A[smallest]$ **then** $smallest := right$;
if $smallest \neq i$ **then**
    Swap $A[i]$ and $A[smallest]$;
    Heapify(A, $smallest$, n);
end

1.6.0.2 Insert

The Insert algorithm adds a new value to the heap by placing it at the end of the array and shifting it up to its correct position, comparing it with its parent until the min-heap property is restored. It runs in $O(\log n)$ time, as it traverses up the tree's height.

Algorithm HeapInsert: Inserting a key into a min-heap

input: array A, value key, heap size n
output: A with key inserted, preserving the min-heap property

```
1 n := n + 1 ;
2 A[n − 1] := ∞ ;
3 i := n − 1 ;
4 while i > 0 and A[⌊(i − 1)/2⌋] > key do
5 |   A[i] := A[⌊(i − 1)/2⌋] ;
6 |   i := ⌊(i − 1)/2⌋;
7 end
8 A[i] := key
```

1.6.0.3 Extract-Min

Algorithm HeapExtractMin removes the minimum element (root) by replacing it with the last element, reducing the heap size, and then calling Heapify to restore the min-heap property. It operates in $O(\log n)$ time due to the heapification step.

Algorithm HeapExtractMin: Extracting the minimum from a min-heap

input: array A, heap size n
output: the minimum element; A with it removed, preserving the min-heap property

```
1 if n < 1 then return error ;
2 min := A[0] ;
3 A[0] := A[n − 1] ;
4 n := n − 1 ;
5 Heapify(A, 0, n) ;
6 return min
```

1.6.0.4 Build-Heap

Algorithm BuildHeap constructs a min-heap from an unsorted array by iteratively applying Heapify from the last non-leaf node up to the root. It runs in $O(n)$ time, as the cumulative effect of heapifying all subtrees is linear in the number of nodes.

Algorithm BuildHeap: Building a min-heap from an unsorted array

input: array A, size n
output: A rearranged in place to satisfy the min-heap property
$i := \lfloor n/2 \rfloor - 1$;
while $i \geq 0$ **do**
  Heapify(A, i, n);
  $i := i - 1$;
end

The time complexity of various operations on a heap can be summarized as follows.

- **Heapify**: $O(\log n)$
- **Insert**: $O(\log n)$
- **HeapExtractMin**: $O(\log n)$
- **BuildHeap**: $O(n)$

1.7 Organization of the Book

Chapter 2 presents the general Lattice Linear Predicate (LLP) algorithm. It shows how many combinatorial optimization problems can be solved by a single framework and introduces the notions of forbidden indices and advance operations.

Chapter 3 presents the stable marriage problem. It gives the classical Gale–Shapley algorithm and shows that the LLP algorithm yields a more general version that handles additional constraints.

Chapter 4 discusses various sorting algorithms, including bubble sort, insertion sort, merge sort, quicksort, bitonic sort, and radix sort.

Chapter 5 discusses basic graph algorithms, including various traversal techniques (BFS, DFS), layering of directed acyclic graphs, and connected components.

Chapter 6 covers greedy algorithms: interval scheduling, interval partitioning, scheduling to minimize lateness, and Huffman coding.

Chapter 7 discusses algorithms for computing the shortest path in a graph, including Dijkstra's algorithm, Bellman–Ford, Floyd–Warshall, and Johnson's algorithm. Both classical and LLP formulations are presented.

Chapter 8 describes algorithms for computing the minimum spanning tree in a weighted undirected graph. By considering different lattices on the set of edges, one can derive classical Kruskal's, Prim's, and Boruvka's algorithms.

Chapter 9 discusses the divide-and-conquer strategy, with applications to closest pair, counting inversions, convex hull, Karatsuba multiplication, Strassen matrix multiplication, and the FFT.

Chapter 10 discusses dynamic programming. Problems include the longest increasing subsequence, the optimal binary search tree, the knapsack problem, weighted interval scheduling, and segmented least squares. All of these are shown to be solvable using the LLP algorithm.

Chapter 11 discusses network flow, including the Ford–Fulkerson and Edmonds–Karp algorithms, the max-flow min-cut theorem, and the lattice of min-cuts.

Chapter 12 describes bipartite matching, including augmenting-path algorithms and maximum antichain computation via LLP.

Chapter 13 describes the theory of NP-completeness. This theory relates the difficulty of solving one problem to that of another and includes reductions among SAT, Clique, Vertex Cover, Hamiltonian Path, and Subset Sum.

Chapter 14 discusses approximation algorithms for NP-hard optimization problems, including a 2-approximation for vertex cover, the greedy H_n-approximation for set cover, and a fully polynomial-time approximation scheme (FPTAS) for the knapsack problem. Both classical and LLP formulations are presented, including primal-dual treatments of weighted vertex cover and weighted set cover. A standard course in algorithms may cover all the chapters up to this point. The remaining chapters may be covered depending on the interest of the instructor.

Chapter 15 discusses the housing allocation problem, a one-sided version of the stable marriage problem solved via the top-trading-cycles algorithm and its LLP formulation.

Chapter 16 describes linear programming, another general algorithm design technique. We show that many problems described using the LLP paradigm can also be solved using linear programming.

Chapter 17 discusses the assignment problem (weighted bipartite matching), with LLP algorithms for computing market-clearing prices.

Chapter 18 discusses subclasses of Boolean satisfiability — Horn formulas and 2-SAT — whose satisfiability can be evaluated efficiently, including an LLP formulation for Horn-SAT.

Chapter 19 presents LLP formulations of number-theoretic algorithms: GCD, extended GCD, least common multiple, the Chinese Remainder Theorem, and multiplicative order.

Chapter 20 discusses the predicate detection problem in a poset. It explores special cases of LLP algorithms such as conjunctive predicates and the complexity of determining whether a given predicate is lattice-linear.

1.8 Summary

This chapter provided an introduction to algorithms and their analysis. It covered the definition of an algorithm, asymptotic notation for characterizing growth rates, and efficiency analysis of several fundamental algorithms. The chapter also reviewed essential data structures—linked lists, stacks, queues, binary search trees, and heaps—that are used throughout the book.

Algorithm	**Problem**	**Time Complexity**
Euclid's GCD	Greatest Common Divisor	$O(\log(\min(a, b)))$
Linear Search	Search in unsorted array	$O(n)$
Binary Search	Search in sorted array	$O(\log n)$
Tower of Hanoi	Move n disks	$O(2^n)$
BuildHeap	Build a min-heap from array	$O(n)$

1.9 Problems

1.1. **(Asymptotic Notation Proofs)** Prove or disprove each of the following:

(a) $3^n = O(2^n)$

(b) If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$

(c) If $f(n) = O(g(n))$, then $f(n)^3 = O(g(n)^3)$

1.2. **(Big-O Algebraic Properties)** Prove the following statements.

(a) Prove that for any constant $d > 0$, $f(n) = O(d \cdot g(n))$ iff $f(n) = O(g(n))$.

(b) Prove that if $f(n) = O(h_1(n))$ and $g(n) = O(h_2(n))$ then

$$f(n) + g(n) = O(\max\{h_1(n), h_2(n)\}).$$

(c) Prove that all logarithms are asymptotically equivalent. That is, prove that $\log_a(n) = O(\log_b(n))$ for any positive a and b.

1.3. **(Build Heap Linear Time)** Show that the time complexity of BuildHeap is $O(n)$.

1.4. **(kth Smallest Heap)** Suppose that you have a min Heap of size n. Give the pseudo-code to find and delete the kth smallest element in the heap. What is the time complexity of your algorithm in terms of n and k ? Assume that you have a method available to you for a Heap that can insert any element in the heap in $O(\log n)$ time.

1.5. **(Two-Sum Min Max Heaps)** Suppose that we have an array A of n *distinct* integers in two forms: a min Heap H and a max heap K. We are given the following functions on the min-heap:
$H.getMin()$ // return the least value in the heap H
$H.deleteMin()$ // delete the least value from the heap H
Similarly, we have analogous methods for the max-heap. Give pseudo-code of an efficient program to return a value $A[i]$ in the array A such that there exists j where $A[i] + A[j]$ equals m. If there are no two such distinct entries, then the program should return null. Give the tight asymptotic time complexity of your method.

1.6. **(Function Growth Ranking)** Rank the following functions by growth rate from slowest to fastest. For each consecutive pair in your ordering, state whether the relationship is O, Θ, or neither.

$$n^2, \quad 2^n, \quad n \log n, \quad n!, \quad \log n, \quad n^{1.5}, \quad n \log^2 n$$

1.7. **(Linear Recurrence Solution)** Solve the recurrence $T(n) = 2\,T(n/2)+n$ with $T(1) = 1$ and prove that $T(n) = \Theta(n \log n)$.

1.8. **(k Largest Min Heap)** Given an unsorted array A of n integers and an integer $k \leq n$, describe an algorithm that returns the k largest elements in A in $O(n \log k)$ time using a min-heap of size k. Prove the time bound.

1.9. **(Binary Search Correctness)** Prove the correctness of Algorithm BinarySearch using a loop invariant. State the invariant precisely and show that it holds at initialization, is maintained by each iteration, and implies correctness at termination.

1.10. **(Stack Merging Sums)** Let A be an array of n integers and let S be an initially empty stack. Consider the following algorithm: for $i = 1$ to n, push $A[i]$ onto S. After each push, while S is not empty and the top two elements of S are equal, pop both and push their sum. What does S contain when the algorithm finishes on input $A = [2, 2, 4, 8, 2, 2, 4]$? What is the worst-case time complexity of the algorithm? Justify your answer using an amortized argument.

1.11. **(Euclid GCD Implementation)** Implement both versions of Euclid's GCD algorithm in Java: (a) the subtraction-based version, which replaces the larger value by the difference of the two values, and (b) the version that uses the *mod* operator instead of repeated subtraction (Algorithm GCD from this chapter). Run both on the input pair $(a, b) = (10^9 + 7,\ 10^8 + 7)$ and compare the number of iterations each version requires. Report the ratio.

1.12. **(Min Heap Implementation)** Implement a min-heap in Java with the following operations:
`insert(int key)`, `extractMin()`, `buildHeap(int[] A)`.
Use a 1-based array as the underlying storage. Test your implementation by inserting the values $[15, 10, 20, 17, 8, 25, 18, 5]$ one at a time, then extracting all elements in order; verify that they come out sorted. As a second test, build a heap from the same array in $O(n)$ time using `buildHeap` and verify the output is the same.

1.13. **(Merge Sort Comparisons)** Implement Merge Sort in Java. Instrument your code to count the exact number of element comparisons made during the sort. Run it on random arrays of sizes $n = 10^3,\ 10^4,\ 10^5$, and 10^6. For each n, report the measured comparison count and the ratio $\text{count}/(n \log_2 n)$. Is the ratio approximately constant?

1.14. **(Next Greater Element)** Given an array A of n integers, the *next greater element* of $A[i]$ is the first element to the right of $A[i]$ that is strictly larger than $A[i]$, or -1 if no such element exists. Implement a Java program that computes the next-greater-element array in $O(n)$ time using a stack. Test on $A = [4, 5, 2, 25, 7, 8, 6, 1]$ and verify the output is $[5, 25, 25, -1, 8, -1, -1, -1]$.

1.10 Bibliographic Remarks

Standard algorithms references include Cormen, Leiserson, Rivest, and Stein [CLRS09] ("CLRS"), Kleinberg and Tardos [KT06], Sedgewick and Wayne [SW11], Skiena [Ski08], Dasgupta, Papadimitriou, and Vazirani [DPV08], Erickson [Eri19], Aho, Hopcroft, and Ullman [AHU74], and Horowitz and Sahni [HSR01]. Moore and Mertens [MM11] connect algorithms with computational complexity and statistical physics. Knuth's multivolume *The Art of Computer Programming* [Knu97a, Knu98] covers asymptotic notation and the analysis of fundamental data structures.

For a discussion of the asymptotic notation of the algorithm time complexity, we refer the reader to Knuth [Knu97a]. Heaps were introduced by J.W.J. Williams in 1964 as part of the heapsort algorithm, which uses the heap structure for sorting in $O(n \log n)$ time. The linear-time BuildHeap construction was later analyzed by Floyd, showing its $O(n)$ complexity. Heaps are also used in priority-queue implementations, including Dijkstra's algorithm and Huffman coding.

Chapter 2

Lattice Linear Predicate Detection

2.1 Introduction

Many classical combinatorial optimization problems — stable marriage, shortest paths, minimum spanning trees, market-clearing prices, housing allocation — appear at first glance to require entirely different algorithmic techniques. The Gale–Shapley algorithm proposes and rejects; Dijkstra's algorithm relaxes edges in priority order; Boruvka's algorithm merges components. Yet a closer look reveals a shared structure: in every case, the set of feasible solutions is closed under a natural "meet" operation, and the optimal solution is the *least* element of a *distributive lattice* that satisfies a Boolean predicate expressing feasibility.

Concretely, every lattice in this book is a set of n-dimensional vectors $G = (G[1], \ldots, G[n])$. We say that $G \leq G'$ iff $G[i] \leq G'[i]$ for every component i. The operation *meet* of two vectors G and G' is simply the componentwise minimum of the two vectors. Similarly, the operation *join* of two vectors G and G' is simply the componentwise maximum of the two vectors. A lattice is *distributive* when meet and join distribute over each other. The reader is referred to Appendix A for more details on the lattice theory.

In this chapter, we formalize the notion of Lattice-Linear Predicate (LLP) algorithm. The framework has three ingredients:

1. A *distributive lattice* L that models the search space. Each element of L is a candidate solution represented as a vector of component values.

2. A Boolean *predicate* B on L that captures the goal: $B(G)$ is true when G is a feasible (or optimal) solution.

3. A *lattice-linearity* condition on B: whenever $B(G)$ is false, at least one component j of G is **forbidden** — meaning that no solution above G with the same value of $G[j]$ can satisfy B. Advancing that component brings G closer to a feasible solution.

Once these ingredients are in place, a single generic algorithm repeatedly advances forbidden components until the predicate is satisfied (or the top of the lattice is reached, signaling that no solution exists). The algorithm is *correct by construction*: lattice-linearity guarantees

that every advance makes irrevocable progress, so the algorithm terminates when there is no forbidden component.

The LLP framework unifies a wide range of problems that are traditionally taught with separate algorithms:

Problem	**Classical algorithm**	**LLP lattice**
Stable marriage	Gale–Shapley	proposal vectors
Single-source shortest path	Dijkstra, Bellman–Ford	distance vectors
Minimum spanning tree	Kruskal, Prim, Boruvka	edge-selection vectors
Market-clearing prices	Demange–Gale–Sotomayor	price vectors
Housing allocation	Top-trading cycles	allocation vectors
Horn satisfiability	Unit propagation	truth-assignment vectors
Job scheduling	Critical-path method	completion-time vectors

Table 2.1: Problems unified by the LLP framework. Each classical algorithm is a specific schedule of the generic LLP algorithm on the corresponding lattice.

In each case, the classical algorithm emerges as a *specific schedule* of the generic LLP algorithm — a particular order in which forbidden components are advanced. Different schedules yield different classical algorithms for the same problem (for example, Dijkstra's algorithm and Bellman–Ford are two schedules of the same LLP instance).

A key advantage of the LLP viewpoint is *composability*: because the conjunction of two lattice-linear predicates is itself lattice-linear (Lemma 2.3(c)), one can add side constraints to any of the above problems — forced or forbidden pairs in stable marriage, fairness constraints in shortest paths, budget caps in market-clearing prices — and the same algorithm solves the constrained version with no additional machinery.

This chapter is organized as follows. Section 2.2 formally defines the lattice-linear predicates. These predicates are defined on a distributive lattice. The problem is framed as the search for an element in the lattice that satisfies the predicate. Generally, we are interested in the least element in the lattice that satisfies the predicate. Section 2.3 gives the notation that we use for programs based on lattice-linear predicates. It specifies the lattice that we are working on, the starting element in the lattice, the top element of the lattice, and the predicate **forbidden**, which allows us to advance in the lattice. Section 2.4 gives LLP-Sequential, a general sequential implementation of the algorithm that maintains a set of forbidden indices. Section 2.5 lists some desirable properties of the LLP algorithm. The algorithm is naturally *nondeterministic*. There are multiple ways in which the algorithm can advance in the lattice. The algorithm can advance on multiple components simultaneously. The algorithm is *online*. It does not inspect any element higher than the current element of the lattice that is under consideration. Finally, the algorithm returns an optimal answer for all components. Section 2.6 works through an additional example, computing the greatest common divisor of an array. Section 2.7 introduces the optional *priority* tag, which lets the designer influence the schedule of forbidden-index advances without changing the predicate or the final result. Section 2.8 gives the dual of the lattice-linearity property. Many problems such as the stable marriage problem satisfy not only lattice-linearity but also its dual. For such predicates, one can also start from the top element of the lattice and traverse it downwards

looking for an element that satisfies the given predicate. When applied to the stable marriage problem, one can find the woman-optimal stable marriage in this manner.

2.2 Lattice-Linear Predicates

A Warm-up Trace

Before stating the formal definitions, we walk through a tiny instance to expose the mechanics. Consider a job-scheduling problem with three jobs $1, 2, 3$ that require times $t = [2, 3, 1]$ to complete, where jobs 2 and 3 each depend on job 1. Let $G[j]$ denote the (candidate) completion time of job j. A feasible schedule must satisfy

$$B_{jobs}(G) \equiv \forall j : G[j] \geq t_j \wedge \forall i \in pre(j) : G[j] \geq G[i] + t_j.$$

We search the lattice of completion-time vectors starting from the bottom, $G = [0, 0, 0]$. At each step we look for a single index whose constraint is violated — such an index is called *forbidden* — and *advance* (raise) its value just enough to remove the violation:

G before	violation found	G after
$[0, 0, 0]$	$G[1] < t_1 = 2$	$[2, 0, 0]$
$[2, 0, 0]$	$G[2] < G[1] + t_2 = 5$	$[2, 5, 0]$
$[2, 5, 0]$	$G[3] < G[1] + t_3 = 3$	$[2, 5, 3]$
$[2, 5, 3]$	none — B_{jobs} holds	—

The trajectory $[0, 0, 0] \to [2, 0, 0] \to [2, 5, 0] \to [2, 5, 3]$ is monotone in the componentwise order: no value ever decreases, and the algorithm terminates at a feasible vector. Two properties make this work. First, when index j is identified as forbidden, B_{jobs} cannot be satisfied without raising $G[j]$, so we never overshoot. Second, whenever B_{jobs} is false, at least one forbidden index exists — so progress is always available. The rest of this section formalizes these two properties as the *forbidden* predicate and *lattice-linearity*; a predicate with both admits the simple iterative algorithm sketched above, climbing from the bottom of the lattice straight to the minimum feasible point.

Let L be the lattice of all n-dimensional vectors of reals greater than or equal to the zero vector and less than or equal to a given vector T where the order on the set of vectors is defined by the natural componentwise $\leq$. The minimum element of this lattice is the zero vector. The lattice is used to model the search space of the combinatorial optimization problem. For simplicity, we are considering the lattice of vectors of nonnegative reals; later, we show that our results are applicable to any distributive lattice. The combinatorial optimization problem is modeled as finding the minimum element in L that satisfies a boolean *predicate* B, where B models *feasible solutions*. We are interested in algorithms to solve the combinatorial optimization problem. We will assume that the system maintains as its state the current candidate vector $G \in L$ in the search lattice. We call G, the global state.

Finding an element in the lattice that satisfies the given predicate B is called the *predicate detection problem*. Finding the *minimum* element that satisfies B (whenever it exists) is the combinatorial optimization problem. We now define *lattice-linearity*, which enables efficient

computation of this minimum element. Lattice-linearity was originally introduced in the context of detecting global conditions in a distributed system, where it is simply called linearity. We use the term *lattice-linearity* to avoid confusion with the standard usage of linearity. A key concept in the development of an efficient predicate detection algorithm is that of a *forbidden* state. Given a predicate B and a vector $G \in L$, a state $G[i]$ is **forbidden** (or equivalently, the index i is forbidden) if for any vector $H \in L$, where $G \leq H$, if $H[i]$ equals $G[i]$, then B is false for H. Formally,

Definition 2.1 (Forbidden State) *Given any distributive lattice L of n-dimensional vectors of $\mathbf{R}_{\geq 0}$, and a predicate B, we define $forbidden(G, i, B) \equiv \forall H \in L : G \leq H : (G[i] = H[i]) \Rightarrow \neg B(H)$.*

We define a predicate B as *lattice-linear* with respect to a lattice L if, for any global state G, B is false in G implies that G contains a *forbidden state*. Formally,

Definition 2.2 (Lattice-Linear Predicate) *A boolean predicate B is* lattice-linear *with the polytime algorithm $\mathcal{A}$ with respect to a distributive lattice L generated from a poset P iff $\forall G \in L : \neg B(G) \Rightarrow \mathcal{A}(G) \neq \varnothing \wedge \forall i \in \mathcal{A}(G) : forbidden(G, i, B)$.*

Note that $\mathcal{A}$ returns a nonempty *set* of indices, every one of which must be forbidden; its running time is the cost of producing that set, which has at most n elements.

The algorithm $\mathcal{A}$ is efficient in the size of the poset P. It tells us which indices are forbidden. From now on, the algorithm $\mathcal{A}$ is implicit. We now give some examples of lattice-linear predicates.

1. **Job Scheduling Problem**: Our first example relates to scheduling of n jobs. Each job j requires time t_j for completion and has a set of prerequisite jobs, denoted by $pre(j)$, such that it can be started only after all its prerequisite jobs have been completed. Our goal is to find the minimum completion time for each job. We let our lattice L be the set of all possible completion times. A completion vector $G \in L$ is feasible iff $B_{jobs}(G)$ holds where $B_{jobs}(G) \equiv \forall j : (G[j] \geq t_j) \wedge (\forall i \in pre(j) : G[j] \geq G[i] + t_j)$. B_{jobs} is lattice-linear because if it is false, then there exists j such that either $G[j] < t_j$ or $\exists i \in pre(j) : G[j] < G[i] + t_j$. We claim that $forbidden(G, j, B_{jobs})$. Indeed, any vector $H \geq G$ cannot be feasible with $H[j]$ equal to $G[j]$. The minimum of all vectors that satisfy feasibility corresponds to the minimum completion time.

2. **Prefix Sum Problem**: Our second example relates to (exclusive) prefix sum of an array A with non-negative reals. We are required to output an array G such that $G[j]$ equals sum of all entries in A from 0 to $j-1$. We define G to be feasible iff B_{prefix} holds where $B_{prefix} \equiv (\forall j > 0) : (G[j] \geq G[j-1] + A[j-1])$. Again, it is easy to verify that B_{prefix} is lattice-linear. The minimum vector G that satisfies B_{prefix} corresponds to the exclusive prefix sum of the array A.

3. **Continuous Optimization Problem**: We are required to find minimum nonnegative x and y such that $B \equiv (x \geq y^2/4 + 5) \wedge (y \geq x - 4)$. We view this problem as finding

the minimum (x, y) pair such that B holds. It is easy to verify that B is lattice-linear. If the first conjunct is false, then x is forbidden. Unless x is increased, the predicate cannot become true, even if other variables (y for this example) increase. If the second conjunct is false, then y is forbidden. In this case, $x = 6$ and $y = 2$ is the pointwise minimum solution. For some predicates, there may not be any solution. For example, when $B \equiv (x \geq 2y^2 + 5) \wedge (y \geq x - 4)$, there is no nonnegative (x, y) pair such that B holds (verify this!). The predicate B is still lattice-linear, but by advancing along x and y we go beyond any bounded (x, y).

4. **A Non-Lattice-Linear Predicate**: As an example of a predicate that is not lattice-linear, consider the predicate $B \equiv \sum_j G[j] \geq 1$ defined on the space of two-dimensional vectors. Consider the vector G equal to $(0, 0)$. The vector G does not satisfy B. For B to be lattice-linear, either the first index or the second index should be forbidden. However, none of the indices are forbidden in $(0, 0)$. The index 0 is not forbidden because the vector $H = (0, 1)$ is greater than G, has $H[0]$ equal to $G[0]$ but it still satisfies B. The index 1 is also not forbidden because $H = (1, 0)$ is greater than G, has $H[1]$ equal to $G[1]$ but it satisfies B.

The following lemma is useful for proving lattice-linearity of predicates.

Lemma 2.3 *Let B be any Boolean predicate defined on a lattice L of vectors.*

(a) *Let $f : L \rightarrow \mathbf{R}_{\geq 0}$ be any monotone function defined on the lattice L of vectors of $\mathbf{R}_{\geq 0}$. Consider the predicate $B \equiv G[i] \geq f(G)$ for some fixed i. Then, B is lattice-linear.*

(b) *Let L_B be the subset of the lattice L of the elements that satisfy B. Then, B is lattice-linear implies that L_B is closed under meets.*

(c) *If B_1 and B_2 are lattice-linear then $B_1 \wedge B_2$ is also lattice-linear.*

Proof:

(a) Suppose B is false for G. This implies that $G[i] < f(G)$. Consider any vector $H \geq G$ such that $H[i]$ is equal to $G[i]$. Since $G[i] < f(G)$, we get that $H[i] < f(G)$. The monotonicity of f implies that $H[i] < f(H)$ which shows that $\neg B(H)$.

(b) We show the contrapositive. Let $Y = \{H_1, H_2, \ldots, H_k\}$ be any subset of L_B such that its infimum G does not belong to L_B. Since G is the infimum of Y, for any i, there exists $j \in \{1 \ldots k\}$ such that $G[i] = H_j[i]$. Since $B(H_j)$ is true for all j, for each i, the witness H_j shows i is not forbidden in G. Thus none of the indices in G are forbidden, yet $\neg B(G)$, so lattice-linearity does not hold.

(c) Suppose that $\neg(B_1 \wedge B_2)$. This implies that one of the conjuncts is false and therefore, from the lattice-linearity of that conjunct, a forbidden state exists.

■

Thus lattice-linearity implies meet-closure of the satisfying set, but the converse need not hold: a predicate can be meet-closed without being lattice-linear. Lattice-linearity also requires an efficient algorithm to determine forbidden indices.

For the job scheduling example, we define B_j as $G[j] \geq max(t_j, max\{G[i]+t_j \mid i \in pre(j)\})$. Since $f_j(G) = max(t_j, max\{G[i] + t_j \mid i \in pre(j)\})$ is a monotone function, it follows from Lemma 2.3(a) that B_j is lattice-linear. The predicate $B_{jobs} \equiv \forall j : B_j$ is lattice-linear due to Lemma 2.3(c). Also note that the problem of finding the minimum vector that satisfies B_{jobs} is well-defined due to Lemma 2.3(b).

We now discuss detection of lattice-linear predicates. Definition 2.2 already provides what this requires — the *efficient advancement property*: an efficient (polynomial time) algorithm $\mathcal{A}$ to determine the forbidden indices. This property holds for all the problems considered in this book. Once we determine j such that $forbidden(G, j, B)$, we also need to determine how to advance along the index j. To that end, we extend the definition of forbidden as follows.

Definition 2.4 (α-forbidden) *Let B be any Boolean predicate on the lattice L of all assignment vectors. For any G, j and positive real $\alpha > G[j]$, we define $forbidden(G, j, B, \alpha)$ iff*

$$\forall H \in L : H \geq G : (H[j] < \alpha) \Rightarrow \neg B(H).$$

Given any lattice-linear predicate B, suppose $\neg B(G)$. This means that G must be advanced on all indices j such that $forbidden(G, j, B)$. We use a function $\alpha(G, j, B)$ such that $forbidden(G, j, B, \alpha(G, j, B))$ holds whenever $forbidden(G, j, B)$ is true. With the notion of $\alpha(G, j, B)$, we have Algorithm LLP. The algorithm LLP has two inputs, the predicate B and the top element of the lattice T. It returns the least vector G which is less than or equal to T and satisfies B (if it exists). Whenever B is not true in the current vector G, the algorithm advances on all forbidden indices j in parallel. This simple algorithm can be used to solve a wide variety of combinatorial optimization problems by instantiating different $forbidden(G, j, B)$ and $\alpha(G, j, B)$.

Theorem 2.5 *Suppose that there exists a fixed constant $\delta > 0$ such that $\alpha(G, j, B) - G[j] \geq \delta$ whenever $forbidden(G, j, B)$. Then, the algorithm LLP finds the least vector $G \leq T$ that satisfies B, if one exists.*

Proof: Since $G[j]$ increases by at least δ for at least one forbidden j in every iteration of the *while* loop, the algorithm terminates in at most $\sum_i \lceil T[i]/\delta \rceil$ number of steps.

We show that the algorithm maintains the invariant $(I1)$ that for all indices j, any vector V such that $V[j]$ is less than $G[j]$ cannot satisfy B. Formally, the invariant $(I1)$ is

$$\forall j : (\forall V \in L : (V[j] < G[j]) \Rightarrow \neg B(V)).$$

Initially, the invariant holds trivially because G is initialized to 0. Suppose $forbidden(G, j, B)$. Then, we increase $G[j]$ to $\alpha(G, j, B)$. We need to show that this change maintains the invariant. Pick any V such that $V[j] < \alpha(G, j, B)$. We now do a case analysis. If $V \geq G$, then $\neg B(V)$

Algorithm LLP: Finding the minimum vector at most T that satisfies B

```
input: T: vector (top element), B: predicate
output: the minimum vector G ≤ T that satisfies B, or null
var G: vector of reals initially ∀i : G[i] := 0
1 while ∃j : forbidden(G, j, B) do
2 |   forall j such that forbidden(G, j, B) in parallel :
3 |   |   if α(G, j, B) > T[j] then return null;
4 |   |   else G[j] := α(G, j, B);
5 |   end
6 end
7 return G ;                                   // the optimal solution
```

holds from the definition of $\alpha(G, j, B)$. Otherwise, there exists some k such that $V[k] < G[k]$. In this case, $\neg B(V)$ holds due to $(I1)$.

We can now show Theorem 2.5 using the invariant. First, suppose that the algorithm LLP terminates because $\alpha(G, j, B) > T[j]$. In this case, there is no feasible vector in L due to the invariant (because the predicate B is false for all values of $G[j]$). Now suppose that the algorithm terminates because there does not exist any j such that $forbidden(G, j, B)$. This implies that G satisfies B due to lattice-linearity of B. It is also the least vector that satisfies B due to the invariant $(I1)$.

■

For the job scheduling example, we get an algorithm to find the minimum completion time using $forbidden(G, j, B_{jobs}) \equiv (G[j] < t_j) \vee (\exists i \in pre(j) : G[j] < G[i] + t_j)$, and $\alpha(G, j, B_{jobs}) = \max\{t_j, \max\{G[i] + t_j | i \in pre(j)\}\}$.

For the prefix sum example, we get an algorithm using $forbidden(G, j, B_{prefix}) \equiv (G[j] < G[j-1] + A[j-1])$ and $\alpha(G, j, B_{prefix}) = G[j-1] + A[j-1]$ for all $j > 0$.

We now show, on account of Lemma 2.3(c), that if we have an algorithm for a problem, then we also have one for the constrained version of that problem.

Lemma 2.6 *Let LLP be the algorithm to find the least vector G that satisfies B_1 if one exists. Then, LLP can be adapted to find the least vector G that satisfies $B_1 \wedge B_2$ for any lattice-linear predicate B_2.*

Proof: The algorithm LLP can be used with the following changes:
$forbidden(G, j, B_1 \wedge B_2) \equiv forbidden(G, j, B_1) \vee forbidden(G, j, B_2)$, and
$\alpha(G, j, B_1 \wedge B_2) = \max\{\alpha(G, j, B_1), \alpha(G, j, B_2)\}$.

■

For example, suppose that we want the minimum completion time of jobs with the additional lattice-linear constraint that $B_2(G) \equiv (G[1] = G[2])$. B_2 is lattice-linear with $forbidden(G, 1, B_2) \equiv (G[1] < G[2])$ and $forbidden(G, 2, B_2) \equiv (G[2] < G[1])$. Applying Lemma 2.6, we get an algorithm for the constrained version.

2.3 LLP Notation

We now describe the notation used in LLP algorithms. Fig. 2.1 shows LLP algorithms for the job scheduling problem. We have a single variable G in all the examples shown in Fig. 2.1.

All other variables are derived directly or indirectly from G. G is an array of objects such that $G[j]$ is the state of component j. There are three sections of the program.

The **var section** declares the state variables and their initial values. For example, the **var** section of the job scheduling program in Fig. 2.1 specifies that $G[j]$ is an integer initially equal to $t[j]$.

The **always section** defines additional variables that are derived from G. The actual implementation of these variables is left to the system. They can be viewed as macros.

The third section gives the desirable predicate, either by using the **forbidden** predicate or the **ensure** predicate. The **forbidden** predicate has an associated *advance clause* that specifies how $G[j]$ must be advanced whenever the forbidden predicate is true. For many problems, it is more convenient to use the complement of the forbidden predicate. The *ensure section* specifies the desirable predicates of the form $(G[j] \geq expr)$ or $(G[j] \leq expr)$. The statement **ensure** $G[j] \geq expr$ simply means that whenever component j finds $G[j]$ to be less than $expr$, it advances $G[j]$ to $expr$. Since $expr$ may refer to G, just by setting $G[j]$ equal to $expr$, there is no guarantee that $G[j]$ continues to be equal to $expr$ — the value of $expr$ may change due to changes in other components. We use the **ensure** statement whenever $expr$ is a monotonic function of G and therefore the predicate is lattice-linear.

A program may state more than one **forbidden** predicate, each with its own advance clause, joining them with **&&**. The rules then share a single computation: an index is forbidden when *any* of the rules forbids it, and that rule supplies the advance. On every round each index is tested against every rule, and the program terminates when no index is forbidden by any rule. Algorithm LLP-IntervalScheduling is an example with two rules.

Consider four jobs with execution times $t = (3, 2, 4, 1)$ and prerequisites $pre(2) = \{1\}$, $pre(3) = \{1\}$, $pre(4) = \{2, 3\}$ (job 1 has no prerequisites). The LLP algorithm initializes $G = (3, 2, 4, 1)$. Depending on the evaluation order, a job may need to be advanced more than once.

Algorithm LLP-JobScheduling-Forbidden: Job scheduling via forbidden/advance

input: t: array[1..n] of int; pre: array[1..n] of sets of int
output: G: array[1..n] of int, initially $t[i]$

1 **forbidden**($j \in [1..n]$): $G[j] < \max\{G[i] + t[j] \mid i \in pre(j)\}$
2 $\Rightarrow$ **advance**: $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$;

Algorithm LLP-JobScheduling-Ensure: Job scheduling via ensure

input: t: array[1..n] of int; pre: array[1..n] of sets of int
output: G: array[1..n] of int, initially $t[i]$

1 **ensure**($j \in [1..n]$): $G[j] \geq \max\{G[i] + t[j] \mid i \in pre(j)\}$;

Figure 2.1: LLP program for the job scheduling problem: (a) using the **forbidden/advance** notation, (b) using the compact **ensure** clause. Both are equivalent; the **ensure** form is used when the advance expression is a monotone function of G.

Example 2.7 Suppose we evaluate job 4 before job 3:

1. Job 1 not forbidden. Advance job 2: $G = (3, 5, 4, 1)$.
2. Job 4: requires $\max(G[2] + 1, G[3] + 1) = \max(6, 5) = 6$, but $G[4] = 1$. Advance: $G[4] := 6$. Now $G = (3, 5, 4, 6)$.
3. Job 3: requires $G[3] \geq G[1] + 4 = 7$, but $G[3] = 4$. Advance: $G[3] := 7$. Now $G = (3, 5, 7, 6)$.
4. Job 4 is forbidden *again*: requires $\max(6, 8) = 8$, but $G[4] = 6$. Advance: $G[4] := 8$. Now $G = (3, 5, 7, 8)$.

Job 4 was advanced *twice* because it was processed before its prerequisite job 3 was settled.

An Efficient Reformulation Using Fixed Indices

To avoid such repeated advances, we reformulate the forbidden predicate using a $fixed$ array. Note that this is an implementation refinement: the state is augmented with the $fixed$ array, so the lattice is no longer just the space of completion-time vectors G. Let $fixed[j]$ be a boolean, initially false for all j. A job j is forbidden if $\neg fixed[j] \wedge (\forall i \in pre(j) : fixed[i])$. When j is forbidden, we advance $G[j] := \max\{G[i] + t[j] \mid i \in pre(j)\}$ and set $fixed[j] := true$. Since all predecessors are already fixed (their values will not change), this advance is final — each job becomes forbidden at most once. Algorithm LLP-JobScheduling-Fixed gives the LLP program.

Algorithm LLP-JobScheduling-Fixed: Job scheduling with fixed predicates

input: t: array[1..n] of int; pre: array[1..n] of sets of int
output: G: array[1..n] of int, initially $t[i]$; *fixed*: array[1..n] of boolean, initially *false*

1 **forbidden**($j \in [1..n]$): $\neg$*fixed*$[j] \wedge \forall i \in pre(j) :$ *fixed*$[i]$
2 $\Rightarrow$ **advance**: $G[j] := \max(t[j], \max\{G[i] + t[j] \mid i \in pre(j)\})$; *fixed*$[j] :=$ *true*;

The efficiency of the algorithm may depend on the formulation of $\alpha(G, j, B)$. We will see such optimizations for many problems later in the book, including the shortest-path algorithm.

Example 2.8 Using the same four-job instance, we trace Algorithm LLP-JobScheduling-Fixed. Initialize $G = (3, 2, 4, 1)$ and $fixed = (\text{false}, \text{false}, \text{false}, \text{false})$.

1. Job 1: $pre(1) = \emptyset$, so all predecessors are vacuously fixed. Forbidden. Advance: $G[1] := 3$ (unchanged); $fixed[1] := true$. $G = (3, 2, 4, 1)$.
2. Job 2: $pre = \{1\}$ and $fixed[1]$ is true. Forbidden. Advance: $G[2] := G[1] + t[2] = 5$; $fixed[2] := true$. $G = (3, 5, 4, 1)$.
3. Job 3: $pre = \{1\}$ and $fixed[1]$ is true. Forbidden. Advance: $G[3] := G[1] + t[3] = 7$; $fixed[3] := true$. $G = (3, 5, 7, 1)$.
4. Job 4: $pre = \{2, 3\}$ and both are fixed. Forbidden. Advance: $G[4] := \max(6, 8) = 8$; $fixed[4] := true$. $G = (3, 5, 7, 8)$. All jobs fixed. Done.

Each job is advanced exactly once; when job j is advanced, its predecessors are scanned to compute the maximum, costing $O(|pre(j)|)$ time. Summing over all jobs gives $O(n + m)$ total work (where $m = \sum_j |pre(j)| = |E|$, the number of prerequisite edges).

The $fixed$ predicate ensures that a job is only processed after all its prerequisites are settled, eliminating repeated advances.

Fig. 2.2 illustrates the difference between the two formulations, projected onto the $(G[3], G[4])$ plane. In both traces, $G[1]$=3 and $G[2]$=5 are settled early. A state (g_3, g_4) is

feasible when

$$g_3 \geq G[1] + t[3] = 7 \quad \text{and} \quad g_4 \geq \max(G[2] + t[4],\ g_3 + t[4]) = \max(6,\ g_3 + 1).$$

The feasible states (shaded) are $(7, 8)$, $(7, 9)$, and $(8, 9)$; the least feasible state is $(7, 8)$.

(a) LLP-JobScheduling **(b) With Fixed Indices**

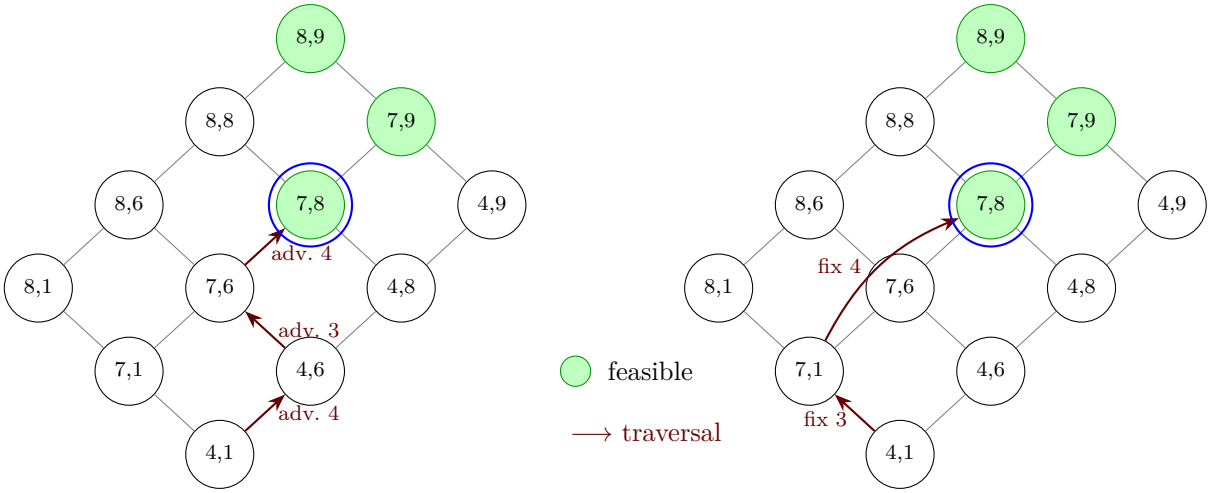


Figure 2.2: LLP traversal projected onto $(G[3], G[4])$ after settling $G[1] = 3$ and $G[2] = 5$. The feasible states $\{(7, 8), (7, 9), (8, 9)\}$ are shaded; the least is $(7, 8)$. (a) basic formulation (bad evaluation order): advance job 4 to $(4, 6)$, then job 3 to $(7, 6)$, then job 4 *again* to $(7, 8)$. Job 4 is advanced twice. (b) with fixed indices: fix job 3 first (reaching $(7, 1)$), then fix job 4 which advances directly to $(7, 8)$. Each job advances once.

2.4 A Sequential Implementation of the LLP Algorithm

The LLP algorithm can be implemented sequentially by maintaining a set S of forbidden indices. Algorithm LLP-Sequential gives LLP-SEQ, a general sequential implementation.

The algorithm maintains the invariant: if S is empty then G is the optimal solution. The set S need not contain all forbidden indices at every step; it may also contain non-forbidden indices, which are simply skipped. The key implementation choices are:

- *Storage for S*: a linked list, queue, stack, or heap depending on the removal strategy. For the shortest path problem, a priority queue keyed by the value the advance would install, $\min\{G[i] + w(i, j) \mid i \in pre(j)\}$, is efficient.
- *Initialization*: insert all indices into S, or only those initially forbidden. For job scheduling with acyclic prerequisites, initialize with jobs having no predecessors.
- *Removal order*: simple FIFO, or priority-based (e.g., smallest tentative distance first for shortest path). A priority affects only how quickly the fixpoint is reached.

Algorithm LLP-Sequential: Sequential algorithm to find the minimum vector $\leq T$ that satisfies B

input: T: vector (top element), B: predicate
output: the minimum vector $G \leq T$ that satisfies B, or null
var G: vector initially $\forall i : G[i] := 0$; S: set of indices

```
1  initialize(S, G) ;                          // add initial forbidden indices to S
2  while ¬S.empty() do
3  |  j := S.remove(G);
4  |  if forbidden(G, j, B) then
5  |  |  G[j] := α(G, j, B);
6  |  |  if G[j] > T[j] then return null;
7  |  |  S.updateForbidden(G, j);
8  |  end
9  end
10 return G
```

- *Updating* S: use a dependency list $dependList(j)$ listing indices whose forbidden status may change when $G[j]$ changes. For graph problems, these are typically the neighbors of j.

The while loop executes at most $O(mn)$ times, since each of n components advances at most m times (where m is the maximum number of time a single component can advance). If each iteration costs $O(n)$ for remove, advance, and update, the total is $O(mn^2)$; for problems where these operations are $O(1)$, the total is $O(mn)$.

2.5 Properties of the LLP Algorithm

The LLP algorithm has many useful properties. We list them here so that the reader can apply them to various problems studied in this book. These properties are applicable to all the problems for which the LLP algorithm is used. We illustrate them with the job scheduling problem; in Chapter 3, we reinforce them in the context of the stable marriage problem.

1. **Nondeterminism in Evaluation of Forbidden Predicate**: Given a global state G, there may be multiple indices j for which $G[j]$ is forbidden. The LLP algorithm is correct irrespective of the order in which these indices are updated. The efficiency may differ depending upon the order, but the correctness is independent of the order. For example, in the job scheduling problem, the minimum completion time vector is the same regardless of the order in which forbidden jobs are advanced.

