Algorithms

Workbook

University of Waterloo

CS 341, Fall 2026

Revised 22 October 2026

Preface

This workbook accompanies the *Algorithms: Reference Notes* for CS 341. Each problem asks for an algorithm, a proof of correctness, and a proof of the stated running-time bound.

Part I contains the problems, Part II the hints, and Part III the model solutions.

Workbook chapter	Problems	Notes part
Divide & Conquer	1–24	I
Graph Search	25–57	II
Greedy Algorithms	58–75	III
Dynamic Programming	76–114	IV
Max Flow and Min Cut	115–141	V

We use the same conventions and notation as in the Reference Notes. You may use algorithms and results from those notes in your solutions without reproving them.

I encourage you to use this workbook in two ways. First, practise problem solving. For each problem, look for the right algorithmic technique and work out how to prove that your algorithm is correct and meets the stated time bound. This is the familiar way of using a problem set.

Second, practise writing clear solutions. In the design of algorithms, communicating our results is just as important as solving problems. The two tasks call for different skills, and both require practice. Sit down with pen and paper and write out the algorithm, proof of correctness, and time analysis without referring to the solution in the book. This is a valuable exercise even when you already know how to solve the problem. The more you practise, the easier it becomes, so I strongly recommend doing this regularly.

One last note: this version of the workbook is preliminary and will continue to be updated throughout the rest of the term.

Eric Blais
Waterloo, October 2026

Contents

I Problems **1**

1 Divide & Conquer 2

2 Graph Search 7

3 Greedy Algorithms 13

4 Dynamic Programming 17

5 Max Flow and Min Cut 25

II Hints **31**

6 Divide & Conquer 32

7 Graph Search 36

8 Greedy Algorithms 41

9 Dynamic Programming 44

10 Max Flow and Min Cut 50

III Solutions **54**

11 Divide & Conquer 55

12 Graph Search 79

13 Greedy Algorithms 112

14 Dynamic Programming 130

15 Max Flow and Min Cut 169

List of Problems 196

Index 200

Part I

Problems

Chapter 1

Divide & Conquer

Merge Sort

1. Stable Sorting

Input: An array of n records, each with an integer key.
Output: The records in nondecreasing key order, with equal-key records in their original relative order.

Target: $O(n \log n)$ time.

Hint · p. 32 Solution · p. 55

2. Merging Several Arrays

Input: $k \geq 2$ nonempty sorted integer arrays, with N entries in total.
Output: One sorted array containing all their entries, with multiplicity.

Target: $O(N \log k)$ time.

Hint · p. 32 Solution · p. 56

3. Multiset Intersection

Input: Sorted integer arrays A and B, of lengths n and m.
Output: A sorted array in which each integer x occurs exactly $\min(a_x, b_x)$ times, where a_x, b_x are its multiplicities in A, B.

Target: $O(n + m)$ time. The output may be empty.

Hint · p. 32 Solution · p. 57

Counting Inversions

4. Significant Inversions

Input: An integer array A of length n.
Output: The number of pairs $i < j$ such that $A[i] > 2A[j]$.

Target: $O(n \log n)$ time, including when entries are negative.

Hint · p. 32 Solution · p. 58

5. Weighted Inversions

Input: An integer array A and nonnegative integer weights w, both of length n.
Output: The sum $\sum_{i<j,\, A[i]>A[j]} w[i]w[j]$.

Target: $O(n \log n)$ time.

Hint · p. 32 Solution · p. 59

6. Disagreement between Rankings

Input: Two permutations P, Q of the same set of n distinct integers.
Output: The number of unordered pairs of values whose relative orders differ in P and Q.

Target: $O(n \log n)$ time. The values need not lie in $[1, n]$.

Hint · p. 33 Solution · p. 60

Maximum Subarray

7. Circular Maximum Subarray

Input: An integer array A of length n, whose indices are arranged in a circle.
Output: A starting index and a length $1 \le \ell \le n$ specifying a consecutive block of maximum sum. Indices after n continue at 1.

Target: $O(n \log n)$ time. The chosen block must be nonempty.

Hint · p. 33 Solution · p. 61

8. Two Disjoint Maximum Subarrays

Input: An integer array A of length $n \ge 2$.
Output: Two nonempty intervals $[i, j]$ and $[u, v]$, with $j < u$, maximizing $\sum_{t=i}^{j} A[t] + \sum_{t=u}^{v} A[t]$. Adjacent intervals are allowed.

Target: $O(n)$ time.

Hint · p. 33 Solution · p. 62

9. Maximum Submatrix

Input: An n-by-m integer matrix A, with $n, m \ge 1$.
Output: The row and column endpoints of a nonempty rectangle of consecutive rows and consecutive columns with maximum sum.

Target: $O(n^2 m \log m)$ time.

Hint · p. 33 Solution · p. 63

Median and Selection

10. Selection with Repetitions

Input: An integer array A of length n and a rank $1 \le k \le n$.
Output: The value in position k of a sorted rearrangement of A. Each occurrence counts toward the rank.

Target: $O(n)$ time.

Hint · p. 33 Solution · p. 64

11. Multiple Selection

Input: An array A of n distinct integers and sorted ranks $1 \le k_1 < \cdots < k_r \le n$.
Output: The entries of these r ranks, in rank order.

Target: $O(n \log r)$ time.

Hint · p. 33 Solution · p. 65

12. Weighted Median

Input: Distinct integer values $a_1, \ldots, a_n$ with positive integer weights $w_1, \ldots, w_n$.
Output: An input value a_j such that the total weight strictly below a_j and the total weight strictly above a_j are each at most half the total weight.

Target: $O(n)$ time.

Hint · p. 34 Solution · p. 66

Closest Pair

13. Closest Pair with Equal Coordinates

Input: $n \ge 2$ points with integer coordinates, allowing equal x-coordinates and coincident points.
Output: Two distinct input occurrences at minimum Euclidean distance.

Target: $O(n \log n)$ time.

Hint · p. 34 Solution · p. 67

14. Bichromatic Closest Pair on Parallel Lines

Input: Red points on $x = 0$ and blue points on $x = 1$, with integer y-coordinates, n points in total, and at least one point of each colour.
Output: A red–blue pair at minimum Euclidean distance.

Target: $O(n \log n)$ time. Repeated points within either colour are allowed.

Hint · p. 34 Solution · p. 68

15. Manhattan Closest Pair

Input: $n \ge 2$ points with integer coordinates.
Output: Two distinct input occurrences minimizing $|p_x - q_x| + |p_y - q_y|$.

Target: $O(n \log n)$ time. Equal coordinates are allowed.

Hint · p. 34 Solution · p. 69

Integer Multiplication

16. Polynomial Multiplication

Input: Integer coefficient arrays A, B, each of length n, representing $P(x) = \sum_{i=1}^{n} A[i]x^{i-1}$ and $Q(x) = \sum_{i=1}^{n} B[i]x^{i-1}$.
Output: The $2n-1$ coefficients of $P(x)Q(x)$, including any leading zeroes needed for this fixed length.

Target: $O(n^{\log_2 3})$ scalar arithmetic operations.

Hint · p. 34 Solution · p. 70

17. Truncated Integer Multiplication

Input: Two nonnegative integers in binary, with n input bits in total, and an integer $k \geq 1$.
Output: Exactly the k least significant bits of their product, adding leading zeroes if needed; equivalently, the product modulo 2^k.

Target: $O(n + k^{\log_2 3})$ bit operations.

Hint · p. 34 Solution · p. 71

18. Signed Integer Multiplication

Input: Two integers, each represented by a sign and a binary magnitude of at most n bits, where $n \geq 1$.
Output: The sign and binary magnitude of their product. Use sign + and magnitude 0 as the unique representation of zero.

Target: $O(n^{\log_2 3})$ bit operations.

Hint · p. 35 Solution · p. 72

Matrix Multiplication

19. Rectangular Matrix Multiplication

Input: Integer matrices A of dimensions n-by-m and B of dimensions m-by-k, with all dimensions positive.
Output: Their n-by-k product.

Target: $O(N^{\log_2 7})$ scalar arithmetic operations, where $N = \max(n, m, k)$.

Hint · p. 35 Solution · p. 73

20. Matrix Powers

Input: An n-by-n integer matrix A and an integer $k \geq 0$.
Output: A^k, with $A^0 = I_n$.

Target: $O(n^{\log_2 7} \log k)$ scalar arithmetic operations.

Hint · p. 35 Solution · p. 74

21. Boolean Matrix Multiplication

Input: Two n-by-n matrices A, B with entries in $\{0, 1\}$.
Output: A Boolean matrix C with $C[i, j] = 1$ exactly when some index t satisfies $A[i, t] = B[t, j] = 1$.
Target: $O(n^{\log_2 7})$ scalar arithmetic operations.

Hint · p. 35 Solution · p. 75

Maximum Independent Set

22. Maximum-Weight Independent Set

Input: A graph with a nonnegative integer weight $w(v)$ on each vertex.
Output: An independent set S maximizing $\sum_{v \in S} w(v)$.
Target: $O(n^2 \varphi^n)$ time for $n \geq 1$, where $\varphi = (1 + \sqrt{5})/2$.

Hint · p. 35 Solution · p. 76

23. Counting Independent Sets

Input: A graph.
Output: The number of its independent vertex sets, including the empty set.
Target: $O(n^2 \varphi^n)$ time for $n \geq 1$, where $\varphi = (1 + \sqrt{5})/2$. Account for the size of the count.

Hint · p. 35 Solution · p. 77

24. Minimum Vertex Cover

Input: A graph.
Output: A smallest set $C \subseteq V$ containing at least one endpoint of every edge.
Target: $O(n^2 \varphi^n)$ time for $n \geq 1$, where $\varphi = (1 + \sqrt{5})/2$.

Hint · p. 35 Solution · p. 78

Chapter 2

Graph Search

Reachability by Breadth-First Search

25. Reachability from a Set

Input: A graph $G = (V, E)$, a nonempty set $S \subseteq V$, and a vertex t.
Output: Whether some vertex of S has a path to t.

Target: $O(n + m)$ time.

Hint · p. 36 Solution · p. 79

26. Reachability after Deletions

Input: A graph G, vertices s, t, and a set $F \subseteq E$ specified by edge identifiers.
Output: Whether s and t are connected after deleting every edge in F.

Target: $O(n + m)$ time.

Hint · p. 36 Solution · p. 80

27. Even-Length Reachability

Input: A graph G and vertices s, t.
Output: Whether there is an even-length walk from s to t. A walk may repeat vertices and edges; length zero is allowed.

Target: $O(n + m)$ time.

Hint · p. 36 Solution · p. 81

Shortest Paths

28. Multiple-Source Shortest Paths

Input: An unweighted graph G and a nonempty set $S \subseteq V$.
Output: For every vertex v, its minimum distance from S and a source attaining it. Return $+\infty$ and no source if unreachable.

Target: $O(n + m)$ time.

Hint · p. 36 Solution · p. 82

29. Counting Shortest Paths

Input: An unweighted graph G and a source s.
Output: For every vertex v, the number of shortest s–v paths, distinguished by vertex sequences. Use zero if unreachable and one for s.

Target: $O(n(n+m))$ time, counting bit operations on path counts.

Hint · p. 36 Solution · p. 83

30. Edges on Shortest Paths

Input: An unweighted graph G and vertices s, t.
Output: All edges lying on at least one shortest s–t path; return the empty set if no path exists.

Target: $O(n+m)$ time.

Hint · p. 36 Solution · p. 84

Connected Components

31. Component Weights

Input: A graph G with an integer weight $w[v]$ on each vertex.
Output: The connected-component partition and the sum of vertex weights in each component.

Target: $O(n+m)$ time.

Hint · p. 37 Solution · p. 85

32. Connecting a Graph

Input: A nonempty graph G.
Output: A smallest set of new edges whose addition makes G connected. Any pair of distinct vertices may be joined.

Target: $O(n+m)$ time.

Hint · p. 37 Solution · p. 86

33. Clique Components

Input: A graph G.
Output: Whether every connected component is a clique: every pair of its distinct vertices is adjacent.

Target: $O(n+m)$ time.

Hint · p. 37 Solution · p. 87

Testing Bipartiteness

34. Extending a Partial Two-Colouring

Input: A graph G with prescribed colours $b[v] \in \{0, 1\}$ on a subset $P \subseteq V$.
Output: A proper two-colouring extending every prescription, or a report that none exists.

Target: $O(n + m)$ time.

Hint · p. 37 Solution · p. 88

35. Counting Two-Colourings

Input: A graph G.
Output: The number of proper assignments of labelled colours $0, 1$ to its vertices. Swapping the colours gives a different assignment.

Target: $O(n + m)$ time, including writing the answer in binary.

Hint · p. 37 Solution · p. 89

36. Parity Constraints

Input: A graph G with a bit $b_e \in \{0, 1\}$ on every edge.
Output: Bits x_v satisfying $x_u \oplus x_v = b_{uv}$ for every edge uv, or a report that no assignment exists.

Target: $O(n + m)$ time.

Hint · p. 37 Solution · p. 90

Reachability by Depth-First Search

37. Path Recovery

Input: A graph G and vertices s, t.
Output: An s–t path, or a report that none exists.

Target: $O(n + m)$ time.

Hint · p. 37 Solution · p. 91

38. Subtree Sizes

Input: A rooted forest, given by its roots and the child list of each vertex.
Output: The number of vertices in every rooted subtree, including its root.

Target: $O(n)$ time.

Hint · p. 38 Solution · p. 92

39. Ancestor Queries

Input: A rooted forest given by child lists, and q ordered pairs (u, v).
Output: For every pair, whether u is an ancestor of v, allowing $u = v$.

Target: $O(n + q)$ time.

Hint · p. 38 Solution · p. 93

Cut Vertices

40. Components after Vertex Deletion

Input: A graph G.
Output: For every vertex v, the number of connected components of $G - \{v\}$. The empty graph has zero components.

Target: $O(n + m)$ time.

Hint · p. 38 Solution · p. 94

41. Cut Vertices in a Disconnected Graph

Input: A graph G, possibly disconnected.
Output: All vertices whose removal increases the number of connected components.

Target: $O(n + m)$ time.

Hint · p. 38 Solution · p. 95

42. Separated Pairs after Vertex Deletion

Input: A connected graph G.
Output: For every vertex u, the number of unordered pairs of distinct vertices in $V \setminus \{u\}$ that become disconnected when u is removed.

Target: $O(n + m)$ time.

Hint · p. 38 Solution · p. 96

Cut Edges

43. Edges on Cycles

Input: A graph G.
Output: All edges belonging to at least one simple cycle.

Target: $O(n + m)$ time.

Hint · p. 38 Solution · p. 97

44. Components after Removing All Cut Edges

Input: A graph G.
Output: A component label for every vertex after all cut edges of G have been removed.

Target: $O(n + m)$ time.

Hint · p. 38 Solution · p. 98

45. Unavoidable Edges

Input: A connected graph G and vertices s, t.
Output: All edges contained in every s–t path. For $s = t$, return the empty set.

Target: $O(n + m)$ time.

Hint · p. 38 Solution · p. 99

Directed Reachability

46. Directed Path Recovery

Input: A directed graph G and vertices s, t.
Output: A directed s–t path, or a report that none exists.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 100

47. Avoiding Specified Vertices

Input: A directed graph G, vertices s, t, and $S \subseteq V \setminus \{s, t\}$.
Output: Whether a directed s–t path avoiding every vertex of S exists.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 101

48. Shortest Directed Path

Input: An unweighted directed graph G and vertices s, t.
Output: A directed s–t path with the fewest edges, or a report that none exists.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 102

Strong Connectivity

49. Strong Connectivity with a Witness

Input: A nonempty directed graph G.
Output: A report that G is strongly connected, or an ordered pair (u, v) with no directed path from u to v.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 103

50. Strongly Connected Component of a Vertex

Input: A directed graph G and a vertex s.
Output: All vertices v such that s reaches v and v reaches s.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 104

51. A Sparse Strongly Connected Subgraph

Input: A strongly connected directed graph G.
Output: A set of at most $2n - 2$ edges whose spanning subgraph remains strongly connected.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 105

Directed Acyclic Graphs

52. Directed Cycle Witness

Input: A directed graph G.
Output: A directed cycle, or a report that G is acyclic.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 106

53. Acyclicity after One Edge Insertion

Input: A DAG G and distinct vertices u, v with $(u, v) \notin E$.
Output: Whether adding the edge $u \to v$ leaves the graph acyclic.

Target: $O(n + m)$ time.

Hint · p. 39 Solution · p. 107

54. Longest Directed Path in a DAG

Input: A DAG G and vertices s, t.
Output: A directed s–t path with the greatest number of edges, or a report that none exists.

Target: $O(n + m)$ time.

Hint · p. 40 Solution · p. 108

Strongly Connected Components

55. Condensation Graph

Input: A directed graph G.
Output: Its strongly connected component labels and a simple DAG with one vertex per component and one edge $C \to D$ whenever an original edge joins distinct components C, D.

Target: $O(n + m)$ time.

Hint · p. 40 Solution · p. 109

56. Universal Source Vertices

Input: A nonempty directed graph G.
Output: All vertices from which every vertex of G is reachable.

Target: $O(n + m)$ time.

Hint · p. 40 Solution · p. 110

57. Minimum Starting Set

Input: A nonempty directed graph G.
Output: A smallest set $S \subseteq V$ such that every vertex is reachable from at least one vertex of S.

Target: $O(n + m)$ time.

Hint · p. 40 Solution · p. 111

Chapter 3

Greedy Algorithms

Interval Scheduling

58. Scheduling with Turnaround Time

Input: Closed intervals $I_i = [s_i, f_i]$, $1 \le i \le n$, with integer endpoints $s_i \le f_i$, and an integer $\Delta \ge 0$.
Output: A largest subcollection that can be ordered so each chosen interval satisfies $f_i + \Delta < s_j$ for the next chosen interval.

Target: $O(n \log n)$ time.

Hint · p. 41 Solution · p. 112

59. Scheduling with a Required Interval

Input: Closed intervals $I_i = [s_i, f_i]$ with integer endpoints $s_i \le f_i$, and a required index r.
Output: A largest pairwise disjoint subcollection containing I_r.

Target: $O(n \log n)$ time.

Hint · p. 41 Solution · p. 113

60. Stabbing Intervals

Input: Closed intervals $I_i = [s_i, f_i]$ with integer endpoints $s_i \le f_i$.
Output: A smallest set of points such that every interval contains at least one selected point.

Target: $O(n \log n)$ time.

Hint · p. 41 Solution · p. 114

Interval Coloring

61. Coloring Half-Open Intervals

Input: Half-open intervals $I_i = [s_i, f_i)$ with integer endpoints $s_i < f_i$.
Output: A colouring with the fewest colours such that overlapping intervals receive different colours.

Target: $O(n \log n)$ time.

Hint · p. 41 Solution · p. 115

62. Maximum Simultaneous Intervals

Input: Closed intervals $I_i = [s_i, f_i]$ with integer endpoints $s_i \le f_i$.
Output: A time t contained in as many intervals as possible, together with the indices of all intervals containing it.

Target: $O(n \log n)$ time.

Hint · p. 41 Solution · p. 116

63. Interval Coloring with Demands

Input: Closed integer-endpoint intervals $I_1, \ldots, I_n$ and positive integer demands $d_1, \ldots, d_n$. Let $N = \sum_i d_i$.
Output: Assign d_i distinct colours to each interval, with disjoint colour sets on overlapping intervals, using the fewest colours overall.

Target: $O(N \log N)$ time.

Hint · p. 41 Solution · p. 117

Total Completion Time

64. Weighted Total Completion Time

Input: Jobs $1, \ldots, n$ with positive integer processing times p_i and weights w_i.
Output: An order on one machine, starting at time zero without idle time, minimizing $\sum_i w_i C_i$, where C_i is job i's completion time.

Target: $O(n \log n)$ time.

Hint · p. 42 Solution · p. 118

65. Minimum Total Waiting Time

Input: Jobs with positive integer processing times $p_1, \ldots, p_n$.
Output: An order on one machine, starting at time zero without idle time, minimizing the sum of the jobs' starting times.

Target: $O(n \log n)$ time.

Hint · p. 42 Solution · p. 119

66. Maximum Lateness

Input: Jobs with positive integer processing times p_i and integer deadlines d_i.
Output: An order on one machine, starting at time zero without idle time, minimizing $\max_i(C_i - d_i)$. Lateness may be negative.

Target: $O(n \log n)$ time.

Hint · p. 42 Solution · p. 120

Shortest Paths with Nonnegative Weights

67. Nearest Facility

Input: A directed graph with nonnegative integer edge weights and a nonempty set $S \subseteq V$ of facilities.
Output: For every vertex v, the minimum weighted distance from a facility to v and a facility attaining it. Use $+\infty$ and no facility if unreachable.

Target: $O((n+m)\log n)$ time.

Hint · p. 42 Solution · p. 121

68. One Free Edge

Input: A directed graph with nonnegative integer edge weights and vertices s, t.
Output: The minimum cost of an s–t path when at most one edge on it may be traversed for free, or $+\infty$ if no path exists.

Target: $O((n+m)\log n)$ time.

Hint · p. 42 Solution · p. 122

69. Vertex Tolls

Input: A directed graph with nonnegative integer edge costs $w(u,v)$ and vertex tolls $c(v)$, and vertices s, t.
Output: The minimum sum of edge costs and vertex tolls on an s–t path, including tolls at both endpoints, or $+\infty$ if unreachable.

Target: $O((n+m)\log n)$ time.

Hint · p. 42 Solution · p. 123

Minimum Spanning Trees

70. Minimum Spanning Forest

Input: An undirected graph G with integer edge weights, possibly disconnected.
Output: A minimum-total-weight forest containing one spanning tree of each connected component.

Target: $O((n+m)\log n)$ time.

Hint · p. 42 Solution · p. 124

71. MST after One Edge Insertion

Input: An MST T of a connected weighted graph G, and a new edge $e = uv \notin E(G)$ with $u \neq v$ and its integer weight. The tree is given by adjacency lists.
Output: An MST of $G + e$.

Target: $O(n)$ time.

Hint · p. 42 Solution · p. 125

72. Minimum Bottleneck Spanning Tree

Input: A connected graph with integer edge weights and $n \geq 2$.
Output: A spanning tree minimizing the weight of its heaviest edge.

Target: $O((n+m)\log n)$ time.

Hint · p. 43 Solution · p. 126

Minimum Spanning Trees (revisited)

73. Maximum Spanning Tree

Input: A connected graph G with integer edge weights.
Output: A spanning tree of maximum total weight.

Target: $O((n+m)\log n)$ time.

Hint · p. 43 Solution · p. 127

74. MST with Required Edges

Input: A connected graph G with integer edge weights and a set $F \subseteq E$ of required edges.
Output: A minimum-weight spanning tree containing every edge in F, or a report that no such tree exists.

Target: $O((n+m)\log n)$ time.

Hint · p. 43 Solution · p. 128

75. Threshold Connectivity Queries

Input: A graph G with integer edge weights and q queries (s_i, t_i, b_i).
Output: For each query, whether there is an s_i–t_i path all of whose edge weights are at most b_i.

Target: $O((n+m+q)\log(n+q))$ time.

Hint · p. 43 Solution · p. 129

Chapter 4

Dynamic Programming

Weighted Interval Scheduling

76. Scheduling at Most k Intervals

Input: Closed intervals $I[1..n]$, nonnegative integer weights $w[1..n]$, and $1 \le k \le n$.
Output: A pairwise disjoint subcollection of at most k intervals with maximum total weight.

Target: $O(n \log n + nk)$ time.

Hint · p. 44 Solution · p. 130

77. Weighted Scheduling with a Required Interval

Input: Closed intervals $I[1..n]$, nonnegative integer weights $w[1..n]$, and a designated interval I_r.
Output: A maximum-weight pairwise disjoint subcollection containing I_r.

Target: $O(n \log n)$ time.

Hint · p. 44 Solution · p. 131

78. Weighted Scheduling with Turnaround Time

Input: Closed intervals $I[1..n]$, nonnegative integer weights $w[1..n]$, and an integer $\Delta \ge 0$.
Output: A maximum-weight subcollection such that consecutive chosen intervals satisfy $f_i + \Delta < s_j$.

Target: $O(n \log n)$ time.

Hint · p. 44 Solution · p. 132

Subset Sum

79. Equal-Sum Partition

Input: A positive integer array $A[1..n]$; write $S = \sum_i A[i]$.
Output: A partition of the indices into two sets of equal sum, or a report that none exists.

Target: $O(nS)$ time.

Hint · p. 44 Solution · p. 133

80. Most Balanced Partition

Input: A positive integer array $A[1..n]$; write $S = \sum_i A[i]$.
Output: A partition of the indices minimizing the absolute difference between the two sums.

Target: $O(nS)$ time.