2. **Parallel Evaluation of Forbidden Predicate**: Suppose that G is shared among different threads, such that thread j is responsible for evaluating $forbidden(G, j)$. While

this thread is evaluating the predicate, other threads may have advanced on other indices, i.e., thread j may have old information of $G[i]$ for $i \neq j$. However, this would still keep the algorithm correct. For example, in job scheduling, multiple forbidden jobs can evaluate their prerequisites and advance their completion times simultaneously, even with stale values of G.

3. **No Lookahead Required for Evaluation of Forbidden Predicate**: The LLP algorithm determines whether an index j is forbidden depending upon only the current global state G. This means that these algorithms are applicable in online settings where the future part of the lattice is revealed only when a forbidden index needs to advance. For job scheduling, the forbidden predicate for job j depends only on the current completion times of its prerequisites — not on jobs that may become forbidden later. For some problems, such as the shortest path problem, the weights on the edges are required to evaluate the forbidden predicate, so the graph must be known and static.

4. **The Optimal Feasible Global State is Optimal for Individual Components**: Suppose that in the job scheduling problem, we are interested in the minimum completion time for a single job j, ignoring all other jobs. The LLP algorithm returns a vector G_{llp} such that $G_{llp}[j]$ equals this individually optimal value. More generally, let G be the global state that is feasible and optimal with respect to index i, i.e., for all feasible H, $G[i] \leq H[i]$. Let G_{llp} be the global state computed by the LLP algorithm, then $G[i] = G_{llp}[i]$. This follows from the meet-closure property of feasible states. If $G[i] < G_{llp}[i]$, then the global state given by $G \sqcap G_{llp}$ is also feasible and strictly smaller than G_{llp}, contradicting that G_{llp} is the least global state that satisfies the feasibility predicate.

2.6 Additional Examples of LLP Algorithms

To further illustrate the generality of the LLP framework, we present an additional example: computing the greatest common divisor (GCD) of an array. Fig. 2.3 shows the LLP program for this problem.

Algorithm LLP-ArrayGCD: Finding the GCD of an array via LLP

input: A: array$[0..n-1]$ of int
output: G: array$[0..n-1]$ of int, initially $A[i]$

1 **forbidden**$(j \in [0..n-1])$: $\exists i \in [0..n-1] : G[j] > G[i]$
2 $\Rightarrow$ **advance**: **if** $G[j] \bmod G[i] = 0$ **then** $G[j] := G[i]$ **else** $G[j] := G[j] \bmod G[i]$;

Figure 2.3: Finding the GCD of an array via LLP.

For computing the GCD of an array of positive integers A, the lattice is ordered by

componentwise $\geq$ (decreasing order). Component j is forbidden if $G[j] > G[i]$ for some i. In this case, we replace $G[j]$ by $G[i]$ if $G[i]$ divides $G[j]$; otherwise, we replace $G[j]$ by $G[j] \bmod G[i]$. When the algorithm terminates, all components are equal to the GCD of A. The invariant is that every common divisor of A is also a common divisor of G. This holds initially because $G = A$. After each advance, if $G[i]$ divides $G[j]$, then every divisor of $G[i]$ also divides $G[j]$, so setting $G[j] := G[i]$ preserves the invariant. If $G[i]$ does not divide $G[j]$, the common divisors of $G[i]$ and $G[j]$ are exactly those of $G[i]$ and $G[j] \bmod G[i]$.

2.7 The *priority* Tag

The **forbidden** and **advance** clauses of an LLP program specify *which* indices can be advanced and *how* to advance them, but they deliberately leave the *order* in which forbidden indices are processed unspecified. Every fair schedule that keeps advancing forbidden indices reaches the same least fixed point, so correctness is independent of the order. For efficiency, however, the order often matters. The LLP framework provides an optional *priority* tag that lets the designer influence the schedule without changing the predicate.

An LLP program may carry a third clause alongside **forbidden** and **advance**:

$$\textbf{priority}(j)\text{: } \textit{expression}$$

where *expression* is a real-valued (or ordinal-valued) function of the current state G and the index j. The semantics is:

> *Whenever more than one index is forbidden, remove (and advance) the forbidden index j whose **priority**(j) is smallest first, breaking ties arbitrarily.*

Because the least-fixed-point solution is independent of the schedule, adding a *priority* clause never changes the final answer of a lattice-linear program; it can only change the running time and, in some cases, the number of advance steps. A typical implementation maintains a priority queue keyed on **priority**(j) that contains all currently forbidden indices. After each advance, any indices whose forbidden status may have changed (either newly forbidden, or newly not forbidden) are inserted into or removed from the queue. The choice of data structure (binary heap, bucket queue, Fibonacci heap) depends on the range and update pattern of the priority.

We now give some examples where the tag *priority* will come in handy.

Breadth-first search. In the LLP formulation of BFS (Chapter 5), an index j is forbidden when some predecessor i has $G[j] > G[i] + 1$. Setting

$$\textbf{priority}(j) := \min\{G[i] + 1 \mid i \in \mathit{pre}(j)\}$$

— the value the advance would install, not the current $G[j]$ — processes vertices in increasing order of their tentative distance, matching the classical BFS-with-queue schedule.

A *priority* tag chooses among forbidden indices.

Many dynamic-programming recurrences have the form

$$G[j] = \min_{i \in pre(j)} (G[i] + w(i, j)),$$

on a DAG (e.g., shortest path in a DAG, longest increasing subsequence ending at each index, optimal binary search tree). Taking

$$\textbf{priority}(j) := \text{topological-order rank of } j$$

forces predecessors to be fixed before their successors, so every $G[j]$ is advanced exactly once — recovering the familiar "fill the DP table in topological order" schedule. Chapter 10 exploits exactly this pattern.

In every case, correctness is inherited from lattice-linearity; the *priority* clause only changes the order of evaluation and thus the running time. This separation of concerns — correctness from the predicate, efficiency from the schedule — is a recurring theme throughout the book.

2.8 Dual of Lattice-Linearity

We briefly discuss detection of predicates when the search starts from the top of the lattice in addition to the bottom of the lattice. Many predicates, such as $B_{stableMarriage}$ satisfy not only the lattice-linearity but also its dual. Thus, the set of elements satisfying these predicates is closed not only for the meet operation but also for the join operation. In addition, whenever the predicate is false, we can efficiently determine the forbidden indices. If we are okay with returning either of the elements as our final answer, then searching for any of the elements in parallel may speed up the algorithm by a factor of the height of the lattice. The dual of the lattice-linear property can be formally defined as follows.

Definition 2.9 (dual of Lattice-Linearity Property) *A Boolean predicate B has the dual of lattice-linearity property with the polytime algorithm $\mathcal{A}$ with respect to a distributive lattice L generated from a poset P if*

$$\forall G \in L : \neg B(G) \Rightarrow \mathcal{A}(G) \neq \varnothing \wedge \forall i \in \mathcal{A}(G) : \mathit{dual\text{-}forbidden}(i, G, B)$$

where $\mathit{dual\text{-}forbidden}(i, G, B)$ is defined as

$$\forall H \leq G : (H[i] = G[i]) \Rightarrow \neg B(H).$$

The above definition states that whenever B is false in G, the algorithm $\mathcal{A}(G)$ returns a nonempty set of indices, each with the property that any global state H less than G that matches G on that index also has B false.

We next define a predicate B as a regular predicate when it satisfies lattice-linearity as well as its dual.

Definition 2.10 (Regular Predicate) *A predicate B is regular in a lattice L iff B is lattice-linear and also dual lattice-linear.*

From the definition of regular predicates, it follows that the subset of elements that satisfy B is a sublattice of L. Since a sublattice of a distributive lattice is also distributive, we conclude that the subset of elements in L satisfying B is also distributive.

For a regular predicate, one can search for the satisfying element starting from both the bottom and the top of the lattice, as shown in Algorithm Regular.

Algorithm Regular: Detecting predicates with the efficient advancement property and its dual

```
   input: B: a regular predicate on lattice L
   output: an element of L satisfying B, or false
   var G: element of L initially ⊥ // the bottom element of L
 1 Z: element of L initially ⊤ (the top element of L);
 2 while ¬B(G) and ¬B(Z) do
 3 |   forall i: forbidden(i, G) :
 4 |   |   if G cannot be advanced on i then return false;
 5 |   |   else advance G on i;
 6 |   end
 7 |   forall j: dual-forbidden(j, Z) :
 8 |   |   if Z cannot be retreated on j then return false;
 9 |   |   else retreat Z on j;
10 |   end
11 end
12 if B(G) then return G;
13 else return Z ;                                  // an optimal solution
14 ;
```

Applying the idea to the stable marriage problem, one can search for the stable marriage starting from the top choices for all men (the $\bot$ element) of the lattice (in parallel) with the bottom choices for all men (the $\top$) element of the lattice. This algorithm will traverse the distance in a lattice given by the minimum of the distance of a stable marriage from the top or the bottom. Hence, depending upon the distance of man-optimal and man-pessimal stable marriages, it will return the stable marriage that is closer to the bottom or the top of the lattice.

In the above analysis, we ignore the 2 penalty factor that we incur because we run the algorithm from both the bottom and top of the lattice.

2.9 Summary

This chapter presented the Lattice-Linear Predicate (LLP) algorithm, a general method for finding the least element in a distributive lattice satisfying a given predicate. The LLP algorithm is nondeterministic and online: it does not inspect any element of the lattice above the current state. We showed that lattice-linear predicates are closed under conjunction, which means that constrained versions of problems are solved by the same algorithm with no

additional machinery.

We also discussed the dual of lattice-linearity, which enables traversal from the top of the lattice downward. For predicates that are both lattice-linear and dual lattice-linear (called *regular* predicates), one can search from both ends simultaneously.

2.10 Problems

2.1. **(Acyclic Relation Predicate)** Let G be an n-dimensional vector of positive numbers. Let *pred* be a binary acyclic relation on $[n]$. Show that the predicate $B \equiv \forall (i,j) \in pred : G[j] \geq G[i] + 1$ is a lattice-linear predicate.

2.2. **(Order Ideal Predicate)** Let $(X, \leq)$ be a poset. A subset $Y \subseteq X$ is an order ideal if it satisfies

$$\forall u, v \in X : (v \in Y) \wedge (u \leq v) \Rightarrow (u \in Y)$$

Show that the predicate $B(Y) \equiv$ "Y is an order ideal of $(X, \leq)$" is lattice-linear in the boolean lattice of all subsets of X.

2.3. **(Precedence Job Scheduling Trace)** Consider the job scheduling instance with $n = 5$ jobs, processing times $t = [2, 3, 1, 4, 2]$, and precedence constraints $1 \to 3$, $1 \to 4$, $2 \to 4$, $3 \to 5$, $4 \to 5$. Trace the execution of Algorithm LLP-JobScheduling-Fixed. At each step, state which indices are forbidden, show the advance, and give the updated vector G. What is the final completion-time vector and the critical path?

2.4. **(Testing Lattice-Linearity)** Determine whether each of the following predicates on vectors $G \in \mathbb{Z}^n_{\geq 0}$ is lattice-linear on the componentwise lattice. Prove your answer.

(a) $B_1(G) \equiv \forall i < n : \; G[i] \leq G[i+1]$ (the vector is non-decreasing).

(b) $B_2(G) \equiv G[1] + G[2] + \cdots + G[n] \geq k$ for a fixed constant k.

(c) $B_3(G) \equiv \max_i G[i] - \min_i G[i] \leq d$ for a fixed constant d (bounded range).

2.5. **(LCM Lattice Linear)** Given an array $A[1..n]$ of positive integers, the *least common multiple* (LCM) of A is the smallest positive integer divisible by every $A[i]$. Formulate the LCM computation as an LLP problem: define the lattice, the state vector, the predicate B, the forbidden condition, and the advance operation. Prove that B is lattice-linear.

2.6. **(Release Time Constraint)** Consider the job scheduling problem with the additional constraint that each job i has a *release time* $r[i]$: job i cannot start before time $r[i]$, so its completion time must satisfy $G[i] \geq r[i] + t[i]$.

(a) Show that the release-time predicate $B_r(G) \equiv \forall i : \; G[i] \geq r[i] + t[i]$ is lattice-linear.

(b) Using Lemma 2.3(c), give the combined forbidden and advance operations for the constrained problem (precedence constraints $\wedge$ release-time constraints).

2.7. **(Permutation Not Lattice Linear)** Prove that the predicate $B(G) \equiv$ "G is a permutation of $(1, 2, \ldots, n)$" is *not* lattice-linear on the componentwise lattice of vectors in $\{1, \ldots, n\}^n$.

2.8. **(Incremental Job Insertion)** A least feasible completion vector G^* has been computed by LLP for the job-scheduling predicate B_{jobs} on jobs $1, \ldots, n$. A new job $n+1$ arrives with processing time t_{n+1} and a set of prerequisites $pre(n+1) \subseteq \{1, \ldots, n\}$. Give an algorithm that updates G^* to the new least feasible vector G^{**} on $n+1$ jobs without re-running LLP from $\perp$.

2.9. **(Closure Properties of Lattice-Linearity)** Prove or disprove the following:

(a) If B is lattice-linear, then $\neg B$ is also lattice-linear.

(b) If B_1 and B_2 are lattice-linear, then $B_1 \vee B_2$ is also lattice-linear.

2.10. **(Lattice-Linear but Not Regular)** Give an example of a predicate that is lattice-linear but not regular.

2.11 Bibliographic Remarks

The Lattice-Linear Predicate (LLP) algorithm is a general technique for designing algorithms for combinatorial optimization problems. The lattice-linearity property was introduced by Chase and Garg [CG98] in the context of detecting global predicates in distributed computations. In that setting, a distributed system of n processes generates a lattice of global states, and the goal is to find the least global state satisfying a given predicate. The key observation — that if the predicate is lattice-linear, a single pass through the lattice suffices — was later applied to algorithm design.

The application of LLP to the stable marriage problem is shown in Garg [Gar17], and its systematic application to algorithm design — including the shortest path problem, the assignment problem, and many others — is developed in Garg [Gar20b]. The LLP framework has since been extended to dynamic programming problems [Gar22], to the housing allocation problem [Gar21], and to the minimum spanning tree problem [AG22]. Gupta and Kulkarni extend LLP algorithms for deriving self-stabilizing algorithms [GK23, GK21], and study their tolerance to asynchronous execution for multiplication and modulo [GK25]. Garg [Gar23] shows that many generalizations of the stable matching problem, including the constrained stable marriage problem with ties, can also be solved using the LLP framework.

When the feasibility predicate has the form $\forall i : G[i] \geq f_i(G)$ for monotone functions f_i, the LLP algorithm is closely related to computing the least fixed point of a monotone function on a lattice via the Knaster–Tarski fixed-point theorem [Tar55, LNS82]. However, the LLP framework is more general in three respects: (1) the predicate B need not have the form $G \geq f(G)$ — we only require B to be closed under meets; (2) the Knaster–Tarski theorem requires the function to map the lattice to itself, guaranteeing a fixed point exists, whereas in LLP the algorithm may return null (e.g., cyclic job scheduling with positive weights has no solution); (3) the LLP framework supports advancement on multiple components, which the classical fixed-point iteration does not address.

The technique of *chaotic iteration* (also called *asynchronous iteration*) in abstract interpretation and dataflow analysis [CC77] also computes least fixed points by iterating on individual components in arbitrary order. The LLP algorithm can be viewed as a generalization of chaotic iteration to predicates that are not necessarily of the fixed-point form.

The classical algorithms unified by the LLP framework include Dijkstra's algorithm [Dij59] for shortest paths, the Gale–Shapley algorithm [GS62] for stable marriage, and Kruskal's and Prim's algorithms for minimum spanning trees. For lattice theory background, see Appendix A and the texts by Birkhoff [Bir67], Davey and Priestley [DP90], and Garg [Gar15].

Chapter 3

The Stable Marriage Problem

EVALUATION COPY
Valid until June 1, 2028

3.1 Introduction

The U.S. National Resident Matching Program (NRMP) places more than 40,000 medical-school graduates into hospital residency positions each year, using an algorithm that traces directly to the work of Gale and Shapley. Kidney-exchange clearinghouses use the same idea, pairing incompatible donor–recipient pairs into chains so that more transplants can take place. In each setting the underlying question is the same: given a set of agents on each side and a preference order from every agent, find a pairing in which no two unpaired agents would both prefer each other.

The Stable Matching Problem formalizes this question. Gale and Shapley introduced it in their 1962 paper [GS62] and showed two surprising results: a stable matching always exists, and the deferred-acceptance algorithm they introduced constructs one in $O(n^2)$ time. The economic significance of this line of work was recognized by the 2012 Nobel Prize in Economics awarded to Lloyd Shapley and Alvin Roth for the theory of stable allocations and the practice of market design.

Beyond its applications, stable matching is also the first concrete instance of the lattice-linear-predicate (LLP) framework introduced in Chapter 2. The set of *proposal vectors* forms a finite distributive lattice; the stability requirement is a lattice-linear predicate; and the Gale–Shapley algorithm is exactly the LLP main loop applied to that predicate. The chapter therefore reads as the first concrete LLP application and provides a template that later chapters — shortest paths, minimum spanning trees, network flow, and several others — will reuse.

In the standard problem there are n men numbered $1, 2, \ldots, n$ and n women numbered $1, 2, \ldots, n$, each with a totally ordered preference list. The matrices $mpref$ and $wpref$ encode the lists: $mpref[i][k] = j$ iff woman j is man i's k^{th} preference, and analogously for $wpref$. A matching has a *blocking pair* (m, w) if m and w are not married to each other yet each prefers the other to their current partner; the matching is *stable* if it has no blocking pair. Fig. 3.1 shows a small instance.

mpref

$\boldsymbol{m_1}$	w_2	w_3	w_1
$\boldsymbol{m_2}$	w_2	w_3	w_1
$\boldsymbol{m_3}$	w_3	w_1	w_2

wpref

$\boldsymbol{w_1}$	m_1	m_3	m_2
$\boldsymbol{w_2}$	m_2	m_1	m_3
$\boldsymbol{w_3}$	m_1	m_2	m_3

Figure 3.1: Stable matching problem with men's preference list (*mpref*) and women's preference list (*wpref*). The first column of each table (in bold) indexes by man / woman; the remaining cells give that person's ordered preference list, most preferred first.

This chapter is organized as follows. Section 3.2 introduces the proposal-vector lattice on which all algorithms in this chapter operate. Section 3.3 presents the Gale–Shapley algorithm with its $O(n^2)$ analysis. Section 3.4 gives the LLP formulation of stable matching together with the constrained version, in which additional lattice-linear constraints — regret bounds (e.g., Peter's regret must be less than Paul's), *forced pairs*, and *forbidden pairs* — can be imposed without changing the algorithm. Section 3.5 discusses how the algorithms of this chapter might adapt to other settings such as incremental or dynamic preference changes, distributed or privacy-limited participants, additional side constraints, and ties in the preference lists.

3.2 Proposal Vector Lattice

We use the notion of a *proposal vector* for our algorithms. A vector of (man) proposals, G, is of dimension n, the number of men. We view any vector G as follows: $(G[i] = k)$ if man i has proposed to his k^{th} preference, i.e., the woman given by $mpref[i][k]$. If $mpref[i][k]$ equals j, then $G[i] = k$ corresponds to man i proposing to woman j. The woman that man i proposes to in G is $mpref[i][G[i]]$. The vector $(1, 1, \ldots, 1)$ corresponds to the proposal vector in which every man has proposed to his top choice. Similarly, $(n, n, \ldots, n)$ corresponds to the vector in which every man has proposed to his last choice. Our algorithms can also handle the case when the lists are incomplete, that is, a man prefers to stay alone to be matched to some woman. However, for simplicity, we assume complete lists. It is clear that the set of all proposal vectors forms a distributive lattice under the natural less than order in which the meet and join are given by the component-wise minimum and the component-wise maximum, respectively. This lattice has n^n elements.

Given any proposal vector, G, there is a unique matching defined as follows: man i and $mpref[i][G[i]]$ are matched in G if the proposal of man i is the best proposal for that woman in G or any proposal before G. The man i is unmatched otherwise. A woman q is unmatched in G if she does not receive any proposal in G or earlier.

A proposal vector G represents a *man-saturating matching* iff no woman receives more than one proposal in G. Formally, G is a man-saturating matching if $\forall i, j : i \neq j : mpref[i][G[i]] \neq mpref[j][G[j]]$. When the number of men equals the number of women, a man-saturating matching is a perfect matching (all men and women are matched). When the number of men is less than the number of women, then G is a man-saturating matching if every man is matched (but some women are unmatched). We say that a matching M_1 (or a marriage) is less than another matching M_2 if the proposal vector for M_1 is less than that of M_2.

A proposal vector G may have one or more blocking pairs. A pair (m, w) is a *blocking pair* in G iff m is not matched to w in G, yet m prefers w to his current match, and w prefers m

to her current match (or to staying unmatched, if she has none).

A proposal vector G is a stable marriage (or a stable proposal vector) iff it is a man-saturating matching and there are no blocking pairs in G. The man-optimal marriage is the *least* stable matching in the proposal lattice, and the woman-optimal marriage is the *greatest* stable matching.

3.3 Gale–Shapley Algorithm

In this section, we first present an algorithm due to Gale and Shapley for this problem (also known as the deferred acceptance algorithm). In this algorithm, a free man proposes to women in order of his preference list, i.e., he first proposes to his top choice. Only when a man is rejected by his top choice would he move to his next top choice. We maintain the list of all men who are not engaged in the variable $mList$. Initially, all men are free (not engaged) and are on this list. For this section, we assume that the number of men is equal to the number of women.

When any woman z receives a proposal from a man i, she always accepts it if she is not engaged. In this case, they both get engaged, and the variable $partner[z]$ is set to i. If the woman z is engaged, then she compares this proposal with her existing partner. If she prefers this proposal to her existing partner, then she breaks the engagement with her existing partner and makes i her partner. The previous partner joins $mList$, the list of free men. If the woman z prefers her existing partner, then the proposal of the man i is rejected, and the man i goes back to $mList$. Algorithm Gale–Shapley shows the pseudocode for these steps.

It is easy to verify the following properties of the algorithm.

Lemma 3.1 *All the following statements are valid.*

1. *As the algorithm progresses, the partner for a man can only worsen and the partner for a woman can only improve.*
2. *Once a woman is engaged, she stays engaged.*

Observe that once all men are engaged, the list $mList$ is empty and the *while* loop in the algorithm terminates. Is it possible for all men to run out of all the choices and still not have a stable marriage? We claim that whenever a man i proposes to his last choice, the proposal is always accepted. The proposal is always accepted if the woman is not engaged. If the woman is engaged, we can deduce that all women are engaged because the man had previously proposed to all other women who are now engaged. However, this is a contradiction because all n women are engaged to different men and the man i is not engaged. Hence, we conclude that the last proposal of any man is always accepted.

How many times can the *while* loop iterate? Observe that in every iteration $G[i]$ increases by 1 for some i. Since $G[i]$ starts with 0 and cannot increase beyond n, the total number of iterations for the *while* loop is $O(n^2)$. Each iteration of the *while* loop takes $O(1)$ time. Hence, the time complexity of the Gale–Shapley Algorithm is $O(n^2)$.

We now show that the GS algorithm returns the man-optimal stable marriage. Call a pair (m, w) *valid* if m and w are matched to each other in some stable marriage of the instance;

Algorithm Gale–Shapley: Finding the man-optimal marriage

```
input: mpref: array[1..n][1..n] of int; rank: array[1..n][1..n] of int;
output: G: man-optimal stable marriage

var mList: list of 1..n // men that are free, initially all men
partner: array[1..n] of 0..n initially partner[i] = 0 for all i // current fiancé for
 woman i
G: array[1..n] of 0..n initially G[i] = 0 for all i // number of proposals made by
 man i
 1 mList := [1..n] // Initialize all men as part of this list
 2 while mList is nonempty do
 3 |   i := mList.removeFirst();
 4 |   G[i] := G[i] + 1 // Move to the next available top choice for man i
 5 |   z := mpref[i][G[i]] // Woman corresponding to that choice
 6 |   if partner[z] = 0 then
 7 |   |   partner[z] := i
 8 |   end
 9 |   else
10 |   |   if rank[z][i] < rank[z][partner[z]] then
11 |   |   |   mList.add(partner[z]);
12 |   |   |   partner[z] := i;
13 |   |   end
14 |   |   else
15 |   |   |   mList.add(i)
16 |   |   end
17 |   end
18 end
19 return G
```

a man's *best valid partner* is the highest-ranked woman among those with whom he forms a valid pair. Suppose that a man m is rejected by his best valid partner woman w in the GS algorithm. Let this be the first time a valid pair is rejected. Suppose w rejected m for some man m'. Thus, we know that w prefers m' to m.

Let M be a marriage in which m is married to w because (m, w) is a valid pair. In the marriage, M, let m' be married to w'. When w rejected m for m' in the GS marriage, it was the first rejection of any valid pair. This means that m' must have proposed to w before proposing to w'. Thus, we also get that m' prefers w to w'. However, this results in (m', w) forming a blocking pair for M. The man m' prefers w to w' and the woman w prefers m' to m, a contradiction. Hence, we conclude that the GS algorithm returns the man-optimal stable marriage.

Example 3.2 We trace the Gale–Shapley algorithm on the instance in Fig. 3.1. Initially, $G = (0, 0, 0)$, $partner = (0, 0, 0)$, and $mList = [m_1, m_2, m_3]$.

1. m_1 proposes to $mpref[1][1] = w_2$. $G = (1, 0, 0)$. w_2 is free; $partner[w_2] := m_1$.
2. m_2 proposes to $mpref[2][1] = w_2$. $G = (1, 1, 0)$. w_2 prefers m_2 to m_1, since $rank[w_2][m_2] = 1$ is less than $rank[w_2][m_1] = 2$. $partner[w_2] := m_2$; m_1 returns to $mList$.
3. m_3 proposes to $mpref[3][1] = w_3$. $G = (1, 1, 1)$. w_3 is free; $partner[w_3] := m_3$.
4. m_1 proposes to $mpref[1][2] = w_3$. $G = (2, 1, 1)$. w_3 prefers m_1 to m_3, since $rank[w_3][m_1] = 1$ is less than $rank[w_3][m_3] = 3$. $partner[w_3] := m_1$; m_3 returns to $mList$.
5. m_3 proposes to $mpref[3][2] = w_1$. $G = (2, 1, 2)$. w_1 is free; $partner[w_1] := m_3$.

$mList$ is empty. The algorithm terminates with $G = (2, 1, 2)$: $m_1 \leftrightarrow w_3$, $m_2 \leftrightarrow w_2$, $m_3 \leftrightarrow w_1$. This is the man-optimal stable marriage.

3.4 LLP Stable Matching Algorithm

We now derive the algorithm for the stable matching problem using Lattice-Linear Predicates. We assume that the number of men equals the number of women in this section. We let $G[i]$ be the choice number that man i has proposed to. Initially, $G[i]$ is 1 for all men. The woman that man i proposes to in G is $mpref[i][G[i]]$.

Definition 3.3 *An assignment G is feasible for the stable marriage problem if (1) it corresponds to a perfect matching (all men are paired with different women) and (2) it has no blocking pairs.*

We show that the predicate "G is a stable marriage" is a lattice-linear predicate.

Lemma 3.4 *The predicate that a vector G corresponds to a stable marriage is lattice-linear.*

Proof: Let $z = mpref[j][G[j]]$. G is not a stable marriage iff $\exists j : forbidden(G, j)$, where $forbidden(G, j)$ is expressed formally as $(\exists i : \exists k \leq G[i] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j]))$: some man i has, at some point no later than his current proposal, proposed to the woman j is currently proposing to, and she prefers i to j.

We show that if $forbidden(G, j)$ holds, then for any $H \geq G$ with $H[j] = G[j]$, H is also not a stable marriage. Since $H[j] = G[j]$, $mpref[j][H[j]] = z$. Since $H[i] \geq G[i]$, the same k that satisfies $k \leq G[i]$ also satisfies $k \leq H[i]$, so the forbidden condition carries over to H. Hence $forbidden(H, j)$ holds, so H is not a stable marriage.

■

Lemma 3.4 immediately gives us the Algorithm LLP-StableMarriage-Spec. To keep the forbidden clause readable we introduce an auxiliary variable $z = mpref[j][G[j]]$ — the woman that man j is currently proposing to — using the **always** section. The **always** section defines variables that are derived from G, viewed as macros: whenever $G[j]$ changes, so does z.

If man j is forbidden, it is clear that any vector in which man j is matched with z and man i is matched with his current or a worse choice can never be a stable marriage. Thus, it is safe for man j to advance to the next choice. If he has no next choice — he is forbidden while already proposing to the last woman on his list, $G[j] = n$ — then no vector at or above G is a stable marriage, and since the algorithm only ever raises G from I, no stable marriage extends I at all. Starting from $I = [1, 1, \ldots, 1]$ this case never arises, because a stable marriage always exists; it can arise for an arbitrary I, which is why the advance tests for it.

Algorithm LLP-StableMarriage-Spec: Man-optimal stable marriage via LLP — a specification, not yet an efficient algorithm

input: $mpref$: array$[1..n][1..n]$ of int; $rank$: array$[1..n][1..n]$ of int; I: array$[1..n]$ of int
`// initial proposals,` $1 \le I[i] \le n$
output: G: array$[1..n]$ of int, initially $I[i]$, or "no stable matching extending I"

1 **always**: $z = mpref[j][G[j]]$;
2 **forbidden**$(j \in [1..n])$:
$\exists i \in [1..n] : \exists k \in [1..G[i]] : (z = mpref[i][k]) \wedge (rank[z][i] < rank[z][j])$
3 $\Rightarrow$**advance**: **if** $G[j] = n$ **then return** "no stable matching extending I" **else** $G[j] := G[j] + 1$;

The general properties of the LLP algorithm (Section 2.5) are well illustrated by the stable marriage problem:

- *Nondeterminism*: The man-optimal stable marriage is the same regardless of the order in which forbidden men advance on their preference lists.
- *No lookahead*: A man need not reveal his full preference list in advance. Only when he is rejected (forbidden) does he reveal his next choice. This makes the algorithm applicable in online settings.
- *Individual optimality*: The LLP solution is optimal for each man individually. If man m seeks the stable marriage that gives him his best possible partner (among all stable marriages), the LLP algorithm returns exactly that partner for m.

Algorithm LLP-StableMarriage-Spec is only a specification: it pins down the correct answer, but as written it is not an efficient algorithm, and we use it only as a path to one. The $\exists i : \exists k \in [1..G[i]]$ scan in its forbidden condition exists only because an arbitrary initial

G gives no way to tell, from G alone, which woman currently holds the best proposal among those made so far. Maintaining that information directly gives us LLP-StableMarriage, the algorithm we actually use.

Algorithm LLP-StableMarriage: Gale–Shapley as a lattice-linear program on two vectors

input: *mpref*: array[1..n][1..n] of int; *rank*: array[1..n][1..n] of int
output: G: array[1..n] of int initially 0

var *partner*: array[1..n] of int initially 0 `//` $partner[z] = 0$ `means woman` z `is free`
1 **always**: $z = mpref[i][G[i]]$;
2 **forbidden**$(i \in [1..n])$:
$(G[i] = 0) \ \vee \ ((partner[z] \neq 0) \wedge (rank[z][partner[z]] < rank[z][i]))$
3 $\Rightarrow$ **advance**: $G[i] := G[i] + 1$;
4 **if** $((partner[z] = 0) \ \vee \ (rank[z][i] < rank[z][partner[z]]))$ **then** $partner[z] := i$;

Unlike Algorithm LLP-StableMarriage-Spec's guard, which must re-examine every man's whole proposal history to find out who currently holds z, this guard reads it straight off $partner[z]$: man i is forbidden either because he has not yet proposed to anyone ($G[i] = 0$), or because his current proposal z already holds someone z prefers. The advance performs the proposal itself, updating $partner[z]$ exactly when the new arrival is the better choice — so *partner* never needs to be rebuilt from G, it is maintained incrementally. This is precisely the Gale–Shapley algorithm, and each advance now costs $O(1)$ instead of $O(n)$, giving the $O(n^2)$ bound of Section 3.3.

We now present an algorithm to find stable marriages that satisfy additional constraints. The following lemma proves lattice-linearity of many such constraints.

Lemma 3.5 *The following constraints are lattice-linear.*

1. *The regret of man i is at most that of man j.*
2. *Man i cannot be married to woman j.*
3. *The regret of man i is equal to that of man j.*

Proof: Let B be the predicate that G is a stable marriage and it satisfies the corresponding additional constraint.

1. Suppose that G is a stable marriage but it does not satisfy B. This means that the regret of man i is more than the regret of man j. In this case, we have $forbidden(G, j)$, because unless $G[j]$ is advanced, the predicate cannot become true.
2. If $mpref[i][G[i]] = j$, then $forbidden(G, i)$ holds.

3. This condition is a conjunction of two lattice-linear conditions of type in part (1).

■

Observe that we have not given the argument that the LLP predicate eventually becomes true. In general, this may not happen for all LLP predicates. For the standard stable marriage, however, the LLP predicate becomes true. We can define the predicate $D(k)$ on the set of assignments as "At least k different women have been proposed to, either in G or in some assignment preceding it." It is clear that before all the proposals are done $D(n)$ holds. It is easy to show that $D(k)$ implies that there are k valid pairs in the assignment. Therefore, $D(n)$ implies that B holds on G.

We now enumerate advantages of the LLP algorithm.

1. It gives us a more general algorithm. Instead of starting from the initial vector, we can start from any vector and find a stable marriage greater than or equal to that vector, if one exists.

2. We can use the LLP algorithm for finding a stable marriage that satisfies additional constraints such as the regret of man i is at most that of man j.

3. We can automatically deduce that the meet of two stable marriages is also a stable marriage.

Example 3.6 As an example of a constrained stable marriage, consider the instance in Fig. 3.1 with the additional constraint that the regret of m_1 must be at most the regret of m_2 (i.e., $G[1] \leq G[2]$). This constraint is lattice-linear: $forbidden(2) \equiv (G[1] > G[2])$ with $advance(2) : G[2] := G[1]$. By Lemma 2.3(c), the conjunction of the stability predicate and this constraint is also lattice-linear, so the LLP algorithm finds the least stable marriage satisfying the constraint. In the unconstrained man-optimal marriage $G = (2, 1, 2)$, m_1 has regret 2 while m_2 has regret 1, violating the constraint. Since $(2, 1, 2)$ is in fact the only stable marriage of this instance, no stable marriage satisfies $G[1] \leq G[2]$; the LLP algorithm with the combined predicate advances past the top of the lattice and reports that the constrained problem is infeasible (returns null).

3.5 Extending Algorithms

The Gale–Shapley algorithm and its LLP variants presented in this chapter all assume a clean, static, centralized setting: a fixed set of n men and n women, complete and truthful preference lists, and a single process that has access to all of them. Readers are encouraged to study how the algorithms of this chapter adapt, generalize, or fail under the following conditions:

1. **Incremental setting:** Men and/or women may join or leave the system dynamically after an initial matching has been computed. Does the existing matching need to be recomputed from scratch, or can it be repaired locally while preserving stability?

2. **Dynamic setting:** The preference lists change over time (for example, a participant revises their ranking after learning more about the candidates). How should the algorithm track a moving stable matching?

3. **Distributed setting:** Each participant has access only to their own preference list and can communicate only with a subset of the other side. Design a stable matching algorithm in which no single node sees the entire instance.

4. **Privacy setting:** Each participant is willing to reveal only partial information about their preferences (e.g., their top k choices, or a coarse ranking). Can a stable matching still be computed, possibly with a degradation in the quality of the result?

5. **Constrained setting:** The matching must satisfy additional side constraints—for example, forced pairs that must appear in the matching, forbidden pairs that must not, regional or diversity caps, or couples who must be matched to nearby partners. Some such constraint families are already treated in the book; others make for good exercises.

6. **Ties and indifference:** Participants are allowed ties in their preference lists. Stable matchings may no longer be unique, and several notions of stability (weakly, strongly, super-) diverge. Which of them can be computed with LLP-style algorithms?

7. **Parallel setting:** The algorithm is parallelized to reduce the wall-clock time of computing a stable matching. What is the complexity of the parallel algorithm?

8. **Approximation setting:** When no stable matching exists (e.g., in the stable roommate problem with odd-length preference cycles), the goal is to find a near-stable solution that minimizes the number or severity of blocking pairs.

9. **Fairness setting:** Gale–Shapley is known to be men-optimal and women-pessimal (or vice versa). The objective is to produce a stable matching that is fair across the two sides, for example by minimizing the sum or the maximum of regrets.

10. **Enumeration setting:** The task is to enumerate all stable matchings of a given instance, or the top-k stable matchings according to a given objective. The lattice of stable matchings makes this a natural LLP target.

11. **Online setting:** Participants arrive one at a time in an adversarial or random order, and the algorithm must make irrevocable decisions as each participant is revealed. How close to an offline stable matching can one stay?

12. **Byzantine setting:** Some participants act strategically or maliciously: misreporting preferences, refusing to execute the protocol, or colluding to disrupt the matching for truthful participants. Which of the algorithms in this chapter are robust to such behavior, and what defenses are possible?

3.6 Summary

This chapter studied the stable marriage problem through the lens of lattice-linear predicate detection. We modeled the problem using the proposal vector lattice, where each vector

G assigns a choice number to every man. The key insight is that the predicate "G is a stable marriage" is lattice-linear: if man j is forbidden in G (another man is preferred by $mpref[j][G[j]]$), then j remains forbidden in every $H \geq G$ with $H[j] = G[j]$.

We presented the Gale–Shapley algorithm as a special case of the LLP framework. LLP-StableMarriage expresses the same algorithm in the LLP forbidden/advance notation, matching Gale–Shapley's own $O(n^2)$ running time; it was reached via an intermediate specification, LLP-StableMarriage-Spec, whose forbidden condition is simple to state and prove correct but costs an $O(n)$ rescan per guard rather than $O(1)$.

Table 3.1 lists the algorithms discussed in this chapter.

Algorithm	**Problem**	**Time Complexity**
Gale–Shapley	Stable Marriage	$O(n^2)$
LLP-StableMarriage	Stable Marriage	$O(n^2)$

Table 3.1: Algorithms for the stable marriage problem

3.7 Problems

3.1. **(GS Invariant And No Blocking Pair)** Show that Gale–Shapley algorithm maintains the invariant that every woman is partnered with the most preferred man of all the proposals made so far to her. Use this claim to show that there is no blocking pair for all the engaged couples during the execution of Gale–Shapley algorithm.

3.2. **(Woman Pessimal Stable Marriage)** Show that the man-oriented Gale–Shapley algorithm returns the stable marriage, which is the worst stable marriage from the perspective of women.

3.3. **(Stable Marriage Sublattice Closure)** Show that the set of stable marriages is closed under the meet operation and the join operation of the proposal lattice.

3.4. **(Check Stable Marriage Assignment)** Given an assignment vector, how will you check whether it forms a stable marriage?

3.5. **(Trace Gale Shapley n4)** Consider $n = 4$ with the following preferences.

$mpref$

$\boldsymbol{m_1}$	w_1	w_2	w_3	w_4
$\boldsymbol{m_2}$	w_2	w_1	w_4	w_3
$\boldsymbol{m_3}$	w_1	w_4	w_3	w_2
$\boldsymbol{m_4}$	w_4	w_1	w_2	w_3

$wpref$

$\boldsymbol{w_1}$	m_2	m_1	m_4	m_3
$\boldsymbol{w_2}$	m_4	m_3	m_1	m_2
$\boldsymbol{w_3}$	m_1	m_3	m_2	m_4
$\boldsymbol{w_4}$	m_3	m_4	m_2	m_1

Trace the man-oriented Gale–Shapley algorithm. Show the proposal vector G after each round and give the final stable matching.

3.6. **(Find Blocking Pairs n3)** Consider $n = 3$ with the following preferences.

$mpref$

$\boldsymbol{m_1}$	w_1	w_2	w_3
$\boldsymbol{m_2}$	w_2	w_1	w_3
$\boldsymbol{m_3}$	w_1	w_3	w_2

$wpref$

$\boldsymbol{w_1}$	m_2	m_1	m_3
$\boldsymbol{w_2}$	m_1	m_3	m_2
$\boldsymbol{w_3}$	m_3	m_2	m_1

The matching $M = \{(m_1, w_2), (m_2, w_3), (m_3, w_1)\}$ is given. Find all blocking pairs, or prove that M is stable.

3.7. **(Constrained Stable Matching)** Using the instance from the Gale–Shapley trace problem above (the $n = 4$ instance), find the man-optimal stable matching subject to the constraint that m_1 must be matched to either w_1 or w_2 (i.e., $G[1] \leq 2$). Does a constrained stable matching exist?

3.8. **(Stable Roommate Nonexistence)** The *stable roommate problem* is the non-bipartite version of stable matching: given $2n$ people (no men/women distinction), each with a preference ranking over all others, find a matching into n pairs with no blocking pair. Show that stable matchings need not exist in the roommate setting by constructing a 4-person instance with no stable matching. Explain why bipartiteness guarantees existence in the stable marriage problem.

3.9. **(Opposing Preferences Across Matchings)** Let S_1 and S_2 be two distinct stable matchings for the same instance. A man m is said to *prefer* S_1 if he prefers his partner in S_1 to his partner in S_2, and similarly for women. Prove: if man m prefers S_1, then his partner in S_1 prefers S_2.

3.10. **(Quadratic Proposals Worst Case)** Construct an instance with n men and n women in which the man-oriented Gale–Shapley algorithm makes $\Omega(n^2)$ proposals.

3.11. **(Median Stable Matching)** Let L be any finite distributive lattice of global states, and let B be any Boolean predicate defined on L that is both lattice-linear and dual lattice-linear (a *regular predicate*; see Section 2.8). Let $M = \{M_1, M_2, \ldots, M_k\}$ be any subset of L of size k all of whose elements satisfy B. For each index m, the multiset $\{M_i[m] : 1 \le i \le k\}$ has a sorted form; for any $j \in [1..k]$, define the generalized j-median state G^j by

$$G^j[m] \;:=\; \text{the } j\text{-th smallest value in the multiset } \{M_i[m] : 1 \le i \le k\}.$$

(a) Show that G^j also satisfies B.

(b) Apply (a) to the stable marriage problem to conclude that the j-th median of any set of k stable matchings is itself a stable matching — a theorem of Teo and Sethuraman [TS98].

3.8 Bibliographic Remarks

The stable matching problem was introduced by Gale and Shapley [GS62] and has been studied extensively, with several books and survey articles devoted to the topic [GI89, Knu97c, RS92, IM08, Man13]. The notion of *regret* in a matching is discussed in [GI89]. Stable marriage with restricted pairs (forced and forbidden pairs) is studied by Dias et al. [DdFdFS03]. The lattice theory background on order ideals and distributive lattices used in this chapter can be found in Davey and Priestley [DP90] and in Garg [Gar15]. The existence of a stable matching via Tarski's fixed-point theorem is shown by Adachi [Ada00].

The lattice-linear predicate (LLP) approach to stable marriage presented in this chapter was introduced by Garg [Gar17]. The application of predicate detection to combinatorial optimization problems, including stable matching, is discussed in [Gar20b]. Hu and Garg [HG21] give a characterization of super-stable matchings. Garg and Hu [GH23] present improved paths to stability for the stable marriage problem. Extensions to the constrained stable marriage problem with ties are discussed by Garg [Gar23]. A generalization of Teo and Sethuraman's median stable marriage theorem is given in [Gar20a].

Chapter 4

Sorting Algorithms

4.1 Introduction

In this chapter, we consider some basic ideas in the design of algorithms that are crucially based on the notion of *order* in a set of elements of size n. We will assume that the order is total, that is, for every two elements x and y, either x is less than or equal to y, or y is less than or equal to x in that order.

A large fraction of the running time of real systems is spent arranging data into some order, and many algorithms that appear unrelated to sorting reduce to it as a subroutine. Once the data is sorted, many tasks become cheaper: searching for a value takes $O(\log n)$ time by binary search instead of $O(n)$; computing the median, quantiles, or any order statistic becomes $O(1)$ after an $O(n \log n)$ preprocessing; and detecting duplicates or taking the union, intersection, or difference of two sets takes linear time. Sorting also enables efficient algorithms for geometric problems such as finding the closest pair of points and constructing convex hulls, and it is a building block for data compression, indexing, and scheduling.