Hint · p. 44 Solution · p. 134

81. Subset Sum with Fewest Items

Input: A positive integer array $A[1..n]$ and a positive integer target K.
Output: A smallest set of indices summing to K, or a report that none exists.

Target: $O(nK)$ time.

Hint · p. 44 Solution · p. 135

Knapsack

82. Unbounded Knapsack

Input: Positive integer weights $w[1..n]$, nonnegative integer values $v[1..n]$, and positive integer capacity W.
Output: Nonnegative multiplicities a_i maximizing $\sum_i a_i v_i$ subject to $\sum_i a_i w_i \le W$.

Target: $O(nW)$ time.

Hint · p. 45 Solution · p. 136

83. Exact-Weight Knapsack

Input: Positive integer weights $w[1..n]$, nonnegative integer values $v[1..n]$, and positive integer target W.
Output: A maximum-value subset of total weight exactly W, or a report that none exists.

Target: $O(nW)$ time.

Hint · p. 45 Solution · p. 137

84. Knapsack with Two Resource Limits

Input: Positive integer resource requirements $a[1..n], b[1..n]$, nonnegative values $v[1..n]$, and positive integer limits A, B.
Output: A maximum-value subset J with $\sum_{i \in J} a_i \le A$ and $\sum_{i \in J} b_i \le B$.

Target: $O(nAB)$ time.

Hint · p. 45 Solution · p. 138

Longest Increasing Subsequence

85. Maximum-Weight Increasing Subsequence

Input: An integer array $A[1..n]$ and nonnegative integer weights $w[1..n]$.
Output: Increasing indices $i_1 < \cdots < i_k$ with strictly increasing values and maximum total weight.

Target: $O(n^2)$ time.

Hint · p. 45 Solution · p. 139

86. Longest Bitonic Subsequence

Input: An integer array $A[1..n]$.
Output: A longest subsequence that strictly increases to a peak and then strictly decreases; either side may consist only of the peak.

Target: $O(n^2)$ time.

Hint · p. 45 Solution · p. 140

87. Counting Longest Increasing Subsequences

Input: An integer array $A[1..n]$ and an integer modulus $p \geq 2$.
Output: The maximum increasing-subsequence length and the number of index sequences attaining it, modulo p.

Target: $O(n^2)$ time.

Hint · p. 45 Solution · p. 141

Longest Increasing Subsequence (revisited)

88. Longest Nondecreasing Subsequence

Input: An integer array $A[1..n]$.
Output: Indices of a longest subsequence whose values are nondecreasing.

Target: $O(n \log n)$ time.

Hint · p. 46 Solution · p. 142

89. Fewest Deletions to Make an Array Increasing

Input: An integer array $A[1..n]$.
Output: A smallest set of indices to delete so that the retained entries are strictly increasing.

Target: $O(n \log n)$ time.

Hint · p. 46 Solution · p. 143

90. Nesting Rectangles

Input: n rectangles with positive integer widths a_i and heights b_i; rotations are not allowed.
Output: A longest chain with both width and height strictly increasing.
Target: $O(n \log n)$ time.

Hint · p. 46 Solution · p. 144

Longest Common Subsequence

91. Longest Palindromic Subsequence

Input: A string $A[1..n]$.
Output: A longest subsequence that reads the same forwards and backwards.
Target: $O(n^2)$ time.

Hint · p. 46 Solution · p. 145

92. Shortest Common Supersequence

Input: Strings $A[1..n]$ and $B[1..m]$.
Output: A shortest string containing both inputs as subsequences.
Target: $O(nm)$ time.

Hint · p. 46 Solution · p. 146

93. Common Subsequence of Three Strings

Input: Strings $A[1..n]$, $B[1..m]$, and $C[1..\ell]$.
Output: The length of a longest subsequence common to all three strings.
Target: $O(nm\ell)$ time.

Hint · p. 46 Solution · p. 147

Edit Distance

94. Edit Distance with Unequal Costs

Input: Strings $A[1..n]$, $B[1..m]$ and nonnegative integer costs d, a, s for a deletion, insertion, and substitution.
Output: The minimum edit cost and an alignment attaining it; matching equal characters costs zero.
Target: $O(nm)$ time.

Hint · p. 47 Solution · p. 148

95. Closest Substring by Edit Distance

Input: Strings $A[1..n]$ and $B[1..m]$.
Output: A contiguous substring of B having minimum edit distance from A, together with that distance. The empty substring is allowed.

Target: $O(nm)$ time.

Hint · p. 47 Solution · p. 149

96. Editing with Insertions and Deletions

Input: Strings $A[1..n]$ and $B[1..m]$.
Output: The minimum number of edits needed to transform A into B when only insertions and deletions are permitted.

Target: $O(nm)$ time.

Hint · p. 47 Solution · p. 150

Maximum Independent Set on Trees

97. Maximum-Weight Independent Set on a Tree

Input: A tree T with nonnegative integer vertex weights w.
Output: An independent set of maximum total weight.

Target: $O(n)$ time.

Hint · p. 47 Solution · p. 151

98. Minimum-Weight Vertex Cover on a Tree

Input: A tree T with nonnegative integer vertex costs c.
Output: A minimum-cost vertex set containing at least one endpoint of every edge.

Target: $O(n)$ time.

Hint · p. 47 Solution · p. 152

99. Counting Independent Sets of a Tree

Input: A tree T and an integer modulus $p \geq 2$.
Output: The number of all independent sets, including the empty set, modulo p.

Target: $O(n)$ time.

Hint · p. 47 Solution · p. 153

Optimal Binary Search Trees

100. Height-Bounded Optimal Search Tree

Input: Increasing keys $x[1..n]$, positive integer access frequencies $f[1..n]$, and a positive integer height limit h. Height counts vertices on a longest root-to-leaf path.
Output: A binary search tree of height at most h minimizing the weighted number of comparisons, or a report that none exists.

Target: $O(hn^3)$ time.

Hint · p. 48 Solution · p. 154

101. Optimal Alphabetic Tree

Input: Positive integer weights $w[1..n]$ in a fixed order.
Output: A full binary tree whose leaves, left to right, are $1, \ldots, n$, minimizing $\sum_i w_i \operatorname{depth}(i)$. A one-leaf tree is allowed.

Target: $O(n^3)$ time.

Hint · p. 48 Solution · p. 155

102. Search Trees with Unsuccessful Queries

Input: Increasing keys $x[1..n]$, nonnegative successful-query frequencies $f[1..n]$, and nonnegative gap-query frequencies $q[0..n]$. Gap i is between x_i and x_{i+1}, with unbounded end gaps.
Output: A search tree minimizing the total number of key comparisons over both kinds of query, weighted by their frequencies.

Target: $O(n^3)$ time.

Hint · p. 48 Solution · p. 156

Shortest Paths with Negative Weights

103. Cheapest Walk with an Edge Limit

Input: A directed graph with integer edge weights, vertices s, t, and a positive integer k. Negative cycles are permitted.
Output: A minimum-weight s–t walk with at most k edges, or a report that none exists. Vertices and edges may repeat.

Target: $O(k(n+m))$ time.

Hint · p. 48 Solution · p. 157

104. Shortest Paths in a Weighted DAG

Input: A directed acyclic graph with integer edge weights and a source s. Weights may be negative.
Output: The minimum path weight from s to every vertex, with $+\infty$ for unreachable vertices.

Target: $O(n+m)$ time.

Hint · p. 48 Solution · p. 158

105. Distances to a Destination

Input: A directed graph with integer edge weights and no negative cycle, and a destination t.
Output: For every vertex v, the minimum weight of a path from v to t, or $+\infty$ if no path exists.
Target: $O(n + nm)$ time.

Hint · p. 48 Solution · p. 159

Negative Cycles

106. A Reachable Negative Cycle

Input: A directed graph with integer edge weights and a source s.
Output: A negative-weight cycle reachable from s, or a report that none exists.
Target: $O(n(n + m))$ time.

Hint · p. 48 Solution · p. 160

107. Vertices with Unboundedly Small Walk Weights

Input: A directed graph with integer edge weights and a source s.
Output: All vertices v for which s–v walks have arbitrarily negative weight.
Target: $O(n(n + m))$ time.

Hint · p. 48 Solution · p. 161

108. A Cycle below a Mean-Weight Threshold

Input: A directed graph with integer edge weights and a rational threshold a/b, where a is an integer and b is a positive integer.
Output: A directed cycle whose average edge weight is less than a/b, or a report that none exists.
Target: $O(n(n + m))$ time.

Hint · p. 49 Solution · p. 162

All-Pairs Shortest Paths

109. Transitive Closure

Input: A directed graph G.
Output: A Boolean matrix R with $R[i, j] =$ true exactly when a directed path from i to j exists. Zero-edge paths are allowed.
Target: $O(n^3)$ time.

Hint · p. 49 Solution · p. 163

110. Minimum-Bottleneck Paths

Input: A directed graph with nonnegative integer edge weights.
Output: For every ordered pair, the smallest possible maximum edge weight on a path; use zero for a zero-edge path and $+\infty$ for an unreachable pair.
Target: $O(n^3)$ time.

Hint · p. 49 Solution · p. 164

111. Minimum-Weight Directed Cycle

Input: A directed graph without self-loops, with integer edge weights and no negative cycle.
Output: The minimum weight of a directed cycle, or a report that no cycle exists.
Target: $O(n^3)$ time.

Hint · p. 49 Solution · p. 165

Travelling Salesperson

112. Minimum-Weight Hamiltonian Path

Input: A complete graph on $n \geq 2$ vertices with nonnegative edge weights and distinct endpoints s, t.
Output: A minimum-weight path from s to t visiting every vertex exactly once.
Target: $O(n^2 2^n)$ time.

Hint · p. 49 Solution · p. 166

113. Travelling Salesperson with Precedence Constraints

Input: A complete graph on $n \geq 3$ vertices with nonnegative weights, and ordered pairs (u, v) requiring u to be visited before v.
Output: A minimum-weight tour starting at vertex 1 and respecting every constraint before its final return to 1, or a report that none exists.
Target: $O(n^2 2^n)$ time.

Hint · p. 49 Solution · p. 167

114. Bottleneck Travelling Salesperson

Input: A complete graph on $n \geq 3$ vertices with nonnegative edge weights.
Output: A tour visiting every vertex exactly once that minimizes its largest edge weight.
Target: $O(n^2 2^n)$ time.

Hint · p. 49 Solution · p. 168

Chapter 5

Max Flow and Min Cut

Maximum Flow and Minimum Cut

115. Flow with Several Sources and Sinks

Input: A directed graph with nonnegative integer edge capacities and disjoint nonempty source and sink sets A, B.
Output: A capacity-respecting flow conserving flow outside $A \cup B$, with nonnegative net outflow at each source and nonnegative net inflow at each sink, maximizing total net outflow from A.
Target: $O(n(n+m)^2)$ time.

Hint · p. 50 Solution · p. 169

116. Flow with Vertex Capacities

Input: A flow network (G, c, s, t) and nonnegative integer bounds $b(v)$ for vertices other than s, t.
Output: A maximum flow whose total inflow at every internal vertex v is at most $b(v)$.
Target: $O(n(n+m)^2)$ time.

Hint · p. 50 Solution · p. 170

117. Circulation with Lower Bounds

Input: A directed graph and integer edge bounds $0 \le \ell(e) \le u(e)$.
Output: An assignment $\ell(e) \le f(e) \le u(e)$ conserving flow at every vertex, or a report that none exists.
Target: $O(n(n+m)^2)$ time.

Hint · p. 50 Solution · p. 171

Maximum Flow (revisited)

118. One Unit of Additional Capacity

Input: A flow network, an integral maximum flow f, and an edge e whose capacity is increased by one.
Output: A maximum flow for the modified network.
Target: $O(n+m)$ time.

Hint · p. 50 Solution · p. 172

119. Checking a Maximum-Flow Claim

Input: A flow network (G, c, s, t) and an integer assignment $f(e)$ to its edges.
Output: If f is infeasible, a violated capacity or conservation constraint. Otherwise, either an augmenting path proving nonoptimality or a cut proving that f is maximum.
Target: $O(n + m)$ time.

Hint · p. 50 Solution · p. 173

120. Is the Minimum Cut Unique?

Input: A flow network and an integral maximum flow f.
Output: Whether its minimum cut is unique as a source-side vertex set; if not, two distinct minimum cuts.
Target: $O(n + m)$ time.

Hint · p. 50 Solution · p. 174

Edge-Disjoint Paths

121. Edge-Disjoint Paths between Terminal Sets

Input: A directed graph and disjoint nonempty vertex sets A, B.
Output: A largest collection of edge-disjoint paths, each starting in A and ending in B. Endpoints may be shared; write F for the optimum number.
Target: $O((F + 1)(n + m))$ time.

Hint · p. 51 Solution · p. 175

122. Edge-Disjoint Paths to Distinct Destinations

Input: A directed graph, a source s, and a set $T \subseteq V \setminus \{s\}$.
Output: A largest collection of edge-disjoint paths from s to distinct vertices of T.
Target: $O(n(n + m))$ time.

Hint · p. 51 Solution · p. 176

123. Minimum Directed Edge Separator

Input: A directed graph and distinct vertices s, t.
Output: A smallest edge set whose deletion makes t unreachable from s.
Target: $O(n(n + m))$ time.

Hint · p. 51 Solution · p. 177

Vertex-Disjoint Paths

124. Vertex-Disjoint Paths between Terminal Sets

Input: A directed graph and disjoint nonempty terminal sets A, B.
Output: A largest collection of paths from A to B with no shared vertices, including endpoints.

Target: $O(n(n+m))$ time.

Hint · p. 51 Solution · p. 178

125. Minimum Internal Vertex Separator

Input: A directed graph and distinct vertices s, t with $(s,t) \notin E$.
Output: A smallest set of vertices other than s, t whose deletion makes t unreachable from s.

Target: $O(n(n+m))$ time.

Hint · p. 51 Solution · p. 179

126. Edge-Disjoint Paths with Vertex Limits

Input: A directed graph, distinct vertices s, t, and nonnegative integer limits $b(v)$ for internal vertices.
Output: A largest collection of edge-disjoint s–t paths in which each internal vertex v occurs in at most $b(v)$ paths.

Target: $O(n(n+m))$ time.

Hint · p. 51 Solution · p. 180

Edge-Disjoint Paths in Undirected Graphs

127. Edge Connectivity of an Undirected Graph

Input: A connected undirected graph on $n \geq 2$ vertices.
Output: A smallest edge set whose deletion disconnects the graph.

Target: $O(n^2(n+m))$ time.

Hint · p. 51 Solution · p. 181

128. Undirected Paths with Edge-Use Capacities

Input: An undirected graph, distinct s, t, and nonnegative integer capacities $c(e)$.
Output: A maximum-size multiset of s–t paths using each edge at most $c(e)$ times, represented by at most m paths with positive integer multiplicities.

Target: $O(n + m + nm^2)$ time.

Hint · p. 51 Solution · p. 182

129. Two Edge-Disjoint Routes or a Small Separator

Input: An undirected graph and distinct vertices s, t.
Output: Two edge-disjoint s–t paths, or a set of at most one edge whose deletion separates them.

Target: $O(n + m)$ time.

Hint · p. 52 Solution · p. 183

Bipartite Matching

130. Bipartite Matching with Vertex Quotas

Input: A bipartite graph $G = (X \cup Y, E)$ and nonnegative integer quotas $b(v)$.
Output: A largest edge subset in which each vertex v is incident to at most $b(v)$ chosen edges.

Target: $O(n(n + m)^2)$ time.

Hint · p. 52 Solution · p. 184

131. Bipartite Matching with Required Edges

Input: A bipartite graph $G = (X \cup Y, E)$ and a specified edge set $R \subseteq E$.
Output: A largest matching containing every edge of R, or a report that none exists.

Target: $O(n(n + m))$ time.

Hint · p. 52 Solution · p. 185

132. Matching Covering Specified Left Vertices

Input: A bipartite graph $G = (X \cup Y, E)$ and a subset $Q \subseteq X$.
Output: A largest matching that matches every vertex of Q, or a report that none exists.

Target: $O(n(n + m))$ time.

Hint · p. 52 Solution · p. 186

Minimum Vertex Cover in Bipartite Graphs

133. Minimum-Weight Bipartite Vertex Cover

Input: A bipartite graph $G = (X \cup Y, E)$ and nonnegative integer vertex costs $c(v)$.
Output: A vertex cover of minimum total cost.

Target: $O(n(n + m)^2)$ time.

Hint · p. 52 Solution · p. 187

134. Maximum-Weight Bipartite Independent Set

Input: A bipartite graph $G = (X \cup Y, E)$ and nonnegative integer weights $w(v)$.
Output: An independent vertex set of maximum total weight.

Target: $O(n(n + m)^2)$ time.

Hint · p. 52 Solution · p. 188

135. Vertex Cover with Required Vertices

Input: A bipartite graph $G = (X \cup Y, E)$ and a specified vertex set R.
Output: A smallest vertex cover containing all of R.

Target: $O(n(n+m))$ time.

Hint · p. 52 Solution · p. 189

Baseball Elimination

136. Possible First Place with Ties Allowed

Input: Current wins $w[1..n]$, a symmetric matrix $g[1..n, 1..n]$ of remaining game counts with zero diagonal, and a team x. Each game has one winner.
Output: An allocation of remaining wins in which no team finishes above x, or a report that none exists. Counts are nonnegative integers.

Target: $O(n^6)$ time.

Hint · p. 53 Solution · p. 190

137. League Winner with Prescribed Results

Input: Current wins w, remaining counts g, a designated team x, and nonnegative counts h_{ij} prescribing wins for i against j, with $h_{ij} + h_{ji} \le g_{ij}$ and $h_{ii} = 0$. There are n teams.
Output: A remaining-win allocation obeying the prescriptions and making x strictly first, or a report that none exists.

Target: $O(n^6)$ time.

Hint · p. 53 Solution · p. 191

138. League Completion under Win Caps

Input: Current wins $w[1..n]$, a symmetric nonnegative remaining-game matrix g with zero diagonal, and integer final-win caps $U[1..n]$.
Output: An allocation in which every team i finishes with at most U_i wins, or a report that none exists. Each remaining game has one winner.

Target: $O(n^6)$ time.

Hint · p. 53 Solution · p. 192

Project Selection

139. Projects with Prerequisites

Input: A directed graph on n projects with m prerequisite arcs and integer profits $w(v)$, which may be negative. An arc $u \to v$ requires choosing v whenever u is chosen.
Output: A prerequisite-closed project set of maximum total profit. The empty set is allowed.

Target: $O(n(n+m)^2)$ time.

Hint · p. 53 Solution · p. 193

140. Required and Forbidden Projects

Input: A directed prerequisite graph with integer project profits, a required set R, and a forbidden set F. Write $n = |V|$, $m = |E|$.
Output: A maximum-profit prerequisite-closed set containing R and avoiding F, or a report that none exists.

Target: $O(n(n+m)^2)$ time.

Hint · p. 53 Solution · p. 194

141. Projects with Penalties for Missing Prerequisites

Input: A directed graph on n projects with integer profits $w(v)$ and a nonnegative integer penalty $p(u, v)$ on each of its m arcs.
Output: A set A maximizing $\sum_{v \in A} w(v) - \sum_{u \in A,\, v \notin A} p(u, v)$. A missing prerequisite incurs its penalty instead of making the choice infeasible.

Target: $O(n(n+m)^2)$ time.

Hint · p. 53 Solution · p. 195

Part II

Hints

Chapter 6

Divide & Conquer

Merge Sort

1. Stable Sorting

Trace two equal-key records through a recursive call. Where could their relative order change?

Problem · p. 2

2. Merging Several Arrays

The order in which arrays are merged can make the same entry move many times. What would a good merge order try to control?

Problem · p. 2

3. Multiset Intersection

Try tracing one value through both sorted arrays. When can an occurrence be ruled out without searching further?

Problem · p. 2

Counting Inversions

4. Significant Inversions

For one entry in a sorted half, where can its qualifying partners lie in the other sorted half?

Problem · p. 2

5. Weighted Inversions

Look at a batch of inversions counted together by the ordinary algorithm. What information about that batch would you need here?

Problem · p. 2

6. Disagreement between Rankings

Can you relabel the items so that one ranking becomes $1, 2, \ldots, n$? What does the other ranking then represent?

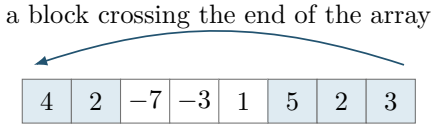
P :	30	10	40	20
Q :	10	30	20	40

Problem · p. 3

Maximum Subarray

7. Circular Maximum Subarray

Look at a block that crosses the end of the array. What can you say about the entries it leaves out?



Problem · p. 3

8. Two Disjoint Maximum Subarrays

Draw a boundary separating two disjoint intervals. What useful information could be prepared on either side?

Problem · p. 3

9. Maximum Submatrix

Suppose some rectangle boundaries were fixed in advance. Could the remaining choice become one-dimensional?

Problem · p. 3

Median and Selection

10. Selection with Repetitions

What extra information could distinguish equal entries without changing the order of unequal entries?

Problem · p. 3

11. Multiple Selection

The logarithm in the time bound depends on the number of requested ranks. What could you divide into two balanced groups?

Problem · p. 4

12. Weighted Median

Choosing a pivot and deciding which side to keep serve different purposes. Does weight need to control both decisions?

Problem · p. 4

Closest Pair

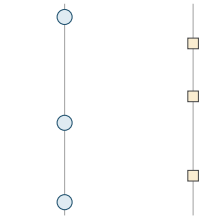
13. Closest Pair with Equal Coordinates

Equal coordinates need not mean equal input occurrences. How could you make the recursive split unambiguous?

Problem · p. 4

14. Bichromatic Closest Pair on Parallel Lines

Try sorting both colours together by height. What would an optimal pair look like in that order?



red: $x = 0$ blue: $x = 1$

Problem · p. 4

15. Manhattan Closest Pair

Revisit the geometric step that limits the number of nearby candidates. How does changing the distance measure affect that argument?

Problem · p. 4

Integer Multiplication

16. Polynomial Multiplication

Write each polynomial as a low-degree part and a high-degree part. Which features of fast integer multiplication depend on working with integers?

Problem · p. 5

17. Truncated Integer Multiplication

Which input bits can influence a product modulo 2^k?

Problem · p. 5

18. Signed Integer Multiplication

Start from an algorithm that multiplies nonnegative integers. What work remains outside that call?

Problem · p. 5

Matrix Multiplication

19. Rectangular Matrix Multiplication

Could you enlarge the instance so that a square-matrix algorithm applies without disturbing the product you need?

Problem · p. 5

20. Matrix Powers

Is decreasing the exponent by one the only useful way to obtain a smaller subproblem?

Problem · p. 5

21. Boolean Matrix Multiplication

What information would an ordinary arithmetic matrix product reveal about an entry of the desired Boolean product?

Problem · p. 6

Maximum Independent Set

22. Maximum-Weight Independent Set

Which decisions in the unweighted recursive algorithm depend on the objective, and which depend only on feasibility?

Problem · p. 6

23. Counting Independent Sets

What would you need to know about the overlap between recursive cases before using them to count solutions?

Problem · p. 6

24. Minimum Vertex Cover

What property must the vertices outside a vertex cover satisfy?

Problem · p. 6

Chapter 7

Graph Search

Reachability by Breadth-First Search

25. Reachability from a Set

Consider adding an additional vertex to the input graph.

Problem · p. 7

26. Reachability after Deletions

Is it necessary to physically delete an edge to prevent a search from using it?

Problem · p. 7

27. Even-Length Reachability

Can you turn this into an ordinary reachability problem on a graph with $2n$ vertices?

Problem · p. 7

Shortest Paths

28. Multiple-Source Shortest Paths

Consider giving the source set a single representative in an enlarged graph.

Problem · p. 7

29. Counting Shortest Paths

Look at the last step of a shortest path. Which earlier path counts could help determine the count at its endpoint?

Problem · p. 8

30. Edges on Shortest Paths

Try combining information from a search rooted at s with one rooted at t. What could this tell you about an edge?

Problem · p. 8

Connected Components

31. Component Weights

Suppose a component label is already available for every vertex. Could a single pass over the vertices finish the task?

Problem · p. 8

32. Connecting a Graph

Draw one vertex for each connected component. How many edges are needed to connect that smaller graph?

Problem · p. 8

33. Clique Components

Could a few numerical statistics certify that a component is complete?

Problem · p. 8

Testing Bipartiteness

34. Extending a Partial Two-Colouring

Start with a connected graph. How much freedom remains once one vertex has a colour?

Problem · p. 9

35. Counting Two-Colourings

Fix a connected component and the colour of one of its vertices. How many choices remain, and how do separate components interact?

Problem · p. 9

36. Parity Constraints

Consider replacing an edge by a short path in an ordinary two-colouring instance. How does the path length constrain its endpoint colours?

Problem · p. 9

Reachability by Depth-First Search

37. Path Recovery

What information does a search know when it first reaches a vertex that may be hard to recover later?

Problem · p. 9

38. Subtree Sizes

At the moment a recursive DFS call returns, what is already known about that vertex's children?

Problem · p. 9

39. Ancestor Queries

During a depth-first traversal, how are the visits to one subtree arranged in time?

Problem · p. 9

Cut Vertices

40. Components after Vertex Deletion

What information does the cut-vertex test discard when it reports only whether a deletion disconnects the graph?

Problem · p. 10

41. Cut Vertices in a Disconnected Graph

What can removing a vertex change outside its own connected component?

Problem · p. 10

42. Separated Pairs after Vertex Deletion

Look at the pieces identified by the cut-vertex argument. What additional information about each piece would let you count separated pairs?

Problem · p. 10

Cut Edges

43. Edges on Cycles

What would happen to connectivity if you removed an edge from a cycle?

Problem · p. 10

44. Components after Removing All Cut Edges

Could the search operate on a filtered view of the adjacency lists?

Problem · p. 10

45. Unavoidable Edges

Find any one path. Which of its edges can never be bypassed?

Problem · p. 10

Directed Reachability

46. Directed Path Recovery

When a directed search discovers a vertex, what evidence of reachability could it keep?

Problem · p. 11

47. Avoiding Specified Vertices

Could you encode the forbidden set in the graph supplied to the reachability algorithm?

Problem · p. 11

48. Shortest Directed Path

Compare the order in which DFS and BFS discover vertices. Which order is useful for minimizing the number of edges?

Problem · p. 11

Strong Connectivity

49. Strong Connectivity with a Witness

If a strong-connectivity test fails, what information from the failed search could explain why?

Problem · p. 11

50. Strongly Connected Component of a Vertex

How could you use a reachability search to answer a question about paths ending at s?

Problem · p. 11

51. A Sparse Strongly Connected Subgraph

Could a small collection of search-tree edges certify all the reachability you need?

Problem · p. 11

Directed Acyclic Graphs

52. Directed Cycle Witness

At what moment does DFS learn that a directed cycle exists? What information is still available then?

Problem · p. 12

53. Acyclicity after One Edge Insertion

Suppose the insertion does create a cycle. Where could that cycle lie relative to the new edge?

Problem · p. 12

54. Longest Directed Path in a DAG

Try computing an answer for every vertex, in topological order. What information is available when a vertex is processed?

Problem · p. 12

Strongly Connected Components

55. Condensation Graph

Several edges may represent the same edge of the answer. How could you arrange to encounter these repetitions together?

Problem · p. 12

56. Universal Source Vertices

Could two vertices in the same strongly connected component differ in whether they reach the whole graph?

Problem · p. 12

57. Minimum Starting Set

First contract the strongly connected components. Which vertices of the resulting DAG must be represented in a starting set?

Problem · p. 12

Chapter 8

Greedy Algorithms

Interval Scheduling

58. Scheduling with Turnaround Time

Can the turnaround requirement be absorbed into the way an interval is represented?

Problem · p. 13

59. Scheduling with a Required Interval

Sketch the intervals compatible with I_r. Can the remaining choices be separated into independent groups?

Problem · p. 13

60. Stabbing Intervals

Could an optimal set of stabbing points be moved onto interval endpoints?

Problem · p. 13

Interval Coloring

61. Coloring Half-Open Intervals

Can you replace the half-open intervals by closed intervals without changing which pairs overlap? Integer endpoints may help.

Problem · p. 13

62. Maximum Simultaneous Intervals

At which times can the set of active intervals change?

Problem · p. 14

63. Interval Coloring with Demands

The time bound depends on the total demand N. What kind of enlargement of the instance would that allow?

Problem · p. 14

Total Completion Time

64. Weighted Total Completion Time

Compare the contribution of two adjacent jobs in the two possible orders.

Problem · p. 14

65. Minimum Total Waiting Time

How does the total change when you measure each job at its start rather than at its finish?

Problem · p. 14

66. Maximum Lateness

Consider exchanging two adjacent jobs. Which features of their deadlines determine whether the maximum lateness can increase?

Problem · p. 14

Shortest Paths with Nonnegative Weights

67. Nearest Facility

Consider adding an auxiliary vertex to represent the facilities.

Problem · p. 15

68. One Free Edge

Can a graph on $2n$ vertices turn this into an ordinary shortest-path problem?

Problem · p. 15

69. Vertex Tolls

Could the cost of visiting a vertex be charged somewhere else along the path?

Problem · p. 15

Minimum Spanning Trees

70. Minimum Spanning Forest

If a spanning-tree search exhausts one connected component, what could it do next?

Problem · p. 15

71. MST after One Edge Insertion

Examine the structure formed by adding the new edge to T. How much of the old tree might need to change?

Problem · p. 15

72. Minimum Bottleneck Spanning Tree

Compare a tree's largest edge with the edges that could reconnect the tree if that edge were removed.

Problem · p. 16

Minimum Spanning Trees (revisited)

73. Maximum Spanning Tree

Could a change of edge weights turn a preference for larger total weight into a preference for smaller total weight?

Problem · p. 16

74. MST with Required Edges

Must a greedy algorithm always start from the empty set?

Problem · p. 16

75. Threshold Connectivity Queries

Must the queries be answered in the order they are given?

Problem · p. 16

Chapter 9

Dynamic Programming

Weighted Interval Scheduling

76. Scheduling at Most k Intervals

Try adding a second index to the weighted-interval scheduling table. How should that index change when an interval is chosen?

Problem · p. 17

77. Weighted Scheduling with a Required Interval

Draw the intervals compatible with I_r. Could the remaining optimization split into independent instances?

Problem · p. 17

78. Weighted Scheduling with Turnaround Time

Could a different representation of the intervals make ordinary compatibility sufficient?

Problem · p. 17

Subset Sum

79. Equal-Sum Partition

Try reducing the task to one call to a subset-sum algorithm. What target would you use?

Problem · p. 17

80. Most Balanced Partition

Try expressing the difference between the two sums using information about just one side.

Problem · p. 18

81. Subset Sum with Fewest Items

Try keeping the usual subset-sum table indices but changing what each entry stores.

Problem · p. 18

Knapsack

82. Unbounded Knapsack

Try defining a subproblem by capacity alone. After taking one copy of an item, which smaller subproblem remains?

Problem · p. 18

83. Exact-Weight Knapsack

What should an entry mean when its exact weight cannot be achieved? Check the base cases under that interpretation.

Problem · p. 18

84. Knapsack with Two Resource Limits

Try replacing the capacity index in the knapsack table by a pair of capacities.

Problem · p. 18

Longest Increasing Subsequence

85. Maximum-Weight Increasing Subsequence

Keep the usual subproblems ending at each array position. What should change when two candidate predecessors are compared?

Problem · p. 19

86. Longest Bitonic Subsequence

Try temporarily fixing one index of the optimal subsequence. Which index would make the remaining choices simpler?

Problem · p. 19

87. Counting Longest Increasing Subsequences

When two equally good partial solutions can be extended in the same way, what information should survive the comparison?

Problem · p. 19

Longest Increasing Subsequence (revisited)

88. Longest Nondecreasing Subsequence

Try breaking ties between equal values using their original positions. How could this help you use the strict-increase algorithm?

Problem · p. 19

89. Fewest Deletions to Make an Array Increasing

Try turning the minimization problem into a maximization problem on subsequences.

Problem · p. 19

90. Nesting Rectangles

Can ordering the rectangles by one dimension help with the other? Pay particular attention to ties.

Problem · p. 20

Longest Common Subsequence

91. Longest Palindromic Subsequence

Consider subproblems on contiguous intervals $A[i..j]$, even though the selected characters need not be contiguous.

Problem · p. 20

92. Shortest Common Supersequence

Try constructing the output from left to right while tracking how far you have progressed in each input.

Problem · p. 20

93. Common Subsequence of Three Strings

What information identifies a subproblem in the two-string version, and what would be missing for three strings?

Problem · p. 20

Edit Distance

94. Edit Distance with Unequal Costs

Trace the three ways an alignment can begin. Where should each operation's cost enter the recurrence?

Problem · p. 20

95. Closest Substring by Edit Distance

In the usual edit-distance table, where is the requirement to use all of the second string enforced?

Problem · p. 21

96. Editing with Insertions and Deletions

Consider the original characters that survive all deletions. What relationship must they have to the target string?

Problem · p. 21

Maximum Independent Set on Trees

97. Maximum-Weight Independent Set on a Tree

Which parts of the independent-set recurrence describe compatibility, and which describe the quality of a choice?

Problem · p. 21

98. Minimum-Weight Vertex Cover on a Tree

What can you say about the vertices left outside a cover?

Problem · p. 21

99. Counting Independent Sets of a Tree

Once a choice is fixed at a parent, how much freedom remains in the different child subtrees?

Problem · p. 21

Optimal Binary Search Trees

100. Height-Bounded Optimal Search Tree

Try adding a table dimension for the height available to a subtree. How does that allowance change below the chosen root?

Problem · p. 22

101. Optimal Alphabetic Tree

What restrictions does the left-to-right leaf order place on the possible subtrees?

Problem · p. 22

102. Search Trees with Unsuccessful Queries

When a root is chosen for a key interval, which gap queries also pass through that root?

Problem · p. 22

Shortest Paths with Negative Weights

103. Cheapest Walk with an Edge Limit

Why is the number of rounds important in the usual shortest-path dynamic program?

Problem · p. 22

104. Shortest Paths in a Weighted DAG

Could an ordering of the vertices make repeated relaxations unnecessary?

Problem · p. 22

105. Distances to a Destination

Can paths toward a fixed destination be viewed as paths away from a fixed source?

Problem · p. 23

Negative Cycles

106. A Reachable Negative Cycle

Could a reachability search simplify the instance before you look for a cycle?

Problem · p. 23

107. Vertices with Unboundedly Small Walk Weights

After $n - 1$ rounds of the shortest-path dynamic program, try one more round. What do further improvements tell you?

Problem · p. 23

108. A Cycle below a Mean-Weight Threshold

How could a change of edge weights make zero the relevant threshold?

Problem · p. 23

All-Pairs Shortest Paths

109. Transitive Closure

Try adapting the all-pairs shortest-path recurrence to Boolean table entries. What should it mean to join two partial routes or to choose between them?

Problem · p. 23

110. Minimum-Bottleneck Paths

How does the bottleneck of a route behave when two smaller routes are joined?

Problem · p. 24

111. Minimum-Weight Directed Cycle

If you remove one edge from a directed cycle, what remains?

Problem · p. 24

Travelling Salesperson

112. Minimum-Weight Hamiltonian Path

Which step of the tour algorithm insists that you return to the starting vertex?

Problem · p. 24

113. Travelling Salesperson with Precedence Constraints

What information about earlier visits is already available in a visited-set state?

Problem · p. 24

114. Bottleneck Travelling Salesperson

Keep the visited-set states of the tour algorithm. Reconsider how an edge extends a stored value.

Problem · p. 24

Chapter 10

Max Flow and Min Cut

Maximum Flow and Minimum Cut

115. Flow with Several Sources and Sinks

Could a larger network replace the two sets of terminals by one source and one sink?

Problem · p. 25

116. Flow with Vertex Capacities

Consider replacing a vertex by a small network.

Problem · p. 25

117. Circulation with Lower Bounds

Imagine the compulsory flow is already present. Which flow conditions might still fail?

Problem · p. 25

Maximum Flow (revisited)

118. One Unit of Additional Capacity

What does the old minimum cut tell you about how much the optimum could change?

Problem · p. 25

119. Checking a Maximum-Flow Claim

A failed augmenting-path search contains useful information. What can it certify?

Problem · p. 26

120. Is the Minimum Cut Unique?

Once a maximum flow is fixed, what must the residual graph look like across any minimum cut?

Problem · p. 26

Edge-Disjoint Paths

121. Edge-Disjoint Paths between Terminal Sets

Consider enlarging the graph to simplify the sets of allowed endpoints.

Problem · p. 26

122. Edge-Disjoint Paths to Distinct Destinations

Try adding an auxiliary sink. What could the capacity of an edge entering it represent?

Problem · p. 26

123. Minimum Directed Edge Separator

What should a cut's capacity measure if you want it to represent a number of deleted edges?

Problem · p. 26

Vertex-Disjoint Paths

124. Vertex-Disjoint Paths between Terminal Sets

Consider replacing every original vertex—including a terminal—by a small network.

Problem · p. 27

125. Minimum Internal Vertex Separator

Try replacing each internal vertex by two copies. Which auxiliary edges should a minimum cut be allowed to use?

Problem · p. 27

126. Edge-Disjoint Paths with Vertex Limits

Consider splitting the internal vertices in an auxiliary flow network. Which capacities should enforce edge-disjointness, and which should enforce $b(v)$?

Problem · p. 27

Edge-Disjoint Paths in Undirected Graphs

127. Edge Connectivity of an Undirected Graph

Choose one vertex as a reference point. What must any disconnection separate it from?

Problem · p. 27

128. Undirected Paths with Edge-Use Capacities

How much flow can you remove along a single positive-flow path at once?

Problem · p. 27

129. Two Edge-Disjoint Routes or a Small Separator

How much of a maximum-flow computation is actually needed to distinguish the two possible outcomes?

Problem · p. 28

Bipartite Matching

130. Bipartite Matching with Vertex Quotas

In the standard matching network, what does each kind of capacity enforce?

Problem · p. 28

131. Bipartite Matching with Required Edges

Try reducing to an ordinary matching problem on a smaller graph after committing to the required edges.

Problem · p. 28

132. Matching Covering Specified Left Vertices

What happens to a vertex that is already matched when an augmenting path is used?

Problem · p. 28

Minimum Vertex Cover in Bipartite Graphs

133. Minimum-Weight Bipartite Vertex Cover

Start with the usual bipartite matching network. Which capacities represent choosing a vertex, and which must prevent an edge from being uncovered?

Problem · p. 28

134. Maximum-Weight Bipartite Independent Set

What kind of set is the complement of an independent set, and how are the two weights related?

Problem · p. 28

135. Vertex Cover with Required Vertices

Can you construct a smaller vertex-cover instance containing only the obligations left after choosing R?

Problem · p. 29

Baseball Elimination

136. Possible First Place with Ties Allowed

Could changing just one team's initial win total let you reuse the strict-first-place algorithm?

Problem · p. 29

137. League Winner with Prescribed Results

Which quantities would change if the prescribed games had already been played?

Problem · p. 29

138. League Completion under Win Caps

What do the team capacities in the usual league network express?

Problem · p. 29

Project Selection

139. Projects with Prerequisites

Try using the source side of a cut to represent the chosen projects. How could a prerequisite arc make a forbidden choice too expensive?

Problem · p. 29

140. Required and Forbidden Projects

Can a suitable capacity make one placement of a vertex prohibitively expensive?

Problem · p. 30

141. Projects with Penalties for Missing Prerequisites

Under what placement of its endpoints does a directed arc contribute to a cut?

Problem · p. 30

Part III

Solutions

Chapter 11

Divide & Conquer

1. Stable Sorting

Assume that n is a power of two. A merge compares keys and moves whole records.

Algorithm StableSort($A[1..n]$)

```
1 if n = 1 then return A
2 L ← StableSort(A[1..n/2])
3 R ← StableSort(A[n/2 + 1..n])
4 return Merge(L, R), taking the left record on equal keys
```

Theorem 1.1. StableSort *solves* Stable Sorting.

Proof. We prove that the output is a sorted rearrangement of the input in which every pair of equal-key records retains its original order. We use induction on n.

Base case. For $n = 1$, the algorithm returns the sole record, so the claim holds.

Induction step. Suppose $n > 1$. By the induction hypothesis, L and R are stably sorted rearrangements of their respective input halves. The correctness of Merge implies that the merged array is a sorted rearrangement of A. To show stability, consider two records x, y with equal keys, where x precedes y in A. There are two cases:

(i) If x and y lie in the same half, the recursive call preserves their order by the induction hypothesis. The merge also preserves this order, since it consumes each half in order.

(ii) If x and y lie in different halves, x lies in the left half and y in the right. Since L is sorted, all its records through x have key at most the key of y. The merge emits all these records before y, taking the left record on equal keys.

In either case, x precedes y in the output. Thus the output is stable, completing the inductive step. □

Theorem 1.2. StableSort *takes* $O(n \log n)$ *time.*

Proof. A merge performs constant work per emitted record, hence takes $O(n)$ time. Copying the halves also takes $O(n)$. Thus $T(1) = O(1)$ and $T(n) \leq 2T(n/2) + O(n)$. Applying the Master Theorem to this recurrence gives $T(n) = O(n \log n)$. □

Problem · p. 2

2. Merging Several Arrays

Assume that the number k of arrays is a power of two. The input $\mathcal{A}$ is a list of sorted arrays.

Algorithm MULTIMERGE($\mathcal{A}[1..k]$)

```
1 if k = 1 then return A[1]
2 L ← MultiMerge(A[1..k/2])
3 R ← MultiMerge(A[k/2 + 1..k])
4 return Merge(L, R)
```

Theorem 2.1. MULTIMERGE *solves* Merging Several Arrays.

Proof. We prove that the output is the sorted multiset union of the input arrays. We use induction on k.

Base case. For $k = 1$, the input array already has the required form.

Induction step. The induction hypothesis says that L and R are the sorted multiset unions of the two groups of input arrays. The correctness of MERGE implies that their merge is sorted and contains their entries with unchanged multiplicities. The two groups partition the inputs, so the merged array is their sorted multiset union. □

Theorem 2.2. *For N total entries,* MULTIMERGE *takes $O(N \log k)$ time.*

Proof. The recursion has $\log k$ merging levels. At each level the arrays being merged contain each original entry exactly once. The merges at that level cost $O(N)$ in total. Copying lists of array references adds at most $O(k)$ per level, which is $O(N)$ because every original array is nonempty. The k leaves add $O(N)$, so the total is $O(N \log k)$. □

Problem · p. 2

3. Multiset Intersection

Algorithm INTERSECT($A[1..n], B[1..m]$)

```
1 allocate C[1.. min(n, m)]; i ← 1; j ← 1; q ← 0
2 while i ≤ n and j ≤ m do
3     if A[i] < B[j] then i ← i + 1
4     else if B[j] < A[i] then j ← j + 1
5     else
6         q ← q + 1; C[q] ← A[i]
7         i ← i + 1; j ← j + 1
8 return C[1..q]
```

Theorem 3.1. INTERSECT *solves* Multiset Intersection.

Proof. We prove that after any number of iterations, the desired intersection is the multiset union of $C[1..q]$ and the intersection of the unconsumed suffixes $A[i..n]$ and $B[j..m]$. We also prove that $C[1..q]$ is sorted and every entry in it is at most every entry in either suffix. We use induction on the number of iterations.

Base case. Before the first iteration, $C[1..q]$ is empty and the suffixes are the full input arrays, so both claims hold.

Induction step. Suppose both claims hold before an iteration. There are two cases:

(i) If $A[i] \neq B[j]$, the smaller front value cannot occur in the other sorted suffix. Discarding it leaves the intersection of the suffixes unchanged. The output is unchanged and the suffixes only shrink, so the order properties are also preserved.

(ii) If $A[i] = B[j] = x$, removing one x from each suffix reduces the intersection's multiplicity of x by one and leaves all other multiplicities unchanged. Appending x to C compensates for this change and preserves sortedness by the induction hypothesis. Since the suffixes are sorted, all their remaining entries are at least x.

Thus both claims hold after the iteration. When the loop ends, one suffix is empty, so the intersection of the suffixes is empty. The claims therefore show that $C[1..q]$ is exactly the desired sorted intersection. □

Theorem 3.2. INTERSECT *takes* $O(n + m)$ *time.*

Proof. Every iteration advances at least one index, and there are at most $n + m$ index advances in total. Each iteration takes $O(1)$ time. Allocating the buffer and returning its used prefix take $O(\min(n, m))$ time. Thus the total time is $O(n + m)$. □

Problem · p. 2

4. Significant Inversions

Assume that n is a power of two. Return a sorted array together with the count.

Algorithm SignificantCount($A[1..n]$)

```
1 if n = 1 then return (A, 0)
2 (L, c_L) ← SignificantCount(A[1..n/2])
3 (R, c_R) ← SignificantCount(A[n/2 + 1..n])
4 j ← 1; c_X ← 0
5 for i ← 1 to n/2 do
6     while j ≤ n/2 and 2R[j] < L[i] do j ← j + 1
7     c_X ← c_X + (j − 1)
8 return (Merge(L, R), c_L + c_R + c_X)
```

Theorem 4.1. SignificantCount *returns a sorted rearrangement of A and the number of its significant inversions.*

Proof. By induction on n.

Base case. A singleton is sorted and has no significant inversions.

Induction step. The induction hypothesis gives the sorted halves and their correct internal counts. We claim that line 7 counts exactly the cross pairs whose left entry is $L[i]$. Since R is sorted, the entries satisfying $2R[j] < L[i]$ form a prefix. Its endpoint cannot decrease as i increases, because L is sorted. Thus the scan stops just after this prefix and adds its length $j - 1$. This argument also applies to negative entries.

Every significant inversion lies wholly in the left half, wholly in the right half, or crosses the split. These classes are disjoint, and sorting within a half preserves its cross pairs. Their counts therefore sum to the required count. The correctness of Merge gives the sorted rearrangement. □

Theorem 4.2. SignificantCount *takes $O(n \log n)$ time.*

Proof. The two scanning indices advance $O(n)$ times in total. Copying and merging are also linear. Hence $T(n) \le 2T(n/2) + O(n)$ with constant base cost. Applying the Master Theorem to this recurrence gives $T(n) = O(n \log n)$. □

Problem · p. 2

5. Weighted Inversions

Assume that n is a power of two. Each record contains a value and its weight.

Algorithm WeightedCount$(A[1..n], w[1..n])$

```
1 if n = 1 then return ([(A[1], w[1])], 0)
2 (L, c_L) ← WeightedCount(A[1..n/2], w[1..n/2])
3 (R, c_R) ← WeightedCount(A[n/2 + 1..n], w[n/2 + 1..n])
4 W ← 0; c_X ← 0
5 merge L, R by value into C, taking the left record on a tie
6     when a right record of weight u is emitted: W ← W + u
7     when a left record of weight u is emitted: c_X ← c_X + uW
8 return (C, c_L + c_R + c_X)
```

Theorem 5.1. WeightedCount *returns the records sorted by value and the total weight of the inversions.*

Proof. By induction on n.

Base case. A singleton is sorted and has inversion weight zero.

Induction step. The induction hypothesis gives sorted halves and correct internal totals. We claim that, when a left record of value x is emitted, W is the total weight of the right records with value below x. Every consumed right record was chosen before a left record of value at most x, so its value is strictly below x. Every remaining right record has value at least x, because ties favour the left record. This proves the claim.

Consequently uW is exactly the weight of the cross inversions with this left record. Each cross inversion is counted once, at its left endpoint. Adding the two internal totals accounts for all inversions. The merge also produces a sorted rearrangement, completing the induction. □

Theorem 5.2. WeightedCount *takes $O(n \log n)$ time.*

Proof. Each merge emits n records and does constant additional arithmetic per record. Copying the two halves is linear as well. Therefore $T(n) \le 2T(n/2) + O(n)$ with $T(1) = O(1)$, and the Master Theorem gives $T(n) = O(n \log n)$. □

Problem · p. 2

6. Disagreement between Rankings

Assume that n is a power of two.

Algorithm RANKINGDISAGREEMENT($P[1..n], Q[1..n]$)

```
1 U ← [(P[i], i) : i = 1, ..., n]
2 V ← [(Q[j], j) : j = 1, ..., n]
3 sort U, V by their first coordinates using MERGESORT
4 allocate R[1..n]
5 for t ← 1 to n do
6     R[U[t].position] ← V[t].position
7 (B, c) ← SORTCOUNT(R)
8 return c
```

Theorem 6.1. RANKINGDISAGREEMENT *solves* Disagreement between Rankings.

Proof. We prove that disagreements between the rankings correspond bijectively to inversions of R. To prove this directly, first note that MERGESORT places the same value at position t of U and V, since the inputs are permutations of the same distinct values. Hence $R[i]$ is the position of $P[i]$ in Q.

For any $i < j$, the pair $P[i], P[j]$ appears in the opposite order in Q exactly when $R[i] > R[j]$. Every unordered pair has exactly one ordering $i < j$, proving the bijection. The correctness of SORTCOUNT now gives exactly the number of disagreements. □

Theorem 6.2. RANKINGDISAGREEMENT *takes* $O(n \log n)$ *time.*

Proof. Constructing the record arrays and R costs $O(n)$. The two sorts cost $O(n \log n)$; comparing or moving a record is constant-time. SORTCOUNT has the same bound. Adding the costs proves the claim. □

Problem · p. 3

7. Circular Maximum Subarray

Each MaxSubarray call below appends $-\infty$ entries to reach a power-of-two length. Its result (a, b, t) gives endpoints and sum; padding cannot be part of an optimal interval with finite sum.