This chapter is organized as follows. Section 4.2 discusses sorting algorithms based on the swapping of consecutive entries that are out of order. This is a general class of algorithms that are very natural. However, these algorithms take $O(n^2)$ sequential time in the worst case. Section 4.3 discusses *QuickSort*. Section 4.4 discusses *MergeSort*. Since MergeSort and QuickSort are based on the divide-and-conquer paradigm, they are revisited in Chapter 9. Section 4.5 proves an $\Omega(n \log n)$ lower bound on the number of comparisons required by any comparison-based sorting algorithm, using a decision-tree argument. All these sorting algorithms are based on the comparison of two entries in the array. Finally, Section 4.6 describes sorting algorithms that are not based on the comparison of two entries. Section 4.7 casts sorting itself as an LLP search for the permutation that sorts the array.

4.2 Algorithms Based on Swapping Consecutive Entries

Let A be an array of distinct integers. Our goal is to sort the array.

Algorithm BubbleSort swaps entries that are out of order in one iteration of the *for* loop.

Algorithm BubbleSort: Sorting by repeatedly swapping adjacent out-of-order elements

input: A: array$[1..n]$ of int
output: A: array$[1..n]$ of int `// sorted`

```
1  found := true;
2  while found do
3  |  found := false;
4  |  for j = 1 to n − 1 do
5  |  |  if A[j] > A[j + 1] then
6  |  |  |  found := true;
7  |  |  |  swap(A[j], A[j + 1]);
8  |  |  end
9  |  end
10 end
```

Example 4.1 Consider sorting the array $A = [5, 3, 8, 1]$ using Bubble Sort.

Pass 1 ($j = 1, 2, 3$): Compare $(5, 3)$, swap $\rightarrow [3, 5, 8, 1]$. Compare $(5, 8)$, no swap. Compare $(8, 1)$, swap $\rightarrow [3, 5, 1, 8]$. The largest element 8 has "bubbled" to the end.

Pass 2: Compare $(3, 5)$, no swap. Compare $(5, 1)$, swap $\rightarrow [3, 1, 5, 8]$. Compare $(5, 8)$, no swap.

Pass 3: Compare $(3, 1)$, swap $\rightarrow [1, 3, 5, 8]$. No more swaps needed; *found* remains false in the next pass, and the algorithm terminates.

Each pass moves at least one element to its correct position, so at most $n - 1$ passes do useful work; one additional pass with no swaps is needed to detect termination. Hence, the repeat loop is executed at most n times, giving a time complexity of $O(n^2)$.

Another schedule is to increase the size of the sorted array one at a time, as done in Algorithm InsertionSort. Here we assume that the array $A[1..i - 1]$ is sorted and we find the

right place to insert $A[i]$ so that $A[1..i]$ is sorted.

Algorithm InsertionSort: Sorting by inserting each element into the sorted prefix

input: A: array$[1..n]$ of int
output: A: array$[1..n]$ of int `// sorted`

```
1 for i = 2 to n do
2 |   for j := i − 1 downto 1 do
3 |   |   if A[j] ≤ A[j + 1] then
4 |   |   |   break;
5 |   |   end
6 |   |   swap(A[j], A[j + 1]);
7 |   end
8 end
```

The time complexity of Algorithm InsertionSort is clearly $O(n^2)$ due to nested *for* loops.

Example 4.2 Consider sorting $A = [5, 3, 8, 1]$ using Insertion Sort.
$i = 2$: Insert $A[2] = 3$ into the sorted subarray $[5]$. Compare $(5, 3)$, swap $\rightarrow [3, 5, 8, 1]$.
$i = 3$: Insert $A[3] = 8$ into $[3, 5]$. Compare $(5, 8)$, no swap (break). Array: $[3, 5, 8, 1]$.
$i = 4$: Insert $A[4] = 1$ into $[3, 5, 8]$. Compare $(8, 1)$, swap $\rightarrow [3, 5, 1, 8]$. Compare $(5, 1)$, swap $\rightarrow [3, 1, 5, 8]$. Compare $(3, 1)$, swap $\rightarrow [1, 3, 5, 8]$. Sorted.

We now prove that any comparison-based sorting algorithm that restricts itself to comparing consecutive entries (adjacent elements) in an array of size n requires at least $\Omega(n^2)$ comparisons in the worst case. This applies to algorithms such as Bubble Sort or Insertion Sort, where comparisons are limited to pairs of elements at positions i and $i + 1$.

Theorem 4.3 *Consider an array $A = [a_1, a_2, \ldots, a_n]$ of n distinct elements. A sorting algorithm that uses only consecutive comparisons requires at least $\Omega(n^2)$ comparisons in the worst case.*

Proof:
The maximum number of inversions in a permutation of n elements is achieved by the reverse order $[n, n-1, \ldots, 2, 1]$, with:

$$\text{number of inversions} = \binom{n}{2} = \frac{n(n-1)}{2}.$$

Each adjacent comparison can resolve at most one inversion (e.g., swapping a_i and a_{i+1}), so a permutation with $\frac{n(n-1)}{2}$ inversions requires at least that many comparisons to sort fully.

■

4.3 Quicksort

Quicksort (due to C.A.R. Hoare) is one of the fastest sorting algorithms on sequential computers. It has $O(n^2)$ worst case time complexity but requires $O(n \log n)$ time on average. The algorithm works by partitioning the array around a chosen element called the *pivot*: it rearranges the array so that every element less than or equal to the pivot ends up to its left, and every element greater than the pivot ends up to its right. In particular, once this rearrangement is done, the pivot itself is sitting exactly where it belongs in the final sorted array, and it never needs to move again. The two sides of the partition can then be sorted independently of each other and of the pivot, so applying the same partition-and-recurse procedure to each side — and stopping once a side has at most one element — sorts the entire array.

There are multiple methods to choose a pivot to partition the array A. Choosing an element at random to be the pivot is an easy method that will result in approximately equal sized partitions on average. This gives us the average case sequential time complexity of $O(n \log n)$. In the worst case, the recursion may reduce the range by only 1, resulting in the sequential time complexity of $O(n^2)$. For example, if the array is already sorted and the pivot is always the first element, then each partition produces one part of size 1 and one part of size $n-1$, giving the recurrence $T(n) = T(n-1) + O(n)$, which yields $T(n) = O(n^2)$.

Algorithm QuickSort: Sorting by partitioning around a pivot

input: A: array$[low..high-1]$ of int; $low, high$: int
output: A: array$[low..high-1]$ of int `// sorted in non-decreasing order`
1 **if** $low < high$ **then**
2 $\quad pivot := choosePivot()$;
3 $\quad (p, q) :=$ Partition$(A, pivot, low, high)$;
4 $\quad$ QuickSort(A, low, p);
5 $\quad$ QuickSort$(A, q, high)$;
6 **end**

We describe a slight variant of the Quicksort algorithm in which we partition the array into three parts. Algorithm ThreeWayPartition takes array A, low, and $high$ as input parameters. The part of the array this method partitions is given by

$$\{A[i] \mid low \leq i < high\}$$

Note that when low equals $high$, the range is empty.

The method *partition* returns two indices p and q. All elements in the range $[low \ldots p)$ are strictly less than the pivot, in the range $[p \ldots q)$ are equal to the pivot, and in the range $[q \ldots high)$ are greater than the pivot. We only need to recurse on the first and the third parts. Depending upon the pivot, any (or both!) of the first and third parts may be empty.

Once we have a pivot, how do we partition the array into three parts: the first partition with all elements less than the pivot, the second one with all elements equal to the pivot, and the third partition with elements strictly greater than the pivot? Algorithm ThreeWayPartition is due to Dijkstra, who called this problem the Dutch National Flag problem. Recall that we maintain the invariant that the entries $[low \ldots p)$ are less than the pivot, $[p \ldots q)$ are equal to the pivot and $[k \ldots high)$ are greater than the pivot. The algorithm initializes p and q to low, therefore the first two ranges are empty initially and trivially satisfy the invariants. It also initializes k to $high$, making the range $[k \ldots high)$ empty and thereby ensuring that the invariant holds initially. The range $[q..k)$ corresponds to the initial input and initially contains entries that may be less than, equal to, or greater than the pivot. In each iteration of the *while* loop, this range is reduced by one by increasing q or decreasing k. Depending on the comparison between $A[q]$ and the pivot, the entry $A[q]$ is placed in the appropriate range keeping the invariants.

Algorithm ThreeWayPartition: Partitioning an array into three groups (Dutch national flag)

input: A: array$[low..high-1]$ of int; $pivot$: int
output: (p, q): boundary indices

```
1  p := low;
2  q := low;
3  k := high;
4  while q < k do
5  |  if A[q] < pivot then
6  |  |  swap(A[p], A[q]);
7  |  |  p := p + 1;
8  |  |  q := q + 1;
9  |  end
10 |  else
11 |  |  if A[q] > pivot then
12 |  |  |  k := k − 1;
13 |  |  |  swap(A[q], A[k]);
14 |  |  end
15 |  |  else
16 |  |  |  q := q + 1
17 |  |  end
18 |  end
19 end
20 return (p, q)
```

It is easy to verify that the algorithm takes $O(n)$ time when the range has n elements.

Example 4.4 Consider partitioning $A = [3, 7, 2, 5, 1, 8, 4]$ with $pivot = 4$, $low = 1$, $high = 8$. Initially $p = q = 1$, $k = 8$.

p	q	k	Array	Action
1	1	8	$[3, 7, 2, 5, 1, 8, 4]$	$A[1]$=3 $<$ 4: swap $A[1], A[1]$; p=2, q=2
2	2	8	$[3, 7, 2, 5, 1, 8, 4]$	$A[2]$=7 $>$ 4: k=7; swap $A[2], A[7]$
2	2	7	$[3, 4, 2, 5, 1, 8, 7]$	$A[2]$=4 $=$ 4: q=3
2	3	7	$[3, 4, 2, 5, 1, 8, 7]$	$A[3]$=2 $<$ 4: swap $A[2], A[3]$; p=3, q=4
3	4	7	$[3, 2, 4, 5, 1, 8, 7]$	$A[4]$=5 $>$ 4: k=6; swap $A[4], A[6]$
3	4	6	$[3, 2, 4, 8, 1, 5, 7]$	$A[4]$=8 $>$ 4: k=5; swap $A[4], A[5]$
3	4	5	$[3, 2, 4, 1, 8, 5, 7]$	$A[4]$=1 $<$ 4: swap $A[3], A[4]$; p=4, q=5
4	5	5	$[3, 2, 1, 4, 8, 5, 7]$	$q = k$: loop ends

The loop ends with $(p, q) = (4, 5)$. The array is $[3, 2, 1, 4, 8, 5, 7]$: positions [1..3] are less than 4, position [4] equals 4, and positions [5..7] are greater than 4.

The same partition routine yields a linear-expected-time algorithm for finding the k-th smallest element of an array, called QuickSelect: instead of recursing on both sides of the pivot, recurse only on the side that contains rank k. This is explored in the chapter exercises.

4.4 Merge Sort

Merge Sort is the classic example of the divide-and-conquer method in algorithm design. To sort an array A, we divide it into two halves, recursively sort each half, and then merge the two sorted halves. The base case is when a half has only a single element and is already sorted. The key step is the merge: given two sorted arrays B and C, we would like to merge them into another array D so that D is sorted.

Example 4.5 Sort $A = [5, 3, 8, 1]$ using Merge Sort.

Split: $[5, 3]$ and $[8, 1]$
Recursively sort left: split [5] and [3], merge $\rightarrow [3, 5]$
Recursively sort right: split [8] and [1], merge $\rightarrow [1, 8]$
Merge $[3, 5]$ and $[1, 8]$: compare 3 and 1 $\rightarrow$ 1; compare 3 and 8 $\rightarrow$ 3;
compare 5 and 8 $\rightarrow$ 5; copy 8. Result: $[1, 3, 5, 8]$.

So, it is sufficient to consider the following problem: we have two sorted arrays B and C, each of size n. We would like to merge them into another array D so that D is sorted.

It is easy to design an algorithm that merges two arrays B and C into D. We simply keep two indices i and j in the arrays B and C, respectively. In any step of the algorithm, we compare $B[i]$ and $C[j]$. If $B[i]$ is smaller than or equal to $C[j]$, then we copy $B[i]$ into the next available slot in D and advance the index i; breaking ties this way, in favor of B, is what makes the algorithm stable (Section 4.6.1 defines stability). Otherwise $C[j]$ is strictly smaller,

Algorithm MergeSort: Sorting by recursively merging sorted halves

input: A: array[$low..high$] of int; $low, high$: int
output: A: array[$low..high$] of int `// sorted in non-decreasing order`

```
1 if low < high then
2 |  mid := ⌊(low + high)/2⌋ ;                    // split the range in half
3 |  B := MergeSort(A, low, mid) ;                // recursively sort the left half
4 |  C := MergeSort(A, mid + 1, high) ;           // recursively sort the right half
5 |  return MergeTwo(B, C) ;                      // merge the two sorted halves
6 end
7 else return [A[low]] ;
```

so we copy $C[j]$ into the next available slot in D and advance the index j. This algorithm takes $O(n)$ time.

Sequentially, $MergeTwo$ takes $O(m+n)$ time.

Example 4.6 Merge $B = [2, 5, 8]$ and $C = [1, 4, 9]$.
Step 1: Compare $B[1]$=2 and $C[1]$=1. Since $1 < 2$, copy 1 to D. $D = [1]$.
Step 2: Compare $B[1]$=2 and $C[2]$=4. Since $2 < 4$, copy 2. $D = [1, 2]$.
Step 3: Compare $B[2]$=5 and $C[2]$=4. Copy 4. $D = [1, 2, 4]$.
Step 4: Compare $B[2]$=5 and $C[3]$=9. Copy 5. $D = [1, 2, 4, 5]$.
Step 5: Compare $B[3]$=8 and $C[3]$=9. Copy 8. $D = [1, 2, 4, 5, 8]$.
B is exhausted; copy remaining $C[3]$=9. $D = [1, 2, 4, 5, 8, 9]$.

For n total elements, the recurrence of the merge sort is $T(n) = 2T(n/2) + O(n)$, resulting in $O(n \log n)$ time.

4.5 Lower Bound for Comparison-Based Sorting

We show that any comparison-based sorting algorithm for an array of n distinct elements requires at least $\Omega(n \log n)$ comparisons in the worst case. This result applies to algorithms using unrestricted pairwise comparisons (e.g., Merge Sort), contrasting with the $\Omega(n^2)$ bound for consecutive-only comparisons. We use a decision tree model.

Theorem 4.7 *Given an array $A = [a_1, a_2, \ldots, a_n]$ of n distinct elements, a comparison-based sorting algorithm uses binary comparisons ($a_i < a_j$) to sort A into $a_{\pi(1)} < a_{\pi(2)} < \cdots < a_{\pi(n)}$, where π is a permutation. The minimum number of comparisons needed to sort any input permutation in the worst case is $\Omega(n \log n)$.*

Algorithm MergeTwo: Merging two sorted arrays

input: B: array$[1..m]$ of int `// sorted`
C: array$[1..n]$ of int `// sorted`
output: D: array$[1..m+n]$ of int `// merged and sorted`

```
1  i := 1; j := 1; k := 1;
2  while i ≤ m and j ≤ n do
3  |  if B[i] ≤ C[j] then
4  |  |  D[k] := B[i];
5  |  |  i := i + 1;
6  |  end
7  |  else
8  |  |  D[k] := C[j];
9  |  |  j := j + 1;
10 |  end
11 |  k := k + 1;
12 end
13 while i ≤ m do
14 |  D[k] := B[i]; i := i + 1; k := k + 1 ;        // copy leftover elements from B
15 end
16 while j ≤ n do
17 |  D[k] := C[j]; j := j + 1; k := k + 1 ;        // copy leftover elements from C
18 end
```

Proof: We model the sorting algorithm as a decision tree, where internal nodes represent comparisons, and leaves correspond to permutations. The worst-case number of comparisons is the tree's height.

For n distinct elements, there are $n!$ possible permutations. The algorithm must distinguish all $n!$ permutations, so the decision tree requires at least $n!$ leaves.

In a comparison-based algorithm, each node compares two elements (a_i vs. a_j), branching on the less-than operator ($<$) or the greater-than operator ($>$). With distinct elements, each node has two children (binary tree). After k comparisons, the maximum number of leaves is 2^k.

To cover all permutations:

$$2^k \geq n!$$

Taking the base-2 logarithm:

$$k \geq \log_2(n!).$$

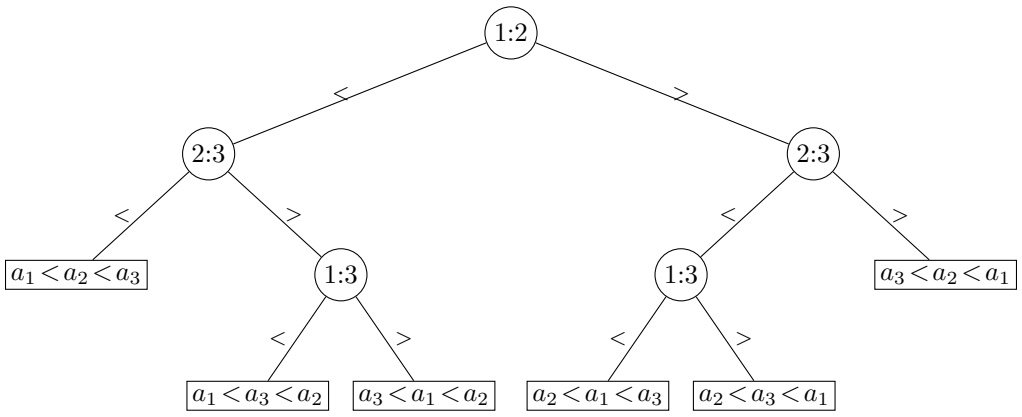


Figure 4.1: A decision tree for sorting three elements. Internal nodes are comparisons $i{:}j$ (is $a_i < a_j$?); left branches mean "yes," right branches mean "no." The six leaves correspond to the six permutations.

To show that $\log_2(n!) = \Omega(n \log n)$, observe that

$$n! \;\geq\; \left(\frac{n}{2}\right)^{n/2},$$

since the product $n!$ contains at least $n/2$ factors each at least $n/2$. Taking logarithms:

$$\log_2(n!) \;\geq\; \frac{n}{2}\,\log_2\!\left(\frac{n}{2}\right) \;=\; \Omega(n \log n).$$

For $n = 3$, there are $3! = 6$ permutations, so the decision tree needs at least $\lceil \log_2 6 \rceil = 3$ levels. Fig. 4.1 shows a decision tree that sorts three elements using at most 3 comparisons. At each internal node, the label $i{:}j$ denotes the comparison $a_i < a_j$; the left child corresponds to "yes" and the right child to "no" (i.e., $a_i > a_j$). Each leaf is labeled with the sorted order of the indices.

Since $k \geq \log_2(n!)$ and $\log_2(n!) = \Omega(n \log n)$, we conclude:

$$k = \Omega(n \log n).$$

■

4.6 Non-comparison Based Sorting

All our sorting algorithms, so far, have been based on comparison. Any comparison-based sorting algorithm must do at least $\Omega(n \log n)$ work. We now show two non-comparison-based sorting algorithms that escape this lower bound by exploiting structure in the keys: *counting sort*, when the keys come from a small integer range $[0..r-1]$, and *radix sort*, when the keys have a fixed number of digits in some radix r (generally a power of 2). Radix sort uses counting sort as its inner stable pass, so we present counting sort first.

Before doing so, we make precise a property that both algorithms rely on. A sorting algorithm is *stable* if it preserves the relative order of elements with equal keys: whenever two

elements have the same key and the first appears before the second in the input, the first must also appear before the second in the output. Stability matters here because it lets us compose sorting passes: radix sort sorts one digit at a time, and correctness requires that sorting on a later digit never disturbs the relative order already established by the earlier, less significant digits — which is exactly what a stable inner sort guarantees (Section 4.6.2 works through why in detail). Of the algorithms already presented in this chapter, Bubble Sort and Insertion Sort are stable, since both only ever swap two *adjacent* elements, and only when they are strictly out of order, so equal elements are never exchanged. Merge Sort is stable because Algorithm MergeTwo breaks ties in favor of the left array B (Section 4.4). Quicksort's partitioning step, by contrast, is *not* stable in general: moving elements past the pivot can reorder two equal elements that started on the same side of it.

4.6.1 Counting Sort

A single linear pass suffices to sort stably when values lie in $[0..r-1]$: count the occurrences of each value, convert the counts to cumulative offsets, then scatter each element to its target slot.

Algorithm CountSort: Sorting integers by counting occurrences

input: A: array$[1..n]$ of int with $A[i] \in \{0, \ldots, r-1\}$
output: B: array$[1..n]$ of int `// stably sorted`

```
1  C[v] := 0 for all v ∈ {0, ..., r − 1};
2  for i := 1 to n do
3  |  C[A[i]] := C[A[i]] + 1 ;                          // frequencies
4  end
5  for v := 1 to r − 1 do
6  |  C[v] := C[v] + C[v − 1] ;                        // cumulative offsets
7  end
8  for i := n to 1 by −1 do
9  |  B[C[A[i]]] := A[i];   C[A[i]] := C[A[i]] − 1;
10 end
```

Time and space are both $O(n+r)$. Scanning the input from right to left preserves stability: equal values are placed in reverse input order at decreasing positions of their bucket, so the leftmost equal element ends up first.

4.6.2 Radix Sort

Radix sort processes the keys one digit at a time. In our examples we use base 10, as it is easy for humans to read. Suppose that we need to sort $[85, 72, 94, 45, 13, 12, 61, 81]$. Each key has two digits, so $k = 2$. When sorting on a single digit, we exploit the fact that there are only r

possibilities and use counting sort as the inner pass. The number of passes equals the number of digits.

It may appear that it is easier to sort starting from the most significant digit (msd) first. If we used this strategy, we would get $[13, 12, 45, 61, 72, 85, 81, 94]$ after the first pass. The second pass would consist of sorting all subarrays with the same msd, and we would get the array $[12, 13, 45, 61, 72, 81, 85, 94]$. The problem with this approach is that we are forced to maintain different subarrays, one for each digit after the first pass. With every subsequent pass, it becomes even more cumbersome. Hence, somewhat counterintuitively, we will employ the least significant digit first strategy. After the first pass, we get $[61, 81, 72, 12, 13, 94, 85, 45]$. We do not need to remember any sublists that are created during the first pass. Now, we sort by the second least significant digit. We need to ensure that if two numbers have the same digit, then their relative order from the previous pass is preserved — in other words, we need each pass's sorting algorithm to be *stable* in the sense defined above (Section 4.6.1). After the second pass, we get the sorted array $[12, 13, 45, 61, 72, 81, 85, 94]$. Since 12 appeared before 13 after the first pass, the order is preserved after the second pass.

Example 4.8 Sort $[85, 72, 94, 45, 13, 12, 61, 81]$ by LSD radix sort (base 10).

Pass 1 (sort by ones digit): distribute into buckets:

Digit	Numbers
1	61, 81
2	72, 12
3	13
4	94
5	85, 45

Concatenating: $[61, 81, 72, 12, 13, 94, 85, 45]$.

Pass 2 (sort by tens digit, stably):

Digit	Numbers
1	12, 13
4	45
6	61
7	72
8	81, 85
9	94

Concatenating: $[12, 13, 45, 61, 72, 81, 85, 94]$. Sorted!

Note that stability is essential: in Pass 2, both 81 and 85 have tens digit 8, and their relative order from Pass 1 (81 before 85) is preserved, ensuring correctness.

Thus, the algorithm is simply stated as Algorithm RadixSort, using Counting Sort (Section 4.6.1) for the inner stable pass.

Each CountSort pass runs in $O(n + r)$ time, so the overall time complexity is $O(k(n + r))$, which is $O(kn)$ for constant radix. For $r = 10$, sorting $[85, 72, 94, 45, 13, 12, 61, 81]$ yields $O(kn)$ time (here, $k = 2$, $O(n)$ per pass).

Algorithm RadixSort: Radix sort for integers with k digits

```
   input: A: array[1..n] of k-digit integers in radix r
   output: A: array[1..n] of int // sorted
1  for i = 1 to k do
2  |  CountSort(A, digit i);                          // stable inner pass
3  end
```

4.7 LLP Perspective

At face value, the sorting problem does not appear to be searching for a satisfying element in a lattice. However, it can be viewed from that angle. Sorting an array A is the same as finding a permutation π such that applying π to A yields the sorted array. For example, if $A = [45, 12, 15]$, then the permutation $[3, 1, 2]$ sorts A once the entry at position i moves to position $\pi(i)$.

There are many ways to represent a permutation. We represent it by its *inversion vector*. The *inversion number* of entry i is the number of entries less than i that appear to the right of i in the permutation. Thus, the permutation $[3, 1, 2]$ is equivalently written as the inversion vector $[2, 0, 0]$: there are two numbers less than 3 that appear to the right of 3, zero numbers less than 1 to the right of 1, and zero to the right of 2. There is a one-to-one correspondence between permutations and inversion vectors.

Consider the set of all inversion vectors. The last entry is always zero (nothing can appear to its right). The first entry can have $0 \ldots n-1$ inversions, the second entry $0 \ldots n-2$, and so on; the total number of inversion vectors is $n!$. We order inversion vectors by componentwise comparison. Under this order, the set of inversion vectors forms a finite distributive lattice. The bottom element is the zero vector, corresponding to the identity permutation. Applying the zero inversion vector to any array leaves it unchanged. *Our goal is to find the inversion vector whose application produces the sorted array.* Figure 4.2 illustrates this lattice for arrays of size three: each node is labeled with an inversion vector $(a, b, 0)$ and the array obtained by applying the corresponding permutation to $A = [45, 12, 15]$. The sorted array $[12, 15, 45]$ sits at $(2, 0, 0)$, so sorting A is equivalent to advancing from the bottom of the lattice to that node.

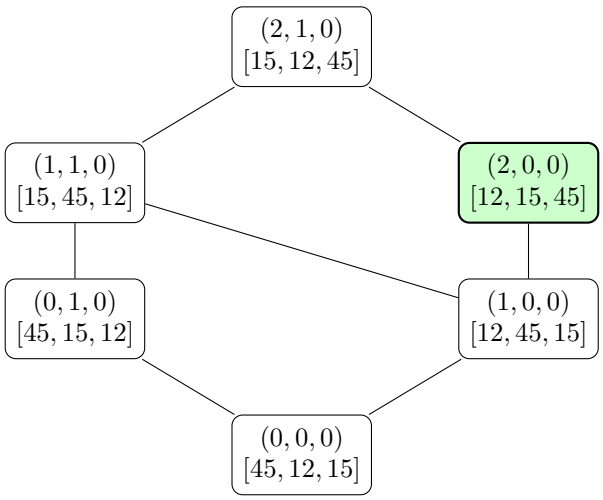


Figure 4.2: Lattice of inversion vectors for arrays of size three, applied to $A = [45, 12, 15]$. Each node shows an inversion vector $(a, b, 0)$ and the array obtained by applying the corresponding permutation to A. The sorted array $[12, 15, 45]$ sits at $(2, 0, 0)$ (shaded); sorting reduces to advancing upward in the lattice to that node.

Suppose G is less than the unique inversion vector that sorts A. Then, when G is applied to A, the array is not sorted, so there exists an index i with a missing inversion: some $j > i$ with $A[j] < A[i]$. The lattice of Figure 4.2 thus represents the permutations of A and contains a unique target that sorts it. Rather than maintain G explicitly, we keep only the effect of applying G to A — that is, we mutate A in place, and with that we give up the correspondence between the two: once A has been permuted, position j of the current array is no longer coordinate j of the inversion vector.

The simplest choice of forbidden predicate checks only *immediate* inversions: $A[i] > A[i+1]$. Swapping $A[i]$ and $A[i+1]$ removes that inversion and reverses no other pair, so each advance leaves the array exactly one inversion closer to sorted. This yields Algorithm LLP-Sort-Adjacent.

Algorithm LLP-Sort-Adjacent: Sorting via adjacent swaps

input: A: array$[1..n]$ of int
output: A: array$[1..n]$ of int `// sorted`

1 **forbidden**$(j \in [1..n-1])$: $A[j] > A[j+1]$
2 $\Rightarrow$ **advance**: $swap(A[j], A[j+1])$; `// one inversion fewer`

Both Bubble Sort and Insertion Sort are deterministic schedules of LLP-Sort-Adjacent: they fix an order in which the forbidden indices are evaluated (round-robin left-to-right for Bubble Sort; incremental for Insertion Sort). The generic LLP framework, by contrast, is

nondeterministic — multiple indices may be forbidden at any point, and any of them may advance.

A slight generalization widens the forbidden predicate to witness *any* right-hand index, not just the immediate neighbor. Algorithm LLP-Sort-Pairwise declares j forbidden whenever any $k > j$ has $A[j] > A[k]$, and advances by swapping $A[j]$ with $A[k]$.

Algorithm LLP-Sort-Pairwise: Sorting via non-adjacent swaps

input: A: array$[1..n]$ of int
output: A: array$[1..n]$ of int `// sorted`
1 **forbidden**$(j \in [1..n-1])$: $\exists\, k > j : A[j] > A[k]$
2 $\Rightarrow$ **advance**: $swap(A[j], A[k])$;

The forbidden predicate of LLP-Sort-Adjacent implies that of LLP-Sort-Pairwise — an adjacent inversion $A[j] > A[j+1]$ is witnessed by $k = j+1$ — so any execution of LLP-Sort-Adjacent is also a valid execution of LLP-Sort-Pairwise. Swapping any out-of-order pair, adjacent or not, also strictly decreases the inversion count, so LLP-Sort-Pairwise terminates for the same reason. The extra freedom in choosing k lets a single advance step resolve multiple inversions at once; Quicksort can be viewed as an implementation of LLP-Sort-Pairwise where the pivot induces a batched advance: every j on the less-than-pivot side and every k on the greater-than-pivot side are resolved in a single partition pass.

4.8 Summary

Table 4.1 lists the sorting algorithms discussed in this chapter, with their sequential time complexities.

Algorithm	Time
Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Merge Sort	$O(n \log n)$
QuickSort	$O(n^2)$ worst, $O(n \log n)$ average
Counting Sort	$O(n + r)$
Radix Sort	$O(kn)$

Table 4.1: Sequential sorting algorithms. For counting sort, r is the range of input values. For radix sort, k is the number of digits and the radix is treated as a constant.

This chapter has surveyed a wide range of sorting algorithms. The simple comparison-based methods, Bubble Sort and Insertion Sort, run in $\Theta(n^2)$ time. The divide-and-conquer methods are far more efficient: Merge Sort runs in $O(n \log n)$ worst-case time, and QuickSort in $O(n \log n)$ expected time. The non-comparison-based Counting Sort and Radix Sort bypass

the comparison lower bound by exploiting structure in the keys, and serve as a reminder that lower bounds are model-specific.

4.9 Problems

4.1. **(Binary Search Implementation)** Implement the binary search algorithm discussed in Section 4.1.

4.2. **(Radix Sort Stability)** Radix Sort requires the per-digit pass to be a *stable* sort. Give a small counterexample showing that an unstable per-digit sort can produce an incorrect final output, even when each individual pass correctly groups elements by their current digit.

4.3. **(k-Way Merge)** Give an algorithm to merge k sorted arrays of size n into a single sorted array.

4.4. **(Two-Part QuickSort)** We have partitioned the array into three parts in the QuickSort algorithm. Give a version of the QuickSort algorithm in which the array is partitioned only in two parts: entries that are less than *pivot* and the entries that are greater than or equal to the pivot.

4.5. **(QuickSelect Expected Linear)** Using the partition routine from QuickSort, give an algorithm that returns the k-th smallest element of $A[1..n]$ in expected $O(n)$ time. Justify the expected complexity.

4.6. **(Sorting Algorithm Stability)** A sorting algorithm is *stable* if it preserves the relative order of records with equal keys. State which of the following are stable, and justify each: Bubble Sort, Insertion Sort, Merge Sort, QuickSort (Lomuto and Hoare partitions), and Radix Sort.

4.7. **(External Memory Merge Sort)** Suppose the array $A[1..n]$ does not fit in memory. RAM holds $M \ll n$ elements, and reads and writes to disk occur in blocks of B elements. Describe an external-memory Merge Sort and count the number of block I/Os it performs.

4.8. **(k-Near-Sorted Sorting)** An array $A[1..n]$ is said to be *k-near-sorted* if every element is at most k positions away from its position in the sorted output. Show that such an array can be sorted in $O(n \log k)$ time. (Hint: maintain a min-heap of size $k+1$.)

4.9. **(Incremental Sort)** A sorted array $A[1..n]$ has a new element a_{n+1} inserted. Update the sort in $O(\log n + k)$ time, where k is the number of elements displaced.

4.10. **(Stable Sort with Ties)** The comparison sort must be *stable*: equal-key elements must keep their input order. Show that LLP-Sort-Adjacent is automatically stable, but LLP-Sort-Pairwise is not.

4.10 Bibliographic Remarks

Quicksort was invented by Hoare in 1961 [Hoa61]; the original paper already contained a two-way in-place partition. The three-way "Dutch National Flag" partition is due to Dijkstra and appears in *A Discipline of Programming* [Dij76]. Merge sort is usually attributed to von Neumann in 1945, and its recurrence-based analysis is given in Knuth's *The Art of*

Computer Programming, Vol. 3: Sorting and Searching [Knu98], which also catalogs Bubble Sort, Insertion Sort, and numerous refinements. Radix sort predates digital computers: it was the basis of Herman Hollerith's electromechanical tabulator used for the 1890 U.S. Census, and a modern analysis (including stability via Counting Sort as the inner pass) appears in [Knu98]. The $\Omega(n \log n)$ lower bound for comparison-based sorting via the decision-tree argument is also treated in [Knu98].

Chapter 5

Graphs

5.1 Introduction

Graph theory has applications across many domains: social-network analysis (detecting communities via clustering), transportation and logistics (shortest-path algorithms such as Dijkstra's for routing, bipartite matching for crew scheduling), computer networks (spanning-tree protocols for loop-free communication, centrality measures for identifying critical nodes), and bioinformatics (traversing protein-interaction and gene-regulatory networks to find functional pathways).

Graphs come in numerous varieties. They may be *directed* or *undirected*. They may be *simple* or with loops and parallel edges. They may be *weighted* or *unweighted*. They may be *acyclic* or not. For example, when modeling transportation networks, the nodes may represent important milestones in a city. A directed edge may represent a one-way street, and an undirected edge may represent a street that can be traversed in either direction. The weights may represent the distance between the nodes.

In this chapter, we cover some basic traversal algorithms in a simple directed graph. This chapter is organized as follows. Section 5.2 describes various commonly used representations of graphs. Section 5.3 gives the breadth-first-search (BFS) method of traversing a graph. Section 5.4 gives the depth-first-search (DFS) method of traversing a graph. Section 5.5 gives an algorithm to "layer" a directed acyclic graph. Section 5.6 shows how to topologically sort a directed acyclic graph. Section 5.7 gives an algorithm to compute the strongly connected components of a directed graph. Section 5.8 revisits these problems from the Lattice-Linear Predicate (LLP) perspective. Many other algorithms on graphs, such as the shortest path algorithms and minimum spanning tree algorithms, are covered in later chapters.

5.2 Graph Representations

We present an undirected graph and its directed counterpart, each with vertices v_0, v_1, v_2, v_3, shown with adjacency matrix and adjacency list representations (as linked lists).

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

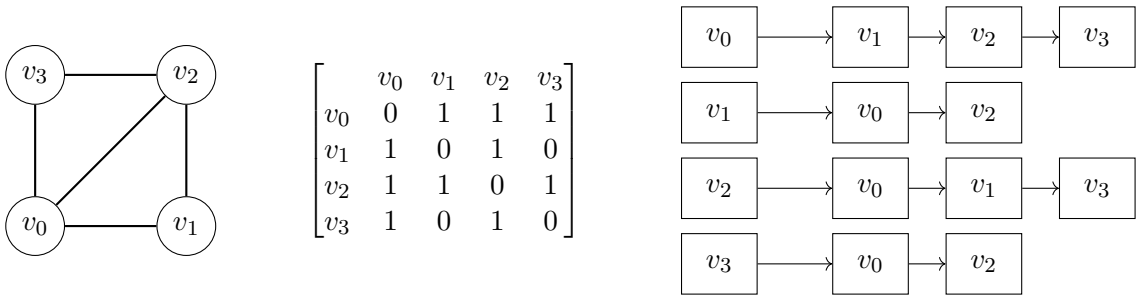


Figure 5.1: An undirected graph (left), its adjacency matrix (middle), and its adjacency list representation (right).

Fig. 5.1 shows an undirected graph together with both representations. In the adjacency matrix, an entry of 1 indicates an edge; because the graph is undirected, the matrix is symmetric. Such a representation allows one to answer any question of the form: is there an edge between v_i and v_j in constant time. However, it takes $O(n^2)$ space irrespective of the number of edges in the graph. When the graph is sparse, the adjacency list representation on the right is less wasteful: it keeps one linked list per vertex, holding exactly that vertex's neighbors.

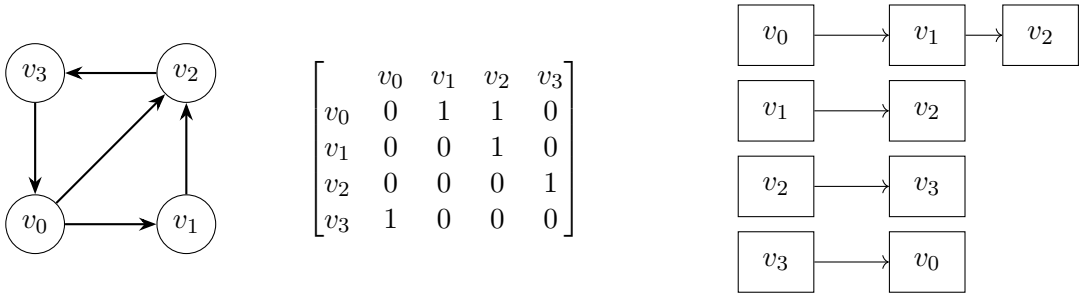
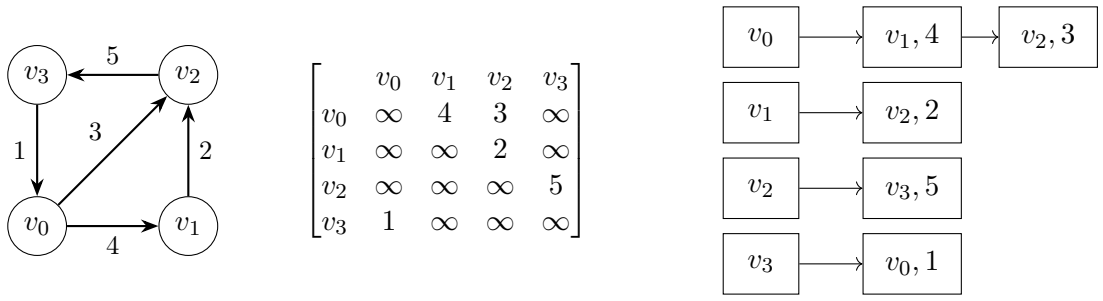


Figure 5.2: A directed graph (left), its adjacency matrix (middle), and its adjacency list representation (right).

Fig. 5.2 shows the directed version of the same graph with the corresponding representations. The adjacency matrix is no longer symmetric: the entry in row v_i and column v_j is 1 exactly when the edge (v_i, v_j) is present, and the adjacency list of v_i holds only the vertices that v_i points to.



$$\begin{bmatrix} & v_0 & v_1 & v_2 & v_3 \\ v_0 & \infty & 4 & 3 & \infty \\ v_1 & \infty & \infty & 2 & \infty \\ v_2 & \infty & \infty & \infty & 5 \\ v_3 & 1 & \infty & \infty & \infty \end{bmatrix}$$

Figure 5.3: A weighted directed graph (left), its adjacency matrix with edge weights in place of 1s and ∞ in place of 0 for pairs with no edge (middle), and its adjacency list with each entry annotated by the weight of the corresponding edge (right).

A weighted graph associates a numeric weight with every edge, representing, for example, a distance or a cost. Fig. 5.3 shows the same directed graph with weights attached to its edges. The adjacency matrix now stores the weight of edge (v_i, v_j) in place of 1, with ∞ indicating no edge; the adjacency list stores, for every vertex v_i, the pairs (v_j, w) for each edge (v_i, v_j) of weight w. Chapters 7 and 8 use this weighted representation for the shortest-path and minimum-spanning-tree algorithms.

5.3 Breadth-First Search (BFS)

In this section, we describe the breadth-first-search (BFS), one of the two most common ways of traversing a graph (the other being depth-first search, covered in the next section). The algorithms are stated for directed graphs; an undirected graph is covered by treating each edge as a pair of opposite arcs. Traversal is required for many reasons. We may be interested in searching for a node, finding the number of components in a graph, or checking for some user-specified property. We first present a general (nondeterministic) traversal algorithm; different strategies for choosing which vertex to explore next yield BFS as a special case.

General Traversal of a Directed Graph

In Algorithm Traversal, we keep *visited* as the Boolean vector marking the set of vertices that can be reached from the source vertex s ($visited[i] = 1$ iff v_i has been reached). We also keep S as a set of vertices that have been visited but not explored. Using different strategies to remove an index from S, we can traverse the graph in a different order. If S is maintained as a queue, the removal corresponds to removing from the head of the queue, and the addition corresponds to appending at the end of the queue, then we traverse the graph in a *breadth-first* manner.

Each vertex is added to and removed from S at most once, and each edge (j, k) is examined exactly once (when j is being explored). Hence Algorithm Traversal runs in $O(n + m)$ time.

Suppose that we are given a source vertex s in the graph. Our task is to find all the nodes that are reachable from s. Algorithm BFS-Traversal can be used for this purpose. It will generate a tree rooted at s such that every node is at the minimum distance from s. We maintain a variable *parent* that gives the parent of any node x in the graph. If x is reachable

Algorithm Traversal: Reachable vertices from a source vertex

input: directed graph (V, E) with $|V| = n$, $|E| = m$
$succ(j)$: list of successors of j, for each $j \in V$
s: source vertex
output: $visited$: array$[0..n-1]$ of int initially 0

```
   var S: set of int // vertices discovered but not yet explored
 1 S.add(s) // add the source s to start
 2 visited[s] := 1;
 3 while ¬S.empty() do
 4 |   j := S.removeAny(visited) // remove any index from S
 5 |   forall (k ∈ succ(j)) :
 6 |   |   if (visited[k] = 0) then
 7 |   |   |   visited[k] := 1 // advance on k
 8 |   |   |   S.add(k) // update frontier
 9 |   |   end
10 |   end
11 end
12 return visited ;                                    // the reachable set
```

from s, then following the *parent* pointer from x to s, we will get a shortest path from s to x. A node v_i has its parent set to v_j if v_j is along a path from s to v_i and has its distance one less than that of v_i.

Each vertex is enqueued at most once and dequeued at most once; each edge (j, k) is examined exactly once when j is dequeued. Hence Algorithm BFS-Traversal runs in $O(n+m)$ time.

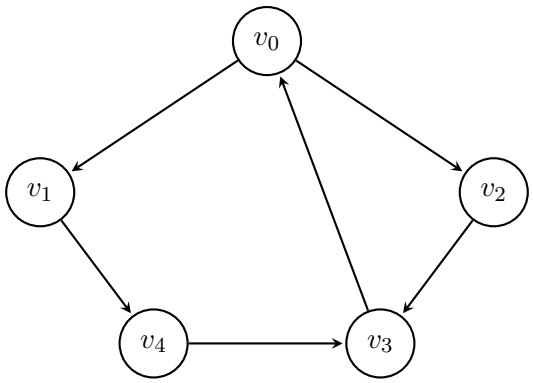


Figure 5.4: A directed graph used to illustrate BFS and DFS traversals.

Algorithm BFS-Traversal: Breadth-first search tree from a source vertex in a directed graph

input: directed graph (V, E) with $|V| = n$, $|E| = m$
$succ(j)$: list of successors of j, for each $j \in V$
s: source vertex
output: $dist$: array$[0..n-1]$ of int `// distance from` s`, initially` ∞
$parent$: array$[0..n-1]$ of int `// BFS tree, initially` -1

```
var S: queue of int // vertices discovered but not yet explored
1  S.enqueue(s);
2  dist[s] := 0;
3  while ¬S.empty() do
4  |  j := S.dequeue();
5  |  forall (k ∈ succ(j)) :
6  |  |  if (dist[k] > dist[j] + 1) then
7  |  |  |  dist[k] := dist[j] + 1;
8  |  |  |  parent[k] := j;
9  |  |  |  S.enqueue(k);
10 |  |  end
11 |  end
12 end
13 return dist ;                    // distances from s; BFS tree given by parent
```

Example 5.1 [BFS on a directed graph] Run BFS from source v_0 on the graph in Figure 5.4. Initially $dist[v_0] = 0$ and the queue is $[v_0]$.

- Dequeue v_0. Neighbors v_1, v_2 have $dist = \infty$. Set $dist[v_1] = 1$, $dist[v_2] = 1$, $parent[v_1] = parent[v_2] = v_0$; enqueue both.
- Dequeue v_1. Neighbor v_4: set $dist[v_4] = 2$, $parent[v_4] = v_1$, enqueue v_4.
- Dequeue v_2. Neighbor v_3: set $dist[v_3] = 2$, $parent[v_3] = v_2$, enqueue v_3.
- Dequeue v_4. Neighbor v_3 already has $dist[v_3] = 2$ (not improved).
- Dequeue v_3. Neighbor v_0 already has $dist[v_0] = 0$ (not improved).

Final distances from v_0:

Node	$dist[\cdot]$ (distance)	$parent[\cdot]$
v_0	0	–
v_1	1	v_0
v_2	1	v_0
v_4	2	v_1
v_3	2	v_2

The BFS tree is shown in Figure 5.5.

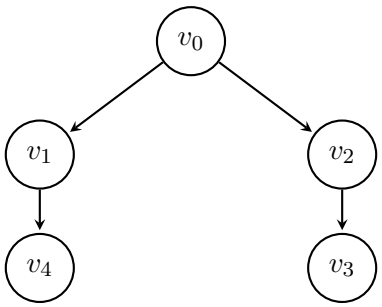


Figure 5.5: BFS tree rooted at v_0 for example 5.1. Every vertex appears at its shortest distance from v_0.

5.4 Depth-First Search (DFS)

Another way to traverse a graph is the depth-first search method. While the breadth-first search traverses the graph one layer at a time, the depth-first search continues to take a path as far as it can go, and then it backtracks to traverse the remaining graph (in the same fashion!). Algorithm Traversal and Algorithm BFS-Traversal are written as flat loops that keep the pending vertices in an explicit set. We present depth-first search as a recursive function instead: the backtracking that defines depth-first order is exactly the unwinding of the call

stack, so no explicit stack is needed.

Algorithm DFS-Traversal: Depth-first search tree from a source vertex in a directed graph

input: directed graph (V, E) with $|V| = n$, $|E| = m$
$succ(j)$: list of successors of j, for each $j \in V$
s: source vertex
output: *visited*: array$[0..n-1]$ of int, initially 0
parent: array$[0..n-1]$ of int, initially -1
discovered: array$[0..n-1]$ of int
finished: array$[0..n-1]$ of int

var *tick*: int initially 1

```
1  function DFS(j):
2      visited[j] := 1;
3      discovered[j] := tick;
4      tick := tick + 1;
5      for k ∈ succ(j) do
6          if visited[k] = 0 then
7              parent[k] := j;
8              DFS(k);
9          end
10     end
11     finished[j] := tick;
12     tick := tick + 1;
13 end
14 DFS(s);
```

The recursive call $DFS(k)$ is made for each vertex k at most once, and each edge (j, k) is examined exactly once when j's adjacency list is scanned. Hence Algorithm DFS-Traversal runs in $O(n + m)$ time.

Algorithm DFS-Traversal visits all vertices reachable from the source vertex s in a depth-first search manner. The variable *visited* is used to mark all the vertices that can be reached from s. The variable *discovered* is used to store the time (tick) when each vertex in the graph is explored. The variable *finished* is used to store the tick when the algorithm is finished exploring a vertex. The variable *parent* is used to store the depth-first tree. Anytime a vertex k is visited for the first time from a vertex j, the $parent[k]$ is recorded as j.

The variables *discovered* and *finished* are used only to record the *tick* when a vertex is starting to be explored and finished for exploration.

Example 5.2 [DFS on a directed graph] Run DFS from source v_0 on the graph in Figure 5.4. From v_0 we visit v_1; from v_1 we visit v_4; from v_4 we visit v_3. When v_3 is explored, its only outgoing edge points to v_0, which is already visited, so v_3 finishes. We backtrack to v_4 (no other outgoing edges), then to v_1, and finally to v_0. From v_0 we now visit v_2, whose outgoing edge leads to v_3 (already visited); v_2 finishes, and DFS terminates.

The resulting *discovered* and *finished* ticks are:

Node	*discovered*	*finished*
v_0	1	10
v_1	2	7
v_4	3	6
v_3	4	5
v_2	8	9

The DFS tree is shown in Figure 5.6.

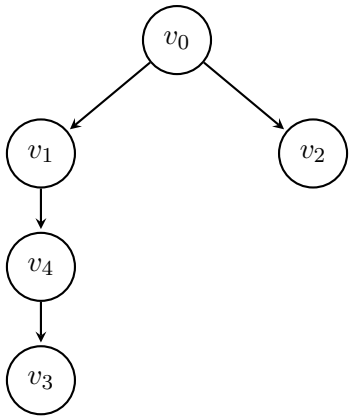


Figure 5.6: DFS tree rooted at v_0 for example 5.2. The edge $v_2 \to v_3$ of the original graph is a *cross edge* (not part of the DFS tree, since v_3 is discovered earlier via v_4).

5.5 Layering of a Directed Acyclic Graph

Suppose that we are given a directed graph with no cycles. This graph can be "layered" as follows . Every vertex i in the graph is assigned a number $G[i]$ such that if there is a directed edge from i to j, then the number assigned to i is strictly less than that of j, i.e., for all edges $(i, j) \in E$, $G[i]$ is less than $G[j]$. An application of this concept is the prerequisite structure of courses in a university, and the integer assigned to the course corresponds to the earliest semester in which the course can be taken by a student.

Our goal is to compute, for each vertex j, the integer $G[j]$ equal to the length of the longest path from any source vertex (a vertex with no incoming edges) to j. We present a classical algorithm that maintains the in-degree of every vertex and processes vertices in the order in which all of their predecessors have been assigned layers.

The idea is simple. Initially, every vertex with in-degree 0 lies at layer 0. We place all such vertices in a queue. We then repeatedly remove a vertex j from the queue, and for every successor k of j (i.e., $(j,k) \in E$), we decrement k's in-degree. Whenever k's in-degree becomes 0, we have seen all of k's predecessors and know their layers; we then set $G[k]$ to one more than the maximum of $G[i]$ over all predecessors i of k, and enqueue k. Algorithm Layering gives the details.

Algorithm Layering: Layering a directed acyclic graph

input: directed acyclic graph (V, E) with $|V| = n$, $|E| = m$
$pre(j)$: list of predecessors of j, for each $j \in V$
$succ(j)$: list of successors of j, for each $j \in V$
output: G: array$[0..n-1]$ of int, the layer of each vertex

```
var Q: queue of int // initially empty
indeg[j] := |pre(j)| for all j;
forall j ∈ V :
    if indeg[j] = 0 then
        G[j] := 0;
        Q.enqueue(j);
    end
end
while ¬Q.empty() do
    j := Q.dequeue();
    forall k ∈ succ(j) :
        indeg[k] := indeg[k] − 1;
        if indeg[k] = 0 then
            G[k] := 1 + max_{i∈pre(k)} G[i];
            Q.enqueue(k);
        end
    end
end
return G;
```

Each vertex is enqueued and dequeued at most once, and each edge (j,k) is processed exactly once when j is dequeued. Hence the time complexity of the algorithm is $O(n+m)$.

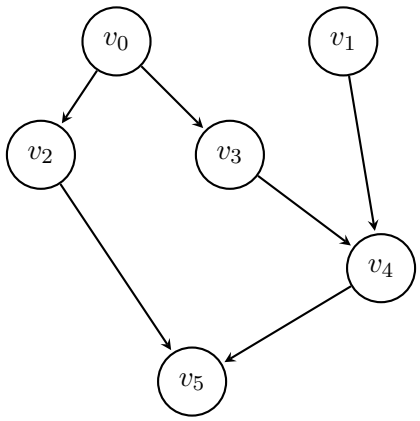


Figure 5.7: A directed acyclic graph used to illustrate layering.

Example 5.3 [Layering a DAG] Run the layering algorithm on the DAG of Figure 5.7. In-degrees are $indeg[v_0] = indeg[v_1] = 0$, $indeg[v_2] = indeg[v_3] = 1$, $indeg[v_4] = indeg[v_5] = 2$.

- Initially v_0 and v_1 have in-degree 0; set $G[v_0] = G[v_1] = 0$ and enqueue both. Queue: $[v_0, v_1]$.
- Dequeue v_0. Successors v_2, v_3: their in-degrees drop to 0; set $G[v_2] = G[v_3] = 1$ and enqueue them.
- Dequeue v_1. Successor v_4: in-degree drops to 1 (not yet ready).
- Dequeue v_2. Successor v_5: in-degree drops to 1 (not yet ready).
- Dequeue v_3. Successor v_4: in-degree drops to 0; set $G[v_4] = 1 + \max(G[v_1], G[v_3]) = 1 + \max(0, 1) = 2$ and enqueue v_4.
- Dequeue v_4. Successor v_5: in-degree drops to 0; set $G[v_5] = 1 + \max(G[v_2], G[v_4]) = 1 + \max(1, 2) = 3$ and enqueue v_5.
- Dequeue v_5 (no successors).

The computed layers are

Vertex	$G[\cdot]$ (layer)
$v_0,\ v_1$	0
$v_2,\ v_3$	1
v_4	2
v_5	3

The layered view of the DAG is shown in Figure 5.8. Each $G[j]$ equals the length of the longest path from any source to j; the longest path to v_5 is $v_0 \to v_3 \to v_4 \to v_5$ of length 3.

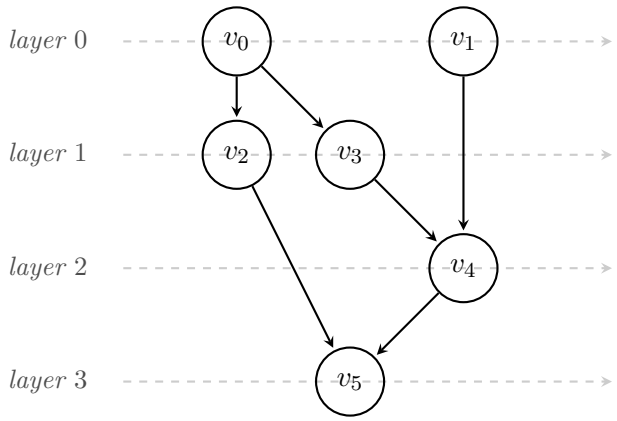


Figure 5.8: The DAG of figure 5.7 redrawn with each vertex placed on its computed layer. Every directed edge goes from a strictly smaller layer to a strictly greater one.

5.6 Topological Sort

Algorithm Layering assigns a number $G[j]$ to each vertex such that $G[i] < G[j]$ along every edge (i, j), but it does not break ties between vertices on the same layer. In many applications — evaluating arithmetic expressions, ordering recipe steps, scheduling courses one per term, processing dependencies in a build system — we want a *total* order on the vertices that respects every edge. This is called a topological sort of the DAG: a linear arrangement $v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}$ such that for every edge $(v_i, v_j) \in E$, v_i precedes v_j in the arrangement.

A simple modification of Algorithm Layering produces a topological sort. Instead of recording the layer of each vertex, we record the order in which it is dequeued: the first vertex dequeued gets position 1, the second gets position 2, and so on. The resulting permutation respects all edges because a vertex can be dequeued only after all of its predecessors have been dequeued. Because the change to Algorithm Layering is so small, we leave the algorithm as an exercise (Problem 5.9).

The time complexity is $O(n + m)$ for the same reason as that of Algorithm Layering: each vertex is enqueued and dequeued at most once and each edge is examined exactly once. If the input graph has a cycle, fewer than n vertices are ever dequeued — a cheap cycle-detection byproduct.

Example 5.4 Consider the DAG of Figure 5.7 (in-degrees $indeg[v_0] = indeg[v_1] = 0$, $indeg[v_2] = indeg[v_3] = 1$, $indeg[v_4] = indeg[v_5] = 2$). Initial queue $[v_0, v_1]$. Dequeueing in order produces $order = [v_0, v_1, v_2, v_3, v_4, v_5]$. The reader can verify that every edge of Figure 5.7 runs left-to-right in this list. The arrangement is not unique: starting from initial queue $[v_1, v_0]$ would yield $[v_1, v_0, v_2, v_3, v_4, v_5]$, also a valid topological order.

Topological sort underlies many later algorithms in this book: dynamic programming on DAGs, such as longest and shortest paths (Chapter 10), and the 2-SAT decision algorithm (Chapter 18).

5.7 Strongly Connected Components

Given a directed graph $\mathcal{G} = (V, E)$, two vertices u and v are said to be in the same *strongly connected component* (SCC) if there is a directed path from u to v and a directed path from v to u. The relation "u and v are mutually reachable" is an equivalence relation, so the vertex set V partitions into maximal strongly connected components. Collapsing each SCC into a single super-vertex yields the *component graph* $\mathcal{G}_{\text{scc}}$, which is always a directed acyclic graph.

Layering handles the acyclic case; SCC computation is the natural counterpart for graphs that may contain cycles. SCC decomposition is used, for example, in compiler dataflow analysis (to identify strongly connected regions of a control-flow graph), in dependency resolution (to detect cyclic package or module dependencies), and in 2-SAT (where satisfiability reduces to computing SCCs of the implication graph).

We present Kosaraju's algorithm, a simple and efficient $O(n+m)$-time algorithm that uses two depth-first searches (DFS). Its correctness rests on the following lemma.

Lemma 5.5 *Let u and v be vertices in distinct strongly connected components S_u and S_v of $\mathcal{G}$ (each S_x is the vertex set of the SCC containing x). Let $f(S) = \max_{w \in S} f[w]$ denote the maximum DFS finishing time among vertices of component S. If there is an edge in $\mathcal{G}_{\text{scc}}$ from S_u to S_v, then $f(S_u) > f(S_v)$.*

Proof: Consider the DFS executed on $\mathcal{G}$ and the vertex that is visited first among $S_u \cup S_v$. If it is in S_u, then since S_u is strongly connected, DFS explores all of S_u and, via the edge to S_v, all of S_v before finishing the start vertex, so $f(S_u) > f(S_v)$. If instead the first visited vertex lies in S_v, then DFS finishes S_v first (since no edge returns from S_v to S_u in $\mathcal{G}_{\text{scc}}$); the earliest vertex visited in S_u is explored later, and its finishing time — hence $f(S_u)$ — is strictly greater than $f(S_v)$.

■

The SCC-id vector $C[\,]$ assigned by Algorithm Kosaraju-SCC satisfies $C[u] = C[v]$ iff the vertex sets $S_u = S_v$, i.e., iff u and v lie in the same strongly connected component.

As a consequence, if we process vertices in decreasing order of finishing time from the first DFS but on the *transpose* graph $\mathcal{G}^{\mathrm{T}}$ (the graph obtained by reversing every edge), then each DFS tree from a new root is exactly one SCC. Edges in $\mathcal{G}^{\mathrm{T}}$ go from lower-f components to higher-f components, so a DFS from the vertex of maximum f in $\mathcal{G}^{\mathrm{T}}$ cannot leave its SCC; after that SCC is exhausted, we pick the next unvisited vertex with the largest remaining f, and so on.

Theorem 5.6 *Algorithm Kosaraju-SCC computes the strongly connected components of $\mathcal{G}$ in $O(n+m)$ time.*

Proof: *Correctness.* By Lemma 5.5, for every edge $(S_u, S_v) \in \mathcal{G}_{\text{scc}}$ we have $f(S_u) > f(S_v)$. Reversing all edges gives $\mathcal{G}^{\mathrm{T}}$ with $(S_v, S_u) \in \mathcal{G}^{\mathrm{T}}_{\text{scc}}$; consequently, when we launch a DFS in $\mathcal{G}^{\mathrm{T}}$ from the vertex with the largest f-value in its component, no transpose edge can leave the component, so the reached set is exactly that SCC. Subsequent iterations remove fully visited components and repeat on the remaining sub-DAG.

Algorithm Kosaraju-SCC: Strongly connected components

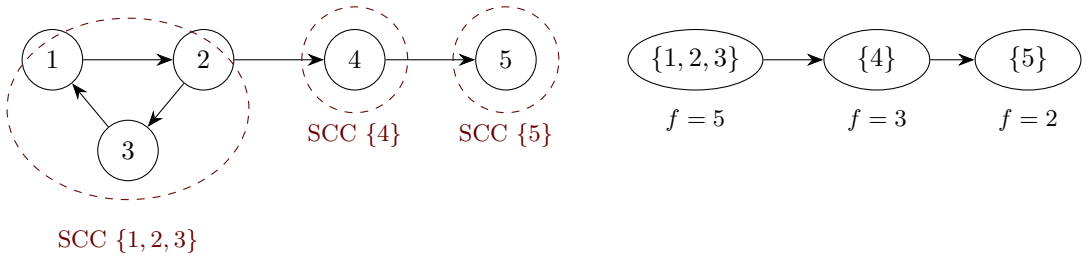
input: directed graph $\mathcal{G} = (V, E)$ with $|V| = n$, $|E| = m$
output: C: array$[1..n]$ of int `// C[v] is the identifier of the SCC containing` v

`// Pass 1: finishing-time DFS on` $\mathcal{G}$
Perform DFS on $\mathcal{G}$ and record the finishing time $f[v]$ for all vertices v;
Let $S = [v_1, v_2, \ldots, v_n]$ be the list of vertices sorted in decreasing order of f;

`// Pass 2: DFS on the transpose in reverse finishing order`
Construct $\mathcal{G}^{\mathrm{T}} = (V, E^{\mathrm{T}})$ where $E^{\mathrm{T}} = \{(v, u) \mid (u, v) \in E\}$;
Mark all vertices in $\mathcal{G}^{\mathrm{T}}$ as unvisited;
$k := 0$;
for $i = 1$ *to* n **do**
    if v_i *is unvisited in* $\mathcal{G}^{\mathrm{T}}$ **then**
        $k := k + 1$;
        Run DFS in $\mathcal{G}^{\mathrm{T}}$ starting at v_i; for all vertices w reached, set $C[w] := k$ and mark w visited;
    end
end
return C

Complexity. Each of the two DFS passes runs in $O(n + m)$ time. Constructing $\mathcal{G}^{\mathrm{T}}$ also takes $O(n + m)$ time using adjacency lists. Sorting vertices by finishing time can be done implicitly by pushing each vertex onto a stack when it finishes during the first DFS, so no additional sorting cost is incurred.

■



(a) Original graph $\mathcal{G}$ (b) Condensation $\mathcal{G}_{\text{scc}}$

Figure 5.9: (a) example directed graph $\mathcal{G}$ with three strongly connected components $\{1, 2, 3\}$, $\{4\}$, and $\{5\}$. (b) the condensation $\mathcal{G}_{\text{scc}}$, with each node labeled by the maximum DFS finishing time $f(S)$ of its component. Edges in the condensation point from larger-f components to smaller, illustrating lemma 5.5.

Example 5.7 Consider the directed graph on $\{1, \ldots, 5\}$ with edges E shown in Figure 5.9, where

$$E = \{(1, 2),\ (2, 3),\ (3, 1),\ (2, 4),\ (4, 5)\}.$$

DFS starting at vertex 1 finishes the vertices in order $3, 5, 4, 2, 1$, giving finishing times $f[3]{=}1$, $f[5]{=}2$, $f[4]{=}3$, $f[2]{=}4$, $f[1]{=}5$. The transpose graph has edges

$$E^{\mathrm{T}} = \{(2, 1),\ (3, 2),\ (1, 3),\ (4, 2),\ (5, 4)\}.$$

Processing vertices in decreasing order of f on the transpose yields three DFS trees: $\{1, 2, 3\}$ from vertex 1, $\{4\}$ from vertex 4, and $\{5\}$ from vertex 5, matching the three SCCs circled in Figure 5.9.

5.8 LLP Perspective

The classical graph algorithms of the preceding sections — traversal, BFS, layering — each have a natural LLP formulation. The value of recasting them is not new algorithms but a uniform correctness argument: once the predicate is shown to be lattice-linear, correctness follows from the LLP framework (Chapter 2) rather than from a case-specific proof. The LLP formulation also reveals which algorithms admit free parallelism, since any set of independently forbidden indices can advance simultaneously.

Graph traversal as an LLP algorithm

Before considering BFS, which tracks distances, we revisit the plain reachability problem of Section 5.3: given a source vertex v_0, compute the set of vertices reachable from v_0. We maintain a Boolean vector G with $G[0] = 1$ and $G[j] = 0$ for $j \neq 0$ and search for the least vector satisfying

$$B_{reach}(G) \equiv G[0] = 1 \ \wedge\ \forall (i, j) \in E : (G[i] = 1 \Rightarrow G[j] = 1).$$

An index j is forbidden when some predecessor i has $G[i] = 1$ but $G[j] = 0$; the advance rule sets $G[j] := 1$. Algorithm LLP-Traversal captures this.

Algorithm LLP-Traversal: Reachable vertices via LLP

input: $pre(j)$: list of predecessors of vertex j
output: G: array$[0..n-1]$ of int, initially $G[0] = 1$ and $G[i] = 0$ for $i \neq 0$
1 **forbidden**$(j \in [0..n-1])$: $G[j] = 0 \wedge \exists i \in pre(j) : G[i] = 1$
2 $\Rightarrow$ **advance**: $G[j] := 1$;

Each $G[j]$ advances at most once (from 0 to 1), and evaluating the forbidden predicate amortizes to $O(n + m)$ work across all indices. The running time is therefore $O(n + m)$.

BFS as an LLP algorithm

For breadth-first search from a source vertex v_0 we maintain a distance vector G with $G[0] = 0$ and $G[j] = \infty$ for $j \neq 0$. We want the greatest vector satisfying

$$B_{bfs}(G) \equiv \forall j \neq 0 : \forall (i, j) \in E : G[j] \leq G[i] + 1.$$

An index j is forbidden if it has a predecessor i with $G[j] > G[i] + 1$; the advance rule sets $G[j] := G[i] + 1$. Algorithm LLP-BFS gives the resulting LLP algorithm.

Algorithm LLP-BFS: BFS distances via LLP

input: $pre(j)$: list of predecessors of vertex j
output: G: array$[0..n-1]$ of int, initially $G[0] = 0$ and $G[i] = \infty$ for $i \neq 0$
1 **forbidden**$(j \in [0..n-1])$: $G[j] > \min\{G[i] + 1 \mid i \in pre(j)\}$
2 $\Rightarrow$ **advance**: $G[j] := \min\{G[i] + 1 \mid i \in pre(j)\}$;

An implementation that evaluates forbidden indices in breadth-first order runs in $O(n+m)$ time and matches the classical BFS bound.

The LLP specification above does not fix an order in which forbidden indices are evaluated. In resource-constrained settings, it can be advantageous to prioritize which forbidden index is processed first. We write the prioritization with a **priority** clause alongside **forbidden** and **advance**. In Algorithm LLP-Traversal-BFS-priority, priority is given by the value the advance would install, $\min\{G[i] + 1 \mid i \in pre(j)\}$, so the search explores vertices in order of their tentative distance from v_0 — exactly as in classical BFS with a queue. The current $G[j]$

Algorithm LLP-Traversal-BFS-priority: Prioritized BFS traversal via LLP

input: $pre(j)$: list of predecessors of vertex j
output: G: array$[0..n-1]$ of int, initially $G[0] = 0$ and $G[i] = \infty$ for $i \neq 0$
1 **ensure**$(j \in [0..n-1])$: $G[j] \leq \min\{G[i] + 1 \mid i \in pre(j)\}$;
2 **priority**(j): $\min\{G[i] + 1 \mid i \in pre(j)\}$

would not serve: an unreached vertex still holds ∞, so all of them would tie and the order among them would be unconstrained.

Processing indices in increasing order of that key matches the classical BFS schedule. The bound this gives depends on how the order is maintained. Because the weights are all 1, the keys are generated in nondecreasing order, so a FIFO queue already delivers them in order: each vertex is enqueued once when it is first discovered, each edge is examined once when its tail is dequeued, and the running time is $O(n + m)$. A general priority queue produces the same order but charges $O(\log n)$ per operation, and recomputing $\min\{G[i] + 1 \mid i \in pre(j)\}$ from scratch each time a predecessor changes costs more still; neither attains the linear bound.

Layering as an LLP algorithm

For layering a directed acyclic graph we want the least vector G of natural numbers satisfying

$$B_{layer}(G) \equiv \forall (i, j) \in E : G[i] < G[j].$$

An index j is forbidden exactly when one of its incoming edges violates B_{layer}, that is, when $G[j] \leq G[i]$ for some predecessor i; the advance rule then raises $G[j]$ to one more than the maximum layer of its predecessors. Algorithm LLP-Layering captures this.

Algorithm LLP-Layering: DAG layering via LLP

input: $pre(j)$: list of predecessors of vertex j, acyclic
output: G: array$[0..n-1]$ of int, initially 0
1 **forbidden**$(j \in [0..n-1])$: $\exists i \in pre(j) : G[j] \leq G[i]$
2 $\Rightarrow$ **advance**: $G[j] := \max_{i \in pre(j)}(G[i] + 1)$;

An implementation that evaluates forbidden indices in any topological order runs in $O(n + m)$ time.

5.9 Summary

This chapter covered fundamental graph algorithms: breadth-first search (BFS), depth-first search (DFS), layering of directed acyclic graphs, topological sorting, and strongly connected components.

Table 5.1 summarizes the time complexities of the algorithms discussed in this chapter.

Problem	**Algorithm**	**Time Complexity**
Breadth-First Search	BFS-Traversal	$O(m+n)$
Depth-First Search	DFS-Traversal	$O(m+n)$
Breadth-First Search	LLP-BFS	$O(m+n)$
Layering	Layering	$O(m+n)$
Layering	LLP-Layering	$O(m+n)$
Strongly Connected Components	Kosaraju-SCC	$O(m+n)$

Table 5.1: Graph algorithms discussed in this chapter.

5.10 Problems

5.1. **(BFS Component Sizes)** Given an undirected graph with n vertices and m edges, find the size (number of vertices) of each connected component using BFS. The graph is represented as an adjacency list. Return a list of sizes for all components.

5.2. **(Undirected Cycle Detection DFS)** Given an undirected graph with n vertices and m edges, determine if the graph contains a cycle using DFS traversal. The graph is represented as an adjacency list.

5.3. **(Bipartite Check By Coloring)** Give an efficient algorithm to check if the given undirected graph $G = (V, E)$ is bipartite. A graph is bipartite if its vertices can be split into two sets such that every edge joins a vertex of one set to a vertex of the other.

5.4. **(DFS Component Count)**

Give an algorithm to find the number of connected components in an undirected graph using DFS.

5.5. **(Find All Bridges)** Give an efficient algorithm to find all bridges in an undirected graph. A bridge is an edge whose removal increases the number of connected components.

5.6. **(Find Articulation Points)** Given an undirected graph with n vertices and m edges, find all articulation points using DFS traversal. An articulation point is a vertex whose removal increases the number of connected components in the graph.

5.7. **(Cycle Error In Layering)** Algorithm Layering gives us the least integral labels satisfying that, for all edges (i, j), $G[i]$ is strictly less than $G[j]$. Such a property is possible only for acyclic graphs. Modify the algorithm to output "error" whenever there is a cycle in the input graph.

5.8. **(Dual Layering From Top)** Algorithm Layering works by assigning level numbers to the minimal vertices in the graph and then to the next layer of vertices. Give the dual algorithm that starts with assigning level number to the maximal vertices and then to the layer of vertices that are below the maximal level.

5.9. **(Topological Sort by Modifying Layering)**

Modify the Layering algorithm (Algorithm Layering) so that, instead of assigning a layer number to each vertex of a DAG, it produces a topological sort: a linear arrangement $v_{\sigma(1)}, v_{\sigma(2)}, \ldots, v_{\sigma(n)}$ of the vertices such that for every edge $(v_i, v_j) \in E$, v_i precedes v_j. Your algorithm should run in $O(n + m)$ time and report "error" when the input graph contains a cycle.

5.10. **(Directed Cycle Three Color DFS)** Problem 2 detects cycles in an *undirected* graph by treating any non-parent visited neighbor as a back edge. This rule fails on a directed graph because a directed edge to an already-visited vertex may simply be a forward or cross edge in the DFS tree, not a cycle. Give an algorithm that decides whether a directed graph contains a cycle, using DFS with three-color marking: *white* (unvisited), *gray* (on the recursion stack), *black* (fully explored).

5.11 Bibliographic Remarks

Breadth-first and depth-first search are classical graph-traversal techniques; both are treated in detail in Cormen, Leiserson, Rivest, and Stein [CLRS09]. The layering of a directed acyclic graph (equivalently, topological sort by longest-path length) is covered in most algorithms texts, including [CLRS09]. The in-degree-based approach used in Algorithm Layering is due to A. B. Kahn (1962). Algorithm Kosaraju-SCC for strongly connected components (two DFS passes on $\mathcal{G}$ and $\mathcal{G}^{\mathrm{T}}$) is due to S. Rao Kosaraju (unpublished, 1978) and was independently described by M. Sharir in 1981. An alternative $O(n + m)$-time one-pass algorithm based on DFS low-link values is due to R. E. Tarjan (1972). A full treatment of both algorithms is given in [CLRS09]. 2-SAT, which reduces in linear time to SCC of the implication graph, is due to Aspvall, Plass, and Tarjan (1979) and is discussed in Chapter 18.

Chapter 6

Greedy Algorithms

6.1 Introduction

A greedy algorithm builds a solution one choice at a time, picking whatever a fixed selection rule declares best at each step and never reconsidering that choice. The algorithm typically starts with the trivial solution of size 0 or 1 and increases the size of the partial solution one step at a time until the original problem of size n is solved. For many applications, this strategy results in a global optimum or a provably near-optimal solution.

A greedy algorithm is cheaper than exhaustive search or dynamic programming precisely because it never backtracks, but this restriction is also its weakness: committing too early can miss the global optimum. Part of the task of this chapter is to identify problems and greedy rules that *provably* yield an optimum, and to prove correctness using the **exchange argument**: take any optimal solution O that disagrees with the greedy solution S_G, identify the earliest point of disagreement, and show that swapping S_G's choice into O produces a solution at least as good; iterating transforms O into S_G without loss.

This chapter is organized as follows. Section 6.2 describes a greedy algorithm for the interval scheduling problem. In this problem, we schedule a maximum number of jobs on a processor. Section 6.3 describes a greedy algorithm for the problem of scheduling intervals using as few rooms as possible so that no two overlapping intervals are assigned to the same room. Section 6.4 discusses the problem of scheduling n jobs so that the maximum lateness of jobs is minimized. Section 6.5 discusses the construction of a Huffman tree that minimizes the expected length of the binary code for a symbol. Section 6.6 discusses the fractional knapsack problem, where a greedy strategy based on value-to-weight ratio yields an optimal solution. Section 6.7 describes the LLP versions of these algorithms. Section 6.8 discusses why the greedy approach fails once the exchange property breaks, using 0/1 Knapsack as a counterexample to Fractional Knapsack's greedy rule.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

6.2 Interval Scheduling Problem

We start with a simple greedy algorithm. Suppose that we have n intervals with start times s_i and finish times f_i for jobs $i = 1..n$, where $s_i < f_i$. We assume that activity i occurs during the half-open interval $[s_i, f_i)$. Two intervals i and j are *compatible* if $f_i \leq s_j$ or $f_j \leq s_i$. Our goal is to select a maximum-sized subset of mutually compatible intervals.

The key idea is that always selecting the activity that finishes earliest leaves as much room as possible in the schedule for the remaining activities. So it is natural to sort the jobs by finish time, and we assume from now on that the intervals are given in that order. We assume that jobs with the same finish times are sorted according to their start times. Furthermore, no two intervals have identical start and finish times.

Let us begin with an algorithm for this problem, shown in Algorithm Interval. Let G be the set of mutually compatible intervals represented as a Boolean array such that $G[i]$ is 1 iff the activity i is in the subset of mutually compatible intervals that can be selected. It is clear that the first activity (with the least finishing time) is always in G. We now traverse over all intervals as follows. We use the variable i to denote the current activity under consideration and the variable *last* as the last activity that was included in G. We initialize *last* to 1. The current activity is included in G if and only if its start time is greater than or equal to the finish time of the last activity. If it is, our current activity becomes the last activity, and we explore the next activity.

Algorithm Interval: Selecting a maximum number of compatible intervals

```
   input: s: array[1..n] of int // start times
   f: array[1..n] of int // finish times
   output: G: array[1..n] of 0..1 initially 0 // selected intervals
1  G[1] := 1;                                  // first interval always selected
2  int last := 1;                              // index of last selected interval
3  for int i := 2 to n do
4  |  if s[i] ≥ f[last] then
5  |  |  G[i] := 1;                                  // interval i is compatible
6  |  |  last := i
7  |  end
8  end
```

The main loop visits each interval once and performs $O(1)$ work per visit, so the loop itself runs in $O(n)$ time. The sort by finish time that the greedy rule relies on costs $O(n \log n)$ and dominates, so the total cost is $O(n \log n)$. When the intervals arrive already sorted, as they often do in scheduling applications, the algorithm runs in $O(n)$ time. No extra space beyond the output array is needed, giving $O(n)$ space.

Theorem 6.1 (Interval Scheduling optimality) *Selecting activities in increasing order of finish times yields a maximum-sized set of mutually compatible activities.*

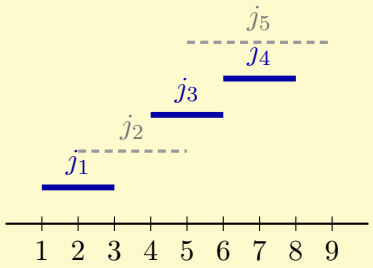
Proof: We use an exchange argument. Suppose the greedy schedule and some optimal schedule are not the same. Compare the two schedules, in order of finish time, and find the first point where they disagree. At that point, greedy picked whichever available activity finishes earliest, so greedy's choice there finishes no later than the optimal schedule's choice. So we can swap the optimal schedule's activity at that point for greedy's: it still fits with everything scheduled so far (the two schedules agreed up to here), and it still fits with everything scheduled later (since it finishes no later than the activity it replaced, it leaves at least as much room). This gives a new optimal schedule that agrees with greedy one step further. Repeating this argument turns the optimal schedule into greedy's own schedule, one agreement at a time, without ever shrinking it. Since we started from an optimal schedule and never made it worse, greedy's schedule must be optimal too.

■

Example 6.2 Suppose that we are given five jobs, each with a start time and an end time. The goal is to select the maximum number of non-overlapping jobs.

Job	Start Time	End Time
j_1	1	3
j_2	2	5
j_3	4	6
j_4	6	8
j_5	5	9

An optimal schedule selects the maximum number of non-overlapping jobs. The optimal solution is $\{j_1, j_3, j_4\}$.



The jobs are drawn in increasing order of finish time, which is the order in which the greedy rule considers them; the selected jobs are solid and the rejected ones are dashed. The algorithm takes j_1, then rejects j_2 because j_2 starts at time 2, before j_1 finishes at time 3. It takes j_3, whose start time 4 is at least the finish time 3 of j_1, and then j_4, whose start time 6 is at least the finish time 6 of j_3. Finally j_5 is rejected, since it starts at time 5, before j_4 finishes at time 8.

6.3 Interval Partition Problem

Consider a university that offers multiple courses for students. Every course i has a fixed slot $[s_i, f_i)$ where s_i indicates the start time of the lecture in course i and f_i indicates the finish time for that lecture. There are n courses in total. The university wants to use as few classrooms as possible for lectures. Our task is to assign a room to each course so that if two lectures overlap, then they must be assigned different rooms. Since each lecture is modeled as an interval, this problem is called the *interval partition problem*.

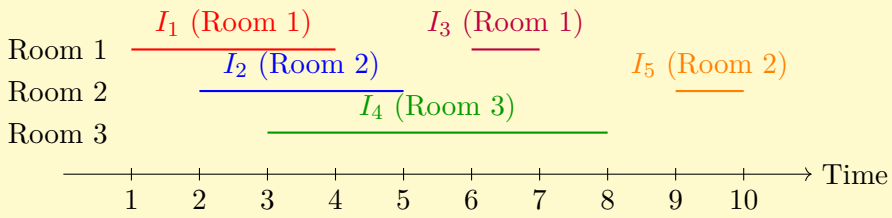
For this problem, we first look at the lower bound on the number of rooms required. Suppose that there are m overlapping lectures. Then it is clear that any room assignment must use at least m rooms.

At any moment, the number of rooms required is determined by how many intervals overlap at that time. Therefore, the goal is simply to reuse rooms whenever possible: a room occupied by a lecture that has already finished should be used again rather than left idle. Rooms are claimed and released as time advances, so it is natural to sweep the lectures in the order in which they begin. So we sort the intervals by start time, and assign courses to rooms one at a time in that order.

> **Greedy rule:** assign the next lecture to the room that became free the earliest among the rooms free at its start time, breaking ties by room number. Open a new room only if no room is free.

The first course is assigned the room 1. Now suppose that k courses have been assigned their rooms. A course is assigned a *new* room only if there are overlapping lectures for all smaller-numbered rooms, so at that moment at least $k + 1$ lectures overlap and the lower bound from above shows $k + 1$ rooms are in fact necessary. Thus, our greedy algorithm achieves the minimum number of rooms.

Example 6.3 Given the intervals $I_1 = [1, 4)$, $I_2 = [2, 5)$, $I_3 = [6, 7)$, $I_4 = [3, 8)$, $I_5 = [9, 10)$, sorted by start time they become $[1, 4), [2, 5), [3, 8), [6, 7), [9, 10)$. The greedy assignment is:



The minimum number of rooms required is equal to the maximum number of overlapping intervals, which is 3.

Theorem 6.4 (Interval Partitioning optimality) *For the Interval Partitioning Problem, assigning intervals to the first available room in order of their start times produces an optimal solution, that is, it minimizes the number of rooms required.*

Proof: The greedy algorithm sorts intervals by start time and assigns each to the first available room, opening a new room only when all existing rooms are busy.

Let d be the maximum number of intervals overlapping at any single time. Any valid assignment needs at least d rooms, since d mutually overlapping intervals must go in d distinct rooms.

Conversely, the greedy algorithm never opens more than d rooms: it opens a new room only when every existing room holds an interval overlapping the current one, so d intervals overlap at that moment, and the new room is at most the d-th.

Therefore the greedy algorithm uses exactly d rooms, which is optimal.

■

Algorithm IntervalPartition is a greedy algorithm for the Interval Partition problem.

Algorithm IntervalPartition: Assigning intervals to the fewest rooms

input: s: array[1..n] of int `// start times, sorted:` $s[1] \leq s[2] \leq \cdots \leq s[n]$
f: array[1..n] of int `// finish times`
output: *room*: array[1..n] of int `// room assigned to each interval`

var: Q: min-priority queue of $(finishTime, roomNumber)$, initially empty
numRooms: int, initially 0

```
1  for int i := 1 to n do
2  |  if Q is not empty and Q.min().finishTime ≤ s[i] then
3  |  |  room[i] := Q.extract-min().roomNumber;      // reuse earliest-free room
4  |  else
5  |  |  numRooms := numRooms + 1;                  // allocate a new room
6  |  |  room[i] := numRooms
7  |  end
8  |  Q.insert(f[i], room[i]);                      // that room is busy until f[i]
9  end
10 return room, numRooms
```

Now, let us determine the time complexity of our algorithm. The sorting of intervals based on the start times can be done in $O(n \log n)$ time. Suppose that we keep *available* rooms in a heap ordered by the $(finishTime, roomNumber)$. Whenever we need a room for a lecture, we simply remove the minimum available in the heap. This heap operation takes $O(\log n)$ time. Thus, we take $O(n \log n)$ time for the entire algorithm.

6.4 Minimizing Maximum Lateness of Jobs

Suppose that we have n jobs such that each job j takes time t_j to execute and should ideally be completed before its deadline d_j. Our task is to schedule these jobs on a single processor. Suppose that the job j is started at time s_j. Then, the job ends at time $s_j + t_j$. If $s_j + t_j$ is less than or equal to the deadline d_j, then this job is not late. Otherwise, this job has *lateness* equal to $s_j + t_j - d_j$. Our task is to schedule these jobs so that the maximum lateness over all jobs is minimized.

A job with an earlier deadline is more urgent to schedule early, since delaying it risks lateness that cannot be recovered, whereas a job with a later deadline has more slack to absorb a delay. This suggests scheduling jobs in increasing order of deadline d_j, rather than by processing time t_j or slack $d_j - t_j$.

Algorithm MinMaxLateness: Scheduling by earliest deadline to minimize maximum lateness

input: t: array$[1 \dots n]$ of int `// processing times, sorted by deadline`
d: array$[1 \dots n]$ of int `// deadlines in non-decreasing order`
output: G: array$[1 \dots n]$ of int initially 0 `// start time of each job`

```
1 int last := 0;
2 int lateness := 0;
3 for int i := 1 to n do
4 |   int finish := last + t[i];
5 |   G[i] := last;
6 |   last := finish;
7 |   if (finish > d[i]) then lateness := max(lateness, finish − d[i]);
8 end
9 return G, lateness;
```

Algorithm MinMaxLateness processes jobs in the order of increasing deadlines. At each step, *last* holds the current end-of-schedule time, $G[i]$ records the starting time of job i, and $finish$ is the moment it completes. The loop performs $O(1)$ work per job, so the loop itself runs in $O(n)$ time. The sort by deadline that the greedy rule relies on costs $O(n \log n)$ and dominates, so the total cost is $O(n \log n)$. When the jobs arrive already sorted by deadline, the algorithm runs in $O(n)$ time. The algorithm uses $O(n)$ space for the output array G.

Theorem 6.5 (EDF optimality) *For the problem of scheduling n jobs with known processing times and deadlines on a single machine to minimize the maximum lateness, the Earliest Deadline First (EDF) algorithm produces an optimal solution.*

Proof:

The EDF algorithm schedules jobs in increasing order of their deadlines. Let S_G be the schedule produced by the EDF algorithm, and let O be an optimal schedule that minimizes maximum lateness. Suppose for contradiction that O is different from S_G and has a different order of jobs while achieving a smaller maximum lateness.

Since $O \neq S_G$, the order of O is not sorted by deadline, so some job j is immediately followed by a job i with $d_i < d_j$. Let the pair start at time t. Before the exchange, $C_j = t + p_j$ and $C_i = t + p_j + p_i$; after it, $C'_i = t + p_i$ and $C'_j = t + p_i + p_j = C_i$. Hence $C'_i - d_i \leq C_i - d_i$, and since $d_i < d_j$, also $C'_j - d_j = C_i - d_j \leq C_i - d_i$. Neither lateness of the pair exceeds $C_i - d_i$, its larger value before the exchange, and taking the maximum with zero preserves this. Every other job keeps its completion time, so the maximum lateness of O does not increase. By iterative application of this argument, we can transform O into S_G without increasing the maximum lateness. Since O was assumed to be optimal, it follows that S_G is also optimal. Therefore, EDF produces an optimal schedule.

■

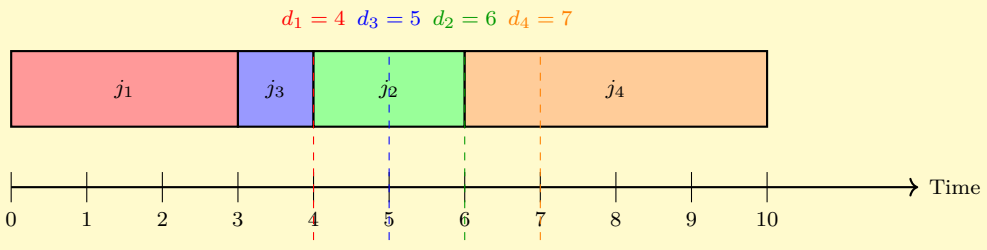
Example 6.6 Consider the following example.

Job j_i	**Processing Time t_i**	**Deadline d_i**
j_1	3	4
j_2	2	6
j_3	1	5
j_4	4	7

Sorting by deadline gives the order j_1, j_3, j_2, j_4. Tracing the algorithm row by row:

i	job	$t[i]$	$d[i]$	$G[i]$ (= *last* on entry)	*finish*	*lateness* after
1	j_1	3	4	0	3	0
2	j_3	1	5	3	4	0
3	j_2	2	6	4	6	0
4	j_4	4	7	6	10	3

Only job j_4 runs past its deadline, contributing lateness $10 - 7 = 3$. The resulting schedule using Earliest Deadline First (EDF) is shown next.