Algorithm CircularMax($A[1..n]$)

1 **if** $n = 1$ **then return** $(1, 1)$
2 $S \leftarrow \sum_{i=1}^{n} A[i]$; $I \leftarrow$ MaxSubarray(A)
3 $(a, b, t) \leftarrow$ MaxSubarray($-A[1..n-1]$); $J_L \leftarrow (a, b, -t)$
4 $(a, b, t) \leftarrow$ MaxSubarray($-A[2..n]$); $J_R \leftarrow (a+1, b+1, -t)$
5 $(a, b, t) \leftarrow$ a smaller-sum triple among J_L, J_R
6 $K \leftarrow ((b \bmod n) + 1,\ n - (b - a + 1))$
7 **return** I as a start and length, or K, whichever has larger sum
(the sum of K is $S - t$)

Theorem 7.1. CircularMax *solves* Circular Maximum Subarray.

Proof. The singleton case is immediate. For $n > 1$, we prove that I and K are feasible and that every feasible block has sum at most one of theirs. We prove the claim by cases.

(i) If the block does not wrap, the correctness of MaxSubarray makes I a best ordinary interval, including the whole array. Padding cannot enter an optimum because it gives sum $-\infty$.

(ii) If the block wraps and is not the whole array, its complement is a nonempty ordinary interval J that excludes the first or last index. Thus J lies in $A[1..n-1]$ or $A[2..n]$. Negation turns minimum sums into maximum sums, so lines 3–5 find a minimum-sum interval among these possibilities. Its circular complement K is nonempty and has sum at least that of the wrapping block.

Both candidates are feasible. The claim therefore shows that returning the better one gives an optimum. □

Theorem 7.2. CircularMax *takes* $O(n \log n)$ *time.*

Proof. Each padded length is less than twice its unpadded length. The three calls therefore cost $O(n \log n)$ in total. Padding, negating, summing, and converting endpoints add $O(n)$. □

Problem · p. 3

8. Two Disjoint Maximum Subarrays

For a nonempty interval X, store its endpoints, total S_X, best nonempty prefix P_X, best nonempty suffix Q_X, and best nonempty subarray M_X. Attach attaining endpoints to each maximum. For adjacent X, Y, define JOIN by

$$S_{XY} = S_X + S_Y, \qquad P_{XY} = \max(P_X, S_X + P_Y),$$
$$Q_{XY} = \max(Q_Y, Q_X + S_Y), \quad M_{XY} = \max(M_X, M_Y, Q_X + P_Y).$$

The summary σ_i of the singleton $A[i]$ has all four sums equal to $A[i]$. Joined candidates join their endpoints as well.

Algorithm TWOINTERVALS($A[1..n]$)

```
1 L[1] ← σ_1; R[n] ← σ_n
2 for i ← 2 to n do L[i] ← JOIN(L[i − 1], σ_i)
3 for i ← n − 1, n − 2, . . . , 1 do
4     R[i] ← JOIN(σ_i, R[i + 1])
5 choose t ∈ {1, . . . , n − 1} maximizing L[t].M + R[t + 1].M
6 return the intervals stored with L[t].M and R[t + 1].M
```

Theorem 8.1. TWOINTERVALS *solves* Two Disjoint Maximum Subarrays.

Proof. We prove that $L[t]$ and $R[t]$ contain the exact summaries of $A[1..t]$ and $A[t..n]$, respectively. We use induction on the number of entries processed in each scan.

Base case. A singleton's total, best prefix, best suffix, and best subarray all have value $A[i]$, as initialized.

Induction step. We claim that JOIN computes the summary of XY from correct summaries of X and Y. A prefix either lies in X or contains all of X and a nonempty prefix of Y, giving the two candidates for P_{XY}. The suffix argument is symmetric. A subarray lies in one half or consists of a suffix of X followed by a prefix of Y, giving the three candidates for M_{XY}. Totals add, and stored endpoints attain each candidate. This proves the claim. The induction hypothesis therefore justifies each extension in both scans.

Every feasible pair has a split t between its intervals, so its sum is at most $L[t].M + R[t+1].M$. Conversely, the stored pair for every split is feasible. Maximizing over the splits therefore returns an optimum. □

Theorem 8.2. TWOINTERVALS *takes* $O(n)$ *time.*

Proof. Each summary has constant size, so JOIN takes $O(1)$ time. The two summary scans and the final scan each have $O(n)$ iterations. Their combined cost is $O(n)$. □

Problem · p. 3

9. Maximum Submatrix

Assume that the number m of columns is a power of two.

Algorithm MaxRectangle($A[1..n, 1..m]$)

```
1 b ← −∞
2 for r ← 1 to n do
3     initialize C[1..m] ← 0
4     for s ← r to n do
5         C ← C + A[s, 1..m]
6         (u, v, t) ← MaxSubarray(C)
7         if t > b then b ← t; H ← (r, s, u, v)
8 return H
```

Theorem 9.1. MaxRectangle *solves* Maximum Submatrix.

Proof. We prove that for fixed r, after line 5 at bottom row s, $C[j] = \sum_{i=r}^{s} A[i, j]$ for every j. We use induction on s.

Base case. For $s = r$, the first update adds row r to the zero array.

Induction step. The induction hypothesis gives the sum of rows $r, \ldots, s-1$. Adding row s gives exactly the claimed sum.

A column interval in C therefore has the same sum as the corresponding rectangle in rows $r, \ldots, s$. The correctness of MaxSubarray implies that line 6 finds a best nonempty rectangle for these rows.

Line 7 keeps a best candidate among the row pairs examined. Every nonempty rectangle has one such pair, so the final candidate is globally optimal. The first candidate has finite sum and is stored, so H is defined even when every entry is negative. □

Theorem 9.2. MaxRectangle *takes* $O(n^2 m \log m)$ *time.*

Proof. There are $n(n+1)/2$ row pairs. Updating C costs $O(m)$ per pair, and MaxSubarray costs $O(m \log m)$. Initializing C for each top row adds $O(nm)$. The sum is $O(n^2 m \log m)$. □

Problem · p. 3

10. Selection with Repetitions

Compare records lexicographically: first by value, then by original position.

Algorithm SELECTREPEATED$(A[1..n], k)$

```
1 B ← [(A[i], i) : i = 1, ..., n]
2 (x, i) ← SELECT(B, k) using lexicographic order
3 return x
```

Theorem 10.1. SELECTREPEATED *solves* Selection with Repetitions.

Proof. We prove that the value coordinate of the kth lexicographic record is the kth entry of a sorted rearrangement of A. Indeed, the records are distinct because their position coordinates are distinct. Sorting them lexicographically orders their values and retains every occurrence, proving the claim directly.

The correctness of SELECT applies to any totally ordered set of distinct records. It returns the record of rank k, whose value coordinate is the required answer by the claim. □

Theorem 10.2. SELECTREPEATED *takes* $O(n)$ *time.*

Proof. Constructing B costs $O(n)$. Comparing or moving a pair takes $O(1)$, so the Linear-time selection theorem gives $O(n)$ time for SELECT on these records as well. Returning one coordinate costs $O(1)$, for $O(n)$ total. □

Problem · p. 3

11. Multiple Selection

Algorithm MULTISELECT$(A[1..n], K[1..r])$

```
1 if r = 0 then return the empty list
2 choose a middle index h of K; k ← K[h]
3 p ← SELECT(A, k)
4 partition the other entries into L below p and R above p
5 U ← MULTISELECT(L, K[1..h − 1])
6 K_R ← [K[t] − k : t = h + 1, . . . , r]
7 V ← MULTISELECT(R, K_R)
8 return the concatenation of U, [p], and V
```

Theorem 11.1. MULTISELECT *solves* Multiple Selection.

Proof. We prove that every call returns the requested entries in rank order. We use induction on the number r of requests.

Base case. With no requests, the empty list is correct.

Induction step. The correctness of SELECT gives p of rank k, so $|L| = k-1$. A requested rank below k is unchanged in L; one above k decreases by exactly k in R. Thus both recursive rank lists are valid. They are shorter than the original list, so the induction hypothesis gives their correct answers in order. Concatenating them with p gives all requested entries in rank order. □

Theorem 11.2. MULTISELECT *takes* $O(n \log r)$ *time.*

Proof. For a call on s entries, selection costs $O(s)$. Partitioning, forming rank lists, and concatenating answers also cost $O(s)$ because there are at most s requests. At a fixed depth, the nonempty-request calls use disjoint subsets of the original array, so their costs sum to $O(n)$.

Choosing a middle request leaves at most half the requests on either side, giving $O(\log r)$ levels. Empty-request calls add $O(r)$ in total. Since $r \le n$, the total is $O(n \log r)$. □

Problem · p. 4

12. Weighted Median

The current records remain paired with their weights. A median is chosen by number of records, using SELECT, rather than by weight.

Algorithm WEIGHTEDMEDIAN$((a, w)[1..n])$

```
1 S ← the input records; t ← (1/2) Σ_{i=1}^{n} w_i
2 while |S| > 1 do
3     choose a median value p of S using SELECT
4     partition S \ {p} into L below p and R above p
5     W_L ← Σ_{a∈L} w(a)
6     if t ≤ W_L then S ← L
7     else if t ≤ W_L + w(p) then return p
8     else S ← R; t ← t − W_L − w(p)
9 return the value of the sole record in S
```

Theorem 12.1. WEIGHTEDMEDIAN *solves* Weighted Median.

Proof. Let W be the total input weight and let x be the first input value whose cumulative weight in increasing order reaches $W/2$. We prove that x remains the first value in S whose cumulative weight reaches t, with $0 < t \le \sum_{a \in S} w(a)$. We use induction on the number of iterations.

Base case. Initially S contains all records and $t = W/2$, so the claim holds by the definition of x.

Induction step. Suppose the claim holds before an iteration. The partition around p gives three cases:

(i) If $t \le W_L$, the first crossing lies in L, with the same target t.

(ii) If $W_L < t \le W_L + w(p)$, the first crossing is p, so returning it is correct.

(iii) Otherwise, the first crossing lies in R, with target $t - W_L - w(p)$.

Each retained target is positive and at most the retained weight, so the claim is preserved. With one record left, that record is x. The weight strictly below x is less than $W/2$, while the cumulative weight through x is at least $W/2$. The weight strictly above x is therefore at most $W/2$, proving that the returned value is a weighted median. □

Theorem 12.2. WEIGHTEDMEDIAN *takes* $O(n)$ *time.*

Proof. On s records, selection costs $O(s)$, and partitioning and summing weights also cost $O(s)$. Each nonreturning iteration reduces the size to at most $s/2$. The total is bounded by $O(n + n/2 + n/4 + \cdots) = O(n)$, including the initial weight sum. □

Problem · p. 4

13. Closest Pair with Equal Coordinates

Assume that n is a power of two. Let d be Euclidean distance; compare squared distances to avoid square roots. Stored x-ranks identify the two halves, even when coordinates tie.

Algorithm ClosestTies($P[1..n]$)

```
1 assign occurrence IDs; sort into X by (x, y, id)
2 store the ranks in X; sort the same points into Y by (y, x, id)
3 if adjacent entries of X coincide then return that pair
4 return EuclideanPair(X, Y)
```

Algorithm EuclideanPair($X[1..n], Y[1..n]$)

```
1 if n = 2 then return (X[1], X[2])
2 L ← X[1..n/2]; R ← X[n/2 + 1..n]
3 ℓ ← (X[n/2]_x + X[n/2 + 1]_x)/2; split Y by membership in L, R
4 e ← a closer pair from EuclideanPair(L, Y_L), EuclideanPair(R, Y_R)
5 δ ← d(e); S ← points of Y with |p_x − ℓ| < δ
6 for each S[i] and each of its next seven successors q do
7     if d(S[i], q) < d(e) then e ← (S[i], q)
8 return e
```

Theorem 13.1. ClosestTies *solves* Closest Pair with Equal Coordinates.

Proof. Coincident occurrences are adjacent in X and give optimum distance zero. Otherwise we prove that EuclideanPair returns a closest pair of its input. We use induction on n.

Base case. At $n = 2$, the only pair is optimal.

Induction step. The induction hypothesis gives the two half optima; let $\delta > 0$ be their smaller distance. A better pair must cross the split, and both points lie in S because their horizontal separation is less than δ. We claim that its later point is among the next seven successors of its earlier point p in S.

Consider the strip rectangle with $y \in [p_y, p_y + \delta)$. Divide each half's portion into four squares of side $\delta/2$, assigning points on the split by half membership. A cell contains at most one point of its half: otherwise two points are closer than δ, contradicting that half's optimum. Hence the rectangle contains at most eight points. The partner and all intervening entries lie there, proving the claim.

The scan checks every possible improvement, so the returned pair is optimal. □

Theorem 13.2. ClosestTies *takes* $O(n \log n)$ *time.*

Proof. The sorts cost $O(n \log n)$. Each combine step uses linear scans and at most seven comparisons per strip point. Thus $T(n) \le 2T(n/2) + O(n)$ with constant base cost. The Master Theorem gives $T(n) = O(n \log n)$, so the total time is $O(n \log n)$. □

Problem · p. 4

14. Bichromatic Closest Pair on Parallel Lines

Assume that n is a power of two. For an eligible pair, use squared distance $D(p, q) = 1 + (p_y - q_y)^2$.

Algorithm ParallelClosest($P[1..n]$)

```
1 sort P by y-coordinate using MergeSort; break ties arbitrarily
2 D_min ← +∞
3 for i ← 1 to n − 1 do
4     if P[i], P[i + 1] have different colours then
5         if D(P[i], P[i + 1]) < D_min then
6             e ← (P[i], P[i + 1]); D_min ← D(e)
7 return e
```

Theorem 14.1. ParallelClosest *solves* Bichromatic Closest Pair on Parallel Lines.

Proof. We prove that some closest eligible pair is consecutive in the sorted list. To prove this directly, take any optimum red–blue pair. Between its two positions there is a consecutive pair that changes colour. Its vertical gap is no larger than the optimum pair's gap. Every eligible pair has horizontal separation 1, so this consecutive pair has no greater Euclidean distance and is also optimal.

The scan keeps a closest checked eligible pair. Both colours occur, so it checks at least one pair and defines e. Every stored pair is eligible, and the claim ensures that one checked pair is optimal. Thus the output is optimal. □

Theorem 14.2. ParallelClosest *takes* $O(n \log n)$ *time.*

Proof. Sorting costs $O(n \log n)$. The scan makes $n - 1$ iterations, each doing constant work. The sum is $O(n \log n)$. □

Problem · p. 4

15. Manhattan Closest Pair

Assume that n is a power of two. Write $d(p, q) = |p_x - q_x| + |p_y - q_y|$. Stored x-ranks identify halves, even when coordinates tie.

Algorithm ManhattanClosest($P[1..n]$)

```
1 assign occurrence IDs; sort into X by (x, y, id)
2 store the ranks in X; sort the same points into Y by (y, x, id)
3 if adjacent entries of X coincide then return that pair
4 return ManhattanPair(X, Y)
```

Algorithm ManhattanPair($X[1..n], Y[1..n]$)

```
1 if n = 2 then return (X[1], X[2])
2 L ← X[1..n/2]; R ← X[n/2 + 1..n]
3 ℓ ← (X[n/2]_x + X[n/2 + 1]_x)/2; split Y by membership in L, R
4 e ← a closer pair from ManhattanPair(L, Y_L), ManhattanPair(R, Y_R)
5 δ ← d(e); S ← points of Y with |p_x − ℓ| < δ
6 for each S[i] and each of its next seventeen successors q do
7     if d(S[i], q) < d(e) then e ← (S[i], q)
8 return e
```

Theorem 15.1. ManhattanClosest *solves* Manhattan Closest Pair.

Proof. Coincident occurrences are adjacent in X and give optimum distance zero. Otherwise we prove that ManhattanPair returns a closest pair of its input. We use induction on n.

Base case. At $n = 2$, the only pair is optimal.

Induction step. The induction hypothesis gives both half optima; let $\delta > 0$ be their smaller distance. Any better pair crosses the split and lies in S. We claim that its later point is among the next seventeen successors of its earlier point p.

Its partner has $y \in [p_y, p_y + \delta)$. Divide each half's width-δ, height-δ rectangle into nine squares of side $\delta/3$, assigning split-line points by half membership. Each cell has at most one point of its half: otherwise two points have distance at most $2\delta/3 < \delta$, contradicting that half's optimum. Thus there are at most eighteen points, including p. The partner and all intervening entries lie there, proving the claim. Every improvement is checked, so the returned pair is optimal. □

Theorem 15.2. ManhattanClosest *takes* $O(n \log n)$ *time.*

Proof. The sorts cost $O(n \log n)$. Partitioning and forming the strip are linear; each strip point has at most seventeen constant-cost comparisons. Thus $T(n) \le 2T(n/2) + O(n)$ with constant base cost. The Master Theorem gives $T(n) = O(n \log n)$, so the total time is $O(n \log n)$. □

Problem · p. 4

16. Polynomial Multiplication

Assume that n is a power of two. Identify each coefficient array with its polynomial; addition and shifts below operate on coefficient arrays.

Algorithm PolyMultiply$(A[1..n], B[1..n])$

```
1 if n = 1 then return [A[1]B[1]]
2 h ← n/2; write A = A_0 + x^h A_1 and B = B_0 + x^h B_1
3 U ← PolyMultiply(A_0, B_0)
4 V ← PolyMultiply(A_1, B_1)
5 Z ← PolyMultiply(A_0 + A_1, B_0 + B_1)
6 return the 2n − 1 coefficients of U + x^h(Z − U − V) + x^{2h} V
```

Theorem 16.1. PolyMultiply *solves* Polynomial Multiplication.

Proof. We prove that PolyMultiply returns exactly the coefficient array of AB. We use induction on n.

Base case. For $n = 1$, scalar multiplication gives the constant product.

Induction step. All three recursive inputs have length $h = n/2$; adding coefficient arrays introduces no carries. The induction hypothesis gives $U = A_0B_0$, $V = A_1B_1$, and $Z = (A_0 + A_1)(B_0 + B_1)$. Thus $Z - U - V = A_0B_1 + A_1B_0$, and the returned polynomial is

$$A_0B_0 + x^h(A_0B_1 + A_1B_0) + x^{2h}A_1B_1 = (A_0 + x^hA_1)(B_0 + x^hB_1) = AB.$$

Its degree is at most $2n - 2$, so the $2n - 1$ returned coefficients are exactly the required output, including leading zeroes. □

Theorem 16.2. PolyMultiply *uses* $O(n^{\log_2 3})$ *scalar arithmetic operations.*

Proof. Adding and copying coefficient arrays and assembling the result cost $O(n)$ per call outside recursion. Thus $T(1) = O(1)$ and $T(n) \le 3T(n/2) + O(n)$. The Master Theorem gives $T(n) = O(n^{\log_2 3})$. □

Problem · p. 5

17. Truncated Integer Multiplication

The notation low_k returns exactly k bits, supplying leading zeroes when its argument is shorter. Let $n = a + b$.

Algorithm LowProduct$(X[1..a], Y[1..b], k)$

1 $U \leftarrow \text{low}_k(X)$; $V \leftarrow \text{low}_k(Y)$
2 $L \leftarrow$ the least power of two at least k
3 pad U, V with leading zeroes to length L
4 $Z \leftarrow$ Karatsuba(U, V)
5 **return** $\text{low}_k(Z)$

Theorem 17.1. LowProduct *solves* Truncated Integer Multiplication.

Proof. We prove that $xy \equiv uv \pmod{2^k}$, where x, y are the input values and u, v are the values retained in line 1. By a direct calculation, write $x = u + 2^k r$ and $y = v + 2^k s$ to obtain

$$xy = uv + 2^k(us + vr) + 2^{2k}rs \equiv uv \pmod{2^k}.$$

Padding preserves u, v, and the correctness of Karatsuba gives their exact product. The claim shows that its low k bits are the low k bits of xy. The fixed output length supplies any required leading zeroes. □

Theorem 17.2. LowProduct *uses* $O(n + k^{\log_2 3})$ *bit operations.*

Proof. Reading the inputs and forming the retained strings costs $O(n + k)$. Since $k \leq L < 2k$, padding costs $O(k)$ and Karatsuba costs $O(k^{\log_2 3})$. Extracting the result costs $O(k)$. Summing gives $O(n + k^{\log_2 3})$. □

Problem · p. 5

18. Signed Integer Multiplication

Each sign is $+$ or $-$. The two magnitudes have at most n bits each.

Algorithm SIGNEDPRODUCT$(s_X, X[1..a], s_Y, Y[1..b])$

```
1 L ← the least power of two at least max(a, b)
2 pad X, Y with leading zeroes to length L
3 Z ← KARATSUBA(X, Y)
4 if Z = 0 then return (+, [0])
5 s ← + if s_X = s_Y, and s ← − otherwise
6 remove leading zeroes from Z
7 return (s, Z)
```

Theorem 18.1. SIGNEDPRODUCT *solves* Signed Integer Multiplication.

Proof. Padding preserves the input magnitudes, so the correctness of KARATSUBA makes Z their exact product. We check the returned representation by cases.

(i) If the product is zero, line 4 returns the required unique representation of zero, regardless of the input signs.

(ii) If the product is nonzero, the signed product is positive exactly when the input signs agree. Line 5 chooses this sign, and removing leading zeroes preserves the magnitude. The returned pair therefore represents the required integer.

□

Theorem 18.2. SIGNEDPRODUCT *uses* $O(n^{\log_2 3})$ *bit operations.*

Proof. The padded length satisfies $L < 2n$, so the multiplication costs $O(n^{\log_2 3})$. Padding, testing for zero, and removing leading zeroes scan $O(n)$ bits. The sign computation is constant-time. Their sum is $O(n^{\log_2 3})$. □

Problem · p. 5

19. Rectangular Matrix Multiplication

Algorithm RectMultiply$(A[1..n, 1..m], B[1..m, 1..k])$

```
1 L ← the least power of two at least max(n, m, k)
2 initialize zero matrices A'[1..L, 1..L] and B'[1..L, 1..L]
3 A'[1..n, 1..m] ← A; B'[1..m, 1..k] ← B
4 D ← Strassen(A', B')
5 return D[1..n, 1..k]
```

Theorem 19.1. RectMultiply *solves* Rectangular Matrix Multiplication.

Proof. We prove that $(A'B')[i, j] = (AB)[i, j]$ whenever $i \le n$ and $j \le k$. By a direct calculation using the zero padding,

$$(A'B')[i, j] = \sum_{t=1}^{L} A'[i, t]B'[t, j] = \sum_{t=1}^{m} A[i, t]B[t, j] = (AB)[i, j].$$

The correctness of Strassen gives $D = A'B'$. By the claim, the block returned in line 5 is exactly AB. □

Theorem 19.2. *For $N = \max(n, m, k)$,* RectMultiply *uses $O(N^{\log_2 7})$ scalar arithmetic operations.*

Proof. We have $N \le L < 2N$. The square product costs $O(L^{\log_2 7}) = O(N^{\log_2 7})$. Initializing the two squares, copying the input blocks, and extracting the output cost $O(L^2)$. Since $\log_2 7 > 2$, this additional work is absorbed by the product bound. □

Problem · p. 5

20. Matrix Powers

Assume that n is a power of two. Products use STRASSEN with ordinary integer arithmetic.

Algorithm POWER$(A[1..n, 1..n], k)$

```
1 if k = 0 then return I_n
2 if k = 1 then return A
3 if k is even then h ← k/2 else h ← (k − 1)/2
4 B ← POWER(A, h)
5 C ← STRASSEN(B, B)
6 if k is even then return C
7 return STRASSEN(C, A)
```

Theorem 20.1. POWER *solves* Matrix Powers.

Proof. We prove that POWER(A, k) returns A^k. We use induction on k.

Base cases. The returned matrices for $k = 0, 1$ are $I_n = A^0$ and A.

Induction step. For $k \geq 2$, $h < k$, so the induction hypothesis gives $B = A^h$. The correctness of STRASSEN gives $C = B^2 = A^{2h}$.

There are two cases:

(i) If k is even, $2h = k$, so returning C is correct.

(ii) If k is odd, $2h + 1 = k$, so the final product is $CA = A^k$.

□

Theorem 20.2. POWER *uses* $O(n^{\log_2 7} \log k)$ *scalar arithmetic operations.*

Proof. For $k \geq 2$, each recursive exponent is at most half the preceding one, giving $O(\log k)$ calls. Each nonbase call performs at most two matrix products, each costing $O(n^{\log_2 7})$. Copying matrices adds only $O(n^2)$ per call. The base cases use $O(n^2)$ operations to form the identity or return the input. Since $\log_2 7 > 2$, the total is $O(n^{\log_2 7} \log k)$ scalar arithmetic operations. □

This bound counts scalar operations; the entries may have large bit lengths.