The job j_2 finishes at time 6 and has a deadline of 6. The job j_4 finishes at time 10 but has a deadline of 7. Using earliest deadline first, this schedule minimizes *maximum lateness*, which is 3 units.

6.5 Huffman Tree

The goal of the Huffman tree problem is to find an efficient encoding of a set of symbols. We are given n symbols. For each symbol i, the value $p[i]$ gives the probability that it appears in the given text, with $\sum_i p[i] = 1$. Our goal is to design a *prefix* code such that the given text can be translated to a binary string from which the original text can be recovered. We will create a binary tree so that the leaves correspond to the symbols and the path from the root to the leaf corresponds to the *code* for the symbol. Note that only the leaves of the tree correspond to symbols. Also note that leaves may be at different levels, and therefore this method returns a variable-length code for symbols rather than a fixed-length code.

In an optimal prefix code, symbols with higher probability should have shorter codewords, while rare symbols can tolerate longer ones. The greedy strategy builds the code tree from the bottom up by repeatedly merging the two least probable symbols, which pushes them deeper in the tree where the longer codewords are assigned.

Given n symbols with their probabilities of occurrence, the Huffman coding algorithm constructs an optimal prefix code using the following greedy approach. We first insert all symbols into a min-priority queue keyed by probability. As long as the queue contains more than one node, we remove the two nodes with the smallest probabilities, merge them into a new node whose probability is the sum of the two, and insert this merged node back into the queue. When only one node remains, it is the root of the Huffman tree. Finally, we obtain the code for each symbol by traversing the tree from the root, labeling the left branch with 0 and the right branch with 1.

Algorithm HuffmanCoding: Building a Huffman tree by merging the two least probable symbols

input: set of symbols $S = \{s_1, s_2, \ldots, s_n\}$ and probabilities $p(s_1), p(s_2), \ldots, p(s_n)$ with $\sum_i p(s_i) = 1$
output: Huffman tree T

```
Q := priority queue (min-heap) of nodes initially null ;
for each s_i ∈ S do
    Create a leaf node n_i with probability p(s_i) and symbol s_i;
    insert n_i into Q;
end
for i = 1 to n − 1 do
    x := extract-min(Q) ;            // remove node with smallest probability
    y := extract-min(Q) ;                                  // remove next smallest
    z := new internal node with children x and y;
    p(z) := p(x) + p(y) ;                                     // sum probabilities
    insert z into Q;
end
T := extract-min(Q) ;                                          // final tree (root)
return T
```

The time and space complexity of the Huffman coding algorithm depends on the use of a min-heap for the priority queue Q. For the time complexity, building Q from the n leaf nodes costs $O(n)$. The main loop runs $n-1$ times, and each iteration performs two extract-min operations and one insertion, all at cost $O(\log n)$; the loop therefore contributes $O(n \log n)$. The final extract-min is $O(\log n)$, which is absorbed, giving an overall running time of $O(n \log n)$. For the space complexity, the priority queue holds at most n nodes at any time, and the constructed tree has n leaves and $n-1$ internal nodes for a total of $2n-1$ nodes; both contribute $O(n)$, so the algorithm uses $O(n)$ space, where n is the number of symbols.

Example 6.7 Consider six symbols with probabilities that exercise merges of internal nodes with each other:

Symbol	A	B	C	D	E	F
Probability	0.45	0.13	0.12	0.16	0.09	0.05

The priority queue evolves as follows, with probabilities shown in non-decreasing order at each step and internal nodes denoted by the sum of their children:

Step	Extracted (smallest two)	Queue after insertion
0	—	.05, .09, .12, .13, .16, .45
1	.05, .09 → .14	.12, .13, .14, .16, .45
2	.12, .13 → .25	.14, .16, .25, .45
3	.14, .16 → .30	.25, .30, .45
4	.25, .30 → .55	.45, .55
5	.45, .55 → 1.0	1.0

The resulting Huffman codes are:

Symbol	Probability	Huffman Code
A	0.45	0
C	0.12	100
B	0.13	101
F	0.05	1100
E	0.09	1101
D	0.16	111

The expected code length is $0.45 \cdot 1 + (0.12 + 0.13) \cdot 3 + (0.05 + 0.09) \cdot 4 + 0.16 \cdot 3 = 2.24$ bits per symbol, compared with 3 bits per symbol for a fixed-length code, a saving of about 25%.

Theorem 6.8 (Huffman optimality) *Algorithm HuffmanCoding constructs a prefix code of minimum expected length $\sum_i p_i \ell_i$, where ℓ_i is the length of the codeword for symbol i.*

Proof: By induction on n. For $n = 2$, both symbols get a one-bit code, which is trivially optimal.

For $n > 2$, let u and v be the two symbols with the smallest probabilities.

Greedy-choice property. In any optimal tree, there exist two sibling leaves at the deepest level (since the tree is full). Because u and v have the smallest probabilities, we can swap them into those deepest positions without increasing cost — moving a less probable symbol deeper and a more probable symbol shallower can only decrease or maintain the expected length. So there is an optimal tree in which u and v are siblings at the deepest level.

Optimal-substructure property. In such a tree, replace u and v by a single leaf z with probability $p_u + p_v$. This reduces the problem to $n - 1$ symbols, and the expected code length changes

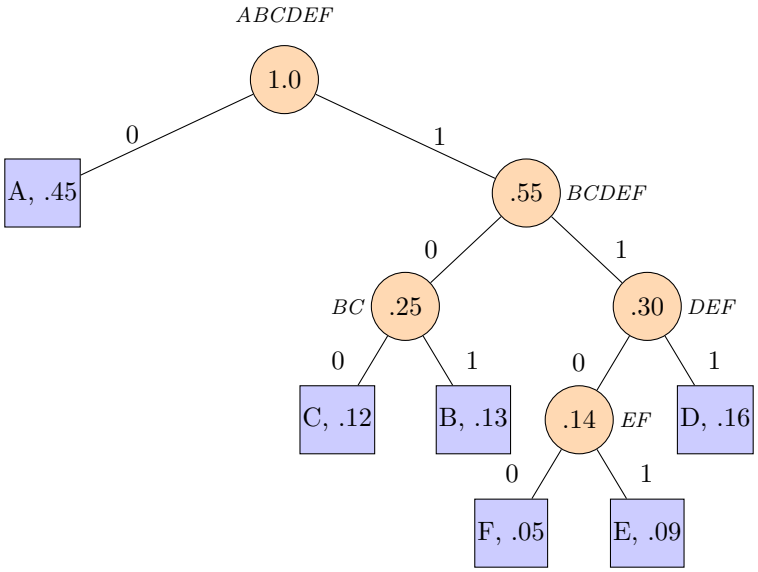


Figure 6.1: Huffman tree for symbols A, B, C, D, E, F with probabilities $0.45, 0.13, 0.12, 0.16, 0.09, 0.05$. Leaves (squares) carry the source symbols; internal nodes (circles) carry the summed probabilities created by merges, labeled with the set of symbols in the subtree below them.

by exactly $p_u + p_v$ — a constant that does not depend on the tree structure. So an optimal tree for the reduced problem yields an optimal tree for the original problem when we expand z back into u and v.

By the inductive hypothesis, Huffman's algorithm produces an optimal tree for the $n-1$ symbol problem. Expanding z back gives an optimal tree for all n symbols.

■

6.6 Fractional Knapsack

The knapsack problem is a natural test case for the reach of greedy methods. We are given n items with values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$, and a knapsack capacity W. In the 0/1 *knapsack problem*, we must choose a subset $S \subseteq [n]$ with $\sum_{i \in S} w_i \leq W$ that maximizes $\sum_{i \in S} v_i$; here no fixed greedy rule is optimal. For capacity $W = 10$ and items $\{(v{=}7, w{=}6), (v{=}5, w{=}5), (v{=}5, w{=}5)\}$, the rule "take the largest value first" and the rule "take the largest value-to-weight ratio first" both pick item 1 alone, for value 7, while items 2 and 3 together give 10. The 0/1 knapsack problem can be solved using dynamic programming (Chapter 10) and is NP-hard in general.

In the *fractional knapsack problem*, items are divisible: we may take a fraction $x_i \in [0, 1]$ of each item, paying weight $x_i w_i$ and earning value $x_i v_i$. The capacity constraint becomes $\sum_i x_i w_i \leq W$. Divisibility restores the power of greed.

Since items are divisible, we should fill the capacity with the highest value-to-weight items first: sort items by v_i/w_i and take as much as possible of each item in that order before moving to the next.

Algorithm FractionalKnapsack: Greedy algorithm for the fractional knapsack problem

input: v: array[1..n] of real; w: array[1..n] of real (presorted once so that $v[i]/w[i]$ is non-increasing); W: real capacity
output: x: array[1..n] of real initially 0 `// fractional selections`

```
1  real rem := W;
2  for i := 1 to n do
3  |  if w[i] ≤ rem then
4  |  |  x[i] := 1;
5  |  |  rem := rem − w[i];
6  |  else
7  |  |  x[i] := rem/w[i];                    // take the remaining fraction
8  |  |  break;                                // knapsack is full
9  |  end
10 end
```

Theorem 6.9 (Fractional Knapsack optimality) *Algorithm FractionalKnapsack returns a selection of maximum total value for the fractional knapsack problem.*

Proof: Assume items are indexed so that $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$. The greedy algorithm fills items in this order, taking as much of each item as the remaining capacity allows.

Let O be any optimal solution. If O differs from the greedy solution, let j be the first item where they disagree — the greedy solution takes more of item j than O does. Since both solutions use the same total capacity, O must compensate by taking more of some later item $j' > j$. We can transfer weight from item j' to item j in O without losing value, because $v_j/w_j \geq v_{j'}/w_{j'}$. Repeating this process eliminates all disagreements without reducing the total value, transforming O into the greedy solution. Hence the greedy solution is optimal.

■

The sort dominates the running time: $O(n \log n)$, with $O(n)$ additional work for the loop.

Example 6.10 Consider four items and a knapsack of capacity $W = 10$:

Item i	Value v_i	Weight w_i	Ratio v_i/w_i
1	10	2	5
2	12	3	4
3	6	2	3
4	8	4	2

The items are already sorted by non-increasing value-per-weight ratio. Tracing the algorithm:

i	w_i	rem (before)	x_i	Value added
1	2	10	1	10
2	3	8	1	12
3	2	5	1	6
4	4	3	3/4	6

The greedy algorithm takes items 1, 2, and 3 in full and three-quarters of item 4. The total weight used is $2 + 3 + 2 + 3 = 10 = W$, and the total value is $10 + 12 + 6 + 6 = 34$.

For contrast, refusing to break item 4 and taking items $1, 2, 3$ alone gives value 28; taking item 4 whole instead of item 3 yields weight $2+3+4 = 9$ and value $10+12+8 = 30$. Both alternatives are worse, confirming the optimality of the greedy fractional selection on this instance.

6.7 LLP Perspective

In this section, we provide LLP versions of these greedy algorithms.

LLP: Interval Scheduling Algorithm

In this subsection, $G[j]$ is a Boolean indicating whether interval j is selected. We are given n jobs sorted by their finish times. Jobs with the same finish times are sorted according to their start times. We are interested in finding a maximum-sized subset of $[n]$ such that each pair of jobs is mutually compatible. We define B on a subset of jobs to be true if the subset is a maximum-sized subset of mutually compatible jobs. For example, consider the set of jobs as $\{[1, 3), [2, 4), [5, 8), [6, 9)\}$. It can be verified that there is no subset of size three that has mutually compatible jobs. However, there are multiple subsets of size two that are mutually compatible. For example, $\{[1, 3), [5, 8)\}$ is one such subset. Another example is $\{[2, 4), [5, 8)\}$. The predicate B as defined is not lattice-linear in the lattice of all subsets. It is not closed under the meet operation. The meet of the two examples is the subset $\{[5, 8)\}$, which is not a maximum-sized subset that has mutually compatible jobs. We redefine B to be the lexicographically least maximum-sized subset of mutually compatible jobs. Now, simply by the definition, B is true exactly on one subset. Hence, B is lattice-linear: for any state G that does not satisfy B, there exists an index j such that every state above G that satisfies B has

a different value of $G[j]$. Because the unique satisfying state G^* differs from G in at least one coordinate j, that coordinate is a forbidden index — advancing j toward $G^*[j]$ makes progress and cannot overshoot the unique target.

We determine the lexicographically least maximum-sized subset of mutually compatible jobs as follows. We maintain a linked list of all the jobs under consideration. These jobs are in the order of their finish times. The algorithm will maintain a Boolean $G[j]$ for each job j. The variable $G[j]$ is initially false and is set to true only if it is guaranteed to be in the maximum-sized set of jobs that the algorithm chooses. We call a job j *fixed* if $G[j]$ is true. The *init* command always fixes the first job. We now make the following observations.

1. If any job j has its start time greater than or equal to the finish time of the previous job, then that job is always included in the final set. Thus, the job j can be fixed.

2. For any job, if the previous job in the linked list is fixed and the start time of the current job is less than the finish time of the previous job, then this job can be deleted.

It follows from the above two rules that eventually all jobs will be fixed or deleted. The algorithm terminates when every job remaining in the linked list has been fixed. The set of all fixed jobs is a maximum-sized set of jobs that can be completed. Furthermore, it is the lexicographically least such set.

The LLP algorithm for the activity selection problem is shown in Algorithm LLP-IntervalScheduling.

Algorithm LLP-IntervalScheduling: Selecting a maximum number of compatible intervals via LLP

```
// Assume that all jobs are in a doubly linked list
// prev and next denote the previous and the next node in the linked
   list
input: s: array[1..n] of int // start times
f: array[1..n] of int // finish times
output: G: array[1..n] of boolean initially G[1] = true and G[j] = false for j ≠ 1
1 forbidden(j ∈ [1..n]): ¬G[j] ∧ f[prev[j]] ≤ s[j]
2   ⇒ advance: G[j] := true;
&&
3 forbidden(j ∈ [1..n]): (j > 1) ∧ (next[prev[j]] = j) ∧ G[prev[j]] ∧ s[j] < f[prev[j]]
4   ⇒ advance: next[prev[j]] := next[j]; prev[next[j]] := prev[j] ;     // delete from
  linked list
```

For example, consider again the five jobs of Example 6.2, indexed in order of finish time:

$$job_1 = [1, 3), \quad job_2 = [2, 5), \quad job_3 = [4, 6), \quad job_4 = [6, 8), \quad job_5 = [5, 9).$$

The first job gets fixed because the first job is always fixed. The fourth job also gets fixed because its starting point 6 is at least as large as the finishing time 6 of the previous job. Since the first job is fixed and the second job starts at 2, before the first job is finished at 3, the second job is deleted from the list. The fifth job is deleted for the same reason: the fourth job is fixed and the fifth starts at 5, before the fourth finishes at 8. Deleting the second job makes the first job the predecessor of the third, and the third job is now fixed as well, because it starts at 4 and the first job finishes at 3. The fixed jobs are job_1, job_3, and job_4, which is the schedule that the classical algorithm of Section 6.2 selects for the same instance. Each job in the linked list is either fixed by the first advance rule or deleted by the second, and both actions are $O(1)$ because the list pointers let us reach *prev* directly. Every job is touched at most twice (once to be fixed, possibly once to be deleted), so the main loop runs in $O(n)$ time once the jobs are sorted by finish time; the overall cost is therefore $O(n \log n)$ including the sort, with $O(n)$ space for the list.

LLP: Minimizing Maximum Lateness of Jobs

In this subsection, $G[j]$ is the start time of job j. We now consider the LLP version of computing a job schedule that minimizes the maximum lateness of jobs. The greedy approach is to schedule jobs in order of increasing deadline. For simplicity, assume that all deadlines are distinct and that the jobs are indexed so that $d[1] < d[2] < \cdots < d[n]$.

Let $G[j]$ denote the starting time of job j, initially 0. The lattice is $\mathbb{N}^n$ with componentwise order, and the bottom element is the all-zeros vector. The target vector G^* is characterized by

$$B \equiv \forall j : G[j] = \sum_{i<j} t[i].$$

Because G^* is uniquely determined, B is lattice-linear on its own.

We state the constraint directly as

$$\textbf{ensure}(j) \equiv G[j] \geq \sum_{i<j} t[i],$$

which is shorthand for the forbidden condition $G[j] < \sum_{i<j} t[i]$ together with an advance that raises $G[j]$ to that target value. Initially every $j \geq 2$ is forbidden (since the right-hand side is positive while $G[j] = 0$), so all $n - 1$ indices can advance.

Algorithm LLP-MinMaxLateness: Minimizing maximum lateness via LLP

input: t: array[1..n] of int `// processing times`
d: array[1..n] of int `// deadlines; jobs indexed in non-decreasing deadline order`
output: G: array[1..n] of int initially 0 `// starting time of each job`
lateness: int initially 0

1 **ensure**($j \in [1..n]$): $G[j] \geq \sum_{i<j} t[i]$;

2 $lateness := \max_j \max(0,\ G[j] + t[j] - d[j])$;

Example 6.11 Continuing Example 6.6, we trace MinMaxLateLLP on the same four jobs with processing times $t = (3, 2, 1, 4)$ and deadlines $d = (4, 6, 5, 7)$. Reindex them in deadline order: $j_1 \rightsquigarrow 1$, $j_3 \rightsquigarrow 2$, $j_2 \rightsquigarrow 3$, $j_4 \rightsquigarrow 4$, so the deadline-sorted processing times are $(3, 1, 2, 4)$ and deadlines are $(4, 5, 6, 7)$. The targets $\sum_{i<j} t[i]$ are $(0, 3, 4, 6)$. Initially $G = (0, 0, 0, 0)$ and indices $2, 3, 4$ are all forbidden. A single round advances $G[2] := 3$, $G[3] := 4$, and $G[4] := 6$; after that no index is forbidden. The final *lateness* reduction reads $G + t - d = (-1, -1, 0, 3)$, giving *lateness* $= 3$, attained by j_4.

round	forbidden set	advances applied	G after
0	$\{2, 3, 4\}$	—	$(0, 0, 0, 0)$
1	$\emptyset$	$G[2] := 3$, $G[3] := 4$, $G[4] := 6$	$(0, 3, 4, 6)$

This matches the schedule drawn in Example 6.6: the jobs start at times $0, 3, 4, 6$ in deadline order, and the maximum lateness is 3 units.

Computing each target $\sum_{i<j} t[i]$ takes $O(n)$ via a left-to-right scan, and the lateness reduction is another $O(n)$, so the total time is $O(n)$ once the deadlines are sorted and $O(n \log n)$ otherwise. The arrays G and the scalar *lateness* use $O(n)$ space.

LLP: Fractional Knapsack

In this subsection, $G[j] \in [0, 1]$ is the fraction of item j taken. We now give an LLP version of the fractional knapsack algorithm. Assume the items are indexed in non-increasing order of v_j/w_j. Let $G[j] \in [0, 1]$ denote the fraction of item j included in the knapsack, initially 0. The lattice is $[0, 1]^n$ with componentwise order and bottom the all-zeros vector.

Let $S_j = \sum_{k \le j} w[k]$ denote the cumulative weight up through item j, and define the target fraction

$$G^*[j] \;=\; \begin{cases} 1 & \text{if } S_j \le W, \\ (W - S_{j-1})/w[j] & \text{if } S_{j-1} < W < S_j, \\ 0 & \text{if } S_{j-1} \ge W. \end{cases}$$

The predicate $B \equiv (\forall j : G[j] = G^*[j])$ is satisfied by a unique vector G^* and is therefore lattice-linear on its own. We take the forbidden condition to be the direct value comparison

$$\text{forbidden}(j) \equiv G[j] < G^*[j],$$

and the advance step raises $G[j]$ to $G^*[j]$. Initially every j with $G^*[j] > 0$ is forbidden, so all such indices can advance.

Computing the prefix sums S_j takes $O(n)$ time once the items are sorted, and each target $G^*[j]$ is obtained in $O(1)$ from S_{j-1}, S_j. The total running time is therefore $O(n \log n)$ (dominated by the sort), with $O(n)$ space.

6.8 When Greedy Fails

The problems we have studied share a feature that makes the greedy approach succeed: each one has an *exchange property* strong enough to carry the correctness proof. Once we identify

Algorithm LLP-FractionalKnapsack: Fractional knapsack via LLP

input: v: array[1..n] of real; w: array[1..n] of real (sorted so that $v[j]/w[j]$ is non-increasing); W: real capacity
output: G: array[1..n] of real in $[0, 1]$, initially 0

// Let $S_j = \sum_{k \le j} w[k]$ and $G^*[j]$ be the target value defined above.
1 **forbidden**($j \in [1..n]$): $G[j] < G^*[j]$
2 $\Rightarrow$ **advance**: $G[j] := G^*[j]$;

the "right" local rule (earliest finish time, earliest available room, earliest deadline, merge the two least probable symbols, highest value-per-weight), any solution that deviates from it can be transformed into the greedy solution without losing value — that is the content of the exchange argument.

Many closely related problems lack this property, and for them greedy fails. Replacing divisibility of items with indivisibility turns Fractional Knapsack into the 0/1 Knapsack problem, which is NP-hard and has no known greedy rule that guarantees an optimum. The counterexample from the chapter introduction makes the failure concrete.

Item i	Value v_i	Weight w_i	Density v_i/w_i
1	7	6	1.17
2	5	5	1.00
3	5	5	1.00

With capacity $W = 10$, both of the natural greedy rules fail, and they fail in the same way. The highest-value-first rule picks item 1, worth 7, and the highest-density-first rule picks item 1 as well, since its density 1.17 is the largest. Either way the remaining capacity is $10 - 6 = 4$, into which neither of the two remaining items fits, so greedy stops with value 7. The optimal solution instead leaves item 1 on the shelf and takes items 2 and 3 together (weight $5 + 5 = 10$, value $5 + 5 = 10$).

Replacing compatibility with a weight on activities similarly turns Interval Scheduling into Weighted Interval Scheduling, for which no fixed greedy rule is optimal and dynamic programming is required. Making the tasks non-preemptive on multiple machines changes the scheduling landscape in a way that local-choice heuristics can only approximate.

The cautionary tale, then, is that greedy is a consequence of structure, not a method of last resort: when the problem admits a sound exchange argument, greedy produces an exact optimum in low polynomial time; when it does not, even the most intuitive local rule can be arbitrarily far from optimal, and a richer technique such as dynamic programming (Chapter 10) or approximation (Chapter 14) is needed.

6.9 Summary

Interval Scheduling selects activities in order of earliest finish time, maximizing the number of non-overlapping intervals chosen. For *Interval Partitioning*, assigning each interval to the earliest available room uses the minimum number of rooms. Ordering jobs by earliest deadline, *Minimizing Maximum Lateness* minimizes the worst-case lateness. *Huffman Tree Construction* repeatedly merges the two least probable symbols, giving more probable symbols shorter codewords and the tree minimum expected length. *Fractional Knapsack* takes items in order of value-to-weight ratio, which maximizes total value for a given capacity.

Table 6.1 summarizes all five problems considered in this chapter. In the table, n denotes the number of intervals (for interval scheduling and interval partitioning), the number of jobs (for minimizing lateness), the number of symbols (for Huffman coding), or the number of items (for fractional knapsack). The time complexity includes the time to sort the input as required by the strategy.

Problem	**Strategy**	**Time Complexity**
Interval Scheduling	Earliest finish time	$O(n \log n)$
Interval Partitioning	Earliest available room	$O(n \log n)$
Minimizing Lateness	Earliest deadline	$O(n \log n)$
Huffman Tree	Merge smallest probabilities	$O(n \log n)$
Fractional Knapsack	Highest value/weight ratio	$O(n \log n)$

Table 6.1: Summary of greedy algorithms.

6.10 Problems

6.1. **(Train Maintenance Scheduling)** A railway station needs to schedule maintenance for the incoming trains before their next departure. Each train has a maintenance duration and a deadline. Find the optimal order of maintenance to minimize the worst delay.

6.2. **(k-Overlap Interval Selection)** Given n intervals and an integer k, find the maximum subset of intervals such that at most k intervals overlap at any time.

6.3. **(Gap Constrained Room Scheduling)** Each task has a required start time and finish time, but there must be a mandatory idle gap of at least g units between consecutive tasks in the same room. Find the minimum number of rooms required.

6.4. **(Typed Machine Scheduling)** Each task has a start and finish time, but also requires a *specific type of machine*. Different machine types may be limited in number. Find the minimum number of machines needed.

6.5. **(Huffman Code Optimality)** Prove that the Huffman coding algorithm produces a prefix code of minimum expected length. State the key greedy-choice and optimal-substructure properties used in the proof.

6.6. **(Interval Point Cover)** Given n closed intervals on the real line, find the minimum number of points required so that every interval contains at least one chosen point. Give a greedy algorithm, prove its correctness, and state its time complexity.

6.7. **(Greedy Coin Change)** Given coin denominations $\{1, 5, 10, 25\}$ and an amount A, give a greedy algorithm that returns the minimum number of coins summing to A. Then exhibit a denomination set on which the same "largest first" greedy rule is *not* optimal.

6.8. **(Profitable Job Sequencing)** Given n jobs, each requiring one unit of processing time, with profit $p_i \geq 0$ and deadline $d_i \geq 1$, schedule a subset on a single machine (one job per integer time slot $1, 2, 3, \ldots$) so that every scheduled job finishes by its deadline. Maximize the total profit. Give a greedy algorithm and state its time complexity.

6.9. **(Optimal Merge Pattern)** Given n sorted files of sizes $s_1, s_2, \ldots, s_n$, we wish to merge them pairwise (using two-way merges) into one sorted file. The cost of merging two files of sizes a, b is $a + b$. Find a merge sequence that minimizes the total merge cost. Give a greedy algorithm and explain its connection to Huffman coding.

6.10. **(Activity Greedy Counterexamples)** For the standard activity selection problem (Section 6.2), the rule "earliest finish time" is optimal. Show by counterexample that each of the following rules can produce a strictly suboptimal schedule: (a) earliest start time first; (b) shortest duration first.

6.11. **(Bounded Fractional Knapsack)** For each item j we are given lower and upper bounds $0 \leq \alpha_j \leq \beta_j \leq 1$ on the fraction taken, in addition to the usual values v_j, weights w_j, and capacity W. The constrained predicate adds $\forall j : G[j] \in [\alpha_j, \beta_j]$ to the fractional-knapsack predicate. Give an LLP-style algorithm and state when the predicate is infeasible.

6.12. **(Length Limited Huffman Codes)** Given symbol probabilities $p_1, \ldots, p_n$ and a maximum codeword length L, find a prefix code that minimizes expected length subject to the constraint that no codeword exceeds L bits. Show that the constrained predicate, with $G[i]$ = depth of leaf i in the code tree, gains the conjunct $\forall i : G[i] \leq L$. Identify when the predicate is infeasible and outline an LLP-style algorithm.

6.13. **(Interval Partitioning is Not Lattice-Linear)** For the Interval Partitioning Problem, let $G[j]$ denote the room assigned to course j, and consider the lattice of room-assignment vectors $G \in \mathbb{N}^n$ under the componentwise order. Let B be the feasibility predicate "no two overlapping courses share a room":

$$B \equiv \forall i, j \in [n] : (i \neq j) \wedge (s_j < f_i) \wedge (s_i < f_j) \Rightarrow G[i] \neq G[j]$$

Show that B is *not* lattice-linear as stated, by exhibiting two vectors that satisfy B whose meet does not. Conclude that B admits no least satisfying vector, so the LLP machinery does not apply to it directly. Then show how strengthening B to single out one particular feasible assignment restores lattice-linearity, and explain why Algorithm IntervalPartition needs no such strengthening.

6.14. **(Incremental Interval Scheduling)** A maximum-compatible set G^* of intervals has been computed by INTERVAL (Algorithm Interval). A new interval (s_{n+1}, f_{n+1}) arrives. Update G^* in $O(\log n)$ time, assuming intervals are stored by finish time in a balanced BST.

6.15. **(Constrained Interval Partition with Forbidden Rooms)** The interval-partition problem (INTERVALPARTITION, Algorithm IntervalPartition) is extended: each interval j has a set $\text{forbid}_j \subseteq \{1, \ldots, m\}$ of rooms it cannot use. Express this as the conjunction [INTERVALPARTITION && FORBIDDENROOMS], give the forbidden predicate of the second operand, and analyze the running time.

6.11 Bibliographic Remarks

The Interval Scheduling Problem and Interval Partitioning Problem are covered in [CLRS09] and [KT06]. For the Minimizing Lateness problem, [LRK76] proves that scheduling jobs by earliest deadline (EDF) minimizes maximum lateness via an exchange argument, for a single machine with deadlines. The Huffman tree construction, which builds an optimal prefix code by greedily merging the least probable symbols, was introduced by [Huf52]. Our sequential description mirrors the classic priority-queue implementation in $O(n \log n)$ time, as detailed in [CLRS09]. The fractional knapsack problem and the contrast with the NP-hard 0/1 variant are treated in [CLRS09, KT06].

Chapter 7

The Shortest Path Problem

7.1 Introduction

The single source shortest path (SSSP) problem has wide applications in transportation, networking, route-planning, and many other fields. It asks, given a weighted directed graph with n vertices and m edges and a distinguished *source* vertex v_0, for $cost[x]$: the minimum total weight of any directed path from v_0 to x, where the cost of a path is the sum of edge weights along it. A related and equally fundamental problem is the *all-pairs shortest-path (APSP) problem*, which asks for the matrix of shortest-path costs between every pair of vertices.

Table 7.1 summarizes the four algorithms presented in this chapter.

Algorithm	**Problem**	**Neg. edges?**	**Time**
Dijkstra	SSSP	No	$O(m \log n)$
BellmanFord	SSSP	Yes	$O(nm)$
FloydWarshall	APSP	Yes	$O(n^3)$
Johnson	APSP	Yes	$O(nm \log n)$

Table 7.1: Overview of shortest-path algorithms in this chapter, where n is the number of vertices and m is the number of edges.

All of these algorithms admit reformulations in the *lattice-linear predicate (LLP)* framework introduced in Chapter 2. In an LLP formulation, a single generic algorithm — advance every forbidden index until the predicate B holds — unifies Dijkstra's single-fix rule, BellmanFord's edge-relaxation loop, and FloydWarshall's triple-nested update, each arising as a special case of the generic advance. The LLP reformulations expose parallelism (many forbidden indices can advance concurrently).

This chapter is organized as follows. Section 7.2 describes Dijkstra's algorithm for SSSP on graphs with non-negative edge weights. Section 7.3 describes the BellmanFord algorithm, which lifts the non-negativity restriction and also detects negative-weight cycles. Section 7.4 presents the FloydWarshall algorithm for APSP in $O(n^3)$ time via the intermediate-vertex recurrence. Section 7.5 develops Johnson's algorithm, which uses the BellmanFord result to

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

reweight edges and then applies Dijkstra from every source, yielding APSP in $O(nm \log n)$ time — faster than FloydWarshall on *sparse* graphs, those with $m = O(n)$ edges rather than the $m = \Theta(n^2)$ of a *dense* graph.

The remaining sections reformulate these algorithms in the LLP framework, presenting LLP versions of BellmanFord, Dijkstra's algorithm, FloydWarshall, and Johnson's algorithm.

7.2 Dijkstra's Algorithm

We consider a directed weighted graph $\mathcal{G} = (V, E, w)$ where V is the set of vertices, E is the set of directed edges, and w is a map from the set of edges to positive reals (see Fig. 7.1 for a running example). We write the weight of the edge from i to j as $w(i, j)$, with parentheses rather than brackets, to emphasise that it is obtained from the edge itself and not from an $n \times n$ array: the algorithms below visit each vertex's outgoing edges, so an implementation stores the weight alongside the adjacency list and never pays the $\Theta(n^2)$ a matrix would cost to build. Brackets are kept where the algorithm really does own a matrix, as for $G[i, j]$ in Algorithm FloydWarshall and the output $D[u, v]$ of Algorithm Johnson. We assume that the graph is loop-free.

Algorithm Dijkstra: Finding the shortest costs from v_0

input: Graph $\mathcal{G} = (V, E, w)$ with edge weights $w(j, k)$; source vertex v_0
output: d: array$[0..n-1]$ of integer, initially $\forall i : d[i] := \infty$

var $fixed$: array$[0 \ldots n-1]$ of boolean initially $\forall i : fixed[i] := false$
H: binary min-heap of (j, c) ordered by c, initially empty `// j is the vertex and c is the cost`

```
1  d[0] := 0;
2  H.add(0, d[0]);
3  while ¬H.empty() do
4  |  (j, c) := H.removeMin();
5  |  if ¬fixed[j] then
6  |  |  fixed[j] := true;
7  |  |  forall k: ¬fixed[k] ∧ (j, k) ∈ E :
8  |  |  |  if d[k] > d[j] + w(j, k) then
9  |  |  |  |  d[k] := d[j] + w(j, k);
10 |  |  |  |  H.add(k, d[k]);
11 |  |  |  end
12 |  |  end
13 |  end
14 end
```

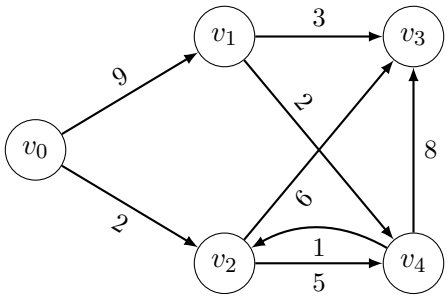


Figure 7.1: A weighted directed graph

Dijkstra's algorithm maintains $d[i]$, which is a cost to reach v_i from v_0. Each vertex x in the graph has initially $d[x]$ equal to ∞. Whenever a vertex is discovered for the first time, its $d[x]$ becomes less than ∞. We use the predicate $discovered(x) \equiv d[x] < \infty$. The variable d decreases for a vertex whenever a shorter path is found due to edge relaxation.

In addition to the variable d, a Boolean array *fixed* is maintained. Thus, every discovered vertex is either *fixed* or *non-fixed*. The invariant maintained by the algorithm is that if a vertex x is *fixed* then $d[x]$ gives the final shortest cost from vertex v_0 to x. If x is *non-fixed*, then $d[x]$ is the cost of the shortest path to x that only goes through fixed vertices.

When all edge weights are non-negative, distances can only increase as we extend a path. Thus, among all currently discovered vertices, the one with the smallest tentative distance must already have its final shortest-path value. Any alternative path to this vertex would have to go through another vertex whose distance is at least as large, and then incur an additional non-negative cost, making it no better. Therefore, we can safely fix the closest unfixed vertex at each step and propagate its distance to its neighbors, growing the set of vertices with known optimal distances in a greedy manner.

The algorithm uses a heap H that contains all vertices that have been discovered but are not fixed along with their distance estimates d. We view the heap as consisting of tuples of the form $(j, d[j])$ where the heap property is with respect to the d values. The algorithm has one main *while* loop that removes the vertex with the minimum distance from the heap with the method H.removeMin(), say v_j, and marks it as fixed. It then *explores* the vertex v_j by relaxing all its adjacent edges going to non-fixed vertices v_k. The value of $d[k]$ is updated to the minimum of $d[k]$ and $d[j] + w(j, k)$. This step is called *edge relaxation*. If v_k is not on the heap, then it is inserted; else, if $d[k]$ has decreased, then the label associated with the vertex k is inserted in the heap. We abstract this step as the method H.add$(k, d[k])$. Since the vertex may already be on the heap, the insertion may cause a vertex to be present in a heap with different distances. Hence, when we remove a vertex from the heap, if it is already fixed, we simply go to the next vertex in the heap. The algorithm terminates when the heap is empty. At this point there are no discovered non-fixed vertices and d reflects the cost of the shortest path to all discovered vertices. If a vertex j is not discovered then $d[j] = \infty$, reflecting that v_j is unreachable from v_0.

Observe that every vertex goes through the following states. Every vertex x is initially *undiscovered* (that is, $d[x] = \infty$). If x is reachable from the source vertex, then it is eventually *discovered* (i.e., $d[x] < \infty$). A discovered vertex is initially *non-fixed*, and is therefore in the

heap H. Whenever a vertex is removed from the heap it is a *fixed* vertex. A fixed vertex may either be *unexplored* or *explored*. Initially, a fixed vertex is unexplored. It is considered explored when all its outgoing edges have been relaxed.

The following lemma simply summarizes the well-known properties of Dijkstra's algorithm. We leave them as an exercise.

Lemma 7.1 *The outer loop in Dijkstra's algorithm satisfies the following invariants.*
(a) For all vertices x: $fixed[x] \Rightarrow (d[x] = cost[x])$.
(b) For all vertices x: $d[x]$ is equal to the cost of the shortest path from v_0 to x such that all vertices in the path before x are fixed.

Theorem 7.2 (Dijkstra's Algorithm correctness) *On a directed graph with non-negative edge weights, Dijkstra's algorithm terminates with $d[x] = cost[x]$ for every vertex x, where $cost[x]$ is the length of the shortest path from v_0 to x (or ∞ if x is unreachable).*

Proof: By Lemma 7.1(a), every vertex marked $fixed$ receives its correct shortest-path cost. It remains to show that every reachable vertex eventually becomes $fixed$. An extracted vertex is inserted into the heap when a relaxed edge discovers it for the first time; since the heap is not drained until it is empty and every relaxation either inserts a new vertex or replaces an old key with a smaller one, every discovered vertex is eventually extracted and fixed. When the heap empties, by Lemma 7.1(b) every unreached vertex has $d[x] = \infty$, which matches $cost[x]$.

The greedy invariant — that the minimum-key vertex on the heap is ready to be fixed — depends on non-negativity: if v has the smallest heap key d, no path through a not-yet-fixed vertex u can improve on d, because the sub-path through u already costs at least the heap key of u, which is $\geq d$, and all remaining edge weights are non-negative.

■

For complexity analysis, let n be the number of vertices and m be the number of edges. We analyze the version of Dijkstra's algorithm in which, instead of adjusting the key in the heap for a vertex, we simply insert the vertex in the heap. As a result, the heap may have a vertex multiple times with different keys. When a vertex is removed, we check if it has already been fixed. If it is fixed, then we do not need to explore it, and we can remove the next vertex in the heap. Since a vertex can be inserted in the heap only when an edge is relaxed, we find that there are at most m insertions in the heap. We can also conclude that there are at most m deletions from the heap. Since an insertion or deletion of a heap takes $O(\log m) = O(\log n)$ time, we get the overall time complexity of $O(m \log n)$.

Example 7.3 Here is a walk-through of Dijkstra's algorithm on the graph of Figure 7.1, starting from the vertex v_0.

1. Set v_0 as the source vertex and assign it a distance of 0. Assign all other vertices a tentative distance of infinity. We insert the source vertex v_0 in the heap H.

2. Since the heap is not empty, we remove the vertex at the top of the heap. It is v_0 with a distance of 0. We mark that vertex as *fixed*. We examine its neighbors v_1 and v_2. For $v_0 \to v_1$, the existing distance to v_1 is infinity. The distance 0 (distance to v_0) + 9 (edge weight from v_0 to v_1) is less than infinity, so we update the distance to v_1 to 9 and add it to the heap. For $v_0 \to v_2$, the existing distance to v_2 is infinity. The distance 0 + 2 is less than infinity, so update the distance to v_2 to 2 and add v_2 to the heap.

3. We remove the vertex at the top of the heap. The smallest distance among the non-fixed vertices is 2 (for v_2), so set v_2 as the current vertex. We examine its neighbors, v_3 and v_4. For $v_2 \to v_3$, the existing distance to v_3 is infinity. Since 2 (distance to v_2) + 6 (edge weight from v_2 to v_3) is less than infinity, we update the distance to v_3 to 8. For $v_2 \to v_4$, the existing distance to v_4 is infinity. Since $2 + 5$ is less than infinity, we update the distance to v_4 to 7.

4. Continuing in this manner until the heap becomes empty, we get the final distances as: v_0: 0, v_1: 9, v_2: 2, v_3: 8, v_4: 7.

7.3 BellmanFord Algorithm

In this section, we give an algorithm that computes the shortest paths from any given vertex even when the edge weights are negative. Why could we not apply Dijkstra's algorithm for this problem? Consider the case where the node that can be reached from the source node on one edge with a minimum cost w has a cost of x. Let this node be v. Dijkstra's algorithm assumes that this is the shortest path to node v, because any other path would already have incurred the cost of w by going to some other vertex. However, in the presence of negative weights, there may be a longer path to v by incurring more cost initially but then traversing an edge with negative cost.

When there are negative cost edges, we have to be careful that there are no negative cost cycles. In the presence of negative cost cycles, the cost of going from s to some vertex v may be decreased in an unbounded manner by going through the cycle. For now, we will assume that the graph may have negative cost edges but no negative cost cycles.

In the 1950s, Bellman and (independently) Ford gave an algorithm for graphs with negative weights. We let $d^k[v]$ be the cost of reaching v from the source vertex s with a path of at most k edges. It is clear that the base case of k equal to 0 is quite simple. The value $d^0[v]$ equals 0 when v equals s and ∞ otherwise. When k equals $n-1$, we claim that $d^k[v]$ is exactly equal to the shortest path from s to v. Why could there not be a path with n edges or more, with a lower cost? If there is a path with n edges, then at least some vertex w appears more than once. The path from w to itself forms a cycle. From our assumption, there are no negative

cost cycles. If the cycle is not negative, then we get a smaller path with equal or lower cost by removing the cycle.

The recurrence relation is as follows: $d^0[v] = 0$ if v equals s and ∞, otherwise.

$$d^k[v] = \min\{d^{k-1}[v], \min_{(u,v)\in E} d^{k-1}[u] + w(u,v)\}$$

We let $p[v]$ denote the predecessor of node v in the shortest path. Initially, $p[v]$ is null for all vertices.

Algorithm BellmanFord shows the procedure used when the edge weights in a directed graph may be negative.

Algorithm BellmanFord: Finding the shortest costs from v_0 when edge weights may be negative

input: Graph $\mathcal{G} = (V, E, w)$, source vertex s, edge weights $w(u, v)$
output: d: array[0..$n-1$] of integer `// shortest-path costs from` s
p: array[0..$n-1$] of integer `// predecessors`

```
1  forall vertex v ∈ V :
2  |  d[v] := ∞;
3  |  p[v] := null;
4  end
5  d[s] := 0;
6  for k = 1 to n − 1 do
7  |  forall edge (u, v) ∈ E :
8  |  |  if d[u] + w(u, v) < d[v] then
9  |  |  |  d[v] := d[u] + w(u, v);
10 |  |  |  p[v] := u;
11 |  |  end
12 |  end
13 end
14 forall edge (u, v) ∈ E :
15 |  if d[u] + w(u, v) < d[v] then return "Negative cycle detected" ;
16 end
17 return d[], p[];
```

The first *for* loop initializes the value of d and p for the base case corresponding to k equal to zero. We now compute d^k for k equal to $1 \ldots n-1$. Instead of computing the recurrence explicitly for each vertex, we *relax* each edge to check if $d^k[v]$ can be reduced by some edge (u, v). Finally, we check if there is a negative cost cycle in the graph. If there is a negative cost cycle that can be reached from the source vertex, then there would be an edge (u, v) in the graph such that $d[v]$ can be reduced further by using that edge even after k has reached the value $n-1$.

The time complexity of BellmanFord is $O(nm)$, making it less efficient than Dijkstra's

algorithm for graphs with non-negative weights but more versatile due to its handling of negative weights. The space complexity is $O(n)$ for the arrays d and p.

In our algorithm, we have iterated k from 1 to $n-1$. What if the values of d do not change in some iteration of k? The reader is invited to change the algorithm so that it checks for this condition and terminates sooner.

Theorem 7.4 (BellmanFord correctness) *Let $\mathcal{G} = (V, E, w)$ be a directed graph with source s, containing no negative-weight cycle reachable from s. After the main loop of Algorithm BellmanFord completes, $d[v]$ equals the cost of a shortest path from s to v for every $v \in V$. The final scan correctly reports a negative-weight cycle if and only if one is reachable from s.*

Proof: By induction on k. Let $d^k[v]$ denote the cost of a shortest path from s to v with at most k edges (or ∞ if no such path exists). The initialization gives $d[v] = d^0[v]$. Assume that after $k-1$ iterations, $d[v] \leq d^{k-1}[v]$ for all v. Relaxing every edge once in iteration k ensures that for any edge (u, v), $d[v] \leq d[u] + w(u, v) \leq d^{k-1}[u] + w(u, v)$, which is at most $d^k[v]$ taken over all such u. Hence $d[v] \leq d^k[v]$ after iteration k.

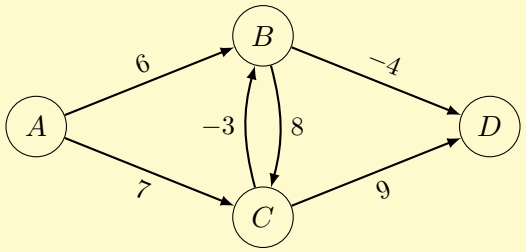
Conversely, $d[v]$ is never smaller than the true shortest-path cost $cost[v]$, because $d[v]$ is only updated via $d[v] := d[u] + w(u, v)$, so $d[v]$ always equals the weight of some path from s to v (or ∞), and no path can be cheaper than the shortest one.

Combining: after $n-1$ iterations, $cost[v] \leq d[v] \leq d^{n-1}[v] = cost[v]$, since any shortest path without a negative cycle has at most $n-1$ edges.

For the negative-cycle test: if the algorithm can still relax some edge after $n-1$ iterations, then there is a path from s to some vertex with cost less than d^{n-1}, which is only possible if a negative-weight cycle is reachable from s. Conversely, in the presence of such a cycle, edges along it can always be relaxed.

■

Example 7.5 Consider the directed graph below on vertices A, B, C, D with source A. The cycle $B \to C \to B$ has weight $8 + (-3) = 5$, and there is no other directed cycle, so the graph has no negative-weight cycle.



Starting from $d[A] = 0$ and $d[B] = d[C] = d[D] = \infty$, the three iterations of the main loop produce:

After iteration	$d[A]$	$d[B]$	$d[C]$	$d[D]$
0 (init)	0	∞	∞	∞
1	0	4	7	2
2	0	4	7	0
3	0	4	7	0

The values stabilize at iteration 2; the subsequent iteration leaves d unchanged, and the negative-cycle check passes. The shortest-path costs from A are therefore $d[A] = 0$, $d[B] = 4$, $d[C] = 7$, $d[D] = 0$.

7.4 FloydWarshall Algorithm

Suppose that $A[i, j]$ represents the weight of the direct edge from i to j denoting the cost of going from i to j using at most one edge. We assume that all diagonal entries are zero and that there is no negative weight cycle. Our goal is to find a matrix $G[i, j]$ such that $G[i, j]$ is the least cost to go from i to j using any number of edges.

Rather than focusing on paths from a single source, we build shortest paths between all pairs of vertices by progressively allowing more intermediate vertices. At stage k, we assume that only vertices $\{1, \ldots, k\}$ may appear as intermediates. For each pair (i, j), the shortest path either avoids vertex k entirely or passes through it, in which case it splits into two shorter subpaths $(i \to k)$ and $(k \to j)$. By systematically considering each vertex as a possible intermediate, we explore all possible path decompositions and obtain the shortest distances between all pairs. We define $d^k[i, j]$ as the cost of going from vertex i to vertex j using intermediate vertices from $1..k$. When k is zero, it is clear that we cannot use any intermediate vertex. Thus,

$$d^0[i, j] = \begin{cases} 0 & \text{if } i = j, \\ w(i, j) & \text{otherwise.} \end{cases}$$

Observe that if there is no edge from i to j when $i \neq j$, $w(i, j)$ is ∞. Now, consider the

case when $k \geq 1$. Then, we get the following recurrence:

$$d^k[i,j] = \min\{d^{k-1}[i,j], d^{k-1}[i,k] + d^{k-1}[k,j]\}$$

Thus, we get the FloydWarshall algorithm (Algorithm FloydWarshall).

Algorithm FloydWarshall: All pairs shortest path algorithm

input: A weighted graph represented by an $n \times n$ cost matrix $G[i,j]$, where $G[i,j]$ is the cost from vertex i to j (possibly ∞ if no edge exists).
output: Updated matrix $G[i,j]$ containing shortest path costs between all pairs of vertices.

```
1 for k = 1 to n do
2 |  for i = 1 to n do
3 |  |  for j = 1 to n do
4 |  |  |  if G[i,k] + G[k,j] < G[i,j] then
5 |  |  |  |  G[i,j] := G[i,k] + G[k,j];
6 |  |  |  end
7 |  |  end
8 |  end
9 end
```

The reader is invited to make two changes to the algorithm. First, modify the algorithm to return an error message when there is a negative cost cycle in the graph. Second, add a variable $p[u][v]$ to the algorithm to return the predecessor of v on the path from u to v.

The time complexity of the FloydWarshall algorithm is $O(n^3)$: the three nested loops iterate n times each and the body is $O(1)$. The space complexity is $O(n^2)$ for the matrix G, with updates done in place.

Theorem 7.6 (FloydWarshall correctness) *On a directed graph without negative-weight cycles, Algorithm FloydWarshall terminates with $G[i,j]$ equal to the cost of a shortest path from vertex i to vertex j, for every pair (i,j).*

Proof: Let $d^k[i,j]$ denote the cost of a shortest path from i to j whose internal vertices all lie in $\{1,2,\ldots,k\}$, or ∞ if no such path exists. By induction on k: the base case $k = 0$ matches the input matrix (only direct edges). For the inductive step, any shortest (i,j)-path using internal vertices from $\{1,\ldots,k\}$ either avoids k — in which case its cost is $d^{k-1}[i,j]$ — or uses k, in which case its two halves are shortest paths from i to k and from k to j using internal vertices from $\{1,\ldots,k-1\}$, of cost $d^{k-1}[i,k] + d^{k-1}[k,j]$. The recurrence taking the minimum of the two is exactly what the algorithm computes in iteration k. After n iterations, $G[i,j] = d^n[i,j]$, which is the unrestricted shortest-path cost since all vertices are now permissible as internal.

■

Example 7.7 Consider a graph on vertices $\{1, 2, 3\}$ with direct-edge costs given by

$$G^{(0)} = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix}.$$

After iteration $k = 1$ (allowing vertex 1 as an intermediate), only the $(3, 2)$ entry could potentially change: $G[3, 2] = \min(\infty, G[3, 1]+G[1, 2]) = \min(\infty, 3+4) = 7$. After iteration $k = 2$, the $(1, 3)$ entry becomes $\min(11, 4 + 2) = 6$ and the $(3, 3)$ entry is checked to be 0. Iteration $k = 3$ updates $(2, 1)$ via $(2, 3, 1)$ to $\min(6, 2 + 3) = 5$. The final all-pairs shortest-path matrix is

$$G = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}.$$

7.5 Johnson's Algorithm

Johnson's algorithm solves all-pairs shortest paths on directed graphs whose edge weights may be negative but contain no negative-weight cycle. It combines BellmanFord and Dijkstra: a single BellmanFord run computes a non-negative *price* vector with which the edges are reweighted, and Dijkstra is then run from every source on the reweighted graph. On sparse graphs this beats the $O(n^3)$ FloydWarshall bound: using Dijkstra yields $O(nm \log n)$ total time.

Negative edge weights prevent the direct use of Dijkstra's algorithm, but we can transform the graph so that all edge weights become non-negative while preserving shortest paths. This is done by assigning a potential (or price) to each vertex and adjusting edge weights accordingly. The reweighting shifts all path costs by a constant depending only on endpoints, so relative ordering of paths is unchanged. Once all edges are non-negative, we can efficiently run Dijkstra from each source. Thus, Johnson's algorithm combines the flexibility of BellmanFord (to handle negative edges) with the efficiency of Dijkstra (for fast shortest-path computation).

Let $\mathcal{G}$ be the input graph. The *reweighted* graph $\mathcal{G}'$ has the same vertices and edges, with edge weights

$$w'(i, j) = w(i, j) + price[j] - price[i], \tag{7.1}$$

where *price* is a non-negative vector such that every $w'(i, j) \geq 0$. This reweighting preserves shortest paths:

Lemma 7.8 *Let s and t be any two vertices in the graph. The weight of any path on the graph $\mathcal{G}'$ from s to t equals the weight in the graph $\mathcal{G}$ plus $(price[t] - price[s])$.*

Proof: Let the path be $p = v_0, v_1, v_2, \ldots, v_k$ where $v_0 = s$ and $v_k = t$. Summing (7.1) along

the edges of p,

$$\begin{aligned} w'(p) &= \sum_{i=0}^{k-1} w'(v_i, v_{i+1}) = \sum_{i=0}^{k-1} \big(w(v_i, v_{i+1}) + \mathit{price}[v_{i+1}] - \mathit{price}[v_i]\big) \\ &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \;+\; \big(\mathit{price}[v_k] - \mathit{price}[v_0]\big), \end{aligned}$$

because the price terms telescope: every intermediate $\mathit{price}[v_i]$ with $0 < i < k$ is added once and subtracted once, leaving only $-\mathit{price}[v_0]$ and $+\mathit{price}[v_k]$. Hence $w'(p) = w(p) + (\mathit{price}[t] - \mathit{price}[s])$.

■

Since path costs shift by a constant $\mathit{price}[t] - \mathit{price}[s]$, shortest paths are unchanged. It remains to find a feasible *price* vector, which exists exactly when $\mathcal{G}$ has no negative-weight cycle:

Lemma 7.9 *A feasible non-negative price vector exists iff the graph $\mathcal{G}$ has no negative-weight cycle. When it does, $\mathit{price}[v] := -\mathit{dist}(s^*, v)$ is feasible, where $\mathit{dist}(s^*, v)$ is the BellmanFord distance to v from a fresh auxiliary vertex s^* joined to every $x \in V$ by a 0-weight edge.*

Proof: ($\Rightarrow$) By Lemma 7.8 with $s = t$, any cycle has the same weight in $\mathcal{G}$ and $\mathcal{G}'$, so a negative-weight cycle in $\mathcal{G}$ leaves some reweighted edge negative.

($\Leftarrow$) Augmenting $\mathcal{G}$ with s^* cannot create a new negative cycle since s^* has no incoming edges; hence $\mathit{dist}(s^*, v)$ is well-defined and ≤ 0. Set $\mathit{price}[v] := -\mathit{dist}(s^*, v) \geq 0$. The BellmanFord triangle inequality $\mathit{dist}(s^*, j) \leq \mathit{dist}(s^*, i) + w(i, j)$ rearranges to $w(i, j) + \mathit{price}[j] - \mathit{price}[i] \geq 0$, so *price* is feasible.

■

BellmanFord from s^* finds such a *price* in at most n edge-relaxation passes (matching the

longest simple path), and reports a negative cycle otherwise.

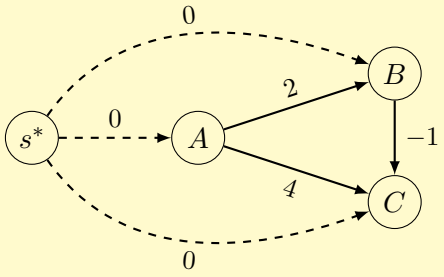
Algorithm Johnson: Johnson's algorithm for all-pairs shortest paths on graphs with negative edges

input: directed graph $\mathcal{G} = (V, E, w)$ with edge weights possibly negative
output: matrix D with $D[u, v]$ the shortest-path cost from u to v; or a report of a negative cycle

```
1  Add vertex s* and an edge (s*, v) of weight 0 for all v ∈ V;
2  Run BellmanFord from s*; if it reports a negative cycle, return "negative-weight
     cycle";
3  price[v] := −dist(s*, v) for all v ∈ V;
4  forall edges (i, j) ∈ E :
5  |  w'(i, j) := w(i, j) + price[j] − price[i];
6  end
7  forall u ∈ V :
8  |  Run Dijkstra from u on the graph with weights w' to get dist'(u, v) for all v;
9  |  forall v ∈ V :
10 |  |  D[u, v] := dist'(u, v) + price[u] − price[v];          // undo the reweighting
11 |  end
12 end
13 return D;
```

Example 7.10 Consider the directed graph on three vertices A, B, C with edges $A \to B$ of weight 2, $B \to C$ of weight -1, and $A \to C$ of weight 4 (solid edges below). It has no negative-weight cycle — indeed no directed cycle at all.

Step 1 (BellmanFord from s^*). Add an auxiliary vertex s^* with a 0-weight edge to A, B, and C (dashed); s^* has no incoming edge, so it creates no new cycle.



BellmanFord returns $dist(s^*, A) = 0$, $dist(s^*, B) = 0$, $dist(s^*, C) = -1$ (realized by $s^* \to B \to C$).

Step 2 (price vector). Set $price := -dist(s^*, \cdot)$, giving $price(A) = 0$, $price(B) = 0$, $price(C) = 1$.

Step 3 (reweighting). The reduced weights $w'(i, j) = w(i, j) + price[j] - price[i]$ are $w'(A, B) = 2 + 0 - 0 = 2$, $w'(B, C) = -1 + 1 - 0 = 0$, $w'(A, C) = 4 + 1 - 0 = 5$, all non-negative.

Step 4 (Dijkstra from every source on the reweighted graph). The reweighted distances are $dist'(A, A) = 0$, $dist'(A, B) = 2$, $dist'(A, C) = 2$ (via $A \to B \to C$), $dist'(B, B) = 0$, $dist'(B, C) = 0$, and $dist'(C, C) = 0$; entries from B or C back to A are ∞.

Step 5 (undo the shift). Recover $D[u, v] = dist'(u, v) + price[u] - price[v]$:

$$D = \begin{pmatrix} 0 & 2 & 1 \\ \infty & 0 & -1 \\ \infty & \infty & 0 \end{pmatrix},$$

where rows are indexed by u and columns by v. Verification: the $A \to C$ entry picks the two-edge path $A \to B \to C$ of cost $2 + (-1) = 1$, cheaper than the direct $A \to C$ edge of cost 4.

Theorem 7.11 (Johnson's Algorithm correctness) *On a directed graph without a negative-weight cycle, Johnson's algorithm returns $D[u, v] = dist(u, v)$ for every pair of vertices. If the graph has a negative-weight cycle, the algorithm reports it.*

Proof: The BellmanFord step correctly finds $dist(s^*, v)$ or a negative cycle; by Lemma 7.9, $price[v] := -dist(s^*, v)$ makes every reduced weight non-negative, so each Dijkstra call is valid. Lemma 7.8 gives $dist'(u, v) = dist(u, v) + price[v] - price[u]$, and the final line of the algorithm inverts this shift.

■

Complexity is dominated by the single BellmanFord call ($O(nm)$) and the n Dijkstra calls. This gives $O(nm \log n)$, beating FloydWarshall's $O(n^3)$ on sparse graphs.

7.6 LLP Perspective

In this section we collect the LLP reformulations of the algorithms introduced earlier in the chapter.

LLP-BellmanFord Algorithm

We now compute the distances of all vertices from v_0 in the LLP framework. The optimal distance vector is the largest $G[x]$, over every vertex $x \neq v_0$, that satisfies the following feasibility predicate:

$$\forall (i,j) \in E : \; G[j] \leq G[i] + w(i,j)$$

We define B_e for any edge $e = (i,j)$ as $G[j] \leq G[i] + w(i,j)$. The feasibility predicate B can be written as

$$B \equiv \wedge_{e \in E} B_e$$

We will view the search for optimal distance vector as searching for G vector in a lattice defined as follows. In this lattice $G \leq_L G'$ iff $\forall i : G[i] \geq G'[i]$. Thus, finding the least vector in this lattice corresponds to finding the largest vector G that satisfies B. The source coordinate is fixed at $G[0] = 0$; over the extended reals, the bottom element is $(0, \infty, \infty, \ldots, \infty)$ and the top element is $(0, -\infty, -\infty, \ldots, -\infty)$.

We now claim that

Theorem 7.12 *B is a lattice linear predicate.*

Proof: It is sufficient to show that B_e is lattice linear because any conjunction of linear predicates is also lattice linear. B_e for $e = (i,j)$ is equivalent to $G[j] \leq G[i] + w(i,j)$. This is equivalent to $G[j] \geq_L G[i] + w(i,j)$. Since the right hand side is a monotonic function of G, we get that B_e is lattice linear.

■

Algorithm LLP-Edge-Relaxation: Shortest-path distances by relaxing any forbidden edge

input: $pre(j)$: list of predecessors of vertex j; $w(i,j)$: int for all $i \in pre(j)$
output: G: array$[0..n-1]$ of real, initially $G[0] = 0$ and $G[i] = \infty$ for $i \neq 0$
1 **ensure**$(j \in [0..n-1])$: $G[j] \leq \min\{G[i] + w(i,j) \mid i \in pre(j)\}$;

Algorithm LLP-Edge-Relaxation gives the correct answer; however, it may result in large computation complexity if the values in G are lowered in a non-disciplined manner. We now optimize this algorithm. Algorithm LLP-BellmanFord chooses forbidden indices more carefully. The algorithm checks across all vertices whether any vertex is forbidden; if there is any forbidden vertex, then all edges are relaxed.

Algorithm LLP-BellmanFord: Shortest-path distances in Bellman-Ford round order

input: $pre(j)$: adjacency list; $w(i,j)$: edge weights
output: G: array$[0..n-1]$ of real, initially $G[0] = 0$ and $G[i] = \infty$ for $i \neq 0$

1 **forbidden**$(j \in [0..n-1])$: $\exists i \in pre(j) : G[j] > G[i] + w(i,j)$
2 $\quad \Rightarrow$ **advance**: $\forall k \neq 0 : G[k] := \min\{G[i] + w(i,k) \mid i \in pre(k)\}$;

Let us analyze the sequential time complexity of Algorithm LLP-BellmanFord. At most n rounds of edge relaxation are needed; each round can be implemented in $O(m)$ time, giving an overall sequential time complexity of $O(mn)$.

Earlier, we had assumed that the graph does not have any negative cost cycles. What if the user input is such that it has a negative cost cycle? In that case, Algorithm LLP-BellmanFord will never terminate. The standard fix, as in the classical BellmanFord algorithm of Section 7.3, is to cap the number of *while*-loop iterations at $|V| - 1$ and report "negative cost cycle" if any index is still forbidden after that.

LLP-Dijkstra Algorithm

We now cast Dijkstra's algorithm (Section 7.2) in the LLP framework. LLP-BellmanFord searches a lattice whose points are distance vectors alone. Dijkstra needs one more thing: a record of which vertices have been *settled*, never to be revisited. We therefore take the lattice points to be *pairs* $(fixed, G)$, where $fixed[0..n-1]$ is a vector of Booleans and $G[0..n-1]$ a vector of distances. The jth component of the state is the pair $(fixed[j], G[j])$, and we climb the lattice by flipping a Boolean from *false* to *true* or by *decreasing* a distance:

$$(fixed, G) \leq_L (fixed', G') \iff (\forall j : fixed[j] \Rightarrow fixed'[j]) \quad \wedge \quad (\forall j : G[j] \geq G'[j]).$$

As in LLP-BellmanFord, small distances sit *higher*. The bottom of the lattice is the initial state: nothing fixed, $G[0] = 0$, and $G[j] = \infty$ for $j \neq 0$.

The predicate has two clauses: one requires the search to be finished, the other requires the distances to be consistent:

$$B \equiv (\forall j : fixed[j] \vee G[j] = \infty) \quad \wedge \quad (\forall (i,j) \in E : G[j] \leq G[i] + w(i,j)).$$

Testing minimality directly would cost a scan of every vertex, so instead we declare the structure that holds the candidate indices:

$$\textbf{setContainer}(PriorityQueue,\ G[j],\ min).$$

The three arguments name the container, the key it orders by, and which end of that order comes first. The index the container offers is already the cheapest, so the guard in the program below only has to check that the candidate is still unsettled and has a finite distance.

Algorithm LLP-Dijkstra: Dijkstra's algorithm as a lattice-linear program

input: graph $\mathcal{G} = (V, E, w)$ with non-negative edge weights $w(i, j)$; source v_0
output: G: array$[0..n-1]$ of real, initially $G[0] = 0$ and $G[i] = \infty$ for $i \neq 0$

var $fixed$: array$[0..n-1]$ of boolean initially $\forall i : fixed[i] := false$

1 **setContainer**($PriorityQueue$, $G[j]$, min);
2 **forbidden**$(j \in [0..n-1])$: $\neg fixed[j] \wedge G[j] < \infty$
3 $\Rightarrow$ **advance**: $fixed[j] := true$;
4 **forall** $(j, k) \in E$ with $\neg fixed[k]$: $G[k] := \min(G[k],\ G[j] + w(j, k))$;

Lowering $G[k]$ in the advance makes k a fresh candidate, so it is pushed onto the container with its new key — exactly the H.add$(k, d[k])$ of Algorithm Dijkstra. Each edge causes at most one push, so the container sees $O(m)$ insertions and $O(m)$ extractions, each $O(\log n)$, and the guard itself is $O(1)$ — giving the $O(m \log n)$ bound of the heap-based Dijkstra of Section 7.2.

LLP-FloydWarshall

We now give an LLP formulation of the FloydWarshall algorithm. Suppose that $A[i, j]$ represents the weight of the direct edge from i to j, denoting the cost of going from i to j using at most one edge. We assume that all diagonal entries are zero and that there is no negative weight cycle. Our goal is to find a matrix $G[i, j]$ such that $G[i, j]$ is the least cost of going from i to j by using any number of edges.

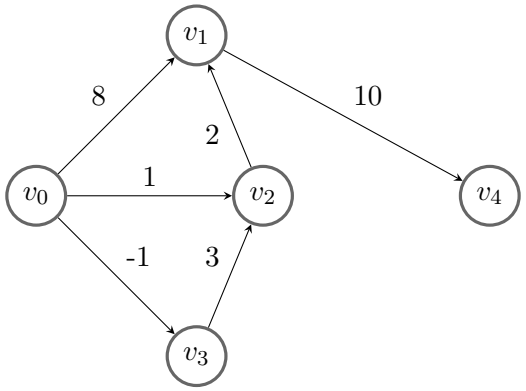


Figure 7.2: A directed graph used as an example for the LLP-FloydWarshall algorithm.

Let us formulate this problem as finding the least vector of size n^2 in a distributive lattice. We initialize G as A. This is the bottom element of our distributive lattice. Our goal is to find the least vector that satisfies the following constraint:

$$B_{FW} \equiv \forall i, j : i \neq j : G[i, j] \leq \min_k \{G[i, k] + G[k, j]\}.$$

When i equals j, $G[i, j]$ equals 0 and will stay fixed through the execution of the algorithm. The cost of going from i to j must be smaller than the cost of going from i to k and k to j. The predicate B_{FW} above quantifies over all k at once, with no fixed order in which the minimization is resolved. Realizing it this way would require repeatedly rechecking every (i, j, k) triple until none remains violated, costing an extra factor of n over the classical algorithm. Fixing an order removes that factor: process the intermediate vertex k in n sequential stages $k = 1, \ldots, n$, applying the **ensure** clause against that single k to every pair (i, j) in parallel within a stage. This is shorthand for the forbidden condition $G[i, j] > G[i, k]+G[k, j]$ together with an advance that lowers $G[i, j]$ to $G[i, k] + G[k, j]$.

Algorithm LLP-FloydWarshall: Computing the shortest cost matrix for a directed graph

input: A: matrix of real ($A[i, j]$ is the weight of the edge from i to j); no negative-weight cycle
output: G: matrix of real initially $\forall i, j : G[i, j] := A[i, j]$

```
1 for k := 1 to n do
2 |  forall (i, j) with i, j ∈ [1..n] :
3 |  |  ensure(i, j ∈ [1..n]): G[i, j] ≤ G[i, k] + G[k, j];
4 |  end
5 end
```

Each of the n stages touches every one of the n^2 pairs (i, j) exactly once, at $O(1)$ work per pair (one addition and one comparison), for a total sequential running time of $O(n^3)$ — matching the classical FloydWarshall algorithm of this chapter.

LLP-Johnson

We now revisit Johnson's reweighting framework of Section 7.5, writing $G[j]$ in place of $price[j]$ (and $G[i]$ in place of $price[i]$) to match the LLP state-vector notation used throughout this book. Recall the reweighting identity $w'(i, j) = w(i, j) + G[j] - G[i]$ from Equation 7.1 and Lemma 7.8, which says that every path from s to t in the reweighted graph $\mathcal{G}'$ has weight $w(s, t) + G[t] - G[s]$, where $w(s, t)$ denotes the weight of that same path in the original graph $\mathcal{G}$. The sequential algorithm of the previous section uses BellmanFord from an auxiliary source s^* to obtain a feasible price vector. Here we give an LLP algorithm that finds such a price

vector directly. Our feasibility predicate B for the pricing vector is

$$\forall (i,j) \in E : w(i,j) + G[j] - G[i] \geq 0$$

Furthermore, we require $G[i] \geq 0$ for all i. We first show that B is lattice linear.

Lemma 7.13 *Consider any graph in which edge (i,j) has weight $w(i,j)$ and every vertex i has price $G[i]$. Consider the lattice of all non-negative price vectors. Then, the predicate*

$$B \equiv \forall (i,j) \in E : w(i,j) + G[j] - G[i] \geq 0$$

is lattice linear.

Proof: Since lattice linearity is closed under conjunction (Lemma 2.3(c)), it suffices to show that $B_e \equiv w(i,j) + G[j] - G[i] \geq 0$ is lattice linear for an arbitrary edge $e = (i,j)$. B_e can be rewritten as $G[j] \geq G[i] - w(i,j)$. The right-hand side of this inequality is a monotone function on G and hence, by Lemma 2.3(a), we get that B_e is lattice linear.

■

By applying the LLP algorithm, we get the following method to find the price vector.

Algorithm LLP-Johnson: Finding the minimum price vector

input: $pre(j)$: list of predecessors of vertex j; $w(i,j)$: int for all $i \in pre(j)$
output: G: array$[0..n-1]$ of int, initially 0
1 **forbidden**$(j \in [0..n-1])$: $\exists i \in pre(j) : G[j] < G[i] - w(i,j)$
2 $\Rightarrow$ **advance**: $\forall k : G[k] := \max(0, \max\{G[i] - w(i,k) \mid i \in pre(k)\})$;

Termination and iteration bounds for Algorithm LLP-Johnson match those of the sequential construction in Section 7.5: by Lemma 7.9, the advance rule converges to a feasible price vector iff the graph has no negative-weight cycle, and at most $|V|$ rounds of edge relaxation suffice. By Lemma 2.3(b), the set of all feasible price vectors is closed under meets, so the LLP algorithm converges to the componentwise-minimum feasible price vector.

Each LLP iteration can be computed in $O(m)$ time, so the graph is "reweighted" in $O(mn)$ time — the same asymptotic bound that BellmanFord achieves in the sequential algorithm. Plugging the resulting price vector into Dijkstra from every source yields an all-pairs shortest-path algorithm with running time $O(nm \log n)$, which is faster than the $O(n^3)$ FloydWarshall algorithm whenever the graph is not dense.

7.7 Summary

This chapter covered four algorithms for shortest-path problems. Dijkstra's algorithm solves single-source shortest paths on graphs with non-negative edge weights in $O(m \log n)$ time.

BellmanFord handles negative edge weights by relaxing all edges over $n-1$ iterations in $O(nm)$ time, and also detects negative-weight cycles. FloydWarshall computes all-pairs shortest paths in $O(n^3)$ time using an intermediate-vertex recurrence. Johnson's algorithm combines BellmanFord and Dijkstra to solve all-pairs shortest paths in $O(nm \log n)$ time, which is faster than FloydWarshall on sparse graphs. The chapter also presented LLP reformulations of these algorithms that expose parallelism by allowing multiple forbidden indices to advance concurrently.

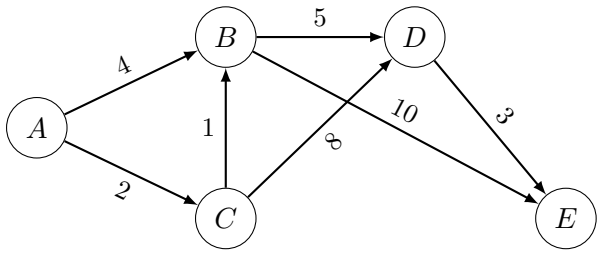
Table 7.2 lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Setting**	**Time Complexity**
Single-Source	Dijkstra	non-negative weights	$O(m \log n)$
Single-Source	BellmanFord	detects negative cycles	$O(mn)$
All-Pairs	FloydWarshall	no negative cycle	$O(n^3)$
All-Pairs	Johnson	no negative cycle	$O(nm \log n)$
Single-Source	LLP-Dijkstra	non-negative weights	$O(m \log n)$
Single-Source	LLP-BellmanFord	detects negative cycles	$O(mn)$
All-Pairs	LLP-FloydWarshall	no negative cycle	$O(n^3)$
All-Pairs	LLP-Johnson	no negative cycle	$O(nm \log n)$

Table 7.2: Classical and LLP shortest-path algorithms discussed in this chapter.

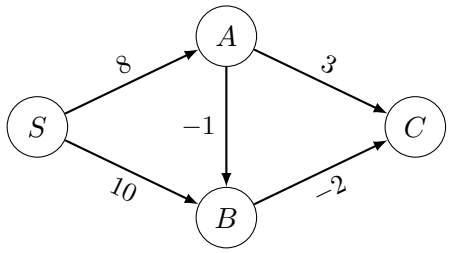
7.8 Problems

7.1. **(Dijkstra Example)** Consider the directed graph below:



Find the shortest path from A to E using Dijkstra's algorithm.

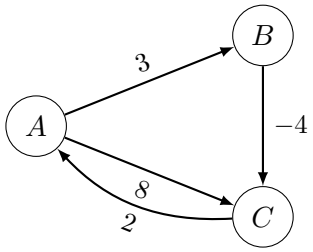
7.2. **(Bellman Ford Negative Weights)** Consider the directed graph below:



Find the shortest path from S to C using the BellmanFord algorithm.

7.3. **(Floyd Warshall All Pairs)**

Consider the directed graph below:



Find the shortest path for all pairs using the FloydWarshall algorithm.

7.4. **(Dijkstra Correctness Lemma)** Prove Lemma 7.1: that the outer loop of Dijkstra's algorithm maintains the following two invariants, where $cost[x]$ is the cost of a shortest path from v_0 to x. (a) For all vertices x: $fixed[x] \Rightarrow (d[x] = cost[x])$. (b) For all vertices x: $d[x]$ is the cost of a shortest path from v_0 to x all of whose vertices before x are fixed.

7.5. **(Rescaling Does Not Remove Negative Weights)** A tempting way to run Dijkstra's algorithm on a graph with negative edge weights is to make every weight non-negative first: let $c = \min_{e \in E} w(e)$ and replace each weight $w(e)$ by $w(e) - c$. Show

that this does not work, by giving a graph on which the shortest path under the rescaled weights is not a shortest path under the original weights. Explain which step of the argument for Johnson's reweighting (7.1) fails for this transformation.

7.6. **(Sensitive Edge Detection)** Give an algorithm to determine which of the edges in a directed graph are sensitive to the shortest path between s and t. An edge is sensitive, if its weight is reduced, then the shortest path from s to t is also reduced.

7.7. **(Bottleneck Shortest Path)** Let $G = (V, E, w)$ be an undirected graph with non-negative edge weights, and let $s, t \in V$. Define the *bottleneck* of a path P as $\max_{e \in P} w(e)$. Describe an algorithm that finds an s-t path of minimum bottleneck, and state its time complexity.

7.8. **(Counting Shortest Paths)** Given a weighted graph with non-negative edge weights and a source s, give an algorithm that computes, for every vertex v, both the shortest-path distance $d[v]$ and the *number* of distinct shortest paths from s to v. Modify Dijkstra's algorithm to maintain a count alongside the distance, and argue why relaxing an edge that ties the current best distance should add to the count rather than replace it. State the time complexity.

7.9. **(Vertex-Weighted Shortest Paths)** Suppose that instead of edge weights, every vertex v has a non-negative weight $c(v)$, and the cost of a path is the sum of the weights of the vertices it visits (including both endpoints). Reduce this problem to the standard (edge-weighted) single-source shortest path problem: describe a graph transformation and show that shortest paths correspond under the reduction. What is the size of the transformed graph in terms of $n = |V|$ and $m = |E|$?

7.10. **(Shortest-Path Tree Uniqueness)** Given a weighted graph with non-negative edge weights and a source s, the shortest-path distances $d[v]$ are uniquely determined, but there may be several different shortest-path trees achieving them (e.g., when a vertex has two incoming edges that both achieve its shortest distance). Give an efficient algorithm to decide whether the shortest-path tree from s is unique.

7.9 Bibliographic Remarks

The single source shortest path problem has a rich history. For the history of Dijkstra's algorithm, the reader is referred to the book by [Eri19]. One popular research direction is to improve the worst case complexity of Dijkstra's algorithm by using different data structures. For example, by using Fibonacci heaps for the min-priority queue, Fredman and Tarjan [FT87] gave an algorithm that takes $O(m + n \log n)$. There are many algorithms that run faster when weights are small integers bounded by some constant γ. For example, Ahuja et al. [AMOT90] gave an algorithm that uses Van Emde Boas tree as the priority queue to give an algorithm that takes $O(m \log \log \gamma)$ time. Thorup [Tho00] gave an implementation that takes $O(n+m \log \log n)$ for graphs with integer weights on the RAM model. Raman [Ram97] gave an algorithm with $O(m+n\sqrt{\log n \log \log n})$ time. The LLP algorithm for the shortest path is taken from [AKG20]. Bellman and Ford's algorithm is from [Bel58] and [For56].

Chapter 8

The Minimum Spanning Tree Problem

8.1 Introduction

In this chapter, we discuss the problem of finding the minimum spanning tree (MST) of an undirected weighted graph. Let $\mathcal{G} = (V, E)$ be a connected undirected graph with $n = |V|$ vertices and $m = |E|$ edges, where each edge $e \in E$ has a weight $w(e) \in \mathbb{R}$. A *minimum spanning tree* (MST) of the graph is a subset $T \subseteq E$ such that:

1. T connects all vertices in V (i.e., (V, T) is connected),
2. The total weight $\sum_{e \in T} w(e)$ is minimized among all such subsets.

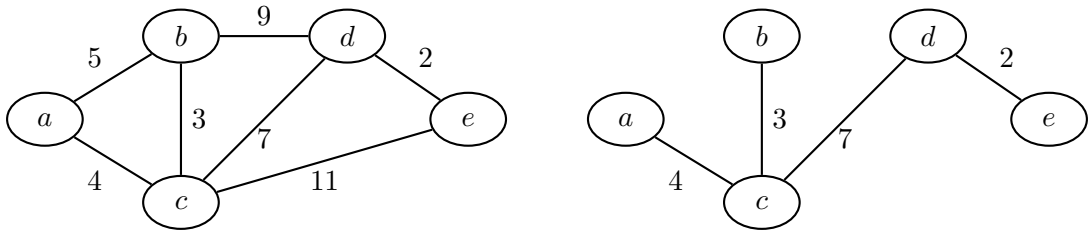
Equivalently, T is a tree that spans all vertices with the minimum possible total edge weight. A maximum spanning tree is defined similarly, but maximizes the total weight instead.

We assume that all edge weights are distinct. It is known that when all edge weights are distinct, a connected graph has a unique minimum spanning tree, and, more generally, a graph that is not necessarily connected has a unique minimum spanning forest. If edge weights may repeat, a minimum spanning tree (or forest) still exists but need not be unique. For simplicity of exposition, we will assume that the underlying graph is connected.

The primary motivation for finding an MST is to connect all nodes in a network with the minimum total cost while ensuring connectivity and avoiding cycles: for example, minimizing the cost of cabling a telecommunications or electrical grid, building roads or pipelines between locations, clustering data points, approximating NP-hard problems such as the Traveling Salesman Problem (an MST gives a lower bound on tour length), and broadcasting or routing efficiently in computer networks.

This chapter is organized as follows. Section 8.2 describes the key properties of a minimum spanning tree, including the notion of a *fragment*. Section 8.3 describes the Find-Union data structure, which is used by several MST algorithms to detect cycles efficiently. Section 8.4 presents Kruskal's algorithm, a greedy algorithm that chooses the least-cost feasible edge. Section 8.5 presents Prim's algorithm, another greedy algorithm that grows a single fragment.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.



(i) the graph (ii) its MST, of weight 16

Figure 8.1: An undirected weighted graph and its minimum spanning tree, the edges of weights $\{2, 3, 4, 7\}$

Section 8.6 describes Boruvka's algorithm, which extends multiple fragments in every iteration. Section 8.7 compares the three algorithms in terms of data structures, graph density, and parallelism. Section 8.8 reformulates these algorithms in the LLP framework. Section 8.9 discusses constrained minimum spanning trees with mandatory edges.

8.2 Key Properties of a Minimum Spanning Tree

The notion of a *fragment* is crucial in understanding MST algorithms. A fragment is simply a subtree of the MST. Consider the graph in Fig. 8.1. The minimum spanning tree in this graph corresponds to the edges $\{2, 3, 4, 7\}$. The subtree formed by the edges with weights 3 and 4 is a fragment with three vertices $\{a, b, c\}$ and two edges $\{(a, c), (b, c)\}$. A crucial property of the MST is as follows.

Lemma 8.1 *Let F be a fragment. Let e be the edge with the minimum weight that is outgoing from F. Then $F \cup \{e\}$ is also a fragment.*

Proof: Let T be the minimum spanning tree. If $e \in T$, then $F \cup \{e\}$ is trivially a fragment. If $e \notin T$, adding e to T creates a unique cycle C. This cycle must contain another edge f that is outgoing from F (since the cycle enters and exits F). Since $w(e) \leq w(f)$ (as e has minimum outgoing weight), $T' = T \setminus \{f\} \cup \{e\}$ is a spanning tree with $w(T') \leq w(T)$. Since T is a minimum spanning tree, T' is also a minimum spanning tree and it contains e. Hence $F \cup \{e\}$ is a fragment of T'.

■

In the fragment formed by edges with weights 3 and 4, there are three outgoing edges — edges with weights 7, 9, and 11. The edge 5 is not outgoing, as it connects vertices that are part of the fragment. According to Lemma 8.1, the edge with weight 7 can be added to the edges with weights 3 and 4 to grow the fragment.

A *cut* of the graph is a partition of the vertices into two non-empty parts S and $V \setminus S$, and an edge *crosses* the cut if exactly one of its endpoints lies in S. Removing any single edge of a

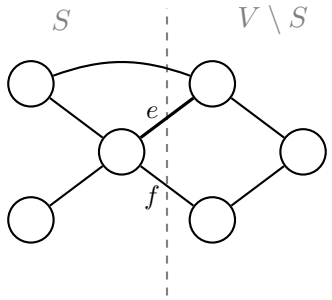


Figure 8.2: The Cut Property. Of the edges crossing the cut $(S, V \setminus S)$, the one of minimum weight, e, belongs to an MST.

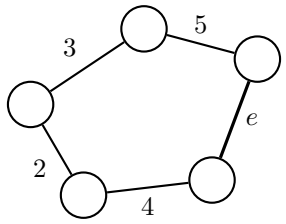


Figure 8.3: The Cycle Property. The strictly heaviest edge e of a cycle belongs to no MST: deleting it from any tree that contains it splits the tree, and some other edge of the cycle reconnects the two parts more cheaply.

spanning tree splits it into two components and so induces a cut. We will use one elementary fact repeatedly: a cycle crosses any cut an *even* number of times, because every edge that leaves S must be matched by one that returns to it.

The following two properties, phrased in terms of edge-weight extremes across a cut and around a cycle, are the workhorses behind every correctness proof in this chapter.

Lemma 8.2 (Cut Property) *Let $S \subsetneq V$ and let e be the minimum weight edge crossing the cut $(S, V \setminus S)$. Then e belongs to some MST; under distinct weights, e belongs to every MST.*

Proof: Let T be any MST and suppose $e \notin T$. Adding e to T creates a unique cycle. That cycle contains e, which crosses the cut, and a cycle crosses any cut an even number of times; it must therefore contain another edge $f \neq e$ that also crosses the cut. By hypothesis $w(e) \leq w(f)$, so $T' = T \cup \{e\} \setminus \{f\}$ is a spanning tree with $w(T') \leq w(T)$; hence T' is an MST containing e. Under distinct weights, $w(e) < w(f)$ and the inequality is strict, forcing e to be in every MST.

■

Lemma 8.3 (Cycle Property) *Let C be a cycle in the graph and let e be the strictly maximum weight edge in C. Then e does not belong to any MST.*

Proof: Suppose, for contradiction, that $e \in T$ for some MST T. Removing e from T partitions V into two components S and $V \setminus S$. The cycle C in $\mathcal{G}$ contains e, which crosses this cut, and C crosses it an even number of times; so some other edge $f \in C \setminus \{e\}$ crosses it too. By hypothesis $w(f) < w(e)$, so $T' = T \setminus \{e\} \cup \{f\}$ is a spanning tree with $w(T') < w(T)$, contradicting the optimality of T among spanning trees of $\mathcal{G}$.

■

8.3 The Find-Union Data Structure

A central operation in several MST algorithms — and in Kruskal's algorithm in particular — is to test whether the next edge under consideration would close a cycle in the partially built forest. The Find-Union data structure (also known as Disjoint Set Union) supports this test efficiently. It maintains a collection of disjoint sets and supports two primary operations:

- **Find**: Determines which set a particular element belongs to by returning its representative (or "root"). With path compression, this operation is nearly constant time, $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function.[1]
- **Union**: Merges two sets into one by linking their roots, typically using rank or size heuristics to keep trees balanced, also achieving $O(\alpha(n))$ amortized time.

Initially, each element is in its own set. When this structure is used to build a spanning forest, the *Union* operation merges sets of vertices connected by selected edges, while the *Find* operation checks if adding an edge would form a cycle (i.e., if its endpoints are already in the same set).

Each set is maintained as a rooted tree in which every node has a *parent* pointer. The root of the tree serves as the identity of the set. The *Find* operation finds the root of the tree. Observe that finding the root of the tree by traversing the parent pointer takes time proportional to the distance of the node from the root. One method of reducing this distance in future traversals is *path compression*: we make all the nodes on the find path point directly to the root, so subsequent traversals from those nodes are essentially constant time.

The *Union* operation merges two trees by making the root of one of them point to the other root. For correctness, it does not matter which root points to the other root. However, to keep the lengths of the longest paths small, we use the notion of *rank* to decide. Our rule is as follows: the root of the tree with the lower rank points to the root with the higher rank. The rank of the new tree is the same as that of the bigger rank. If both roots have the same rank, then it does not matter which one becomes the new root, and the rank of the new tree is one more than the rank of the old trees. It is a simple exercise to show that the rank of any

[1] $\alpha(n)$ is the least k such that $A(k,k) \geq n$, where A is the Ackermann function. It grows so slowly that $\alpha(n) \leq 4$ for every n below the number of atoms in the observable universe, so the bound is constant for all practical purposes.

tree with n nodes is at most $O(\log n)$ and the height of the tree is at most its rank. With these optimizations, any sequence of n Find and Union operations takes $O(n\alpha(n))$ amortized time.

Both algorithms share the following data structures:
parent: array[1..n] of int, initially for all v: $parent[v] = v$
rank: array[1..n] of int, initially for all v: $rank[v] = 0$

Algorithm Find: Finding the root of x's set with path compression

$Find(x)$
input: vertex x
output: root of the set containing x
1 **if** $parent[x] \neq x$ **then** $parent[x] := Find(parent[x])$ //path compression ;
2 **return** $parent[x]$

Algorithm Union: Computing union by rank of sets containing x and y

$Union(x, y)$
input: vertices x, y
output: merges the sets containing x and y (updates *parent*, *rank* in place)
1 $rootX := Find(x)$;
2 $rootY := Find(y)$;
3 **if** $rootX = rootY$ **then return** //already in same set, no union needed;
4 **if** $rank[rootX] < rank[rootY]$ **then** $parent[rootX] := rootY$;
5 **else if** $rank[rootX] > rank[rootY]$ **then** $parent[rootY] := rootX$;
6 **else** $parent[rootY] := rootX$;
7 $rank[rootX] := rank[rootX] + 1$ //increment rank if equal ;

Example 8.4 [Find-Union trace] We trace the data structure on five elements $\{1, 2, 3, 4, 5\}$ under the following sequence of operations:

$$\textsc{Union}(1, 2),\ \textsc{Union}(3, 4),\ \textsc{Union}(1, 3),\ \textsc{Union}(2, 5),\ \textsc{Find}(4).$$

Figure 8.4 shows the forest after each step.