Problem · p. 5

21. Boolean Matrix Multiplication

Assume that n is a power of two. Use ordinary integer arithmetic inside Strassen, and convert the final entries to Boolean values.

Algorithm BooleanProduct$(A[1..n, 1..n], B[1..n, 1..n])$

```
1 D ← Strassen(A, B)
2 allocate C[1..n, 1..n]
3 for each (i, j) ∈ {1, ..., n}^2 do
4     C[i, j] ← 1 if D[i, j] > 0, and C[i, j] ← 0 otherwise
5 return C
```

Theorem 21.1. BooleanProduct *solves* Boolean Matrix Multiplication.

Proof. We prove that $D[i, j] > 0$ exactly when some t has $A[i, t] = B[t, j] = 1$. The correctness of Strassen gives the ordinary integer product, so

$$D[i, j] = \sum_{t=1}^{n} A[i, t]B[t, j].$$

Each summand is zero or one, and equals one precisely for such a witness t. The sum is positive exactly when a witness exists, proving the claim directly. Line 4 therefore gives the required Boolean value at every position. □

Theorem 21.2. BooleanProduct *uses* $O(n^{\log_2 7})$ *scalar arithmetic operations.*

Proof. The integer product costs $O(n^{\log_2 7})$. Allocating and filling C costs $O(n^2)$, which is absorbed because $\log_2 7 > 2$. The sum is $O(n^{\log_2 7})$. □

Problem · p. 6

22. Maximum-Weight Independent Set

Write $N[v]$ for v and its neighbours, and $w(S) = \sum_{v \in S} w(v)$. Induced subgraphs retain the weights and original vertex labels.

Algorithm WeightedIS(G, w)

```
1 if G has no edges then return V(G)
2 choose a vertex v of maximum degree
3 S_0 ← WeightedIS(G − {v}, w)
4 S_1 ← {v} ∪ WeightedIS(G − N[v], w)
5 return a set of larger weight among S_0, S_1
```

Theorem 22.1. WeightedIS *solves* Maximum-Weight Independent Set.

Proof. We prove that the returned set is independent and has maximum weight. We use induction on the number of vertices.

Base case. If there are no edges, all vertices form an independent set. Nonnegative weights make it optimal, including for the empty graph.

Induction step. Classify independent sets by whether they contain v.

(i) If v is excluded, these are exactly the independent sets of $G - \{v\}$. The induction hypothesis makes S_0 an optimum in this class.

(ii) If v is included, removing v gives an independent set of $G - N[v]$, and every such set extends to one containing v. All weights increase by $w(v)$ under this extension. The induction hypothesis therefore makes S_1 an optimum in this class.

The classes are exhaustive and both candidates are feasible. Returning the heavier class optimum gives a global optimum. □

Theorem 22.2. *For $n \geq 1$,* WeightedIS *takes $O(n^2\varphi^n)$ time, where $\varphi = (1 + \sqrt{5})/2$.*

Proof. We prove that a call on s vertices has at most φ^s leaves in its recursion tree. We use induction on s.

Base case. A terminal call has one leaf, at most φ^s.

Induction step. The chosen vertex has positive degree, so the children have $s-1$ and at most $s-2$ vertices. The induction hypothesis bounds their total leaves by $\varphi^{s-1}+\varphi^{s-2} = \varphi^s$.

A full binary tree has twice as many nodes as leaves, minus one, so there are $O(\varphi^n)$ calls. With adjacency matrices, choosing a vertex, forming the induced graphs, and computing or copying weighted answers take $O(n^2)$ time per call, where n is the original size. Thus the total time is $O(n^2\varphi^n)$. □

Problem · p. 6

23. Counting Independent Sets

Write $N[v]$ for v together with all its neighbours.

Algorithm CountIS(G)

```
1 if G has no edges then return 2^|V(G)|
2 choose a vertex v of maximum degree
3 c_0 ← CountIS(G − {v})
4 c_1 ← CountIS(G − N[v])
5 return c_0 + c_1
```

Theorem 23.1. CountIS *solves* Counting Independent Sets.

Proof. We prove that the returned number is the number of independent sets. We use induction on the number of vertices.

Base case. In an edgeless graph every subset is independent, giving $2^{|V(G)|}$ choices, including one choice for the empty graph.

Induction step. Partition the independent sets into two classes.

(i) If v is excluded, these are exactly the independent sets of $G - \{v\}$, so the induction hypothesis gives their count c_0.

(ii) If v is included, removing v gives a bijection with the independent sets of $G - N[v]$, since all neighbours of v must be excluded. The induction hypothesis gives their count c_1.

The classes are disjoint and exhaustive, so $c_0 + c_1$ is the required count. □

Theorem 23.2. *For $n \geq 1$,* CountIS *takes $O(n^2\varphi^n)$ time, including arithmetic on the counts, where $\varphi = (1 + \sqrt{5})/2$.*

Proof. We prove that a call on s vertices has at most φ^s leaves in its recursion tree. We use induction on s.

Base case. A terminal call has one leaf, at most φ^s.

Induction step. The children have $s - 1$ and at most $s - 2$ vertices. By the induction hypothesis, their total leaves are at most $\varphi^{s-1} + \varphi^{s-2} = \varphi^s$.

The full binary recursion tree therefore has $O(\varphi^n)$ nodes. Adjacency-matrix operations take $O(n^2)$ time per node, using the original size n. Every count is at most 2^n and uses at most $n + 1$ bits, so addition takes $O(n)$ bit operations. Writing a base value 2^s as one followed by s zero bits also takes $O(n)$ time. The graph work dominates these costs, giving total time $O(n^2\varphi^n)$. □

Problem · p. 6

24. Minimum Vertex Cover

Algorithm MinVertexCover(G)
1 $S \leftarrow$ IndependentSet(G)
2 **return** $V(G) \setminus S$

Theorem 24.1. MinVertexCover *solves* Minimum Vertex Cover.

Proof. We prove that C is a vertex cover if and only if $V \setminus C$ is independent. Indeed, an uncovered edge is exactly an edge with both endpoints outside C. This proves both directions directly.

Taking complements thus gives a bijection between covers and independent sets, with $|C| = n - |V \setminus C|$. The correctness of IndependentSet gives a largest independent set S. Its complement is therefore a smallest vertex cover, including when the graph is empty. □

Theorem 24.2. *For $n \geq 1$,* MinVertexCover *takes $O(n^2\varphi^n)$ time, where $\varphi = (1 + \sqrt{5})/2$.*

Proof. IndependentSet costs $O(n^2\varphi^n)$. Marking the returned vertices and scanning V to construct the complement adds $O(n)$. The total is $O(n^2\varphi^n)$. □

Problem · p. 6

Chapter 12

Graph Search

25. Reachability from a Set

Algorithm SetReach(G, S, t)

```
1 form H by adding a new vertex z and the edges zs for s ∈ S
2 R ← BFS(H, z)
3 return R[t]
```

Theorem 25.1. SetReach *solves* Reachability from a Set.

Proof. We prove that H has a z–t path if and only if some $s \in S$ has an s–t path in G. We prove both directions.

($\Rightarrow$) A z–t path starts with zs for some $s \in S$. It never revisits z, so its remaining edges form an s–t path in G.

($\Leftarrow$) Prepending zs to an s–t path gives a z–t path in H, including when $s = t$.

By the correctness of BFS, $R[t]$ tests exactly this reachability in H. The claim therefore gives the required answer. □

Theorem 25.2. SetReach *takes* $O(n + m)$ *time.*

Proof. The new graph has $n + 1$ vertices and $m + |S|$ edges. Constructing its lists costs $O(n + m)$ since $|S| \leq n$. BFS has the same bound; the final lookup is constant time. □

Problem · p. 7

26. Reachability after Deletions

Each edge has one identifier shared by its two adjacency entries.

Algorithm DeletedReach(G, s, t, F)

```
1 initialize all edge marks to false; mark each identifier in F
2 build H by copying all vertices and only unmarked edges
3 R ← BFS(H, s); return R[t]
```

Theorem 26.1. DeletedReach *solves* Reachability after Deletions.

Proof. The filtering step retains exactly $E \setminus F$, so $H = (V, E \setminus F)$. We prove that a path in G avoids the deleted edges if and only if it is a path in H.

($\Rightarrow$) Every edge of a path avoiding F is retained in H.

($\Leftarrow$) Every edge of a path in H lies outside F.

The correctness of BFS therefore makes $R[t]$ true exactly when a required path exists. □

Theorem 26.2. DeletedReach *takes* $O(n + m)$ *time.*

Proof. Marking takes $O(m + |F|) = O(m)$ time. Copying the vertex and adjacency lists takes $O(n + m)$. The graph H has n vertices and at most m edges, so BFS also takes $O(n + m)$ time. Their sum gives the bound. □

Problem · p. 7

27. Even-Length Reachability

Algorithm EvenWalk(G, s, t)

```
1 create H with vertices v_0, v_1 for every v ∈ V
2 for each uv ∈ E do
3     add undirected edges u_0v_1 and u_1v_0 to H
4 R ← BFS(H, s_0); return R[t_0]
```

Theorem 27.1. EvenWalk *solves* Even-Length Reachability.

Proof. We prove that an even s–t walk exists in G if and only if t_0 is reachable from s_0 in H. We prove both directions.

($\Rightarrow$) Suppose G has an even s–t walk. Lift it by replacing its jth vertex v with $v_{j \bmod 2}$. Each step is an edge of H by construction. The lifted walk starts at s_0 and ends at t_0, so t_0 is reachable.

($\Leftarrow$) Suppose t_0 is reachable from s_0 in H. Replace each vertex v_i of an s_0–t_0 path by v. Each resulting edge belongs to G. Every edge of H changes the subscript, so the resulting s–t walk has even length.

The correctness of BFS makes $R[t_0]$ true exactly when t_0 is reachable from s_0 in H. By the equivalence above, the returned answer is true exactly when G contains an even s–t walk. □

Theorem 27.2. EvenWalk *takes $O(n + m)$ time.*

Proof. The graph H has $2n$ vertices and $2m$ edges, and is built in $O(n + m)$ time. Its BFS costs $O(n + m)$. All remaining work is constant. □

Problem · p. 7

28. Multiple-Source Shortest Paths

Use the parent tree and depth array returned by BFSTree.

Algorithm SourceDistances(G, S)

```
1 add a new vertex z adjacent to every vertex of S, obtaining H
2 (p, h) ← BFSTree(H, z); build child lists from p
3 traverse this tree from its root toward its leaves
4     for a child v of z, set a[v] ← v
5     for every other reached v ≠ z, set a[v] ← a[p[v]]
6 return (h[v] − 1, a[v]) for reached v ≠ z, and (+∞, nil) otherwise
```

Theorem 28.1. SourceDistances *solves* Multiple-Source Shortest Paths.

Proof. We prove that for every reachable v, $\mathrm{dist}_H(z, v) = 1 + \min_{s \in S} \mathrm{dist}_G(s, v)$. A path from z first enters some $s \in S$ and then stays in G; conversely, prepending zs to a path in G adds exactly one edge. These two directions prove the equality and also identify the unreachable vertices.

The BFS tree lemma makes $h[v]$ the length of a shortest parent-tree path from z to v. We now prove that $a[v]$ is the first vertex after z on this path. We use induction on tree depth.

Base case. At depth one, the vertex is assigned itself.

Induction step. A deeper vertex inherits its parent's first vertex, which is correct by the induction hypothesis. Thus $a[v] \in S$ attains distance $h[v] - 1$, as required by the claim. □

Theorem 28.2. SourceDistances *takes* $O(n + m)$ *time.*

Proof. The augmented graph has $n + 1$ vertices and at most $m + n$ edges. Construction and BFSTree each take $O(n + m)$ time. Building child lists, traversing the tree, and writing the output add $O(n)$. □

Problem · p. 7

29. Counting Shortest Paths

Counts are stored as binary integers of sufficient length.

Algorithm CountShortest(G, s)

```
1 d ← BFSPaths(G, s); bucket finite-distance vertices by d
2 set c[v] ← 0 for all v; c[s] ← 1
3 for each vertex u in increasing distance order do
4     for each v ∈ Adj[u] with d[v] = d[u] + 1 do
5         c[v] ← c[v] + c[u]
6 return c
```

Theorem 29.1. CountShortest *solves* Counting Shortest Paths.

Proof. The correctness of BFSPaths gives the true distances d. We prove that, when a vertex v is processed, $c[v]$ equals its number of shortest paths from s. We use induction on $d[v]$.

Base case. At distance zero only s occurs, and its one shortest path is the empty path.

Induction step. Every shortest path to v ends with an edge uv where $d[u] = d[v] - 1$, preceded by a shortest path to u. Conversely, appending uv to any such path gives a shortest path to v; strictly increasing distances prevent repetition. Partitioning paths by u therefore gives their count as the sum of the predecessor counts. The induction hypothesis makes these counts correct, and exactly these updates occur before v is processed. Unreachable vertices receive no updates and keep count zero. □

Theorem 29.2. CountShortest *takes $O(n(n+m))$ time, counting bit operations on path counts.*

Proof. BFS and bucketing cost $O(n+m)$; finite distances are at most $n-1$. There are at most $2m$ count additions. A shortest path uses strictly increasing distance layers, so its set of vertices determines their order. There are at most 2^n such sets, and every count has at most $n+1$ bits. An addition therefore costs $O(n)$ bit operations. Initializing and returning n counts costs at most $O(n^2)$, giving $O(n(n+m))$ total time. □

Problem · p. 8

30. Edges on Shortest Paths

Algorithm ShortestEdges(G, s, t)

```
1 a ← BFSPaths(G, s); b ← BFSPaths(G, t)
2 D ← a[t]; if D = +∞ then return ∅
3 return all uv ∈ E with a[u] + 1 + b[v] = D or a[v] + 1 + b[u] = D
```

Theorem 30.1. ShortestEdges *solves* Edges on Shortest Paths.

Proof. The correctness of BFSPaths gives the distance arrays. If $D = +\infty$, no path exists, so the empty output is correct. Otherwise we prove that an edge is on a shortest path exactly when it satisfies one of the two tests.

($\Rightarrow$) If a shortest path traverses uv from u to v, its prefix and suffix are shortest paths, giving $a[u] + 1 + b[v] = D$. The opposite orientation gives the other equality.

($\Leftarrow$) If $a[u] + 1 + b[v] = D$, join shortest paths from s to u and from v to t using uv. This walk has length D. For a contradiction, suppose a vertex repeats. Removing the nonempty closed portion would give an s–t walk shorter than D. Hence the walk is a shortest path containing uv. The reverse equality is symmetric. □

Theorem 30.2. ShortestEdges *takes* $O(n + m)$ *time.*

Proof. Each BFSPaths call costs $O(n + m)$. Listing each undirected edge once and testing two equalities adds $O(n + m)$ time. The output has at most m edges. □

Problem · p. 8

31. Component Weights

Algorithm ComponentWeights(G, w)

```
1 (c, k) ← Components(G); initialize W[1..k] ← 0
2 for each v ∈ V do W[c[v]] ← W[c[v]] + w[v]
3 return (c, W)
```

Theorem 31.1. ComponentWeights *solves* Component Weights.

Proof. The correctness of Components gives exactly the connected-component labels. We prove that $W[j]$ is the sum of the weights of processed vertices with label j. We use induction on the number of processed vertices.

Base case. No vertices have been processed and every sum is zero.

Induction step. Processing v adds its weight to precisely the sum for $c[v]$ and leaves the others unchanged.

After all vertices are processed, the invariant gives every component's weight, so (c, W) is the required output. □

Theorem 31.2. ComponentWeights *takes* $O(n + m)$ *time.*

Proof. Components costs $O(n + m)$. There are at most n component labels. Initializing the sums and adding each vertex weight once take $O(n)$ additional time under the stated arithmetic convention. □

Problem · p. 8

32. Connecting a Graph

Algorithm ConnectGraph(G)

```
(c, k) ← Components(G)
choose one representative r_j of each component j
return {r_1 r_j : 2 ≤ j ≤ k}
```

Theorem 32.1. ConnectGraph *solves* Connecting a Graph.

Proof. The correctness of Components gives the exact k components. We prove feasibility and optimality separately.

Feasibility. The returned edges join every component to the first. Their endpoints were in different components, so they are new edges, and the augmented graph is connected.

Optimality. A single new edge reduces the number of components by at most one. Reducing k components to one therefore needs at least $k-1$ edges. The algorithm returns exactly that many, meeting the lower bound. □

Theorem 32.2. ConnectGraph *takes* $O(n+m)$ *time.*

Proof. Components takes $O(n+m)$. A vertex scan selects representatives in $O(n)$ time, and the $k-1 \le n-1$ output edges take $O(n)$ time to list. □

Problem · p. 8

33. Clique Components

Algorithm CliqueComponents(G)

```
1 (c, k) ← Components(G); set s[1..k], e[1..k] ← 0
2 for each v ∈ V do s[c[v]] ← s[c[v]] + 1
3 for each undirected edge uv once do e[c[u]] ← e[c[u]] + 1
4 return whether 2e[j] = s[j](s[j] − 1) for every j
```

Theorem 33.1. CliqueComponents *solves* Clique Components.

Proof. The correctness of Components and the two scans give each component's vertex count $s[j]$ and edge count $e[j]$. We prove that a component is a clique exactly when $2e[j] = s[j](s[j] - 1)$.

A simple graph on $s[j]$ vertices has one possible edge per unordered pair, so at most $s[j](s[j] - 1)/2$ edges. Equality holds precisely when every pair is an edge. This proves the claim directly. Testing it for every component therefore gives exactly the required answer. □

Theorem 33.2. CliqueComponents *takes* $O(n + m)$ *time.*

Proof. Components takes $O(n + m)$. The vertex and edge scans take $O(n + m)$, and there are at most n final tests. Counts and their products fit in a constant number of words, so each update and comparison takes constant time. □

Problem · p. 8

34. Extending a Partial Two-Colouring

Algorithm ExtendColours(G, P, b)

```
1 run Bipartite(G); if it returns an odd cycle then return impossible
2 let a[v] ∈ {0,1} be its colouring; (c, k) ← Components(G)
3 initialize f[1..k] ← nil
4 for each v ∈ P do
5     z ← a[v] ⊕ b[v]
6     if f[c[v]] ≠ nil and f[c[v]] ≠ z then return impossible
7     f[c[v]] ← z
8 replace each undefined f[j] by 0; return a[v] ⊕ f[c[v]] for all v
```

Theorem 34.1. ExtendColours *solves* Extending a Partial Two-Colouring.

Proof. The correctness of Bipartite rules out a colouring when an odd cycle is found; otherwise it gives a proper colouring a.

We prove that a two-colouring b' is proper if and only if it is obtained from a by choosing one flip bit per component.

($\Rightarrow$) For a proper colouring b', the value $a[v] \oplus b'[v]$ agrees at the two ends of each edge, hence throughout each component.

($\Leftarrow$) Flipping a whole component preserves properness.

Every prescribed vertex forces its component's flip bit to be $a[v] \oplus b[v]$. Conflicting forced bits make an extension impossible by the claim. Otherwise the chosen bits give a proper colouring satisfying all prescriptions, so the returned extension is correct. □

Theorem 34.2. ExtendColours *takes* $O(n + m)$ *time.*

Proof. Bipartite and Components cost $O(n + m)$. Initializing flip bits, scanning the at most n prescribed vertices, and writing the colouring add $O(n)$. □

Problem · p. 9

35. Counting Two-Colourings

Algorithm CountColours(G)

```
1 if Bipartite(G) returns an odd cycle then return 0
2 (c, k) ← Components(G)
3 return the binary integer consisting of 1 followed by k zeroes
```

Theorem 35.1. CountColours *solves* Counting Two-Colourings.

Proof. If Bipartite finds an odd cycle, its correctness shows that zero is the required answer. Otherwise fix a proper colouring.

We prove that each connected component has exactly two proper two-colourings. Fixing the colour of one vertex forces every other colour along paths. Thus there are at most two choices. The fixed colouring and its complete flip supply two distinct choices, including for an isolated vertex.

Different components share no edges, so their choices are independent. The claim gives 2^k colourings, which is exactly the returned binary integer. For the empty graph, $k = 0$ gives the single empty assignment. □

Theorem 35.2. CountColours *takes $O(n + m)$ time, including writing the answer in binary.*

Proof. The calls to Bipartite and Components take $O(n + m)$. The result 2^k has $k + 1 \leq n + 1$ bits and is written directly, without repeated large-integer multiplication. This adds $O(n)$ time. □

Problem · p. 9

36. Parity Constraints

Algorithm ParityAssignment(G, b)

```
1 initialize H with the original vertices and no edges
2 for each uv ∈ E do
3     if b_uv = 1 then add uv to H
4     else add a new vertex z_uv and edges u z_uv, z_uv v
5 run Bipartite(H)
6 return impossible if it returns an odd cycle; otherwise restrict its colouring to V
```

Theorem 36.1. ParityAssignment *solves* Parity Constraints.

Proof. We prove that the constraints are satisfiable exactly when H is bipartite. We prove both directions.

($\Rightarrow$) Given a satisfying assignment, keep its colours on V and colour each z_{uv} opposite to its two equal-coloured neighbours. All edges of H then have opposite-coloured endpoints.

($\Leftarrow$) Given a proper colouring of H, an original edge with $b_{uv} = 1$ forces opposite colours. For $b_{uv} = 0$, the two-edge path through z_{uv} forces equal colours at u, v. Restricting a proper colouring of H to V therefore satisfies every constraint.

The correctness of Bipartite and this equivalence justify both the impossibility report and the returned assignment. □

Theorem 36.2. ParityAssignment *takes* $O(n + m)$ *time.*

Proof. The graph H has at most $n + m$ vertices and $2m$ edges. Construction and Bipartite therefore take $O(n + m)$ time. Restricting the colouring costs $O(n)$, so the total is $O(n + m)$. □

Problem · p. 9

37. Path Recovery

Algorithm RecoverPath(G, s, t)
1 $(p, a, f) \leftarrow$ DFSTree(G, s)
2 **if** $a[t] =$ nil **then return** no path
3 initialize an empty list L; $v \leftarrow t$
4 **while** $v \neq s$ **do** append v to L; $v \leftarrow p[v]$
5 append s to L; reverse L; **return** L

Theorem 37.1. RecoverPath *solves* Path Recovery.

Proof. The DFS tree lemma states that the reached vertices are exactly those reachable from s, and that their parent edges form a tree rooted at s. We distinguish the two outputs.

(i) If t is unreached, an undefined start time means that t is unreachable, so reporting no path is correct.

(ii) If t is reached, its parent chain is the unique tree path to s. Reversing it gives an s–t path in G, with distinct vertices. If $s = t$, the singleton output is the length-zero path.

□

Theorem 37.2. RecoverPath *takes* $O(n + m)$ *time.*

Proof. DFSTree takes $O(n + m)$ time. The parent chain has at most n vertices, and building and reversing its list costs $O(n)$. The total is $O(n + m)$. □

Problem · p. 9

38. Subtree Sizes

Algorithm SubtreeSizes(F)

```
1 procedure Visit(u)
2     s[u] ← 1
3     for each child v of u do
4         Visit(v); s[u] ← s[u] + s[v]
5 for each root r do Visit(r)
6 return s
```

Theorem 38.1. SubtreeSizes *solves* Subtree Sizes.

Proof. We prove that after Visit(u), every vertex in u's subtree has its correct subtree size. We use induction on the size of that subtree.

Base case. A leaf has only itself, so its assigned value one is correct.

Induction step. The induction hypothesis gives correct sizes throughout each child subtree. These subtrees are disjoint and, together with u, partition u's subtree. Therefore $1 + \sum_v s[v]$ is its size. The child answers remain unchanged, completing the induction.

The outer loop visits each rooted tree, so every output entry is correct. □

Theorem 38.2. SubtreeSizes *takes* $O(n)$ *time.*

Proof. Each vertex has one call and is initialized once. Every parent–child edge is processed once, with one addition and constant loop work. A forest has at most $n - 1$ edges, so the total is $O(n)$, including the root scan and output. □

Problem · p. 9

39. Ancestor Queries

Algorithm ANCESTORS($F, Q[1..q]$)

```
1 t ← 0; procedure VISIT(u)
2     t ← t + 1; a[u] ← t
3     for each child v of u do VISIT(v)
4     t ← t + 1; f[u] ← t
5 for each root r do VISIT(r)
6 for each (u, v) ∈ Q do output a[u] ≤ a[v] and f[v] ≤ f[u]
```

Theorem 39.1. ANCESTORS *solves* Ancestor Queries.

Proof. We prove that $[a[v], f[v]]$ is contained in $[a[u], f[u]]$ exactly when v belongs to u's subtree, including u itself.