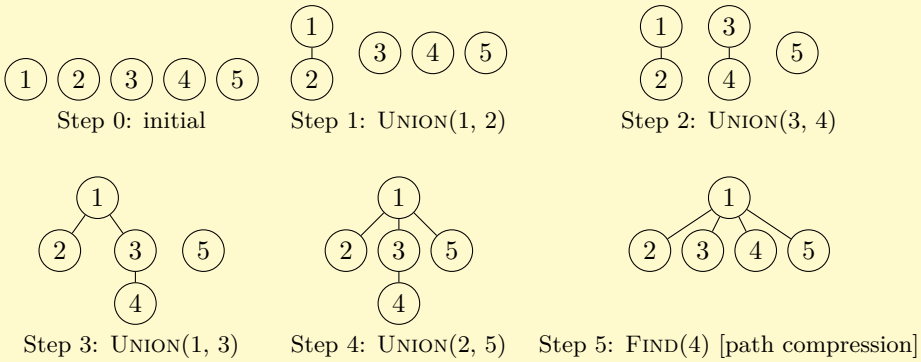


Figure 8.4: Forest after each step of the Find-Union trace. The rank of root 1 is $0, 1, 1, 2, 2, 2$ across the six panels; ranks of all other roots stay at their initial values. Step 5 shows the effect of path compression: $\textsc{Find}(4)$ walks $4 \to 3 \to 1$ and then rewires every node on the find path directly to the root.

A few details are worth noting. At Step 3, both roots have rank 1, so the tie-breaking rule attaches root 3 under root 1 and bumps $rank[1]$ to 2. At Step 4, $\textsc{Find}(2)$ traverses one pointer ($2 \to 1$) and returns 1; $\textsc{Find}(5)$ returns 5. Because $rank[1] = 2 > rank[5] = 0$, vertex 5 is attached under root 1 without changing any rank.

At Step 5, $\textsc{Find}(4)$ walks $4 \to 3 \to 1$. Without path compression, the forest would still look like Step 4 and the next $\textsc{Find}(4)$ would re-traverse the same path. With path compression (Algorithm Find), each node on the find path is rewired directly to the root, giving the flat star-shaped tree in the final panel: subsequent Find calls on 3 or 4 now take a single step. The *rank* values bound tree height but are themselves frozen during Find — compression flattens the tree without invalidating the ranks already assigned. This separation is what gives the combined heuristics their $O(\alpha(n))$ amortized cost.

8.4 Kruskal's Algorithm

Kruskal's algorithm is a canonical greedy algorithm for the minimum-spanning tree problem. It chooses edges one at a time in a greedy fashion. Let T be the set of edges chosen by the algorithm at any stage. The algorithm maintains the invariant that T does not have any cycle. Initially T is empty and therefore trivially satisfies the invariant. The algorithm considers the edges in increasing order of weights. For this reason, it is sometimes also known as the *shortest-edge-next* algorithm. Suppose that the algorithm has chosen the edges in T so far. To grow T, it finds the edge with the least weight that does not form a cycle with existing edges

in T. If any edge e forms a cycle with T, it is rejected, and the algorithm considers the least weight edge of the remaining edges. We use the Find-Union data structure of Section 8.3 to detect cycles efficiently: an edge (u, v) closes a cycle in T iff $Find(u) = Find(v)$.

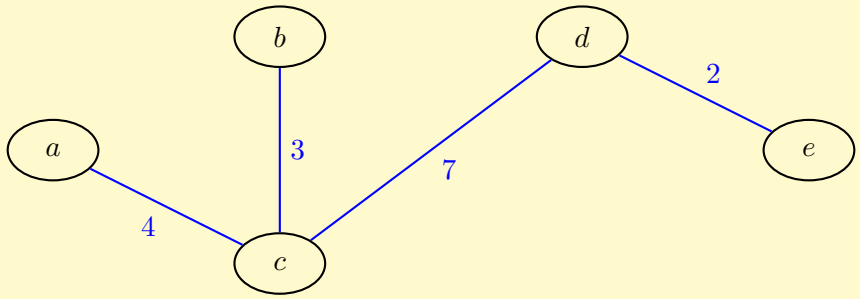
Algorithm Kruskal: Minimum spanning tree by adding edges in weight order

input: Undirected Weighted Graph $\mathcal{G} = (V, E, w)$ with $n = |V|$, edges presorted once in non-decreasing order of weight
output: Minimum Weight Spanning Tree T
var T: set of edges initially $\{\}$

```
1  forall v ∈ V :
2  |  parent[v] := v;   rank[v] := 0 ;        // each vertex starts in its own set
3  end
4  for edge (u, v) ∈ E in that order do
5  |  if Find(u) ≠ Find(v) then
6  |  |  T := T ∪ {(u, v)};
7  |  |  Union(u, v);
8  |  |  if |T| = n − 1 then return T;
9  |  end
10 end
11 return T ;                              // graph is disconnected if |T| < n − 1
```

In Algorithm Kruskal, the variable T keeps the set of all edges chosen. At every step, (V, T) is an acyclic subgraph — that is, a spanning forest. The Find-Union structure represents the connected components of this forest: $Find(u)$ returns the representative of u's component, and $Union(u, v)$ merges two components when an edge is added. Adding the next lightest edge either joins two distinct components (reducing the number of components by one) or closes a cycle within a single component (in which case the edge is rejected). The algorithm terminates when $|T| = n - 1$, at which point the forest has coalesced into a spanning tree, or when the edges have all been processed and $|T| < n-1$, indicating that the input graph is disconnected.

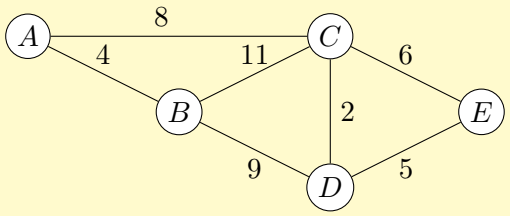
Example 8.5 Consider the graph shown in Fig. 8.1. Kruskal's algorithm first chooses the edge with weight 2 since it has the least weight. Then, it chooses the edges with weight 3 and 4 because these edges do not form any cycle. The next edge has weight 5; this edge is rejected because it forms a cycle with the already chosen edges with weights 3 and 4. The algorithm then chooses the edge with weight 7 and terminates with $T = \{2, 3, 4, 7\}$. The resulting MST is shown below.



Correctness. We show that every edge added by the algorithm belongs to some MST. When the algorithm adds $e = (u, v)$, the vertices u and v are in different components of the forest (V, T). Let S be the vertex set of u's component. Then e crosses the cut $(S, V \setminus S)$, and e is the lightest edge crossing this cut — any lighter edge would have been considered earlier and, since it also crosses the cut, would have been accepted (merging the two sides), contradicting u and v being in different components. By the cut property (Lemma 8.2), e belongs to some MST. Hence every edge in the final T belongs to an MST, so T is an MST.

Complexity. The algorithm is dominated by the cost of sorting the edges: $O(m \log n)$. Each *Find* or *Union* takes $O(\alpha(n))$ amortized time, so the total cost of the $O(m)$ Find-Union operations is $O(m\alpha(n))$, which is dominated by sorting. The overall complexity of Kruskal's algorithm is therefore $O(m \log n)$.

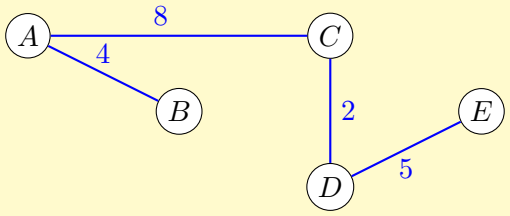
Example 8.6 Consider a graph with five vertices A, B, C, D, E and edges A-B: 4, A-C: 8, B-C: 11, B-D: 9, C-D: 2, C-E: 6, D-E: 5. We apply Kruskal's algorithm.



Edges in sorted order: C-D (2), A-B (4), D-E (5), C-E (6), A-C (8), B-D (9), B-C (11). Initially each vertex is its own set. Process each edge with *Find* and *Union*:

1. C-D (2): $Find(C) \neq Find(D)$, so $Union(C, D) \to \{A\}, \{B\}, \{C, D\}, \{E\}$.
2. A-B (4): $Find(A) \neq Find(B)$, so $Union(A, B) \to \{A, B\}, \{C, D\}, \{E\}$.
3. D-E (5): $Find(D) \neq Find(E)$, so $Union(D, E) \to \{A, B\}, \{C, D, E\}$.
4. C-E (6): $Find(C) = Find(E)$; skip.
5. A-C (8): $Find(A) \neq Find(C)$, so $Union(A, C) \to \{A, B, C, D, E\}$. MST complete; the remaining edges B-D (9) and B-C (11) would close cycles.

The MST consists of C-D (2), A-B (4), D-E (5), A-C (8), with total weight 19.



8.5 Prim's Algorithm

Prim's algorithm is also a greedy algorithm. It builds the minimum spanning tree by increasing the size of a single fragment by adding the minimum weight outgoing edge of the fragment. It simply exploits Lemma 8.1 to increase the size of the fragment until it becomes the MST. At any stage, Prim's algorithm has a fragment F. It finds the minimum outgoing edge of that fragment e. This edge can be viewed as the edge from the fragment to its nearest neighbor. Therefore, this algorithm is sometimes also known as the *nearest-neighbor-next* algorithm. To find the nearest neighbor, every vertex v maintains a label d that corresponds to the cost of adding v to the fragment. At each iteration, the algorithm chooses the vertex v with the minimum value d and adds it to the fragment. The vertex with the minimum label is supplied by a binary min-heap H of (v, c) pairs ordered by c, exactly as in Dijkstra's algorithm. Rather than decrease a key already in the heap, we simply add the vertex again whenever its label falls, so a vertex may appear in H more than once; the stale copies are discarded when they surface and the $\neg fixed[v]$ test rejects them. The array $fixed$ keeps track of the vertices in the

fragment. Whenever a new vertex v is *fixed* and added to the fragment, the d values for any adjacent vertex v' are updated as follows. We check whether the weight of the edge (v, v') is lower than the previous value of $d[v']$. If this is true, then $d[v']$ is updated to $w(v, v')$. We also use a *parent* pointer with each node, which keeps track of the node v that is responsible for the d value of v'.

Algorithm Prim: Minimum spanning tree by growing a single fragment

input: Undirected Weighted Graph: $\mathcal{G} = (V, E, w)$, with the edges of each vertex given by an adjacency list
output: Minimum Weight Spanning Tree T
var d: array$[0..n-1]$ of real initially ∞ `//` $d[v]$ `is the cost to add vertex` v
$parent$: array$[0..n-1]$ of int initially -1 `//` $parent[v]$ `is the node that corresponds to` $d[v]$ `cost`
$fixed$: array$[0..n-1]$ of boolean initially *false* `//vertices whose` d `value is fixed`
T: set of edges initially $\{\}$
H: binary min-heap of (v, c) ordered by c, initially empty `//` v `is the vertex and` c `is the cost`

```
1  d[0] := 0;
2  H.add(0, d[0]);
3  while ¬H.empty() do
4  |  (v, c) := H.removeMin();
5  |  if ¬fixed[v] then
6  |  |  if parent[v] ≠ −1 then add (v, parent[v]) to T ;
7  |  |  fixed[v] := true;
8  |  |  forall v′: ¬fixed[v′] ∧ (v, v′) ∈ E :
9  |  |  |  if w(v, v′) < d[v′] then
10 |  |  |  |  d[v′] := w(v, v′);
11 |  |  |  |  parent[v′] := v;
12 |  |  |  |  H.add(v′, d[v′]);
13 |  |  |  end
14 |  |  end
15 |  end
16 end
17 if |T| < n − 1 then return null // no spanning tree in the graph;
18 return T;
```

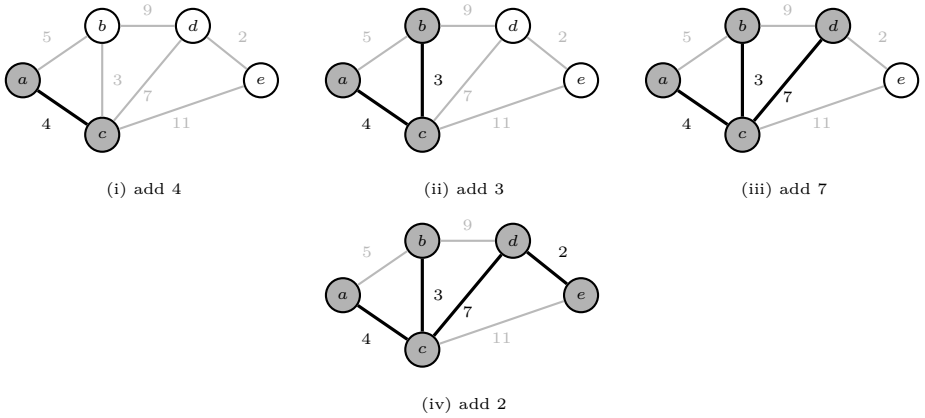


Figure 8.5: Prim's algorithm from vertex a: the shaded fragment grows by one vertex per step, each time across its lightest outgoing edge.

Example 8.7 Consider again the graph shown in Fig. 8.1. For Prim's algorithm, we start from a fixed node. Suppose that we start from the vertex a. Then the nearest neighbor is c with a cost of 4. The next nearest neighbor to the fragment with vertices $\{a, c\}$ is the vertex b. The cost of adding b is 3. At this point, we have the vertices $\{a, b, c\}$ in the fragment. The cost of adding vertex d is 7 and the cost of adding vertex e is 11. We add the vertex d to our fragment with the cost 7. Finally, e is added with the cost 2. Hence, the edges are added to the tree in the order $4, 3, 7, 2$. The set of edges chosen is identical to that of Kruskal's algorithm, even though the order in which they are chosen is different. This is not surprising because there is a unique minimum spanning tree when all edge weights are distinct (Problem 8.3).

Correctness. Prim's algorithm maintains the invariant that the fragment F (the set of fixed vertices together with T) is contained in some MST. Initially $F = (\{v_0\}, \emptyset)$, which is trivially contained in every MST. Suppose the invariant holds just before an iteration that fixes vertex v. By construction, the edge $(parent[v], v)$ is the lightest edge crossing the cut $(F, V \setminus F)$: its weight is $d[v]$, and for any other edge (u', v') with $u' \in F$, $v' \notin F$ we have $d[v'] \leq w(u', v')$ and $d[v] \leq d[v']$. By the cut property (Lemma 8.2), this edge belongs to some MST that extends F. If $\mathcal{G}$ is connected, the heap is never exhausted while some vertex is still unfixed, so after $n - 1$ iterations F is a spanning tree, and hence an MST; otherwise the heap empties with $|T| < n - 1$ and the algorithm reports that no spanning tree exists.

Complexity. The algorithm above keeps the labels in a binary heap. Each edge is relaxed at most once, so at most m vertices are added to H and each is later removed, giving $O(m)$ heap operations of $O(\log n)$ each and $O(m \log n)$ work overall. Walking the adjacency list of each fixed vertex contributes $O(m)$ in total, since every edge is processed at most once; this is why the adjacency-list representation matters, as scanning a weight matrix for the neighbors of v would cost $\Theta(n)$ per iteration however few edges the graph has, and $\Theta(n^2)$ overall. The minimum label can also be found without a heap, by simply traversing the array d on every

iteration. That variant performs $O(n)$ work to find the minimum in each of the $O(n)$ iterations of the while loop, plus $O(m)$ to process the edges, for $O(n^2 + m)$ work in all.

Comparison with Dijkstra's algorithm. Prim's and Dijkstra's algorithms (Chapter 7) share the same control structure: both maintain a priority-keyed set of frontier vertices, repeatedly extract the one with the smallest key, fix it, and relax the keys of its neighbors. The difference lies entirely in what the key means. In Prim, $d[v']$ is the weight of the *single* lightest edge from the current fragment to v', so the relaxation rule reads $d[v'] := \min(d[v'], w(v, v'))$. In Dijkstra, $d[v']$ is the length of the shortest path from the source to v' through already-fixed vertices, so the relaxation rule becomes $d[v'] := \min(d[v'], d[v] + w(v, v'))$. As a consequence, Prim's correctness rests on the cut property and holds for arbitrary real edge weights (including negative), whereas Dijkstra's correctness additionally requires that extending a path cannot shorten it, which is true only for non-negative weights.

8.6 Boruvka's Algorithm

In Prim's algorithm, we started with a trivial fragment that included only the vertex v_0. We kept increasing the size of the fragment until it became a spanning tree. In Boruvka's algorithm, we may have more than one fragment. We increase the size of all fragments by adding the minimum outgoing edge for each fragment.

Algorithm Boruvka presents the sequential Boruvka's algorithm to find the MST. We use T to denote the set of tree edges. Initially, T is empty. When we determine the components in (V, T), we find that there are n components, since each vertex is a component by itself when T is empty. The algorithm finds the minimum-weight outgoing edge for each component as follows. In any iteration, we use BFS to find the least numbered vertex to which any vertex is connected in the graph (V, T). This vertex serves as the identifier for the component of the node i, and we use the variable $cid[i]$ to store it. Once we have determined the component identity of all nodes, we move to the next step of determining the minimum weight outgoing edge for each component. We traverse all edges, and for each edge that connects two different components, we check whether it is cheaper than the previously known outgoing edge for the component on either side. Once we have determined all minimum weight edges for each component, we add these to T and start the next iteration.

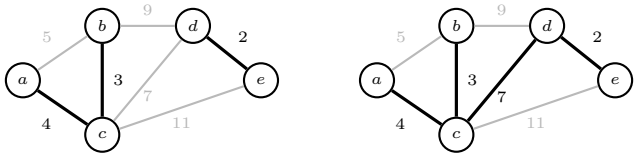
Example 8.8 Consider the graph in Fig. 8.1. Initially, T is empty and there are 5 components. We compute mwe for each component and get the edges $4, 3, 3, 2, 2$ as the minimum weight edges of a, b, c, d, e, respectively. Once these edges are added, we have two components: $\{a, b, c\}$ and $\{d, e\}$. We then find mwe of these two components as the edge 7. On adding this edge, we have chosen $(n - 1)$ edges, and the algorithm terminates with the edges $\{2, 3, 4, 7\}$.

After every iteration, the number of connected components in (V, T) is reduced by at least a factor of two. This is because every component gets attached to some other component. The worst case is when every least weight edge chosen by any component is also chosen as the least weight edge by the component on the other side of the edge. Hence, the algorithm takes at most $O(\log n)$ iterations of the while loop. It is easy to see that the least weight edge outgoing

Algorithm Boruvka: Minimum spanning tree by merging fragments along their lightest edges

input: undirected *connected* weighted graph: $\mathcal{G} = (V, E, w)$
output: minimum weight spanning tree T
var T: set of edges initially $\{\}$
cid: array[1..n] of 0..n initially all 0

```
1  while |T| < n − 1 do
2  |  visited: array[1..n] of boolean initially all false;
3  |  mwe: array[1..n] of edge initially all null;
4  |  dist: array[1..n] of real initially all ∞;
5  |  for i := 1 to n do
6  |  |  if ¬visited[i] then
   |  |  |  // do a BFS in (V,T) from vertex i, setting cid of every visited
   |  |  |     vertex to i
7  |  |  |  BFS(i);
8  |  |  end
9  |  end
10 |  for (i, j) ∈ E such that (cid[i] ≠ cid[j]) do
11 |  |  if w(i, j) < dist[cid[i]] then
12 |  |  |  dist[cid[i]] = w(i, j);
13 |  |  |  mwe[cid[i]] = (i, j);
14 |  |  end
15 |  |  if w(i, j) < dist[cid[j]] then
16 |  |  |  dist[cid[j]] = w(i, j);
17 |  |  |  mwe[cid[j]] = (i, j);
18 |  |  end
19 |  end
20 |  forall i such that cid[i] = i :
21 |  |  T := T ∪ {mwe[i]}
22 |  end
23 end
24 return T;
```



(i) round 1: add $2, 3, 4$ (ii) round 2: add 7

Figure 8.6: Boruvka's algorithm. Every component contributes its minimum weight outgoing edge at once, so the five singleton components become $\{a, b, c\}$ and $\{d, e\}$ in one round, and a single tree in the next.

from each component is found in $O(m)$ work. Thus, the algorithm takes $O(m \log n)$ work.

8.7 Comparison of Algorithms

All three algorithms of this chapter run in $O(m \log n)$ time, but they differ substantially in the data structures they rely on, the input formats they prefer, and the kind of parallelism they expose. This section summarizes those practical trade-offs so that a reader faced with a concrete instance can pick the right algorithm.

Input representation. Kruskal's algorithm is most natural when the edges are already available as a sorted list — the only operations it performs on the graph are "scan the next edge" and "are these two endpoints already connected". In particular, Kruskal's is well suited to streaming settings, where edges arrive one at a time in weight order and the graph need not be kept in memory. Prim's algorithm, on the other hand, needs random access to each vertex's neighbors and is the natural choice when the graph is represented as an adjacency list. Boruvka's algorithm operates on both edge and vertex sets and can be implemented with either representation.

Graph density. When the graph is very dense ($m = \Theta(n^2)$), the $O(m \log n)$ bound of all three algorithms is dominated by the edges themselves. Prim's algorithm implemented with an array-based scan (rather than a heap) achieves $O(n^2)$ work, which beats $O(m \log n) = O(n^2 \log n)$ in this regime. For sparse graphs ($m = O(n)$), all three algorithms are within a $\log n$ factor of linear, and the $O(n^2)$ variant of Prim's becomes strictly worse than Kruskal's.

Required auxiliary structures. Kruskal's algorithm requires a union-find structure to test cycle-closing in $O(\alpha(n))$ time per query, plus a sort on the edge weights. Prim's algorithm requires a priority queue keyed on the cost-to-fragment of each fringe vertex. Boruvka's algorithm, by contrast, needs neither a global sort nor a priority queue: each round is a parallel scan of the edges followed by a connected-component relabeling.

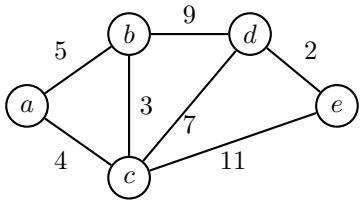
Parallelism. This is where the three algorithms diverge the most. Kruskal's algorithm is inherently sequential: edges must be committed in weight order so that the cut property can

be applied, and each commit may reject its predecessors in the sort. Prim's algorithm is also essentially sequential because it grows one fragment at a time from a single root, though modest parallelism is available in the edge-relaxation step. Boruvka's algorithm, finally, is the parallelism-friendly one: every component picks its minimum outgoing edge independently in each round, and the whole round can be carried out in $O(\log n)$ time with $O(m)$ processors. This is why the parallel MST algorithms in the literature — and the LLP-Boruvka algorithm of the next section — are all Boruvka variants.

Unifying principle. Despite their different strategies, all three algorithms construct an MST by repeatedly selecting edges that are certified safe by the Cut Property (Lemma 8.2). Kruskal's algorithm applies the cut property to the cut defined by the two components that the next lightest edge would join. Prim's algorithm applies it to the cut $(F, V \setminus F)$ separating the current fragment from the rest. Boruvka's algorithm applies it simultaneously to every current component's cut.

8.8 LLP Perspective

In this section we collect the LLP reformulations of the algorithms introduced earlier in the chapter. All three are traced on the same graph, which is repeated here from Fig. 8.1 so that it is at hand throughout the section.



LLP-Kruskal Algorithm

In this subsection, $G[j]$ is a Boolean indicating whether edge e_j is in the spanning tree.

We now give an LLP version for Kruskal's algorithm based on the simple Boolean lattice of all edges. Our distributive lattice is the Boolean lattice formed from all edges. We assume that the edges are given to us in increasing order. Then, an edge e_j between vertices v_k and v_l is in the minimum spanning tree iff there is no path between vertices v_k and v_l using edges $e_1 \ldots e_{j-1}$. Note that when j is 1, e_j corresponds to the lightest edge and is always in the minimum spanning tree. An LLP version is presented as Algorithm LLP-Kruskal.

The guard tests connectivity through *every* earlier edge $e_1, \ldots, e_{j-1}$, not merely the earlier edges already selected. The distinction matters, because an LLP schedule is arbitrary: conditioning on the current selection bits ($G[i] = 1$) gives a different and unsound predicate. On a triangle with sorted edges $(a, b), (b, c), (a, c)$ it permits selecting the last edge first and then both others, forming a cycle, whereas the guard as stated never forbids the last edge of a triangle — its endpoints are already joined through the earlier edges.

Algorithm LLP-Kruskal: Minimum spanning tree by adding the lightest safe edge

input: undirected weighted graph: $\mathcal{G} = (V, E, w)$, edges in increasing order of weights
output: G: array$[1..m]$ of $\{0, 1\}$ `// indicating MST edges`
var G: array$[1..m]$ of $\{0, 1\}$
1 **init**: $\forall j : G[j] := 0$;
2 **forbidden**$(j \in [1..m])$: $(G[j] = 0)$ and there exists no path from v_k to v_l with edges $1..j-1$ for the edge $e_j = (v_k, v_l)$
3 $\Rightarrow$ **advance**: $G[j] := 1$;

Example 8.9 Consider the graph in Fig. 8.1. Sorting the edges by weight gives

$$e_1 = (d, e, 2),\ e_2 = (b, c, 3),\ e_3 = (a, c, 4),\ e_4 = (a, b, 5),$$
$$e_5 = (c, d, 7),\ e_6 = (b, d, 9),\ e_7 = (c, e, 11).$$

Starting from $G = (0, 0, 0, 0, 0, 0, 0)$, an index j is forbidden when $G[j] = 0$ and e_j's endpoints are not connected by any path through the earlier edges $e_1, \ldots, e_{j-1}$ — all of them, whether or not they were selected.

- $j = 1$: no prior edges, so d and e are not connected. Advance $G[1] := 1$.
- $j = 2$: b and c are not connected by $\{e_1\}$. Advance $G[2] := 1$.
- $j = 3$: a and c are not connected by $\{e_1, e_2\}$. Advance $G[3] := 1$.
- $j = 4$: a and b are already connected via a–c–b using $\{e_2, e_3\}$. Not forbidden; $G[4]$ stays 0.
- $j = 5$: c and d are not connected by $\{e_1, e_2, e_3, e_4\}$ — note that e_4 counts here even though it was not selected. Advance $G[5] := 1$.
- $j = 6$: b and d are connected via b–c–d. Not forbidden.
- $j = 7$: c and e are connected via c–d–e. Not forbidden.

The algorithm terminates with $G = (1, 1, 1, 0, 1, 0, 0)$. The MST consists of the edges e_1, e_2, e_3, e_5, that is, $(d, e, 2)$, $(b, c, 3)$, $(a, c, 4)$, $(c, d, 7)$, with total weight 16.

LLP-Prim Algorithm

In this subsection, $G[v]$ stores the weight of the cheapest edge attaching v to the tree-so-far rooted at v_0. Because all edge weights are assumed distinct, the weight uniquely identifies the

chosen edge.

We now show an LLP algorithm to find a minimum spanning tree rooted at a fixed vertex v_0. As before, we assume that all edge weights are unique. Every node other than v_0 has to choose an edge that we call the G edge. A node keeps all its edges in the sorted order and begins by choosing its least edge as its G edge. Initially the chosen edge of v may lead to a non-fixed neighbor, so following G-edges from v need not reach v_0; once v becomes *fixed* (defined below), the chain $v \to G\text{-edge}(v) \to \cdots$ reaches v_0 and $G[v]$ is the weight of the edge by which v is attached to the tree-so-far. For the graph in Fig. 8.1, with root as the vertex a, we get the sorted adjacency lists shown in Fig. 8.7 for the other vertices.

Vertex	**Sorted adjacent edge weights**
b	3, 5, 9
c	3, 4, 7, 11
d	2, 7, 9
e	2, 11

Figure 8.7: Sorted adjacency lists for each non-root vertex of the graph in fig. 8.1, with root a.

We let $G[i]$ be the edge chosen by node i. Thus, initially, the vector G is $\{3, 3, 2, 2\}$. Observe that the set of all possible combinations of edges forms a distributive lattice. Since every node except v_0 has exactly one outgoing edge, following the chosen edge from every node, we either reach v_0, or end up in a cycle. We define a vertex as *fixed* if traversing the path starting from the edge proposed by that vertex leads to v_0. The vertex v_0 is trivially fixed. It is clear that the edge proposed by a non-fixed vertex can only lead to a non-fixed vertex, and the edge proposed by a fixed vertex can only lead to a fixed vertex. Thus, any G partitions the set of vertices into *fixed* and *non-fixed* vertices. In our example, the initial vector $\{3, 3, 2, 2\}$ makes the vertex a fixed and all other vertices $\{b, c, d, e\}$ non-fixed.

We define the set of edges between these two partitions as follows.

$$E'(G) := \{(i, k) \in E \mid fixed(i, G) \wedge \neg fixed(k, G)\}$$

In our example, we get the following edges as $E'(G) = \{(a, b), (a, c)\}$.

Let $x = (i, j)$ be the edge in E' with minimum $w(i, j)$. For our example, it is (a, c) with an edge weight of 4. We claim that the index j is forbidden in G. Consider any H such that $H[j] < w(i, j)$. Suppose, if possible, that H forms the minimum spanning tree. Any spanning tree must have an edge y between the set $fixed(G)$ and *non-fixed*(G). We claim that H cannot be the minimum spanning tree. This claim holds because the edge set $H - \{y\} + \{x\}$ is also a spanning tree and has a lower weight than H. Therefore, unless the index j advances to the edge (i, j), G cannot be the minimum spanning tree.

When we advance $G[j]$ to that edge, v_j becomes fixed. Observe that additional nodes may become fixed if their proposed edge leads to v_j directly or indirectly. The algorithm continues to advance G until all vertices become fixed or the edge set $E'(G)$ is empty. In the latter case, the graph is not connected and there is no minimum spanning tree.

Algorithm LLP-Prim shows the forbidden condition and the advance statement.

Algorithm LLP-Prim: Minimum spanning tree by growing one fragment from a root

input: undirected weighted graph: $\mathcal{G} = (V, E, w)$
output: G: array$[1..n-1]$ of real `// encoding MST edges`
var G: array$[1..n-1]$ of real initially $\forall i : G[i] :=$ minimum edge adjacent to i
1 **always**;
2 $fixed(j, G) \equiv$ there exists a directed path from v_j to v_0 using edges in G;
`// propagate` $fixed$ `along chosen edges from` v_0
3 $E' := \{(i,k) \in E \mid fixed(i,G) \wedge \neg fixed(k,G)\}$;
4 **forbidden**$(j \in [1..n-1])$: $\exists i : (i,j) \in E' \wedge w(i,j) = min\{w(k,l) \mid (k,l) \in E'\}$
5 $\Rightarrow$ **advance**: $G[j] := min\{w(i,j) \mid (i,j) \in E'\}$;

Example 8.10 Take the graph of Fig. 8.1 with root $v_0 = a$. Each non-root vertex j initializes $G[j]$ to the weight of its lightest adjacent edge, giving $G = (G[b], G[c], G[d], G[e]) = (3, 3, 2, 2)$, i.e., $b \to c$, $c \to b$, $d \to e$, $e \to d$. Only a is fixed; the other four vertices form two disconnected cycles, so none is fixed.
Iteration 1. The cross-cut set is $E' = \{(a,b,5), (a,c,4)\}$, with minimum $(a,c,4)$, so $j = c$ is forbidden. Advance $G[c] := 4$. Now c is fixed, and b becomes fixed transitively via $b \to c \to a$. Fixed $= \{a, b, c\}$; $G = (3, 4, 2, 2)$.
Iteration 2. Now $E' = \{(b,d,9), (c,d,7), (c,e,11)\}$, with minimum $(c,d,7)$, so $j = d$ is forbidden. Advance $G[d] := 7$. Now d is fixed, and e becomes fixed transitively via $e \to d \to c \to a$. Fixed $= \{a, b, c, d, e\}$; $G = (3, 4, 7, 2)$.

All vertices are fixed; the algorithm terminates with the MST edges

$$\{(b,c,3),\ (a,c,4),\ (c,d,7),\ (d,e,2)\},$$

total weight 16.

LLP-Boruvka Algorithm

In this subsection, $G[v]$ stores v's parent in the rooted tree: initially v's chosen lightest-edge endpoint, and ultimately — after pointer-jumping — the leader (root) of v's component.

In this section, we give a recursive version of the algorithm and implement each recursive instance of Boruvka's algorithm with the LLP algorithm. As before, we assume that the input to our algorithm is a connected graph without any self-loops. The LLP algorithm uses the single variable G. For each instance, we let the minimum weight edge (mwe) adjacent to each vertex v be denoted by $mwe[v]$. We denote this directed graph by M: the set of vertices is V and the set of edges is $\{(v,w) | (v,w) = mwe[v]\}$. We initialize $G[v]$ with the node w when $mwe[v] = (v,w)$ except when $mwe[w] = (w,v)$ and $v < w$. For the latter case, we make $G[v]$

equal to v. As a result of this initialization, the structure $G[v]$ forms a rooted tree. We say that the edge (v, w) is *incident* to v when $G[v]$ equals w for w different from v.

Algorithm LLP-Boruvka: Finding MST using Boruvka's algorithm via LLP

input: undirected *connected* weighted graph $\mathcal{G} = (V, E, w)$
output: minimum weight spanning tree T

```
1  function Boruvka(V, E):
2    if |V| = 1 then return {};
     var: G: array[1..n] of 1..n, initially G[v] = w such that mwe[v] = (v, w), except
          when mwe[w] = (w, v) and v < w; in that case G[v] := v
3    forbidden(j ∈ [1..n]): G[j] ≠ G[G[j]]
4      ⇒ advance: G[j] := G[G[j]];
5    return T ∪ Boruvka(V',E') where;
6    T := {(v, w) | (v, w) is the minimum weight edge of v};
7    V' := {v ∈ V | G[v] = v};
8    E' := {(G[v], G[w]) | (v, w) ∈ E ∧ G[v] ≠ G[w]};
9      with parallel edges collapsed to the lightest one: for distinct components p, q,;
10     w'(p, q) := min{w(v, w) | (v, w) ∈ E, {G[v], G[w]} = {p, q}};
```

Once we have rooted trees for the graph, we convert the rooted trees to rooted stars using pointer-jumping. A node j is considered forbidden if $G[j] \neq G[G[j]]$. It is advanced by setting $G[j]$ to $G[G[j]]$. The algorithm stops this iteration when no node is forbidden. We show that each instance is indeed an LLP algorithm. First note that $G[v]$ is always reachable from v in the directed graph M. Initially, this claim holds because $G[v]$ is set to w where (v, w) is an edge in M. The only statement executed in the program is $G[v] := G[G[v]]$ which preserves the claim.

We also observe that M is a collection of rooted trees and if w is reachable from v, then there is a unique path from v to w. Furthermore, because all edge weights are assumed distinct, the minimum outgoing edge selected at each vertex is unique, and therefore the weight of edges along any directed path is strictly decreasing. To see this, if v points to w and w points to u then the weight of the edge (w, u) must be strictly smaller than the weight of the edge (v, w) by the property that each vertex points to its minimum weight edge. We are now ready to show the following lemma.

Lemma 8.11 *The predicate $B \equiv \forall j : G[j] = G[G[j]]$ is lattice-linear.*

Proof: Suppose B is false, so there exists j with $G[j] \neq G[G[j]]$. Let $k = G[j]$. Since k is not the root (roots satisfy $G[r] = r$, which would give $G[j] = G[G[j]]$), and M is a rooted tree, the

path from k to the root does not pass through k again. Since $G[k]$ always lies on this path (by the reachability invariant), we have $G[k] \neq k$ permanently. Now, advancing any other index i only changes $G[i]$, not $G[j]$, so $G[j]$ remains k. And $G[G[j]] = G[k] \neq k = G[j]$. Therefore j remains forbidden.

■

Lemma 8.12 *Each instance of finding connected components terminates.*

Proof: Each advance sets $G[j] := G[G[j]]$, moving j's pointer strictly closer to the root along the tree path. The distance from j to the root decreases by at least one with each advance of j. Since the total distance summed over all vertices is finite and strictly decreasing, the algorithm terminates.

■

The complete procedure appears in Algorithm LLP-Boruvka.

Example 8.13 Consider the graph in Fig. 8.1. The minimum-weight adjacent edge of each vertex is

$$mwe[a] = (a, c, 4), \quad mwe[b] = (b, c, 3), \quad mwe[c] = (b, c, 3),$$
$$mwe[d] = (d, e, 2), \quad mwe[e] = (d, e, 2).$$

Vertices b and c share the same minimum edge (b, c), as do d and e via (d, e). Breaking each 2-cycle in favor of the smaller-numbered vertex, the initialization gives

$$G[a] = c, \quad G[b] = b, \quad G[c] = b, \quad G[d] = d, \quad G[e] = d,$$

which corresponds to two rooted trees: $\{a \to c \to b\}$ rooted at b, and $\{e \to d\}$ rooted at d. The contributed MST edges so far are $\{(a, c, 4), (b, c, 3), (d, e, 2)\}$.
Pointer jumping. Only a is forbidden, since $G[a] = c \neq b = G[G[a]]$. Advancing sets $G[a] := b$. Now every vertex points directly to its root: $G = (b, b, b, d, d)$, two rooted stars $\{a, b, c\} \to b$ and $\{d, e\} \to d$.
Recursive call. Contract each star to a single vertex, leaving the two-vertex graph $V' = \{b, d\}$ with the cross-star edges $(b, d, 9), (c, d, 7), (c, e, 11)$ reprojected onto (b, d) as parallel edges of weights $7, 9, 11$. Collapsing these parallel edges to the lightest one leaves a single contracted edge of weight 7, carrying the original edge $(c, d, 7)$. Both components' minimum outgoing edge is thus $(c, d, 7)$, yielding initialization $G[b] = b$, $G[d] = b$ — already a rooted star. The edge $(c, d, 7)$ is added, and the recursion terminates with a single vertex.

The final MST is $\{(d, e, 2), (b, c, 3), (a, c, 4), (c, d, 7)\}$, total weight 16.

8.9 Constrained Minimum Spanning Tree

In some applications a minimum spanning tree must be chosen subject to a side condition that the unconstrained algorithms of this chapter ignore. A network designer building a backbone may insist that certain already-installed links be reused; a road planner extending an existing network may be required to keep all current segments in service.

Mandatory edges

Let $M \subseteq E$ be a set of edges that *must* appear in the chosen spanning tree. The *mandatory-edge MST* problem asks for a minimum-weight spanning tree T with $M \subseteq T$. The problem is feasible iff M is acyclic.

The key observation is that the mandatory edges cannot simply be composed in conjunction with LLP-Kruskal. The forbidden predicate of LLP-Kruskal for edge $e_j = (v_k, v_l)$ checks connectivity only through edges $e_1, \ldots, e_{j-1}$ (those with smaller index, i.e., lighter weight). If a mandatory edge e_i has a larger index than e_j, committing e_i establishes connectivity that e_j's forbidden predicate cannot see. As a result, a conjunction could commit an edge e_j that forms a cycle with a mandatory edge of higher index.

The correct approach is a two-phase algorithm: first commit all mandatory edges, then run LLP-Kruskal on the residual graph.

Algorithm Mandatory-MST: Mandatory-edge commitment for MST

input: $M \subseteq E$: mandatory edges; G: array[1..m] of boolean, initially all false
output: G with all mandatory edges committed, or "infeasible" if M contains a cycle

```
1 forall v ∈ V :
2 |  parent[v] := v;   rank[v] := 0 ;          // each vertex starts in its own set
3 end
4 for e_j = (u, v) ∈ M do
5 |  if Find(u) = Find(v) then return "infeasible";
6 |  G[j] := true;   Union(u, v);
7 end
```

After Algorithm Mandatory-MST commits all mandatory edges and updates the Find-Union structure accordingly, LLP-Kruskal is run on the full edge set. Its forbidden predicate now checks connectivity through *all* committed edges — both mandatory and non-mandatory — so no redundant edge is added. That is, the constrained algorithm is:

1. Run Mandatory-MST(M, G). If infeasible, stop.
2. Run LLP-Kruskal(E, w, G) starting from the G produced by step 1.

This is equivalent to contracting the edges of M, computing an MST of the contracted graph, and uncontracting. The complexity is $O(m \log n)$: the mandatory phase commits at most $|M| \leq n - 1$ edges with $O(\alpha(n))$ per cycle test, and the Kruskal phase is $O(m \log n)$.

Theorem 8.14 *The mandatory-edge MST problem is feasible if and only if the mandatory-edge set M is acyclic. When feasible, running* Mandatory-MST *followed by* LLP-Kruskal *returns the minimum-weight spanning tree containing all edges in M, in $O(m \log n)$ time.*

Proof: If M contains a cycle, any spanning tree including M would also contain a cycle, which is impossible. Conversely, if M is acyclic, committing all edges of M produces a forest. Running LLP-Kruskal on the residual completes this forest to a spanning tree by adding the lightest non-cycle-forming edges, which is exactly the MST of the graph contracted by M.

■

8.10 Summary

Three classical algorithms compute a minimum spanning tree, each exploiting the cut and cycle properties in a different way. *Kruskal's* algorithm scans edges in non-decreasing weight order and admits each edge that does not close a cycle; the Find-Union data structure reduces the cycle check to near-constant amortized time. *Prim's* algorithm grows a single fragment from an arbitrary root, repeatedly adding the lightest edge that leaves the fragment — an instance of the nearest-neighbor-next strategy. *Boruvka's* algorithm extends all fragments in parallel: in each iteration, every current component contributes its minimum outgoing edge, so the number of components is at least halved, and the algorithm terminates in $O(\log n)$ iterations.

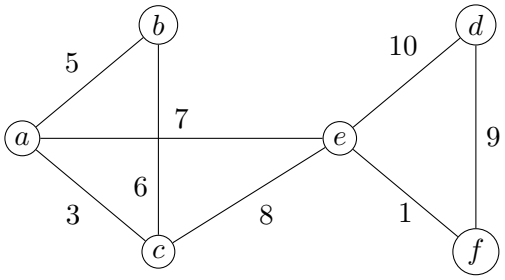
Table 8.1 summarizes the time complexity of the algorithms discussed in this chapter.

Algorithm	**Approach**	**Time Complexity**
Kruskal	add edges in weight order, skipping any that closes a cycle	$O(m \log n)$
Prim	grow a single fragment from a root, always across its lightest outgoing edge	$O(m \log n)$
Boruvka	grow every fragment at once, each across its own lightest outgoing edge	$O(m \log n)$
LLP-Kruskal	Kruskal as a predicate on the Boolean lattice of edges	$O(m \log n)$
LLP-Prim	Prim as a predicate on the edge attaching each vertex to the fragment	$O(m \log n)$
LLP-Boruvka	Boruvka as a predicate on the component leaders	$O(m \log n)$

Table 8.1: Minimum spanning tree algorithms discussed in this chapter, where n is the number of vertices and m is the number of edges.

8.11 Problems

8.1. **(MST Algorithm Comparison)** Consider the undirected weighted graph on six vertices $\{a, b, c, d, e, f\}$ shown below.



Find the minimum spanning tree using Kruskal's algorithm, Prim's algorithm, and Boruvka's algorithm.

8.2. **(Iterative Find)** Algorithm Find implements FIND recursively. Give an iterative version that performs path compression without using recursion or a stack.

8.3. **(Unique MST Distinct Weights)** Show that there is a unique minimum spanning tree in any weighted undirected graph when all edge weights are distinct.

8.4. **(Maximum Spanning Tree Negation)** Show that replacing every edge weight $w(e)$ by $-w(e)$ and then running Kruskal's (or Prim's) algorithm yields a *maximum* weight spanning tree of the original graph.

8.5. **(Excluded Edges MST)** An undirected weighted graph G has a set $F \subseteq E$ of *excluded* edges that must not appear in the spanning tree. Adapt Kruskal's algorithm to compute a minimum spanning tree of $(V, E \setminus F)$ (if one exists) and state a necessary and sufficient condition on F for a feasible spanning tree to exist.

8.6. **(Prim vs Dijkstra Weights)** Both Prim's algorithm and Dijkstra's algorithm repeatedly pick a vertex outside the current frontier whose "cost" is minimum and add it to the frontier. State precisely the *cost* each algorithm minimizes, and use this difference to explain why non-negative edge weights are irrelevant to Prim's correctness (it works for any real weights), whereas Dijkstra's requires non-negative edge weights.

8.7. **(Boruvka Halving Bound)**

Show that in Boruvka's algorithm, regardless of implementation, the number of connected components strictly decreases by at least a factor of two after each iteration. Deduce that the algorithm terminates in at most $\lceil \log_2 n \rceil$ iterations.

8.8. **(Heaviest MST Edge Cut)** Assume all edge weights are distinct. Let T be the MST of $G = (V, E, w)$ and let $e \in T$ be the *heaviest* edge in T. Removing e from T partitions V into two components S and $V \setminus S$. Prove that e is the *lightest* edge crossing the cut $(S, V \setminus S)$ in G.

8.9. **(Incremental MST: Edge Insertion)** An MST T of $\mathcal{G} = (V, E, w)$ has been computed. A new edge $e = (u, v)$ of weight w_e is then inserted. Give an algorithm that updates T in $O(|V|)$ time, exploiting the cycle lemma.