First, we claim that the call on u performs exactly its subtree's events between its entry and exit. We use induction on subtree size.

Base case. A leaf performs only its own entry and exit.

Induction step. By the induction hypothesis, each child call performs exactly that child's subtree events. These calls occur consecutively between u's entry and exit, proving the subclaim.

($\Rightarrow$) We use contraposition. Suppose v lies outside u's subtree. There are two cases:

(i) If v is a proper ancestor of u, its interval strictly contains u's interval, so it cannot be contained in it.

(ii) Otherwise the two vertices lie in different child subtrees or different rooted trees. Their calls occur separately, so their intervals are disjoint.

($\Leftarrow$) If v belongs to u's subtree, its entry and exit occur between those of u by the subclaim, so its interval is contained in u's.

The two non-strict comparisons test exactly this interval containment. □

Theorem 39.2. ANCESTORS *takes* $O(n + q)$ *time.*

Proof. There are two clock updates per vertex and one traversal per parent–child edge, giving $O(n)$ preprocessing. Every query uses two comparisons and one output operation, taking $O(1)$ time. The total is $O(n + q)$. □

Problem · p. 9

40. Components after Vertex Deletion

Build each component as a separate adjacency-list graph with local labels; retain a map to the original vertices.

Algorithm DeletionCounts(G)

```
(c, k) ← Components(G); build its component graphs
for each component graph H with any root r do
    (p, a, f) ← DFSTree(H, r); ℓ ← LowValues(H, p, a, f)
    for each u ∈ V(H) do
        q ← number of children of u in p
        b ← number of children v with ℓ[v] ≥ a[u]
        d[u] ← k − 1 + q if u = r, and d[u] ← k + b otherwise
return d, in original vertex order
```

Theorem 40.1. DeletionCounts *solves* Components after Vertex Deletion.

Proof. The correctness of Components and LowValues gives the component partition and low values. Fix u in component H; the other $k-1$ components survive unchanged. We determine the components of $H - u$ by cases.

(i) If u is a root, distinct child subtrees have no edges between them, since an undirected DFS has no cross edges. Each remains connected by its tree edges, giving exactly q components, including zero for a singleton.

(ii) If u is not a root, a child subtree with $\ell[v] \geq a[u]$ has no edge to a strict ancestor of u, so it becomes a separate component. Every other child subtree has such an edge and joins the connected part containing the root. The absence of cross edges rules out further connections between separated subtrees. Thus $H - u$ has exactly $b + 1$ components.

Adding the unchanged components yields $k - 1 + q$ or $k + b$, as returned. □

Theorem 40.2. DeletionCounts *takes* $O(n + m)$ *time.*

Proof. Components and component-graph construction take $O(n + m)$ time. On each H, DFSTree and LowValues take $O(|V(H)| + |E(H)|)$ time. Count children and qualifying children in one scan of the parent array. Component sizes sum to n and edge counts to m, so the total is $O(n + m)$. □

Problem · p. 10

41. Cut Vertices in a Disconnected Graph

Algorithm AllCutVertices(G)

```
(c, k) ← Components(G)
build induced component graphs H_1, ..., H_k with local labels
for j = 1, ..., k do A_j ← CutVertices(H_j)
return the union of the A_j, mapped to original vertex labels
```

Theorem 41.1. AllCutVertices *solves* Cut Vertices in a Disconnected Graph.

Proof. We prove that $v \in V(H_j)$ is a cut vertex of G exactly when it is a cut vertex of H_j. Deletion leaves all other components unchanged, so the new total component count is $k-1$ plus the number of components in $H_j - v$. It exceeds k exactly when $H_j - v$ has more than one component, proving the claim directly.

The correctness of Components supplies the graphs H_j, and the correctness of CutVertices identifies their cut vertices. The claim therefore makes the returned union exactly the required answer. □

Theorem 41.2. AllCutVertices *takes* $O(n+m)$ *time.*

Proof. Components costs $O(n+m)$. One vertex scan and one edge scan build all locally labelled graphs in $O(n+m)$ time. The calls cost $\sum_j O(|V(H_j)| + |E(H_j)|) = O(n+m)$. Mapping and joining the returned lists adds $O(n)$. □

Problem · p. 10

42. Separated Pairs after Vertex Deletion

Write a for DFS start times and ℓ for low values. Compute subtree sizes by summing child sizes in finish order.

Algorithm SeparatedPairs(G)

```
1 choose r; (p, a, f) ← DFSTree(G, r)
2 ℓ ← LowValues(G, p, a, f); build child lists
3 in increasing finish order, set s[u] ← 1 + Σ_{v child of u} s[v]
4 for each u ∈ V do
5     B ← [s[v] : v child of u,  u = r or ℓ[v] ≥ a[u]]
6     z ← n − 1 − Σ_{b∈B} b
7     q[u] ← ((n − 1)^2 − z^2 − Σ_{b∈B} b^2)/2
8 return q
```

Theorem 42.1. SeparatedPairs *solves* Separated Pairs after Vertex Deletion.

Proof. We prove that line 3 computes correct subtree sizes. We use induction on subtree size.

Base case. A leaf contributes one, its own size.

Induction step. The child subtrees have correct sizes by the induction hypothesis. They are disjoint and, together with the root, partition the subtree, so summing their sizes and adding one is correct.

We claim that B, together with the residual size z when nonzero, gives exactly the component sizes of $G - u$.

(i) If u is a root, each child subtree is connected and there are no cross edges between them. These are exactly the remaining components, and $z = 0$.

(ii) If u is not a root, by the low-value definition, precisely the children with $\ell[v] \geq a[u]$ have no edge to a strict ancestor of u. Their subtrees become separate components. Every other child subtree joins the connected part containing the root. The absence of cross edges proves the claim.

The sizes sum to $n - 1$. For component sizes b_i, $(\sum_i b_i)^2 - \sum_i b_i^2 = 2\sum_{i<j} b_i b_j$. The right side counts twice the pairs in different components, so line 7 gives exactly the number of separated pairs. □

Theorem 42.2. SeparatedPairs *takes $O(n + m)$ time.*

Proof. DFSTree and LowValues take $O(n + m)$ time. Finish times lie in $[1, 2n]$, so buckets give their order in $O(n)$ time. Each child entry is used a constant number of times in the size and deletion scans. Counts are at most n^2, so their arithmetic has constant cost per entry. The total is $O(n + m)$. □

Problem · p. 10

43. Edges on Cycles

Algorithm CycleEdges(G)

```
1 run Components(G) and build locally labelled component graphs
2 run CutEdges on each component; mark the returned original edges
3 return all unmarked edges of G
```

Theorem 43.1. CycleEdges *solves* Edges on Cycles.

Proof. We prove that an edge lies on a cycle if and only if it is not a cut edge of its component. We prove both directions.

($\Rightarrow$) If the edge lies on a cycle, the rest of that cycle gives an alternate path between the edge's endpoints. Any path using the edge can take this detour, so deleting it leaves the component connected.

($\Leftarrow$) If deletion preserves connectivity, a simple path between the endpoints remains. Adding the deleted edge forms a cycle.

The correctness of Components and CutEdges identifies exactly the cut edges. By the claim, their complement is the required output. □

Theorem 43.2. CycleEdges *takes* $O(n + m)$ *time.*

Proof. Components and graph construction take $O(n + m)$. The separate cut-edge calls cost the sum of their component sizes and edge counts, namely $O(n + m)$. Marking and listing edges adds $O(m)$. □

Problem · p. 10

44. Components after Removing All Cut Edges

Algorithm BridgeComponents(G)

```
1 run Components(G) and build locally labelled component graphs
2 run CutEdges on each component; let B be their original edges
3 build H = (V, E \ B) by marking and filtering the edges
4 (c, k) ← Components(H); return c
```

Theorem 44.1. BridgeComponents *solves* Components after Removing All Cut Edges.

Proof. We prove that an edge is a cut edge of G exactly when it is a cut edge of the component containing it. Deleting it leaves every other component unchanged, so the two component-count increases are identical. This proves the claim directly.

The correctness of Components and CutEdges therefore makes B exactly the set of cut edges of G. Filtering builds the specified graph $(V, E \setminus B)$. The final call to Components correctly labels its connected components. □

Theorem 44.2. BridgeComponents *takes* $O(n + m)$ *time.*

Proof. Both component computations cost $O(n + m)$. The locally labelled cut-edge calls cost $O(n + m)$ altogether. Building component graphs, marking edges, and constructing H each take $O(n+m)$. A constant number of these operations gives the stated bound. □

Problem · p. 10

45. Unavoidable Edges

Algorithm UnavoidableEdges(G, s, t)

```
1 B ← CutEdges(G); mark these edges
2 (p, d) ← BFSTree(G, s); v ← t; A ← ∅
3 while v ≠ s do
4     if p[v]v is marked then append it to A
5     v ← p[v]
6 return A
```

Theorem 45.1. UnavoidableEdges *solves* Unavoidable Edges.

Proof. The BFS tree lemma gives a tree path P from s to t, and the correctness of CutEdges gives exactly the cut edges. We prove that an edge is unavoidable if and only if it is a cut edge on P.

($\Rightarrow$) We use contraposition. Suppose an edge is not a cut edge on P. There are two cases:

(i) If the edge lies outside P, the path P itself avoids it.

(ii) If the edge lies on P but is not a cut edge, its endpoints remain connected after deletion. Replacing this step of P by an alternate path gives an avoiding walk; removing repeated portions gives an avoiding path.

($\Leftarrow$) Deleting a cut edge on P separates its endpoints and hence s from t, so every s–t path uses it.

The algorithm returns exactly the cut edges on P, which are therefore the unavoidable edges. □

Theorem 45.2. UnavoidableEdges *takes* $O(n + m)$ *time.*

Proof. CutEdges and BFSTree each take $O(n + m)$ time. Marking edges costs $O(m)$, and the parent chain has at most $n - 1$ steps. Their sum is $O(n + m)$. □

Problem · p. 10

46. Directed Path Recovery

Use outgoing adjacency lists; parent pointers are initially undefined.

Algorithm DirectedPath(G, s, t)

```
1 initialize R[v] ← false and p[v] ← nil for all v
2 procedure Visit(u)
3     R[u] ← true
4     for each v ∈ Adj_G[u] with R[v] = false do
5         p[v] ← u; Visit(v)
6 Visit(s); if not R[t] then return no path
7 follow p from t to s, collecting vertices; reverse the list and return it
```

Theorem 46.1. DirectedPath *solves* Directed Path Recovery.

Proof. We prove that a vertex is visited exactly when it is reachable from s. We establish the two directions separately.

($\Rightarrow$) Every visited vertex is reachable. We use induction on discovery order. The first vertex s is reachable by the empty path. Each later vertex is discovered along an outgoing edge from an earlier vertex, whose path from s exists by the induction hypothesis. Appending that edge proves reachability.

($\Leftarrow$) Every reachable vertex is visited. Fix a directed path from s and use induction on position along it. Its first vertex s is visited. Each visited predecessor scans the edge to the next vertex, visiting it if necessary. Thus every vertex on the path is visited.

The equivalence justifies the no-path answer. Otherwise parent edges have the required direction and lead to earlier discoveries. The parent chain has no repeated vertex and ends at the only discovery without a parent, s. Reversing it gives the required directed path. □

Theorem 46.2. DirectedPath *takes* $O(n + m)$ *time.*

Proof. Initialization costs $O(n)$. Every visited vertex has one recursive call and one outgoing-list scan, giving at most n calls and m scanned edges. Parent assignments add constant work per discovery. Recovering and reversing the path costs $O(n)$, so the total is $O(n + m)$. □

Problem · p. 11

47. Avoiding Specified Vertices

Algorithm AvoidVertices(G, s, t, S)

```
1 mark the vertices of S
2 build H = G[V \ S], retaining a map between new and old labels
3 R ← DFS(H, s) using outgoing adjacency lists
4 return R[t]
```

Theorem 47.1. AvoidVertices *solves* Avoiding Specified Vertices.

Proof. We prove that a directed s–t path lies in H if and only if it lies in G and avoids S. Both endpoints remain by the input condition.

($\Rightarrow$) Every vertex of H lies outside S, and every edge of H is an edge of G.

($\Leftarrow$) A path in G avoiding S retains all its vertices and edges in the induced graph H.

The correctness of DFS makes $R[t]$ true exactly when a path in H exists, which is the required condition by the equivalence above. □

Theorem 47.2. AvoidVertices *takes* $O(n + m)$ *time.*

Proof. Marking takes $O(n + |S|) = O(n)$ time. A vertex scan and outgoing-edge scan build H and its label maps in $O(n + m)$. The resulting graph has at most n vertices and m edges, so DFS costs $O(n + m)$. □

Problem · p. 11

48. Shortest Directed Path

Algorithm DirectedShortest(G, s, t)

```
1 set d[v] ← +∞, p[v] ← nil for all v; d[s] ← 0
2 create a queue containing only s
3 while the queue is nonempty do
4     dequeue u
5     for each v ∈ Adj_G[u] with d[v] = +∞ do
6         d[v] ← d[u] + 1; p[v] ← u; enqueue v
7 if d[t] = +∞ then return no path
8 follow parents from t to s and return the reversed vertex list
```

Theorem 48.1. DirectedShortest *solves* Shortest Directed Path.

Proof. We prove that the final labels are shortest-path distances from s. First, we claim that $d[v] \le d[u] + 1$ for every edge $u \to v$ whose tail is reached. To establish this, we prove that queued labels are nondecreasing and differ by at most one. We use induction on queue operations.

Base case. The initial queue contains only s, with label zero.

Induction step. Dequeuing preserves the order. When a vertex of label h is processed, the remaining queued labels are h or $h + 1$, and each newly discovered vertex is appended with label $h + 1$.

Dequeue labels are therefore nondecreasing. Thus, when u is scanned, a discovered neighbour has label at most $d[u] + 1$, while a new neighbour receives equality. Labels never change, proving the subclaim.

We now use induction on position along any directed path from s. Its first vertex has label zero; the subclaim ensures that each next vertex is reached and has label at most its position. Hence every path to v has length at least $d[v]$, and an infinite label means no path exists. Conversely, parent edges decrease a finite label by one at each step, ending at s. They give a path of length $d[v]$, which is therefore shortest and is exactly the path recovered by the algorithm. □

Theorem 48.2. DirectedShortest *takes* $O(n + m)$ *time.*

Proof. The arrays take $O(n)$ initialization time. Every vertex enters and leaves the queue at most once, and its outgoing list is scanned once. The total search work is $O(n + m)$. Recovering at most n path vertices adds $O(n)$, preserving the bound. □

Problem · p. 11

49. Strong Connectivity with a Witness

Algorithm StrongWitness(G)

```
choose s ∈ V; A ← DFS(G, s)
if some v has A[v] = false then return (s, v)
construct G^R by reversing every edge; B ← DFS(G^R, s)
if some v has B[v] = false then return (v, s)
return strongly connected
```

Theorem 49.1. StrongWitness *solves* Strong Connectivity with a Witness.

Proof. We check each possible output using the correctness of DFS.

(i) If the first search fails, the returned pair (s, v) has no directed path from s to v.

(ii) If the second search fails, there is no s–v path in G^R. Reversing paths shows that there is no v–s path in G, so (v, s) is a witness.

(iii) If both searches succeed, every vertex is reachable from s and can reach s. For any u, v, concatenate a path from u to s with one from s to v and remove repeated portions. This gives a directed u–v path, proving strong connectivity.

□

Theorem 49.2. StrongWitness *takes* $O(n + m)$ *time.*

Proof. Each DFS takes $O(n + m)$. Reversal initializes n lists and inserts one reversed edge per original edge, taking $O(n + m)$. The two flag scans add $O(n)$. Their sum is $O(n + m)$. □

Problem · p. 11

50. Strongly Connected Component of a Vertex

Algorithm ComponentOf(G, s)

```
1 A ← DFS(G, s); construct the reversed graph G^R
2 B ← DFS(G^R, s)
3 return {v : A[v] and B[v]}
```

Theorem 50.1. ComponentOf *solves* Strongly Connected Component of a Vertex.

Proof. The correctness of DFS gives $A[v]$ exactly when s reaches v. Reversing a path gives a path in G^R in the opposite direction, so $B[v]$ holds exactly when v reaches s in G. Thus the two flags both hold if and only if v is mutually reachable with s. This is exactly the membership condition for the strongly connected component of s. □

Theorem 50.2. ComponentOf *takes* $O(n + m)$ *time.*

Proof. Reversal costs $O(n+m)$ by one insertion per edge. Both DFS calls cost $O(n+m)$, and the intersection scan costs $O(n)$. The total is $O(n + m)$. □

Problem · p. 11

51. A Sparse Strongly Connected Subgraph

A discovery edge is the edge that first causes DFS to visit a new vertex.

Algorithm SparseStrong(G)

```
1 choose r ∈ V; run DFS(G, r) and record its discovery edges A
2 construct G^R; run DFS(G^R, r) and record discovery edges B
3 return the union of A and the reversals of the edges in B
```

Theorem 51.1. SparseStrong *solves* A Sparse Strongly Connected Subgraph.

Proof. We prove that the discovery edges of a search from r that reaches every vertex give paths from r to every vertex using exactly $n-1$ edges. We use induction on discovery order to establish the path property.

Base case. The root r has its empty path.

Induction step. Each new vertex extends its parent's path by its discovery edge. The parent was discovered earlier, so its path exists by the induction hypothesis. There is one discovery edge per vertex other than r, giving exactly $n-1$ edges.

Strong connectivity makes the first search reach every vertex. It also makes G^R strongly connected, since paths reverse, so the claim applies to the second search. Reversing its discovery edges gives paths from every vertex to r in G.

The returned union lets every u reach every v through r; removing repetitions gives a path. All edges belong to G, and their union has at most $2n-2$ edges, as required. □

Theorem 51.2. SparseStrong *takes* $O(n+m)$ *time.*

Proof. Recording an edge adds constant work per discovery. Each DFS still costs $O(n+m)$; reversal has the same bound. Mark original edge identifiers while collecting the two lists, so their union is formed in $O(n+m)$ total time. □

Problem · p. 11

52. Directed Cycle Witness

White means unvisited, grey means active, and black means finished. A cycle report halts the entire algorithm.

Algorithm FindDirectedCycle(G)

```
1 set all colours to white and parents to nil
2 procedure Visit(u)
3     colour u grey
4     for each v ∈ Adj_G[u] do
5         if v is grey then report the parent path from v to u, followed by u → v
6         if v is white then p[v] ← u; Visit(v)
7     colour u black
8 for each white vertex r do Visit(r)
9 return acyclic
```

Theorem 52.1. FindDirectedCycle *solves* Directed Cycle Witness.

Proof. We prove that a reported cycle is valid and that an acyclic report is valid.

(i) If a cycle is reported, the grey vertices are exactly the active recursive stack, so a grey neighbour v of u is an ancestor on the parent path. The directed discovery path from v to u has distinct vertices; adding $u \to v$ gives the reported directed cycle.

(ii) If acyclicity is reported, we claim that every edge $u \to v$ has v finish before u. If v is white when examined, its recursive call finishes first. If it is black, it has already finished. It cannot be grey without causing a cycle report. Thus decreasing finish order is topological. For a contradiction, suppose a directed cycle exists. Its positions in this order would increase strictly and return to their starting position. Hence the graph is acyclic.

□

Theorem 52.2. FindDirectedCycle *takes* $O(n + m)$ *time.*

Proof. There are n visits and at most m scanned outgoing edges, with constant work per visit or edge. One-time initialization and the outer scan cost $O(n)$. A cycle is recovered at most once, using at most n parent steps. Total time is $O(n + m)$. □

Problem · p. 12

53. Acyclicity after One Edge Insertion

Algorithm SafeInsertion(G, u, v)
1 $R \leftarrow \mathrm{DFS}(G, v)$
2 **return** not $R[u]$

Theorem 53.1. SafeInsertion *solves* Acyclicity after One Edge Insertion.

Proof. We prove that inserting $u \to v$ creates a cycle exactly when v can already reach u. We prove both directions.

($\Rightarrow$) Since G is acyclic, any new cycle uses the inserted edge. Removing that edge leaves a directed v–u path in G.

($\Leftarrow$) A directed v–u path followed by the new edge forms a cycle.

The correctness of DFS makes $R[u]$ test this path's existence. Its negation therefore says exactly whether the insertion preserves acyclicity. □

Theorem 53.2. SafeInsertion *takes* $O(n + m)$ *time.*

Proof. The single DFS call takes $O(n + m)$ time. Negating and returning one flag takes constant time, so the total remains $O(n + m)$. □

Problem · p. 12

54. Longest Directed Path in a DAG

Algorithm LongestDAGPath(G, s, t)

```
1 run DFSTimes(G); bucket vertices in decreasing finish order
2 set d[v] ← −∞, p[v] ← nil for all v; d[s] ← 0
3 for each u in that order with d[u] > −∞ do
4     for each v ∈ Adj_G[u] do
5         if d[u] + 1 > d[v] then d[v] ← d[u] + 1; p[v] ← u
6 if d[t] = −∞ then return no path
7 follow parents from t to s and return the reversed vertex list
```

Theorem 54.1. LongestDAGPath *solves* Longest Directed Path in a DAG.

Proof. We prove that, when v is processed, $d[v]$ is the longest s–v path length, or $-\infty$ if no such path exists. Decreasing DFS finish order is topological. We use induction on position in this order.

Base case. The first vertex has no incoming edge. Its only possible path from s is the empty path if it equals s, as initialized.

Induction step. Every nonempty path to v ends at a predecessor u, processed earlier. By the induction hypothesis, $d[u]$ is its best prefix length. Conversely, appending $u \to v$ to an attaining prefix gives a path, because a repetition would create a cycle. The updates therefore maximize over exactly all possible final edges. If no reachable predecessor exists, the initial label is correct. No reachable predecessor leads back to s, so its label remains zero.

Parents store attaining predecessors, and following them back gives a path of the computed length. A label of $-\infty$ correctly reports failure. □

Theorem 54.2. LongestDAGPath *takes* $O(n + m)$ *time.*

Proof. DFSTimes takes $O(n + m)$. Its finish times lie in $[1, 2n]$, so bucketing costs $O(n)$. Each outgoing edge is considered at most once. Initialization and recovery add $O(n)$, giving $O(n + m)$ total time. □

Problem · p. 12

55. Condensation Graph

Algorithm CONDENSATION(G)

```
(c, k) ← StrongComponents(G); group original vertices by label
create empty adjacency lists D[1..k]; initialize seen[1..k] ← 0
for i = 1, ..., k do
    for each u with c[u] = i and each edge u → v do
        j ← c[v]
        if j ≠ i and seen[j] ≠ i then
            append j to D[i]; seen[j] ← i
return (c, D)
```

Theorem 55.1. CONDENSATION *solves* Condensation Graph.

Proof. We prove that, while scanning source component i, $\text{seen}[j] = i$ exactly when the edge $i \to j$ has already been emitted. The correctness of StrongComponents gives the component labels. We use induction on the number of edges scanned for component i.

Base case. No edge from i has been emitted, and every existing mark is zero or an earlier source label, so no mark equals i.

Induction step. Consider the next original edge, whose head has component label j. There are two cases:

(i) If $j \neq i$ and $\text{seen}[j] \neq i$, the edge $i \to j$ has not been emitted by the induction hypothesis. Emitting and marking it together preserves the claim.

(ii) Otherwise the edge is internal or its component edge was already emitted. Skipping it preserves the claim and avoids a duplicate.

Every original edge is examined, so the output is precisely the component graph. For a contradiction, suppose this graph has a cycle. Its components would be mutually reachable, contradicting their being distinct strongly connected components. Thus the output is acyclic. □

Theorem 55.2. CONDENSATION *takes* $O(n + m)$ *time.*

Proof. StrongComponents takes $O(n + m)$. Grouping vertices and initializing lists and marks cost $O(n)$. The nested scans visit each vertex and original edge once, doing constant work per edge. At most m edges are emitted, so the total is $O(n + m)$. Marks are not reset separately for each component. □

Problem · p. 12

56. Universal Source Vertices

Algorithm UniversalSources(G)

```
1 (c, k) ← StrongComponents(G); set in[1..k] ← false
2 for each edge u → v with c[u] ≠ c[v] do in[c[v]] ← true
3 if more than one component has no incoming edge then return ∅
4 let j be the unique component with no incoming edge
5 return {v : c[v] = j}
```

Theorem 56.1. UniversalSources *solves* Universal Source Vertices.

Proof. The correctness of StrongComponents gives a nonempty component DAG. We first prove that every vertex of a finite DAG is reachable from a source. Following incoming edges backwards must reach a source; otherwise a repeated vertex would give a cycle. Reversing this chain proves the claim.

(i) If there is one source component, by the claim it reaches every component. Strong connectivity within components then lets each of its vertices reach every original vertex. No vertex outside it can reach it, since that would require an incoming component edge.

(ii) If there are several source components, no component can reach two different sources: reaching a source from outside would require an incoming edge. Hence no original vertex is a universal source.

These cases give exactly the algorithm's output. □

Theorem 56.2. UniversalSources *takes* $O(n + m)$ *time.*

Proof. The component call costs $O(n + m)$. Initializing incoming flags, scanning every edge, counting sources, and listing the answer cost $O(n + m)$. Parallel component edges need no deduplication because only flags are set. □

Problem · p. 12

57. Minimum Starting Set

Algorithm MinimumStarts(G)

```
1 (c, k) ← StrongComponents(G); choose a representative r_j per label
2 initialize in[1..k] ← false
3 for each edge u → v with c[u] ≠ c[v] do in[c[v]] ← true
4 return {r_j : in[j] = false}
```

Theorem 57.1. MinimumStarts *solves* Minimum Starting Set.

Proof. The correctness of StrongComponents gives the component DAG. We prove that one representative from each source component is both necessary and sufficient.

(i) No vertex outside a source component can reach it, because that would give an incoming component edge. Every feasible starting set must therefore contain a vertex in each source component.

(ii) Every component is reachable from a source: follow incoming edges backwards until reaching one; acyclicity prevents repetition. A chosen representative reaches its whole component and every vertex in each downstream component. Thus the returned set reaches all vertices.

Its size meets the lower bound from necessity, so it is minimum. □

Theorem 57.2. MinimumStarts *takes* $O(n + m)$ *time.*

Proof. StrongComponents costs $O(n+m)$. Representatives and incoming flags require $O(n)$ initialization and vertex work; the edge scan costs $O(m)$. Writing at most one vertex per component costs $O(n)$, for $O(n + m)$ total time. □

Problem · p. 12

Chapter 13

Greedy Algorithms

58. Scheduling with Turnaround Time

Algorithm TURNAROUND($I[1..n], \Delta$)

```
1 form J_i = [s_i, f_i + Δ], retaining the original index i
2 A ← SCHEDULEINTERVALS(J)
3 return the original interval indices in A
```

Theorem 58.1. TURNAROUND *solves* Scheduling with TURNAROUND Time.

Proof. We prove that a subcollection satisfies the turnaround constraints if and only if its extended closed intervals are disjoint. For two intervals in temporal order, disjointness is exactly $f_i + \Delta < s_j$, the required inequality. Applying this test to every pair proves both directions of the claim, and the transformation preserves cardinality.

The correctness of SCHEDULEINTERVALS gives a largest disjoint subcollection of extended intervals. Mapping its indices back therefore gives a feasible original schedule of maximum size. □

Theorem 58.2. TURNAROUND *takes* $O(n \log n)$ *time.*

Proof. Constructing the extended records takes $O(n)$ time. SCHEDULEINTERVALS costs $O(n \log n)$, and returning the original indices takes $O(n)$. The total is $O(n \log n)$. □

Problem · p. 13

59. Scheduling with a Required Interval

Algorithm RequiredSchedule($I[1..n], r$)
1 $L \leftarrow \{I_i : f_i < s_r\}$; $R \leftarrow \{I_i : s_i > f_r\}$, retaining indices
2 $A \leftarrow$ ScheduleIntervals(L); $B \leftarrow$ ScheduleIntervals(R)
3 **return** A, followed by $[r]$, followed by B

Theorem 59.1. RequiredSchedule *solves* Scheduling with a Required Interval.

Proof. We prove that a schedule containing I_r is feasible if and only if it consists of a disjoint subcollection of L, the interval I_r, and a disjoint subcollection of R.

($\Rightarrow$) Every other interval in a feasible schedule is compatible with I_r, so it lies in exactly one of L, R. Each side is disjoint.

($\Leftarrow$) All intervals in L are disjoint from I_r and all intervals in R, so any such two subcollections can be combined.

The sizes add. The correctness of ScheduleIntervals gives a largest subcollection on each side, including an empty one when necessary. The claim therefore makes their union with I_r an optimum required schedule. □

Theorem 59.2. RequiredSchedule *takes* $O(n \log n)$ *time.*

Proof. Partitioning the records costs $O(n)$. The two calls cost at most $O(|L| \log n + |R| \log n) = O(n \log n)$; empty lists require constant time. Joining the output lists costs $O(n)$, preserving the bound. □

Problem · p. 13

60. Stabbing Intervals

Algorithm StabIntervals($I[1..n]$)

```
1 sort intervals by nondecreasing f_i using MergeSort
2 P ← an empty list; t ← −∞
3 for each interval [s_i, f_i] in sorted order do
4     if s_i > t then append f_i to P; t ← f_i
5 return P
```

Theorem 60.1. StabIntervals *solves* Stabbing Intervals.

Proof. We prove that choosing the earliest finishing endpoint and then solving the remaining unhit intervals gives a minimum hitting set. We use induction on the number of remaining intervals.

Base case. The empty instance needs no points.

Induction step. Let f be the earliest finishing endpoint. We claim that some optimum contains f, by an exchange argument. An optimum contains a point p in that interval, so $p \leq f$. Every interval containing p starts at most p and finishes at least f. Replacing p by f loses no coverage, proving the subclaim.

Remove the intervals hit by f. The other points of an optimum containing f must hit the remaining intervals. Conversely, any hitting set for the remaining intervals extends by f to hit everything. The induction hypothesis supplies a minimum such completion, proving optimality.

The scan implements these choices. Its last selected point t is at most the current finish. There are two cases:

(i) If $s_i \leq t$, the current interval is already hit.

(ii) Otherwise every earlier chosen point is too small, and f_i is the earliest finish among unhit intervals, so it is the next greedy choice.

Thus the scan returns the minimum hitting set just proved correct. □

Theorem 60.2. StabIntervals *takes* $O(n \log n)$ *time.*

Proof. Sorting costs $O(n \log n)$. The scan does constant work per interval and appends at most n points, taking $O(n)$ additional time. The sum is $O(n \log n)$. □

Problem · p. 13

61. Coloring Half-Open Intervals

Algorithm HalfOpenColours($I[1..n]$)
1 form the closed intervals $J_i = [2s_i, 2f_i - 1]$
2 $(c, k) \leftarrow$ ColorIntervals(J)
3 **return** (c, k) for the original interval indices

Theorem 61.1. HalfOpenColours *solves* Coloring Half-Open Intervals.

Proof. We prove that the transformation preserves exactly the overlapping pairs. By a direct calculation using integer endpoints,

$$\max(s_i, s_j) < \min(f_i, f_j) \quad \Longleftrightarrow \quad 2\max(s_i, s_j) \leq 2\min(f_i, f_j) - 1.$$

The left condition is overlap of the original half-open intervals; the right is overlap of J_i, J_j. Also each J_i is a nonempty closed interval because $s_i < f_i$.

Hence the transformation preserves all valid colourings and their colour counts. The correctness of ColorIntervals gives an optimum colouring of the transformed instance and therefore of the original one. □

Theorem 61.2. HalfOpenColours *takes* $O(n \log n)$ *time.*

Proof. Transforming endpoints costs $O(n)$ time. ColorIntervals costs $O(n \log n)$. The output indices are unchanged, so returning its colouring does not increase the bound. □

Problem · p. 13

62. Maximum Simultaneous Intervals

Algorithm MaximumOverlap($I[1..n]$)

```
create a start event (s_i, +1) and finish event (f_i, −1) for each interval
sort events by time, with all starts before finishes at a tie
a ← 0; b ← 0
for each event (x, ε) do
    a ← a + ε; if a > b then b ← a; t ← x
return t and {i : s_i ≤ t ≤ f_i}
```

Theorem 62.1. MaximumOverlap *solves* Maximum Simultaneous Intervals.

Proof. We prove that a equals the number of processed start events minus the number of processed finish events. We use induction on the number of processed events.

Base case. Initially no events have been processed and $a = 0$.

Induction step. A start event increases both the claimed difference and a by one; a finish event decreases both by one. Thus each update preserves the claim.

At time x, after its starts and before its finishes, a therefore counts exactly the intervals containing x. Intermediate counts at that time are no larger, because starts increase the count and finishes decrease it.

We claim that a maximum overlap occurs at an endpoint. For any nonempty collection sharing a time, its largest starting endpoint also belongs to every interval in the collection. Thus the sweep examines a time attaining the maximum overlap. The running maximum stores such a time t; at least one interval exists, so t is assigned. The final scan lists its intervals. □

Theorem 62.2. MaximumOverlap *takes* $O(n \log n)$ *time.*

Proof. There are $2n$ events. Constructing and sorting them costs $O(n \log n)$. Each event uses constant work, and the final interval scan and output take $O(n)$. The total is $O(n \log n)$. □

Problem · p. 14

63. Interval Coloring with Demands

Algorithm DemandColours($I[1..n], d[1..n]$)

```
1 form a list J containing d_i copies of I_i, each tagged with i
2 (c, k) ← ColorIntervals(J)
3 collect the colours of copies tagged i into C_i
4 return (C_1, ..., C_n, k)
```

Theorem 63.1. DemandColours *solves* Interval Coloring with Demands.

Proof. We prove that the original instance has a feasible solution using at most k colours if and only if its interval copies have a proper colouring using at most k colours.

($\Rightarrow$) Assign the d_i distinct colours of each feasible set to its d_i copies. The demand constraints ensure that every pair of overlapping copies gets different colours.

($\Leftarrow$) In a proper colouring of the copies, the d_i copies of one interval overlap, so they get distinct colours. If two original intervals overlap, every copy of one overlaps every copy of the other. Their collected colour sets are therefore disjoint, as required.

The optimum colour counts are equal. The correctness of ColorIntervals therefore makes the collected output optimal. □

Theorem 63.2. DemandColours *takes* $O(N \log N)$ *time.*

Proof. The expanded list has N records and is formed in $O(N)$ time; positive demands imply $n \leq N$. Its colouring takes $O(N \log N)$ time. Collecting and returning the N colour assignments adds $O(N)$. □

Problem · p. 14

64. Weighted Total Completion Time

Compare p_i/w_i and p_j/w_j by the cross products $p_i w_j$ and $p_j w_i$.

Algorithm WeightedCompletion$(p[1..n], w[1..n])$

```
1 create the index list π = [1, ..., n]
2 sort π by nondecreasing p_i/w_i using MergeSort
3 return π
```

Theorem 64.1. WeightedCompletion *solves* Weighted Total Completion Time.

Proof. We prove that any optimum order can be transformed into a ratio-sorted optimum. We use induction on the number of ratio inversions in that order, with an exchange argument in the inductive step.

Base case. An optimum with no inversions already has the required order.

Induction step. Choose adjacent inverted jobs i, j. We claim that swapping them cannot increase the objective. If they begin after time T, the difference between their contribution before and after the swap is

$$[w_i(T + p_i) + w_j(T + p_i + p_j)] - [w_j(T + p_j) + w_i(T + p_j + p_i)] = w_j p_i - w_i p_j \geq 0.$$

Other completion times are unchanged. Thus the swapped order is also optimal and has one fewer inversion, so the induction hypothesis gives a ratio-sorted optimum.

Equal-ratio swaps leave the displayed difference zero. Any two sorted orders differ only within equal-ratio groups, so every sorted order, including the returned one, is optimal. □

Theorem 64.2. WeightedCompletion *takes* $O(n \log n)$ *time.*

Proof. The index list costs $O(n)$ to construct. Each ratio comparison uses two multiplications and a comparison on a constant number of words, hence constant time. MergeSort therefore takes $O(n \log n)$ time, which bounds the whole algorithm. □

Problem · p. 14

65. Minimum Total Waiting Time

Algorithm MinimumWaiting($p[1..n]$)
1 **return** ShortestFirst(p)

Theorem 65.1. MinimumWaiting *solves* Minimum Total Waiting Time.

Proof. We prove that precisely the same orders minimize total waiting time and total completion time. By a direct calculation, $C_i = W_i + p_i$ for every job, so

$$\sum_i W_i = \sum_i C_i - \sum_i p_i.$$

The subtracted quantity depends only on the input, proving the claim. The correctness of ShortestFirst gives a minimum-total-completion order, which is also a minimum-total-waiting order. □

Theorem 65.2. MinimumWaiting *takes* $O(n \log n)$ *time.*

Proof. The single call to ShortestFirst takes $O(n \log n)$ time. Returning its order adds no larger cost, proving the bound. □

Problem · p. 14

66. Maximum Lateness

Algorithm EarliestDeadline($p[1..n], d[1..n]$)

```
1 create π = [1, ..., n]
2 sort π by nondecreasing d_i using MergeSort
3 return π
```

Theorem 66.1. EarliestDeadline *solves* Maximum Lateness.

Proof. We prove that any optimum order can be transformed into a deadline-sorted optimum. We use induction on the number of deadline inversions in that order, with an exchange argument in the inductive step.

Base case. With no inversions, the optimum already has the required order.

Induction step. Choose adjacent jobs i, j with $d_i > d_j$. We claim that swapping them cannot increase maximum lateness. If they begin after time T, job j originally has lateness $L_j = T + p_i + p_j - d_j$. After the swap, its lateness is $T + p_j - d_j \leq L_j$, and job i has lateness $T + p_j + p_i - d_i \leq L_j$. Other completion times are unchanged. This proves the subclaim. The new order is optimal with one fewer inversion, so the induction hypothesis gives a deadline-sorted optimum.

Equal-deadline swaps preserve the objective because the later completion determines the pair's maximum lateness. Hence every sorted order, including the returned one, is optimal. □

Theorem 66.2. EarliestDeadline *takes* $O(n \log n)$ *time.*

Proof. Constructing the index array takes $O(n)$. Deadline comparisons take constant time, so MergeSort takes $O(n \log n)$. Returning the order adds no larger cost. □

Problem · p. 14

67. Nearest Facility

Algorithm NearestFacility(G, w, S)

```
add a new vertex z and edges z → s of weight zero for all s ∈ S
(d, p) ← Dijkstra(H, w, z) on this augmented graph H
build child lists of the finite-distance parent tree
traverse that tree from the root; set a[v] ← v if p[v] = z
    for each other reached v ≠ z, set a[v] ← a[p[v]]
return (d[v], a[v]) on reached original vertices, and (+∞, nil) otherwise
```

Theorem 67.1. NearestFacility *solves* Nearest Facility.

Proof. We prove that the distance from z to v in H equals the minimum distance from any facility to v in G. Every path from z first uses a zero-weight edge to a facility and then stays in G. Conversely, prepend that edge to any path from a facility. Both directions preserve weight, proving the claim, including the equivalence of reachability.

All weights are nonnegative, so the correctness of Dijkstra gives these distances and attaining parent paths. We also prove that $a[v]$ is the first facility on its parent path. We use induction on parent-tree depth.

Base case. A child of z is a facility and receives itself.

Induction step. A deeper vertex inherits its parent's facility, which is the first facility on the path by the induction hypothesis. This facility attains the reported distance. The unreachable outputs are correct by the claim. □

Theorem 67.2. NearestFacility *takes* $O((n+m)\log n)$ *time.*

Proof. The augmented graph has $n+1$ vertices and at most $m+n$ edges. Construction costs $O(n+m)$, and Dijkstra costs $O((n+m)\log n)$. Building child lists, propagating facilities, and writing the output add $O(n)$. □

Problem · p. 15

68. One Free Edge

Algorithm FreeEdgeDistance(G, w, s, t)

```
create vertices (v,0), (v,1) for every v ∈ V
for each edge u → v of weight w_uv do
    add (u,0) → (v,0) and (u,1) → (v,1) with weight w_uv
    add (u,0) → (v,1) with weight 0
run Dijkstra from (s,0) in the resulting graph H
return min(d[(t,0)], d[(t,1)])
```

Theorem 68.1. FreeEdgeDistance *solves* One Free Edge.

Proof. We prove that the original and layered instances have the same optimum cost. We prove the two inequalities by transformations.

(i) A path with zero or one free edge lifts to a path from $(s, 0)$ to a copy of t, using a layer change exactly at the free edge. Its weight is the original discounted cost.

(ii) A layered path projects to an original walk with at most one free edge, since layers can change only from zero to one. Remove any closed portions. Their charged costs are nonnegative, so this cannot increase cost. If the free edge is removed, the remainder simply uses no discount. The resulting path is feasible and no more expensive.

The transformations prove the claim and the equivalence of reachability. All layered weights are nonnegative, so the correctness of Dijkstra gives exactly the returned optimum. □

Theorem 68.2. FreeEdgeDistance *takes* $O((n + m) \log n)$ *time.*

Proof. The layered graph has $2n$ vertices and $3m$ edges, and is built in $O(n + m)$ time. Dijkstra takes $O((n+m) \log n)$. Taking the smaller of two labels costs constant time. □

Problem · p. 15

69. Vertex Tolls

Algorithm TollDistance(G, w, c, s, t)

```
1 set w'(u, v) ← w(u, v) + c(v) for every edge u → v
2 (d, p) ← Dijkstra(G, w', s)
3 return c(s) + d[t], with a finite number plus +∞ interpreted as +∞
```

Theorem 69.1. TollDistance *solves* Vertex Tolls.

Proof. We prove that each path's original cost is $c(s)$ plus its transformed weight. By a direct calculation for $s = v_0, v_1, \ldots, v_k = t$,

$$c(s) + \sum_{i=1}^{k} w'(v_{i-1}, v_i) = \sum_{i=0}^{k} c(v_i) + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The right side includes exactly the edge weights and all vertex tolls. The additive $c(s)$ is independent of the path. Since all transformed weights are nonnegative, the correctness of Dijkstra minimizes the remaining term. Reachability is unchanged, and when $s = t$ the empty path has cost $c(s)$. Thus the returned value is correct in every case. □

Theorem 69.2. TollDistance *takes* $O((n + m) \log n)$ *time.*

Proof. Creating the transformed weights takes $O(n + m)$ time. The vertices and edges are unchanged, so Dijkstra takes $O((n + m) \log n)$ time. The final addition is constant-time, giving the stated total. □

Problem · p. 15

70. Minimum Spanning Forest

Algorithm MinimumForest(G, w)

```
run Components(G); build locally labelled component graphs H_1, ..., H_k
for each H_j do choose any r_j and compute T_j ← Prim(H_j, w, r_j)
return the union of these trees with original vertex labels
```

Theorem 70.1. MinimumForest *solves* Minimum Spanning Forest.

Proof. The correctness of Components gives the exact components, and the correctness of Prim gives a minimum spanning tree T_j of each. We prove feasibility and optimality separately.

Feasibility. The union spans every component and has no cycles, since each piece is a tree and different pieces have no connecting edges.

Optimality. Every feasible forest consists of one spanning tree per component. Its weight is the sum of these tree weights, each at least $w(T_j)$. The returned union attains the sum of these lower bounds and is therefore minimum. □

Theorem 70.2. MinimumForest *takes* $O((n+m)\log n)$ *time.*

Proof. Components and construction cost $O(n+m)$. For n_j vertices and m_j edges in a component, Prim takes $O((n_j+m_j)\log n_j)$. Since $\log n_j \le \log n$ and the sizes sum to n, m, the calls together cost $O((n+m)\log n)$. Joining their edge lists adds $O(n)$. □

Problem · p. 15

71. MST after One Edge Insertion

Algorithm InsertMSTEdge$(T, w, e = uv)$

```
1 run DFS in T from u, recording parents; recover its u–v path P
2 let f be a maximum-weight edge of P
3 if w(e) < w(f) then return T − {f} + {e}
4 else return T
```

Theorem 71.1. InsertMSTEdge *solves* MST after One Edge Insertion.

Proof. Adding e to T creates exactly the cycle $P + e$, so removing f leaves a spanning tree. Put $W = w(T)$. We claim that every spanning tree S of $G + e$ has weight at least $\min(W, W + w(e) - w(f))$. We prove this by cases.

(i) If $e \notin S$, S is a spanning tree of G, so $w(S) \geq W$.

(ii) If $e \in S$, $S - e$ has two components separating u from v. The path P has an edge g crossing this cut, with $w(g) \leq w(f)$. The tree $S - e + g$ belongs to G, so

$$W \leq w(S) - w(e) + w(g) \leq w(S) - w(e) + w(f).$$

Thus $w(S) \geq W + w(e) - w(f)$, proving the claim.

The algorithm returns a feasible tree of weight exactly the claimed lower bound, and therefore an optimum. □

Theorem 71.2. InsertMSTEdge *takes* $O(n)$ *time.*

Proof. The tree has $n - 1$ edges. Its DFS and path recovery take $O(n)$ time, as does finding the path maximum. Copying the tree with at most one deletion and insertion costs $O(n)$. No scan of the other edges of G is needed. □

Problem · p. 15

72. Minimum Bottleneck Spanning Tree

Algorithm BottleneckTree(G, w)
1 choose any root r
2 **return** Prim(G, w, r)

Theorem 72.1. BottleneckTree *solves* Minimum Bottleneck Spanning Tree.

Proof. The correctness of Prim gives a minimum-total-weight spanning tree T. We prove that its maximum edge weight b is no larger than that of any spanning tree. We use contradiction. Suppose a spanning tree S has every edge lighter than b.

Remove an edge e of weight b from T. The two resulting components form a cut that some edge f of S must cross. Then $T - e + f$ is a spanning tree of smaller total weight, since $w(f) < w(e)$. This contradicts the optimality of T and proves the claim. Thus T is also a minimum bottleneck spanning tree. □

Theorem 72.2. BottleneckTree *takes* $O((n + m) \log n)$ *time.*

Proof. Choosing a root is constant time. Prim costs $O((n + m) \log n)$; returning its tree adds no larger cost. □

Problem · p. 16

73. Maximum Spanning Tree

Algorithm MaximumTree(G, w)
1 set $w'(e) \leftarrow -w(e)$ for every edge e
2 **return** Kruskal(G, w')

Theorem 73.1. MaximumTree *solves* Maximum Spanning Tree.

Proof. We prove that minimizing transformed tree weight is equivalent to maximizing original tree weight. The feasible trees are unchanged, and for every tree T, $w'(T) = -w(T)$. Negation reverses the order of their weights, proving the claim directly.

Kruskal returns a minimum spanning tree for the transformed integer weights, hence a maximum spanning tree for the original weights. □

Theorem 73.2. MaximumTree *takes* $O((n + m) \log n)$ *time.*

Proof. Negating the edge weights takes $O(m)$ time. Kruskal takes $O((n + m) \log n)$. Their sum gives the stated bound. □

Problem · p. 16

74. MST with Required Edges

Use a disjoint-set data structure.

Algorithm REQUIREDMST(G, w, F)

```
1 MakeSets(n); T ← ∅
2 for each uv ∈ F do
3     if Find(u) = Find(v) then return impossible
4     append uv to T; Union(u, v)
5 sort E \ F by nondecreasing weight
6 for each uv in this order do
7     if Find(u) ≠ Find(v) then append uv to T; Union(u, v)
8 return T
```

Theorem 74.1. REQUIREDMST *solves* MST with Required Edges.

Proof. The disjoint sets track the components of T. A required edge within one component creates a cycle, so no feasible tree can contain it. Otherwise F is a forest and extends to a spanning tree because G is connected. We prove that T is contained in an optimum tree satisfying F. We use induction on the number of accepted nonrequired edges.

Base case. Initially $T = F$, so every feasible optimum contains it.

Induction step. Let the next accepted edge e join two components of T, and let C be one. No earlier scanned edge crosses the cut of C, since its endpoints would already be joined. Thus e is a cheapest crossing edge. Let S be an optimum extension supplied by the induction hypothesis. There are two cases:

(i) If $e \in S$, the same optimum extends $T + e$.

(ii) Otherwise adding e to S creates a cycle. Remove another edge f of that cycle crossing the cut. No edge of T crosses the cut, so $f \notin T$ and in particular $f \notin F$. Since $w(e) \le w(f)$, the new tree is feasible and no heavier than the optimum. It is therefore an optimum extension of $T + e$.

The final T is connected: otherwise an edge between two final components would have been accepted. It is a tree and, by the claim, an optimum. □

Theorem 74.2. REQUIREDMST *takes* $O((n + m) \log n)$ *time.*

Proof. There are $O(m)$ finds and unions, each taking $O(\log n)$ time with a disjoint-set data structure. Initialization takes $O(n)$. Sorting takes $O(m \log m) = O(m \log n)$ for a simple graph, since $m < n^2$. Edge marking and listing cost $O(n+m)$, giving $O((n+m) \log n)$. □

Problem · p. 16

75. Threshold Connectivity Queries

Algorithm ThresholdQueries$(G, w, Q[1..q])$

```
sort edges by nondecreasing weight and queries by nondecreasing b_i
keep original query indices; MakeSets(n); j ← 1
for each query (s_i, t_i, b_i) in sorted order do
    while j ≤ m and w(e_j) ≤ b_i do
        let e_j = uv; Union(u, v); j ← j + 1
    A[i] ← (Find(s_i) = Find(t_i))
return A in original query order
```

Theorem 75.1. ThresholdQueries *solves* Threshold Connectivity Queries.