8.10. **(Incremental MST: Edge Deletion)** An MST T of $\mathcal{G}$ has been computed and an MST edge $e = (u, v) \in T$ is then deleted. Give an algorithm that updates T in $O(|E|)$ time, exploiting the cut lemma.

8.11. **(Dynamic MST Under Weight Changes)** The weight of an edge e changes from w to w'. Describe an MST update that handles both increases and decreases, and discuss when a full rebuild is unavoidable.

8.12. **(Fair MST: Bottleneck and Egalitarian Trees)** Define the *bottleneck spanning tree* as a spanning tree whose maximum-weight edge is minimized. Show that every MST is a bottleneck spanning tree, but the converse need not hold. Sketch an $O(|V| + |E|)$ algorithm for the bottleneck problem.

8.12 Bibliographic Remarks

The minimum spanning tree problem has an unusually long and well-documented history. Otakar Boruvka published the first MST algorithm in 1926 in connection with an electrical-grid layout problem in Moravia [Bor26]; the paper also contains the first proof that the MST is unique when edge weights are distinct. A detailed historical survey and an English translation of Boruvka's original papers is given by Nešetřil, Milková, and Nešetřilová [NMN01]. The classical algorithms bearing the names of Kruskal [Kru56] and Prim [Pri57] were developed independently in the 1950s. Prim's algorithm was in fact discovered earlier by Vojtěch Jarník in 1930, and later independently by Dijkstra [Dij59] as a specialization of his shortest-path algorithm — a coincidence that is not accidental: the two algorithms have the same control structure and differ only in the relaxation rule (see Section 8.5).

The efficient implementation of Kruskal's algorithm hinges on the union-find data structure. Tarjan [Tar75] proved the near-constant $O(\alpha(n))$ amortized bound for union-find with path compression and union by rank used in Algorithms Find and Union. An $O(m + n \log n)$ variant of Prim's algorithm, based on Fibonacci heaps, is due to Fredman and Tarjan [FT87]. For a textbook treatment of the three algorithms and their complexity, see Cormen et al. [CLRS09] or Kleinberg and Tardos [KT06].

A line of subsequent research has pushed the sequential complexity below $O(m \log n)$. Karger, Klein, and Tarjan [KKT95] gave a randomized algorithm running in expected $O(m)$ time — the first linear-time MST algorithm in any model. On the deterministic side, Chazelle [Cha00] obtained $O(m\,\alpha(m, n))$, inverse-Ackermann in the edge count, using an intricate soft-heap data structure. Pettie and Ramachandran [PR02] then gave an optimal algorithm whose running time matches the minimum number of comparisons needed to decide any MST instance, though its exact asymptotic complexity remains open.

In the parallel and distributed setting, the textbook of Sanders, Mehlhorn, Dietzfelbinger, and Dementiev [SMDD19] is our reference for the parallel Boruvka algorithm of Section 8.6. The pointer-jumping contraction technique that lies at its heart was popularized for parallel graph algorithms by Tarjan and Vishkin [TV85]. Practical shared-memory implementations

for sparse graphs are described by Bader and Cong [BC06]. The classical distributed MST algorithm of Gallager, Humblet, and Spira [GHS83] computes an MST in a message-passing model with $O(m + n \log n)$ message complexity.

The LLP formulations presented in this chapter recast the three classical algorithms as instances of a single lattice-linear predicate framework. The LLP-Prim algorithm in particular is due to Alves and Garg [AG22].

Appendix A

Lattice Theory

A.1 Introduction

This appendix covers the definitions of partial orders and lattices used throughout this book. For a thorough treatment, see [DP90, Grä03, Gar15].

This appendix is organized as follows. Section A.2 reviews relations, the building block of a partial order. Section A.3 defines reflexive and irreflexive partial orders and introduces Hasse diagrams. Section A.4 defines meet, join, and lattices. Section A.5 covers distributive lattices, join-irreducible elements, and Birkhoff's representation theorem.

A.2 Relations

A partial order is simply a relation with certain properties. A **relation** R over a set X is a subset of $X \times X$. For example, on $X = \{a, b, c\}$, one possible relation is

$$R = \{(a, c), (a, a), (b, c), (c, a)\}.$$

It is sometimes useful to visualize a relation as a directed graph on the vertex set X with an edge from x to y whenever $(x, y) \in R$.

A relation is **reflexive** if $(x, x) \in R$ for every $x \in X$, and **irreflexive** if $(x, x) \notin R$ for every $x \in X$. On the natural numbers $\mathbb{N}$, the relation *divides* is reflexive, while *less than* is irreflexive. Note that a relation need not be either reflexive or irreflexive.

A relation R is **symmetric** if $(x, y) \in R$ implies $(y, x) \in R$, and **antisymmetric** if $(x, y) \in R$ and $(y, x) \in R$ implies $x = y$. On $\mathbb{N}$, *less than or equal to* is antisymmetric. R is **transitive** if $(x, y) \in R$ and $(y, z) \in R$ imply $(x, z) \in R$; the relations *less than* and *equal to* on $\mathbb{N}$ are both transitive.

A relation that is reflexive, symmetric, and transitive is an **equivalence** relation. When R is an equivalence, we write $x \equiv_R y$ for $(x, y) \in R$, and the **equivalence class** $[x]_R = \{y \mid y \equiv_R x\}$. The equivalence classes form a **partition** of X. For example,

$$R = \{(x, y) \mid x \bmod 5 = y \bmod 5\} \tag{A.1}$$

is an equivalence on $\mathbb{N}$ that partitions it into five classes.

© 2026 Vijay K. Garg. *A Systematic Approach to Sequential Algorithms.* All rights reserved.

A.3 Partial Orders

A relation R is a **reflexive partial order** (or **non-strict partial order**) if it is reflexive, antisymmetric, and transitive. A relation R is an **irreflexive partial order** (or **strict partial order**) if it is irreflexive and transitive. The *divides* relation on $\mathbb{N}$ is a reflexive partial order, and *less than* on $\mathbb{N}$ is an irreflexive partial order. When R is a reflexive partial order, we write $x \leq_R y$ to denote $(x, y) \in R$. A reflexive partially ordered set, **poset** for short, is denoted by $(X, \leq)$. When R is irreflexive, we write $x <_R y$. Given an irreflexive partial order, we can define $x \leq y$ as $x < y$ or $x = y$, and vice versa. A relation is a **total order** if it is a partial order and for all distinct $x, y \in X$, either $(x, y) \in R$ or $(y, x) \in R$. The natural order on the integers is a total order, but *divides* on $\mathbb{N}$ is only a partial order.

Finite posets are often depicted using **Hasse diagrams**. For any $x, y \in X$, y **covers** x if $x < y$ and there is no z with $x < z < y$. We write $x <_c y$ to denote that y covers x (y is an **upper cover** of x; x is a **lower cover** of y). A Hasse diagram draws an edge from x to y whenever $x <_c y$, with x below y. For example, consider the poset

$$X \stackrel{\text{def}}{=} \{p, q, r, s\}; \quad \leq \stackrel{\text{def}}{=} \{(p, q), (q, r), (p, r), (p, s)\} \cup \{(x, x) : x \in X\}, \tag{A.2}$$

whose Hasse diagram is shown in Figure A.1. (The reflexive pairs (x, x) are listed once, in full, here; later examples omit them as understood.)

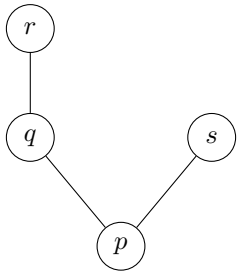


Figure A.1: Hasse diagram of a poset

Elements x, y are **comparable** if $x < y$ or $y < x$; otherwise they are incomparable, written $x \| y$. A poset is a **chain** if every pair is comparable, and an **antichain** if every pair is incomparable. In Figure A.1, $\{p, q, r\}$ is a chain and $\{q, s\}$ is an antichain. The **height** of a poset is the size of a longest chain, and the **width** is the size of a largest antichain. We use $\underline{n}$ for a chain of n elements and A_n for an antichain of n elements. A total order Q that extends a poset P is called a **linear extension** of P. For example, ordering the poset of Figure A.1 as $p < q < r < s$ gives one linear extension.

An element x is a **bottom element** (denoted $\bot$) if $x \leq y$ for all y, and a **top element** (denoted $\top$) if $y \leq x$ for all y. An element x is **minimal** if there is no y with $y < x$, and **maximal** if there is no y with $y > x$.

A.4 Lattices

Let $(X, \leq)$ be a poset. We define two operators on subsets of X: **meet** (also called **infimum**, or **inf**) and **join** (also called **supremum**, or **sup**). For $Y \subseteq X$ and $m \in X$, we say that

$m = \inf Y$ iff

1. $\forall y \in Y : m \leq y$, and
2. $\forall m' \in X : (\forall y \in Y : m' \leq y) \Rightarrow m' \leq m$.

Condition (1) says that m is a lower bound of Y, and condition (2) says that any other lower bound is at most m. For this reason, m is also called the **greatest lower bound** (glb) of Y. The infimum is unique whenever it exists, and m need not itself belong to Y. Dually, for $s \in X$, we say that $s = \sup Y$ iff

1. $\forall y \in Y : y \leq s$, and
2. $\forall s' \in X : (\forall y \in Y : y \leq s') \Rightarrow s \leq s'$.

s is also called the **least upper bound** (lub) of Y. We denote the glb of $\{a, b\}$ by $a \sqcap b$ and the lub of $\{a, b\}$ by $a \sqcup b$. In the natural numbers ordered by *divides*, the meet of two numbers is their gcd and the join is their lcm. The meet or join need not always exist: in Figure A.1, $\{q, s\}$ has no upper bound; in poset (iii) of Figure A.2 below, $\{b, c\}$ has upper bounds d and e but no least upper bound.

Definition A.1 (Lattice) *A poset $(X, \leq)$ is a* **lattice** *iff $\forall x, y \in X$: $x \sqcup y$ and $x \sqcap y$ exist.*

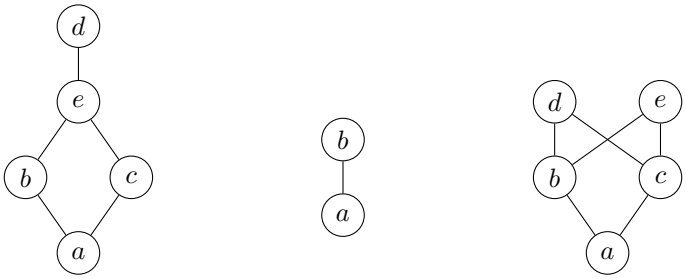


Figure A.2: Only the first two posets are lattices. In (iii), $\{b, c\}$ has no least upper bound.

The first two posets in Figure A.2 are lattices; the third is not, since $\{b, c\}$ has upper bounds d and e but no least upper bound. If only $\sqcup$ (resp. $\sqcap$) exists for all pairs, we have a *sup semilattice* (resp. *inf semilattice*). A lattice in which every subset has both a sup and inf is a *complete lattice*.

A nonempty $S \subseteq L$ is a **sublattice** of a lattice L iff $\forall a, b \in S$: $\sup(a, b) \in S$ and $\inf(a, b) \in S$, where sup and inf are computed in L.

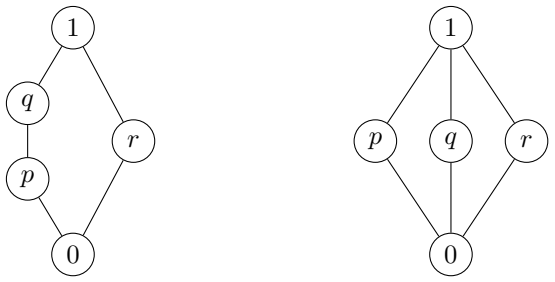
A subset $Y \subseteq X$ of a poset is a **down-set** (or **order ideal**) if $z \in Y$ and $y \leq z$ implies $y \in Y$. The set $D[x] = \{y \mid y \leq x\}$ is a **principal order ideal**. Dually, Y is an **up-set** (or **order filter**) if $y \in Y$ and $y \leq z$ implies $z \in Y$; $U[x] = \{y \mid x \leq y\}$ is a **principal order filter**. In Figure A.1, $D[r] = \{p, q, r\}$ and $U[p] = \{p, q, r, s\}$. We also write $D(x) = \{y \mid y < x\}$ and $U(x) = \{y \mid x < y\}$.

A.5 Distributive Lattices and Birkhoff's Theorem

An element x in a poset P is **join-irreducible** if $\forall Y \subseteq P$: $x = \sup Y$ implies $x \in Y$. The bottom element (if it exists) is not join-irreducible. The set of join-irreducible elements is denoted $J(P)$; in a finite lattice, $x \in J(P)$ iff x has exactly one lower cover. In poset (i) of Figure A.2, $J = \{b, c, d\}$ (and $e \notin J$ since $e = b \sqcup c$). By duality, **meet-irreducible** elements are denoted $M(P)$. The join-irreducibles form a basis: for any $x \in P$, $x = \sup(D[x] \cap J(P))$ (see [DP90]).

Definition A.2 (Distributive Lattice) *A lattice L is* **distributive** *if $\forall a, b, c \in L$: $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.*

Any power-set lattice is distributive, as is the lattice of natural numbers under *divides*. A finite lattice is nondistributive iff it contains a sublattice isomorphic to the Pentagon N_5 or the Diamond M_3 [Bir67, DP90]:



(a) Pentagon N_5 (b) Diamond M_3

Figure A.3: The two smallest nondistributive lattices.

Let $L_I(P)$ denote the set of order ideals of a poset P, ordered by inclusion. Birkhoff's Theorem establishes that every finite distributive lattice is isomorphic to the lattice of order ideals of its join-irreducible elements.

Theorem A.3 *[Birkhoff's Theorem [Bir67]] Let L be a finite distributive lattice. Then the mapping*

$$f : L \to L_I(J(L)), \quad a \mapsto f(a) = \{x \in J(L) : x \leq a\}$$

is an isomorphism from L onto $L_I(J(L))$.

The poset $J(L)$ can be exponentially more succinct than L itself. An example is shown in Figure A.4: the five-element lattice L has three join-irreducibles a, b, c (with $a < b$ and $a < c$ in $J(L)$), and $L_I(J(L)) \cong L$.

Birkhoff's Theorem also translates properties between L and $P = J(L)$: the height of L equals $|P| + 1$, L is Boolean iff P is an antichain, and L is a chain iff P is a chain. See [DP90,

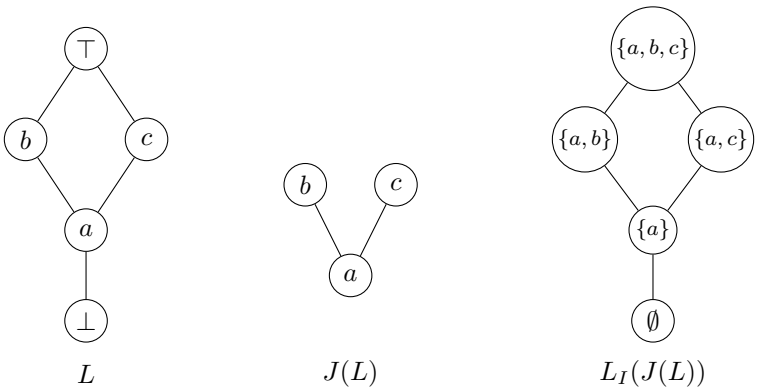


Figure A.4: A lattice L, its join-irreducibles $J(L)$, and the lattice of ideals $L_I(J(L))$.

Grä03] for proofs.

A.6 Problems

A.1. Give an example of a nonempty binary relation which is symmetric and transitive but not reflexive.

A.2. Show that if P and Q are posets defined on set X, then so is $P \cap Q$.

A.3. Draw the Hasse diagram of all natural numbers less than 10 ordered by the relation *divides*.

A.4. Show that if C_1 and C_2 are down-sets for any poset $(E, <)$, then so is $C_1 \cap C_2$.

A.5. Show that the join and meet operators are associative, commutative, and satisfy the absorption laws: $a \sqcup (a \sqcap b) = a$ and $a \sqcap (a \sqcup b) = a$.

A.6. Show that in a finite lattice, an element is join-irreducible iff it has only one lower cover.

A.7. Show that for a finite distributive lattice, L is Boolean iff $J(L)$ is an antichain.

A.7 Bibliographic Remarks

The first book on lattice theory was written by Garrett Birkhoff [Bir67]. The reader will find a discussion of origins of lattice theory and an extensive bibliography in the book by Grätzer [Grä71, Grä03]. The book by Davey and Priestley [DP90] provides an easily accessible account of the field. A treatment with an emphasis on computer science applications is given by Garg [Gar15]. Other books include those by Caspard, Leclerc, and Monjardet [CLM12], and Roman [Rom08].

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[Ada00] Hiroyuki Adachi. On a characterization of stable matchings. *Economics Letters*, 68(1):43–49, 2000.

[AG22] David R. Alves and Vijay K. Garg. Parallel minimum spanning tree algorithms via lattice linear predicate detection. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS Workshops 2022, Lyon, France, May 30 - June 3, 2022*, pages 774–782. IEEE, 2022.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AKG20] David R. Alves, Madan S. Krishnakumar, and Vijay K. Garg. Efficient parallel shortest path algorithms. In *19th International Symposium on Parallel and Distributed Computing, ISPDC 2020, Warsaw, Poland, July 5-8, 2020*, pages 188–195. IEEE, 2020.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[AMOT90] Ravindra K Ahuja, Kurt Mehlhorn, James Orlin, and Robert E Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM (JACM)*, 37(2):213–223, 1990.

[Apo76] Tom M. Apostol. *Introduction to Analytic Number Theory*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1976.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979. Introduces a linear-time algorithm for 2SAT.

[AS98] Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.

[AS99] Atila Abdulkadiroğlu and Tayfun Sönmez. House allocation with existing tenants. *Journal of Economic Theory*, 88(2):233–260, 1999.

[BC06] David A. Bader and Guojing Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[BCH90] Endre Boros, Yves Crama, and Peter L Hammer. Polynomial-time inference of all valid implications for horn and related formulae. *Annals of Mathematics and Artificial Intelligence*, 1(1-4):21–32, 1990.

[BDM12] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. SIAM, revised reprint edition, 2012.

[Bel52] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[Ber92] Dimitri P. Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1(1):7–66, 1992.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bir46] Garrett Birkhoff. Three observations on linear algebra. *Universidad Nacional de Tucumán Revista A*, 5:147–151, 1946.

[Bir67] G. Birkhoff. *Lattice Theory*. American Mathematical Society, Providence, R.I., 1967. third edition.

[Bla77] Robert G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2(2):103–107, 1977.

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3(3):37–58, 1926. In Czech. English translation available in: *Sources in the History of Mathematics and Physical Sciences*, vol. 1, 1985.

[BS96] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory, Volume 1: Efficient Algorithms*. MIT Press, Cambridge, MA, 1996.

[BYE85] Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–46, 1985.

[CBDGF95] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. on Programming Languages and Systems*, 17(1):157–179, January 1995.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[CG98] Craig M Chase and Vijay K Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[Chv79] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[CLM12] Nathalie Caspard, Bruno Leclerc, and Bernard Monjardet. *Finite ordered sets: concepts, results and uses*, volume 144. Cambridge University Press, 2012.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition, 2009.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[Dan63] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963. Introduces the simplex method and foundational linear programming theory.

[DdFdFS03] Vânia M.F. Dias, Guilherme D. da Fonseca, Celina M.H. de Figueiredo, and Jayme L. Szwarcfiter. The stable marriage problem with restricted pairs. *Theoretical Computer Science*, 306(1):391 – 405, 2003.

[DG84] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984. Presents a linear-time algorithm for Horn formula satisfiability.

[DGS86] Gabrielle Demange, David Gale, and Marilda Sotomayor. Multi-item auctions. *Journal of Political Economy*, 94(4):863–872, 1986.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. 51*, pages 161–166, 1950.

[Din70] Yefim A. Dinitz. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11(5):1277–1280, 1970.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[DS05] Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with $0,1$-vertices. *Journal of Research of the National Bureau of Standards B*, 69:125–130, 1965.

[Ege31] E. Egerváry. On combinatorial properties of matrices. *Mat. Lapok*, 38:16–28, 1931.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[Eri19] Jeff Erickson. *Algorithms*. Self-published, 2019. Available at `http://jeffe.cs.illinois.edu/teaching/algorithms/`.

[For56] L. A. Ford. Network flow theory. Technical report, The Rand Corporation, 1956.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[Ful56] D.R. Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. *Proc. American Math. Society*, pages 701–702, 1956.

[Gar15] Vijay K Garg. *Lattice Theory with Computer Science Applications*. Wiley, New York, NY, 2015.

[Gar17] Vijay K. Garg. Brief announcement: Applying predicate detection to the stable marriage problem. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 52:1–52:3. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[Gar20a] Vijay K. Garg. A generalization of Teo and Sethuraman's median stable marriage theorem, 2020.

[Gar20b] Vijay K. Garg. Predicate detection to solve combinatorial optimization problems. In Christian Scheideler and Michael Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 235–245. ACM, 2020.

[Gar21] Vijay K. Garg. A lattice linear predicate parallel algorithm for the housing market problem. In Colette Johnen, Elad Michael Schiller, and Stefan Schmid, editors, *Stabilization, Safety, and Security of Distributed Systems - 23rd International Symposium, SSS 2021, Virtual Event, November 17-20, 2021, Proceedings*, volume 13046 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2021.

[Gar22] Vijay K. Garg. A lattice linear predicate parallel algorithm for the dynamic programming problems. In *Proc. of the Int'l Conf. on Distributed Computing and Networking (ICDCN)*, Delhi, India, 2022. Springer-Verlag.

[Gar23] Vijay Kumar Garg. Keynote talk: Lattice linear predicate algorithms for the constrained stable marriage problem with ties. In *24th International Conference on Distributed Computing and Networking, ICDCN 2023, Kharagpur, India, January 4-7, 2023*, pages 2–11. ACM, 2023.

[Gau01] Carl Friedrich Gauss. Disquisitiones arithmeticae. *English Translation*, 1801. English translation, 1966. Introduces the Chinese Remainder Theorem in modern form.

[GC95] V. K. Garg and C. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the IEEE Intnatl. Conf. on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.

[GH23] Vijay K. Garg and Changyong Hu. Improved paths to stability for the stable marriage problem. In Shlomi Dolev and Baruch Schieber, editors, *Stabilization, Safety, and Security of Distributed Systems: 25th International Symposium, SSS 2023*, volume 14310 of *Lecture Notes in Computer Science*, pages 126–140, Jersey City, NJ, USA, 2023. Springer. Preprint available at https://arxiv.org/abs/2007.07121.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5(1):66–77, January 1983.

[GI89] Dan Gusfield and Robert W Irving. *The stable marriage problem: structure and algorithms*. MIT press, 1989.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[GK21] Arya Tanmay Gupta and Sandeep S. Kulkarni. Extending lattice linearity for self-stabilizing algorithms. In *Stabilization, Safety, and Security of Distributed Systems - 23rd International Symposium, SSS 2021, Virtual Event, November 17-20, 2021, Proceedings*, pages 365–379. Springer, 2021.

[GK23] Arya Tanmay Gupta and Sandeep S Kulkarni. Lattice linearity of multiplication and modulo, 2023.

[GK25] Arya Tanmay Gupta and Sandeep S. Kulkarni. Tolerance to asynchrony in algorithms for multiplication and modulo. *Theoretical Computer Science*, 1024:114914, 2025.

[GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, 2 edition, 1994.

[Grä71] George Grätzer. *Lattice theory*. W.H. Freeman and Company, San Francisco, 1971.

[Gra72] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[Grä03] George Grätzer. *General Lattice Theory*. Birkhäuser, Basel, 2003.

[GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[GS24] Vijay K. Garg and Robert P. Streit. Parallel algorithms for equilevel predicates. In *Proceedings of the 25th International Conference on Distributed Computing and Networking, ICDCN 2024, Chennai, India, January 4-7, 2024*, pages 104–113. ACM, 2024.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[Hås01] Johan Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, 2001.

[HG21] Changyong Hu and Vijay K. Garg. Characterization of super-stable matchings. In Anna Lubiw and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 17th International Symposium, WADS 2021, Virtual Event, August 9-11, 2021, Proceedings*, volume 12808 of *Lecture Notes in Computer Science*, pages 485–498. Springer, 2021.

[HK56] A. J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedra. *Linear Inequalities and Related Systems (Annals of Mathematics Studies 38)*, pages 223–246, 1956.

[HK71] John E. Hopcroft and Richard M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 122 –125, oct. 1971.

[HK73] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[HMRS95] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunction of local predicates. Technical Report 2731, IRISA, Rennes, France, November 1995.

[Hoa61] Charles Antony Richard Hoare. Algorithm 64: quicksort. *Communications of the ACM*, 4(7):321, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[HS74] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974.

[HSR01] Ellis Horowitz, Sartaj Sahni, and Sanguthevar Rajasekaran. *Fundamentals of Computer Algorithms (Computer software engineering series)*. Galgotia Publications, 2001.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[HW08] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, Oxford, 6 edition, 2008. Revised by D. R. Heath-Brown and J. H. Silverman.

[HZ79] Aanund Hylland and Richard Zeckhauser. The efficient allocation of individuals to positions. *Journal of Political economy*, 87(2):293–314, 1979.

[IK75] Oscar H Ibarra and Chul E Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.

[IM08] Kazuo Iwama and Shuichi Miyazaki. A survey of the stable marriage problem and its variants. In *Informatics Education and Research for Knowledge-Circulating Society, 2008. ICKS 2008. International Conference on*, pages 131–136. IEEE, 2008.

[JF56] L. R. Ford Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[Joh74] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[Kan39] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science (English translation, 1960)*, 6(4):366–422, 1939.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. Presents the interior-point method, a polynomial-time alternative to simplex.

[KG05] Sujatha Kashyap and Vijay K. Garg. Intractability results in predicate detection. *Inf. Process. Lett.*, 94(6):277–282, 2005.

[Kha79] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20:191–194, 1979.

[KKT95] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[KM72] Victor Klee and George J. Minty. How good is the simplex algorithm? In *Inequalities III: Proceedings of the Third Symposium on Inequalities*, pages 159–175, 1972.

[Knu71] Donald E. Knuth. Optimum binary search trees. *Acta informatica*, 1(1):14–25, 1971.

[Knu97a] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 3rd edition, 1997.

[Knu97b] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997. Provides a detailed treatment of Euclid's algorithm and its applications.

[Knu97c] Donald Ervin Knuth. *Stable marriage and its relation to other combinatorial problems: An introduction to the mathematical analysis of algorithms*, volume 10. American Mathematical Soc., 1997.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[KO62] Anatoly Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1962.

[Kön31] D. König. Graphen und matrizen. *Mat. es Fiz. Lapok*, 38:116–119, 1931.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, 2006.

[Kuh55] Harold W Kuhn. Naval research logistics quarterly. *The hungarian method for the assignment problem, journal*, 2(1-2):83–97, 1955.

[KV18] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, 6th edition, 2018.

[Lev73] Leonid A. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973. Translated from Russian: Problemy Peredachi Informatsii, 9(3), 115–116.

[Lew78] Harry R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the ACM*, 25(1):134–135, 1978.

[LNS82] J-L. Lassez, V. L. Nguyen, and E. A. Sonenberg. Fixed point theorems and semantics: a folk tale. *Information Processing Letters*, 14(3):112–116, 1982.

[Lov75] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[LP86] László Lovász and Michael D. Plummer. *Matching Theory*, volume 121 of *North-Holland Mathematics Studies*. Elsevier, 1986.

[LRK76] Eugene L. Lawler and A. H. G. Rinnooy Kan. A single-machine scheduling problem with deadlines and lateness penalties. *Management Science*, 22(11):1237–1242, 1976.

[Man13] David F. Manlove. *Algorithmics of Matching Under Preferences*, volume 2 of *Series on Theoretical Computer Science*. World Scientific, Singapore, 2013.

[Men27] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[MM11] Cristopher Moore and Stephan Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[Mun57] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38, 1957.

[NMN01] Jaroslav Nesetril, Eva Milkova, and Helena Nesetrilova. Otakar Boruvka on minimum spanning tree problem Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3):3–36, 2001.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PH77] Franco P. Preparata and Se June Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

[PQ80] Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Studies*, 13:8–16, 1980.

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002.

[Pri57] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[PS08] Parag A. Pathak and Tayfun Sönmez. Leveling the playing field: Sincere and sophisticated players in the Boston mechanism. *American Economic Review*, 98(4):1636–1652, 2008.

[Ram97] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, June 1997.

[Rom08] Steven Roman. *Lattices and ordered sets*. Springer, 2008.

[Ros11] Kenneth H. Rosen. *Elementary Number Theory and Its Applications*. Pearson, 6th edition, 2011. Offers an accessible explanation of the Chinese Remainder Theorem.

[Rot64] Gian-Carlo Rota. On the foundations of combinatorial theory I: Theory of Möbius functions. *Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete*, 2(4):340–368, 1964.

[Rot82] Alvin E Roth. Incentive compatibility in a market with indivisible goods. *Economics letters*, 9(2):127–132, 1982.

[RP77] Alvin E Roth and Andrew Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4(2):131–137, 1977.

[RS92] Alvin E Roth and Marilda Sotomayor. Two-sided matching. *Handbook of game theory with economic applications*, 1:485–541, 1992.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[RSU04] Alvin E. Roth, Tayfun Sönmez, and M. Utku Ünver. Kidney exchange. *Quarterly Journal of Economics*, 119(2):457–488, 2004.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer, 2003.

[SG26a] Robert Streit and Vijay K. Garg. Constrained cuts, flows, and lattice-linearity. In *Proc. Workshop on Parallel/Distributed Combinatorics and Optimization (PDCO), with IPDPS 2026*, New Orleans, LA, USA, May 2026. arXiv preprint arXiv:2512.18141.

[SG26b] Robert P. Streit and Vijay K. Garg. Invited paper: Approximation algorithms via lattice-linear predicates. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Gothenburg, Sweden, October 2026.

[SH75] Michael Ian Shamos and Dan Hoey. Closest-point problems. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 151–162, 1975.

[Sho09] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, Cambridge, 2 edition, 2009.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2013.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, 2008.

[SMDD19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019.

[SS71] Lloyd S Shapley and Martin Shubik. The assignment game i: The core. *International Journal of game theory*, 1(1):111–130, 1971.

[SS74] Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

[SS95] S. D. Stoller and F. B. Schneider. Faster possibility detection by combining two approaches. In *Proc. of the 9th Intnatl. Workshop on Distributed Algorithms*, pages 318–332, France, September 1995. Springer-Verlag.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[Tar55] Alfred Tarski. A lattice-theoretic fixed point theorem and its applications. *Pacific J Math*, 5:285–309, 1955.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[Tho00] Mikkel Thorup. On ram priority queues. *SIAM Journal on Computing*, 30(1):86–109, 2000.

[TS98] Chung-Piaw Teo and Jay Sethuraman. The geometry of fractional stable matchings and its applications. *Mathematics of Operations Research*, 23(4):874–891, 1998.

[TV85] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[Van20] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 5th edition, 2020. Covers simplex, duality, and interior-point methods with modern applications.

[Vaz01] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, Germany, 2001.

[VD92] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *30th Annual Allerton Conf. on Commun., Control and Computing*, pages 137–146, Allerton, Illinois, October 1992.

[vN47] John von Neumann. Discussion of a maximum problem. *Institute for Advanced Study, Princeton (manuscript)*, 1947.

[VV85] Leslie G. Valiant and Vijay V. Vazirani. NP is as easy as detecting unique solutions. In Robert Sedgewick, editor, *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*, pages 458–463. ACM, 1985.

[VV89] John H. Vande Vate. Linear programming brings marital bliss. *Operations Research Letters*, 8(3):147–153, 1989.

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[ZG19] Xiong Zheng and Vijay K. Garg. Parallel and distributed algorithms for the housing allocation problem. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[Zho90] Lin Zhou. On a conjecture by Gale about one-sided matching problems. *Journal of Economic Theory*, 52(1):123–135, 1990.

Algorithm Index

2-SAT, **326**

ApproxSetCover, **263**
ApproxVertexCover, **254**

BellmanFord, **116**
BFS-Traversal, **75**
BinarySearch, **9**
BipartiteMatching, **217**
Boruvka, **145**
BubbleSort, **54**
BuildHeap, **14**

ClosestPair, **163**
CountingInversions, **165**
CountSort, **62**
CRT-via-ExtGCD, **339**

DFS-Traversal, **77**
Dijkstra, **112**

EdmondsKarp, **205**

Factorial, **6**
FFT, **173**
FFTMultiply, **172**
Find, **137**
FloydWarshall, **119**
FordFulkerson, **198**
FPTAS-Knapsack, **265**
FractionalKnapsack, **102**

Gale–Shapley, **44**
GCD, **5**

Hanoi, **10**
HeapExtractMin, **13**
Heapify, **12**
HeapInsert, **13**
HuffmanCoding, **99**

InsertionSort, **55**
Interval, **92**
IntervalPartition, **95**

Johnson, **122**

Karatsuba, **168**
Knapsack, **188**
Kosaraju-SCC, **83**
Kruskal, **139**

Layering, **79**
LinearSearch, **7**
LIS, **184**
LLP, **27**
LLP-Antichain, **219**
LLP-ApproxKnapsack, **267**
LLP-ArrayGCD, **33**
LLP-Assignment, **309**
LLP-BellmanFord, **125**
LLP-BFS, **85**
LLP-Boruvka, **151**
LLP-ConjunctiveDetection, **352**
LLP-CountInversions, **177**
LLP-Dijkstra, **126**
LLP-Edge-Relaxation, **124**
LLP-ExtGCD, **337**
LLP-FloydWarshall, **127**
LLP-FractionalKnapsack, **107**
LLP-GCD, **335**
LLP-HornSAT, **321**
LLP-Housing-Market, **278**

LLP-IntervalScheduling, **104**
LLP-JobScheduling-Ensure, **29**
LLP-JobScheduling-Fixed, **30**
LLP-JobScheduling-Forbidden, **29**
LLP-Johnson, **128**
LLP-Knapsack, **191**
LLP-Kruskal, **148**
LLP-Layering, **86**
LLP-LexMatchVertexCover, **257**
LLP-LIS, **189**
LLP-MinMaxLateness, **105**
LLP-NearestNeighbor, **176**
LLP-OptimalBinarySearchTree, **190**
LLP-Prim, **150**
LLP-Sequential, **32**
LLP-Sort-Adjacent, **65**
LLP-Sort-Pairwise, **66**
LLP-StableMarriage, **47**
LLP-StableMarriage-Spec, **46**
LLP-Traversal, **85**
LLP-Traversal-BFS-priority, **86**
LLP-WeightedIntervalScheduling, **189**
LLP-WeightedVertexCover, **260**

Mandatory-MST, **153**
MaxElement, **6**
MergeSort, **59**
MergeSort-Intro, **10**
MergeTwo, **60**
MinMaxLateness, **96**

NestedSum, **8**

OptimalBinarySearchTree, **186**

PairSum, **8**
Prim, **142**

QuickSort, **56**

RadixSort, **64**
RecFib, **180**
RecFibMemo, **181**
Regular, **36**

Seq-AugmentingPath, **216**
Strassen, **169**

ThreeWayPartition, **57**
Traversal, **74**
TTC, **275**

Union, **137**

VertexCoverFromMatching, **220**

WeightedIntervalScheduling, **182**

General Index

α-forbidden, **26**
s-t cut, **197**
2-SAT, 323
3-Colorability, 244
3-SAT, 233
3D-Matching, **241**

adjacency list, 71
adjacency matrix, 71
advance clause, 28
always section, 28
antichain, 221, 362
antisymmetric relation, 361
APSP, 111
Arithmetization of Horn Clauses, 322
assignment problem, 305
augmenting path, **197**, 213
average case, 8

Bézout pair, 336
Bellman, 14, 22, 115, 179
BellmanFord Algorithm, 115
Big-O notation, 6
binary search, 9
binary search tree, 11
bipartite, 88
bipartite graph, 195, 213, 217, 248, 255, 305
Birkhoff's theorem, 364
blocking pair, 42
Boruvka's Algorithm, 144
Borůvka, 134, 150
bottom element, 362
Bubble Sort, 54
build-heap, 14

chain, 362
Chain Matrix Multiplication, 186
Chain Partition of a Poset, 217
Chinese Remainder Theorem, 338
clauses, 233
Clique, **239**
Closest Pair of Points in the Euclidean Space, 161
comparison-based sorting, 55, 59
Complementary Slackness Conditions, 290
complete lattice, 363
complexity class P, **230**
complexity class co-NP, **245**
Conjunctive Normal Form (CNF), 233
coNP-hard, 354
Constrained Market Clearing Price Problem, 311
Continuous Optimization Problem, 24
core, 273
Counting Inversions, 164
cover relation, 362

deferred acceptance algorithm, 43
Detecting Conjunctive Predicates, 351
Dijkstra, 21, 57, 71, 125, 128, 144
Dijkstra's Algorithm, 112
Dilworth's Theorem, 221
directed acyclic graph (DAG), 78
disjoint set, 136, **241**
Disjoint Set Union, 136
distributive lattice, xvii, 22, 41, 127, 147, 149, 159, 179, 208, 306, 364
divide and conquer, 9
down-set, 363
dual of Lattice-Linearity Property, **35**

Dual-Horn formulas, 322

Earliest Deadline First (EDF) algorithm, 96
edge relaxation, 113
Edge-Menger, 224
Edmonds, 15, 211
EdmondsKarp Algorithm, 204
efficient advancement property, 26
ensure section, 28
equivalence, 361
Euler's totient, 333
Euler's Totient function, 341
exchange argument, 91
Extended GCD Algorithm, 336

Fast Fourier Transform, 171
feasible solution, 23
Find-Union data structure, 136
fixed point, 34, 176, 179
Flow conservation, **196**
flow network, **196**
Floyd, 14
FloydWarshall algorithm, 118, 119
forbidden index, 3, 33, 111, 179
forbidden state, 24
FordFulkerson algorithm, 197
fragment, 133
fully polynomial-time approximation scheme, **264**

Gale, 21, 41, 273, 311
Gale–Shapley algorithm, 42
glb, 363
graph theory, 71
Greatest Common Divisor (GCD) Algorithm, 334
greatest lower bound, 363
greedy algorithm, 91

Hall's Theorem, 214
Hamiltonian Cycle, 244
Hamiltonian Path, 244
Hasse diagrams, 362
heap, 11
heapify, 14
height, 362
Hopcroft, 194, 207, 228
Horn clause, 318
Horn formula, 318
Horn Satisfiability, 318
housing market problem, 273
Huffman, 14
Huffman coding, 19
Huffman tree, 98
Hungarian algorithm, 305

incomparable, 362
Independent Set, **231**
inf, 362
inf semilattice, 363
infimum, 362
Insertion Sort, 55
interval partition problem, 94
Interval Scheduling, 92, 181
inversion vector, 64
irreflexive, 361
irreflexive partial order, 362

join, 362
join-irreducible, 364

König-Egerváry Theorem, 219
Karatsuba, 15, 159
Karatsuba's Multiplication Algorithm, 167
Karp, 15, 207, 228, 241
Knapsack, 187, 244, 253
Knuth, 18, 68, 345
Kosaraju's algorithm, 82
Kruskal, 15, 22, 147
Kruskal's Algorithm, 138
Kuhn-Munkres algorithm, 305
König's theorem, 213, 248

Lattice, 363
lattice-linear, 24
Lattice-Linear Predicate, 21
lattice-linear predicate, 3
lattice-linear predicates, 22, 333
layering, 78
least common multiple, 336
least upper bound, 363

level-based predicate detection, 353
linear programming, 285
linked list, 10
LIS problem, 183
literal, 233
LLP, 21
LLP Notation, 28
lower cover, 362
lub, 363

Möbius function, 340
Möbius inversion formula., 344
man-saturating matching, 42
market clearing price, 306
Master Theorem, 160
matching problems, 213
maximal, 362
maximum flow problem, 195
maximum lateness, 96
maximum matching, 195
meet, 362
meet-irreducible, 364
memoization, 179, 180
Menger's theorem, 206, 213, 223
Merge Sort, 58
minimal, 362
minimum cut, 195, 202, 224
minimum spanning tree (MST), 133
minimum-cost bipartite matching, 306
Multiplicative Functions, 340
multiplicative inverse, 338
multiplicative order, 15

nearest-neighbor-next, 141
Non-Isomorphism problem, **247**
Non-Lattice-Linear Predicate, 25
non-overlapping jobs, 93
NP, **233**
NP-complete, 219, 229, 253, 317, 349
NP-completeness, 229
NP-hard, 101, 133, 230, 253

Omega notation, 6
Optimal Binary Search Tree, 185
order filter, 363
order ideal, 363
overdemanded, 214

partition, 56, 361
partitioning, 14, 56, 95, 213
perfect matching, 214
pivot, 159
Planar Convex Hull, 166
pointer jumping, 152
polynomial-time approximation scheme, **264**
Polytime Reductions, 230
poset, 362
predicate detection, 350
predicate detection problem, 23, 349
Prefix Sum Problem, 24
Prim, 15, 22, 133
Prim's Algorithm, 141
principal order filter, 363
principal order ideal, 363
priority queue, 19, 31, 86, 131, 146
Proposal Vector Lattice, 42
pseudopolynomial, 202, 264

queue, 11, 86
Quicksort, 56

Radix Sort, 62
recurrence relation, 9
recursion tree, 160, 180
reflexive, 361
reflexive partial order, 362
Regular Predicate, **35**
relation, 361
residual graph, 198
Residual Network, **197**
reweighting, 127

SAT, 233
satisfiability problem, 350
self-reducibility, **229**
Set Cover, **233**, 253
Shapley, 21, 41, 273, 316
shortest-edge-next, 138
single source shortest path (SSSP) problem, 111
space complexity, 8

stable, 61, 63
stable marriage, 43
Stable Matching Problem, 41
stack, 11
state vector, 3
Strassen, 15
Strassen's Matrix Multiplication, 169
strict partial order, 362
Strict Split of a Poset, 218
Strong Duality, 290
strongly connected components (SCCs), 71, 82, 324, 325
sublattice, 363
subset sum problem, **241**
sup, 362
sup semilattice, 363
supremum, 362
symmetric relation, 361

Tarjan, 89, 131, 157, 211
Tautology problem, **246**
Theta notation, 6
top element, 362
Top Trading Cycle (TTC) algorithm, 274
topological sort, 81, 327
total order, 362
transitive, 361
Traveling Salesman, 244

Unsatisfiability (UNSAT) problem, **247**
up-set, 363
upper cover, 362

Valiant–Vazirani, **356**
var section, 28
Vertex Cover, **231**, 253
vertex cover, 293
Vertex-Disjoint Paths, 223

Warshall, 14
Weak Duality, 289
Weighted Interval Scheduling, 181
width, 362

Algorithms through the lens of lattice-linear predicates

This book presents the classical algorithms of computer science from two complementary perspectives. The first is the traditional one—greedy choices, divide-and-conquer recursions, dynamic-programming tables, augmenting paths. The second recasts each problem as the search for an extremal element of a distributive lattice that satisfies a lattice-linear predicate. From this single, unifying viewpoint, the book derives algorithms for sorting, graph traversal, shortest paths, minimum spanning trees, dynamic programming, network flow, bipartite matching, stable matching, housing allocation, assignment, satisfiability, number theory, and more.

The lattice-search formulation makes correctness proofs shorter, exposes natural parallelism, and turns algorithmic design into a systematic activity: identify the lattice, write down the predicate, check lattice-linearity, and read off the advance rule. Every algorithm in the book ships as runnable code on a companion website—in Java, Python, C++, and JavaScript—together with interactive demos that step through each LLP loop on small concrete instances. A solution manual is available to instructors on request.

Vijay K. Garg

Cullen Trust for Higher Education Endowed Professor
Department of Electrical and Computer Engineering,
The University of Texas at Austin

Vijay K. Garg leads research on parallel and distributed computing, lattice theory, and algorithms at The University of Texas at Austin. He is the author of *Introduction to Lattice Theory with Computer Science Applications* (Wiley, 2015), *Concurrent and Distributed Computing in Java* (Wiley, 2004), *Elements of Distributed Computing* (Wiley, 2002), *Principles of Distributed Systems* (Springer, 1996), and *Modeling and Control of Logical Discrete Event Systems* (Springer, 1995), and has published extensively on predicate detection, parallel and distributed algorithms, and discrete event systems.



Companion website: `http://www.ece.utexas.edu/~garg`