Proof. We prove that after inserting any prefix of the sorted edge list, the disjoint sets are exactly the connected components of that prefix graph. We use induction on the number of inserted edges.

Base case. With no edges, the graph has only singleton components, as initialized.

Induction step. Inserting uv joins exactly the components containing its endpoints, if they differ, and leaves all other components unchanged. This is precisely the update performed by Union.

Before answering threshold b_i, the scan has inserted all and only edges of weight at most b_i. The sorted orders and non-strict comparison ensure this even at ties. The claim makes the Find comparison true exactly when s_i, t_i are connected using eligible edges. Restoring the original query indices gives the answers in the requested order. □

Theorem 75.2. ThresholdQueries *takes* $O((n + m + q) \log(n + q))$ *time.*

Proof. Sorting costs $O(m \log m + q \log q)$. Since G is simple, $m < n^2$, so this is $O((m + q) \log(n + q))$. There are at most m unions and $2q$ finds, each taking $O(\log n)$ time with a disjoint-set data structure. Initialization and output cost $O(n + q)$. The total is $O((n + m + q) \log(n + q))$. □

Problem · p. 16

Chapter 14

Dynamic Programming

76. Scheduling at Most k Intervals

Sort by start time, retaining original indices. Let q_i be the first index with $s_{q_i} > f_i$, or $n+1$.

Algorithm CappedSchedule$(I[1..n], w[1..n], k)$

```
sort the records; compute every q_i using FirstLarger
set D[n+1, b] ← 0 and D[i, 0] ← 0 for all boundary entries
for i = n, ..., 1 and b = 1, ..., k do
    D[i, b] ← max{D[i+1, b], w_i + D[q_i, b−1]}; record a best case
start at (1, k) and follow recorded cases until i = n+1 or b = 0
when taking i, output its original index and move to (q_i, b−1)
when skipping i, move to (i+1, b); return the output list
```

Theorem 76.1. CappedSchedule *solves* Scheduling at Most k Intervals.

Proof. We prove that $D[i, b]$ is the maximum weight obtainable from suffix i using at most b intervals. We use induction on suffix length, for all b.

Base cases. An empty suffix or zero allowance has optimum zero.

Induction step. Classify feasible schedules by whether they contain i:

(i) If i is omitted, the best weight is $D[i+1, b]$ by the induction hypothesis.

(ii) If i is included, every later compatible interval lies in suffix q_i. The remaining allowance is $b-1$, giving best weight $w_i + D[q_i, b-1]$. Conversely, any feasible solution of that suffix extends by i.

These cases exhaust the possibilities, so their maximum proves the claim. All dependencies have shorter suffixes and are computed first. Recovery follows attaining cases, preserving weight and compatibility while using at most the remaining allowance. At the boundary its remaining optimum is zero, so it returns a feasible collection of optimum weight $D[1, k]$. □

Theorem 76.2. CappedSchedule *takes* $O(n \log n + nk)$ *time.*

Proof. MergeSort and the n FirstLarger calls cost $O(n \log n)$. There are nk interior entries, each using constant time. Boundaries cost $O(n + k)$, and recovery takes $O(n)$. Summing gives $O(n \log n + nk)$. □

Problem · p. 17

77. Weighted Scheduling with a Required Interval

Each subarray retains weights and original indices; an empty subarray returns an empty collection.

Algorithm RequiredWeighted($I[1..n], w[1..n], r$)

```
1 L ← {I_i : f_i < s_r}; R ← {I_i : s_i > f_r}
2 A ← WeightedIntervals(L, w|_L); B ← WeightedIntervals(R, w|_R)
3 return A ∪ {r} ∪ B using original indices
```

Theorem 77.1. RequiredWeighted *solves* Weighted Scheduling with a Required Interval.

Proof. We prove that a schedule containing r is feasible if and only if it consists of r and disjoint subcollections from L and R.

($\Rightarrow$) Every other chosen interval is compatible with I_r, so it lies entirely to its left or right. Each side is disjoint.

($\Leftarrow$) Any such subcollections are mutually compatible and compatible with I_r.

The total weight is the sum of the two contributions and w_r. The correctness of WeightedIntervals maximizes each independent contribution, including an empty side when necessary. Their union with r is therefore feasible and optimal. □

Theorem 77.2. RequiredWeighted *takes* $O(n \log n)$ *time.*

Proof. Partitioning and recovery cost $O(n)$. The calls cost $O(|L| \log |L| + |R| \log |R|) \subseteq O(n \log n)$. The same bound covers the remaining work. □

Problem · p. 17

78. Weighted Scheduling with Turnaround Time

Algorithm WeightedTurnaround($I[1..n], w[1..n], \Delta$)
1 form $J_i = [s_i, f_i + \Delta]$ with weight w_i and original index i
2 $A \leftarrow$ WeightedIntervals(J, w)
3 **return** the original indices in A

Theorem 78.1. WeightedTurnaround *solves* Weighted Scheduling with Turnaround Time*.*

Proof. We prove that feasible original schedules correspond to disjoint subcollections of J, preserving weight. In temporal order, two extended closed intervals are disjoint exactly when $f_i + \Delta < s_j$. These are the turnaround constraints. Requiring them for consecutive chosen intervals also separates every earlier interval from every later one, proving both directions. Each interval retains its weight, proving the claim.

The correctness of WeightedIntervals gives an optimum extended collection. Mapping back its indices therefore gives an optimum schedule. □

Theorem 78.2. WeightedTurnaround *takes* $O(n \log n)$ *time.*

Proof. Construction and recovery take $O(n)$ time. The call takes $O(n \log n)$. Their sum is $O(n \log n)$. □

Problem · p. 17

79. Equal-Sum Partition

Algorithm EqualPartition($A[1..n]$)
1 $S \leftarrow \sum_i A[i]$; **if** S is odd **then return** impossible
2 $J \leftarrow$ SubsetSum($A, S/2$)
3 **if** the call reports failure **then return** impossible
4 **return** $(J, \{1, \ldots, n\} \setminus J)$

Theorem 79.1. EqualPartition *solves* Equal-Sum Partition.

Proof. We prove that a partition exists exactly when S is even and a subset of sum $S/2$ exists.

($\Rightarrow$) Equal integer sums must each equal $S/2$, so S must be even and either part is such a subset.

($\Leftarrow$) A subset of sum $S/2$ has a complement of sum $S - S/2 = S/2$, giving the desired partition.

The correctness of SubsetSum tests precisely this subset condition and supplies it on success. Thus both failure reports and the output are correct. □

Theorem 79.2. EqualPartition *takes* $O(nS)$ *time.*

Proof. Summation and complement construction cost $O(n)$. The subset-sum call costs $O(n(S/2+1)) = O(nS)$ because $S \geq 1$. This also bounds the scans. □

Problem · p. 17

80. Most Balanced Partition

$D[i, z]$ records whether the first i entries have a subset of sum z.

Algorithm BalancedPartition($A[1..n]$)

```
1 S ← ∑_i A[i]; set D[0,0] ← true and D[0,z] ← false for z > 0
2 for i = 1,...,n and z = 0,...,S do
3     D[i,z] ← D[i−1,z] or (z ≥ A[i] and D[i−1,z−A[i]])
4 choose z with D[n,z] = true minimizing |S − 2z|; J ← ∅
5 for i = n,...,1 do
6     if not D[i−1,z] then add i to J; z ← z − A[i]
7 return (J, {1,...,n} \ J)
```

Theorem 80.1. BalancedPartition *solves* Most Balanced Partition.

Proof. We prove that $D[i, z]$ is true exactly when a subset of the first i entries has sum z. We use induction on i.

Base case. The empty prefix has only the empty subset, of sum zero.

Induction step. Classify feasible subsets by whether they contain i:

(i) If i is omitted, such a subset exists exactly when $D[i-1, z]$ is true.

(ii) If i is included, it exists exactly when $z \geq A[i]$ and $D[i-1, z-A[i]]$ is true: remove or append i to obtain the correspondence.

The two cases exhaust all subsets, proving the recurrence and the claim. They use only the previous row, which is computed first. A subset of sum z gives partition difference $|S-2z|$, so the chosen reachable sum is optimal. During recovery the current entry remains true; if a skip is impossible, the recurrence guarantees a take. The base row forces remaining sum zero, so the recovered subset attains the chosen sum. □

Theorem 80.2. BalancedPartition *takes* $O(nS)$ *time.*

Proof. The table has $O(nS)$ entries since $S \geq 1$, and each entry costs constant time. Selecting a sum costs $O(S)$; recovery and complement construction cost $O(n)$. These costs sum to $O(nS)$. □

Problem · p. 18

81. Subset Sum with Fewest Items

Let $D[i, z]$ be the fewest items in prefix $1..i$ having sum z.

Algorithm FewestItems($A[1..n], K$)

```
1 set D[0,0] ← 0 and D[0,z] ← +∞ for 1 ≤ z ≤ K
2 for i = 1,...,n and z = 0,...,K do
3     D[i,z] ← D[i−1,z]; record skip
4     if A[i] ≤ z and 1 + D[i−1,z−A[i]] < D[i,z] then
5         D[i,z] ← 1 + D[i−1,z−A[i]]; record take
6 if D[n,K] = +∞ then return impossible
7 trace recorded cases from (n,K) to row zero; return taken indices
```

Theorem 81.1. FewestItems *solves* Subset Sum with Fewest Items.

Proof. We prove that $D[i, z]$ is the minimum cardinality of a subset of prefix $1..i$ with sum z, or $+\infty$ if none exists. We use induction on i.

Base case. Only sum zero is possible in the empty prefix, using no items.

Induction step. Classify feasible subsets by whether they contain i:

(i) If i is omitted, the best cardinality is $D[i-1, z]$.

(ii) Including i requires $A[i] \le z$. Removing it leaves a subset of the shorter prefix with sum $z - A[i]$, so the best cardinality is $1 + D[i-1, z-A[i]]$. Conversely, appending i realizes this candidate.

By the induction hypothesis, both candidates are correct, with infinity for impossible cases. Their minimum proves the claim. The previous row is already computed. An infinite final entry therefore means impossibility; otherwise tracing attaining choices reaches the base entry of sum zero and yields a subset of sum K with minimum cardinality $D[n, K]$. □

Theorem 81.2. FewestItems *takes* $O(nK)$ *time.*

Proof. There are $(n+1)(K+1) = O(nK)$ entries, each computed in constant time. Recording choices has the same cost. Recovery takes $O(n)$, within the bound. □

Problem · p. 18

82. Unbounded Knapsack

$D[c]$ is the best value with capacity c; a null choice means to leave the remaining space unused.

Algorithm RepeatedItems$(w[1..n], v[1..n], W)$

```
1 D[0] ← 0; P[0] ← nil; set all multiplicities a_i ← 0
2 for c = 1, ..., W do
3     D[c] ← 0; P[c] ← nil
4     for each i with w_i ≤ c do
5         if v_i + D[c − w_i] > D[c] then D[c] ← v_i + D[c − w_i]; P[c] ← i
6 c ← W; while c > 0 and P[c] ≠ nil do
7     i ← P[c]; a_i ← a_i + 1; c ← c − w_i
8 return a
```

Theorem 82.1. RepeatedItems *solves* Unbounded Knapsack.

Proof. We prove that $D[c]$ is the maximum value of a multiset of total weight at most c. We use induction on c.

Base case. Positive weights allow only the empty multiset at capacity zero.

Induction step. For $c > 0$, there are two cases:

(i) If the multiset is empty, this gives the candidate value zero.

(ii) If the multiset is nonempty, remove a copy of any item i it contains. The remainder fits capacity $c - w_i < c$ and has value at most $D[c - w_i]$ by induction. Conversely, adding i to an optimum for that capacity is feasible and attains $v_i + D[c - w_i]$.

Maximizing over these cases proves the claim; all smaller capacities are already computed. Stored choices attain each entry. Recovery preserves accumulated value plus $D[c]$ and never exceeds the original capacity. A null choice leaves value zero, so the returned multiplicities attain $D[W]$. □

Theorem 82.2. RepeatedItems *takes* $O(nW)$ *time.*

Proof. There are W capacities and at most n candidates per capacity. Each candidate costs constant time. Recovery has at most W iterations because weights are positive integers, and writing n multiplicities costs $O(n)$. The total is $O(nW)$. □

Problem · p. 18

83. Exact-Weight Knapsack

Algorithm ExactKnapsack$(w[1..n], v[1..n], W)$

```
1 D[0,0] ← 0; set D[0,c] ← −∞ for 1 ≤ c ≤ W
2 for i = 1,...,n and c = 0,...,W do
3     take the larger of D[i−1,c] and, if w_i ≤ c, v_i + D[i−1,c−w_i]
4     store that value in D[i,c] and an attaining skip/take choice
5 if D[n,W] = −∞ then return impossible
6 trace choices from (n,W) to row zero; return the taken indices
```

Theorem 83.1. ExactKnapsack *solves* Exact-Weight Knapsack.

Proof. We prove that $D[i, c]$ is the maximum value of a subset of the first i items with exact weight c, or $-\infty$ if none exists. We use induction on i.

Base case. The empty prefix permits only weight zero, with value zero.

Induction step. Classify feasible subsets by whether they contain i:

(i) If i is omitted, the optimum is $D[i-1, c]$.

(ii) Including i requires $w_i \le c$. Removing it leaves exact weight $c - w_i$ in the shorter prefix, so this case has optimum $v_i + D[i-1, c-w_i]$; every finite candidate extends to a feasible subset by appending i.

The induction hypothesis makes both candidates correct. Their maximum proves the claim, and the previous-row order is valid. The final entry detects impossibility. Otherwise recorded choices attain it and reach the sole finite base entry, yielding distinct items of weight exactly W and optimum value. □

Theorem 83.2. ExactKnapsack *takes* $O(nW)$ *time.*

Proof. The table has $O(nW)$ entries, each considering at most two candidates in constant time. Recovery uses at most n steps. Including initialization gives $O(nW)$. □

Problem · p. 18

84. Knapsack with Two Resource Limits

Algorithm TwoResources$(a[1..n], b[1..n], v[1..n], A, B)$

```
set D[0, x, y] ← 0 for 0 ≤ x ≤ A, 0 ≤ y ≤ B
for i = 1, ..., n, x = 0, ..., A, y = 0, ..., B do
    D[i, x, y] ← D[i − 1, x, y]; record skip
    if a_i ≤ x and b_i ≤ y then
        consider v_i + D[i − 1, x − a_i, y − b_i]; update the value and choice if larger
trace choices from (n, A, B), subtracting both requirements at a take
return the taken indices
```

Theorem 84.1. TwoResources *solves* Knapsack with Two Resource Limits.

Proof. We prove that $D[i, x, y]$ is the maximum value of a subset of the first i items meeting limits x, y. We use induction on i.

Base case. The empty prefix has optimum value zero for all limits.

Induction step. Classify feasible subsets by whether they contain i:

(i) If i is omitted, the optimum is $D[i-1, x, y]$.

(ii) If i is included, both requirements must fit. Removing i leaves limits $x - a_i, y - b_i$, so this case has optimum $v_i + D[i-1, x-a_i, y-b_i]$. Conversely, any such smaller solution extends feasibly by i.

Taking the better attainable case proves the claim by the induction hypothesis. All dependencies are in the previous row. Recovery follows attaining choices and subtracts exactly the used resources; at row zero the remaining value is zero. Thus the returned set meets both limits and attains $D[n, A, B]$. □

Theorem 84.2. TwoResources *takes* $O(nAB)$ *time.*

Proof. There are $n(A+1)(B+1) = O(nAB)$ transitions and $O(AB)$ base entries. Each has constant work. Recovery takes $O(n)$ time. Their sum is $O(nAB)$. □

Problem · p. 18

85. Maximum-Weight Increasing Subsequence

Algorithm WeightedIncreasing($A[1..n], w[1..n]$)

```
1 for i = 1, ..., n do
2     D[i] ← w_i; P[i] ← nil
3     for j = 1, ..., i − 1 with A[j] < A[i] do
4         if D[j] + w_i > D[i] then D[i] ← D[j] + w_i; P[i] ← j
5 choose i maximizing D[i]; follow P to nil
6 return the visited indices in reverse order
```

Theorem 85.1. WeightedIncreasing *solves* Maximum-Weight Increasing Subsequence.

Proof. We prove that $D[i]$ is the maximum weight of a nonempty increasing subsequence ending at i. We use induction on i.

Base case. For $i = 1$, the only such subsequence is the singleton, with weight w_1.

Induction step. For $i > 1$, consider the two possible forms:

(i) A singleton has weight w_i, as initialized.

(ii) A longer subsequence has a penultimate index $j < i$ with $A[j] < A[i]$. By the induction hypothesis, its prefix has weight at most $D[j]$. Conversely, an attaining subsequence ending at such a j extends by i and has weight $D[j] + w_i$.

These candidates exhaust the possibilities, so their maximum proves the claim. The forward order computes dependencies first. Stored pointers reconstruct an attaining increasing subsequence. Every nonempty subsequence has a last index, so the largest $D[i]$ is optimal. Nonnegative weights ensure that a nonempty optimum exists. □

Theorem 85.2. WeightedIncreasing *takes* $O(n^2)$ *time.*

Proof. There are $n(n-1)/2$ candidate pairs, each checked in constant time. Initialization, selecting the endpoint, and recovery take $O(n)$ time. Thus the total is $O(n^2)$. □

Problem · p. 19

86. Longest Bitonic Subsequence

Maxima over empty candidate sets are zero. Store an attaining predecessor or successor whenever one exists.

Algorithm BITONIC($A[1..n]$)

```
1 for i = 1, ..., n do L[i] ← 1 + max({0} ∪ {L[j] : j < i, A[j] < A[i]})
2 for i = n, ..., 1 do R[i] ← 1 + max({0} ∪ {R[j] : j > i, A[j] < A[i]})
3 choose p maximizing L[p] + R[p] − 1
4 recover the increasing chain ending at p and the decreasing chain starting at p
5 return their concatenation, listing p only once
```

Theorem 86.1. BITONIC *solves* Longest Bitonic Subsequence.

Proof. We prove that $L[i]$ is the longest increasing-subsequence length ending at i, and $R[i]$ is the longest decreasing-subsequence length starting there. We use induction on the number of indices processed in each scan.

Base case. At the first index of either scan, only a singleton is possible, giving length one.

Induction step. There are two possible forms of subsequence:

(i) A singleton has length one.

(ii) A longer subsequence has a penultimate index $j < i$ for L, or a second index $j > i$ for R, with $A[j] < A[i]$. Removing i gives exactly the corresponding smaller state. By the induction hypothesis its best length is known, and an attaining chain extends by i.

Taking the maximum proves both formulas and justifies the stored chains.

For a fixed peak p, every bitonic subsequence has length at most $L[p] + R[p] - 1$. Attaining chains for these entries share only p, since their other indices lie on opposite sides of p. Joining them attains the bound. Maximizing over p returns a longest bitonic subsequence. □

Theorem 86.2. BITONIC *takes* $O(n^2)$ *time.*

Proof. Each of the two passes examines at most n^2 pairs with constant work per pair. Choosing the peak and tracing the two chains take $O(n)$ time. The total is $O(n^2)$. □

Problem · p. 19

87. Counting Longest Increasing Subsequences

All count additions are modulo p. Different index sequences are counted separately, even if their values agree.

Algorithm CountIncreasing$(A[1..n], p)$

```
1 for i = 1, ..., n do
2     L[i] ← 1; C[i] ← 1
3     for j < i with A[j] < A[i] do
4         if L[j] + 1 > L[i] then L[i] ← L[j] + 1; C[i] ← C[j]
5         else if L[j] + 1 = L[i] then C[i] ← (C[i] + C[j]) mod p
6 h ← max_i L[i]; return (h, Σ_{i:L[i]=h} C[i] mod p)
```

Theorem 87.1. CountIncreasing *solves* Counting Longest Increasing Subsequences.

Proof. We prove that $L[i]$ is the longest increasing-subsequence length ending at i, and $C[i]$ counts those subsequences modulo p. We use induction on i.

Base case. At $i = 1$, the singleton is the only subsequence, so its length and count are both one.

Induction step. For $i > 1$, there are two possible forms:

(i) The singleton supplies one subsequence of length one.

(ii) A longer subsequence has a unique penultimate index $j < i$ with $A[j] < A[i]$. By the induction hypothesis, the best length for this j is $L[j] + 1$, with $C[j]$ ways: removing or appending i gives a bijection.

A better length replaces the count; an equal length adds counts from disjoint classes. Modular addition preserves the count modulo p, proving the claim. Finally, longest subsequences with different last indices are disjoint. Summing over endpoints of maximum length therefore gives the required total. □

Theorem 87.2. CountIncreasing *takes* $O(n^2)$ *time.*

Proof. There are $O(n^2)$ pairs and constant work per pair. Counts stay below p after reduction, so their arithmetic costs constant time under the word convention. Initialization and the final sum take $O(n)$. Total time is $O(n^2)$. □

Problem · p. 19

88. Longest Nondecreasing Subsequence

Algorithm NONDECREASING($A[1..n]$)
1 **for** $i = 1, \ldots, n$ **do** $B[i] \leftarrow (n+1)A[i] + i$
2 **return** FASTINCREASINGSUBSEQUENCE(B)

Theorem 88.1. NONDECREASING *solves* Longest NONDECREASING Subsequence.

Proof. We prove that, for $i < j$, $B[i] < B[j]$ if and only if $A[i] \leq A[j]$.

($\Rightarrow$) We use contraposition. If $A[i] > A[j]$, integrality gives $B[i] - B[j] \geq n+1+i-j > 0$, so $B[i] < B[j]$ is false.

($\Leftarrow$) If $A[i] \leq A[j]$, multiplying by $n+1$ preserves the inequality, and adding $i < j$ makes $B[i] < B[j]$.

Thus the same index sequences are nondecreasing in A and strictly increasing in B, with unchanged lengths. The correctness of FASTINCREASINGSUBSEQUENCE therefore gives a longest nondecreasing subsequence of A. □

Theorem 88.2. NONDECREASING *takes* $O(n \log n)$ *time.*

Proof. Constructing B takes $O(n)$ time on a constant number of words per entry. The call takes $O(n \log n)$. This dominates construction. □

Problem · p. 19

89. Fewest Deletions to Make an Array Increasing

Algorithm IncreasingDeletions($A[1..n]$)

```
1 J ← FastIncreasingSubsequence(A)
2 mark all indices of J
3 return all unmarked indices
```

Theorem 89.1. IncreasingDeletions *solves* Fewest Deletions to Make an Array Increasing.

Proof. We prove that a deletion set is feasible if and only if its retained subsequence is strictly increasing.

($\Rightarrow$) A feasible deletion leaves the retained entries in their original order, strictly increasing as required.

($\Leftarrow$) Retaining an increasing subsequence and deleting its complement gives a feasible deletion set.

The deletion count is n minus the retained length.

The correctness of FastIncreasingSubsequence gives a longest such subsequence J. Its complement therefore has minimum deletion count. □

Theorem 89.2. IncreasingDeletions *takes* $O(n \log n)$ *time.*

Proof. The call to FastIncreasingSubsequence takes $O(n \log n)$. Marking and scanning cost $O(n)$, giving the stated total. □

Problem · p. 19

90. Nesting Rectangles

Algorithm NestedRectangles($a[1..n], b[1..n]$)

```
1 sort rectangle records by increasing a_i, breaking ties by decreasing b_i
2 J ← FastIncreasingSubsequence applied to the sorted heights
3 return the original rectangle indices at positions J
```

Theorem 90.1. NestedRectangles *solves* Nesting Rectangles.

Proof. We prove that nesting chains correspond to strictly increasing height subsequences in the sorted records, with unchanged lengths.

($\Rightarrow$) A nesting chain has strictly increasing widths, so its records appear in the sorted order. Its heights are strictly increasing.

($\Leftarrow$) A strictly increasing height subsequence cannot use two equal-width records, because heights within such a block are nonincreasing. Its widths are therefore strictly increasing too, giving a nesting chain.

The correctness of FastIncreasingSubsequence gives a longest height subsequence, which is an optimum chain by the claim. □

Theorem 90.2. NestedRectangles *takes* $O(n \log n)$ *time.*

Proof. Sorting costs $O(n \log n)$; each comparison of two records has constant cost. The subsequence call has the same bound. Recovering original indices costs $O(n)$, so the total is $O(n \log n)$. □

Problem · p. 20

91. Longest Palindromic Subsequence

Initialize $D[i, i-1] = 0$ and $D[i, i] = 1$. A recorded pair case places equal characters on opposite ends of the answer.

Algorithm PalindromicSubsequence($A[1..n]$)

```
initialize the empty-interval and singleton entries
for interval lengths ℓ = 2, ..., n do
    for each interval [i, j] of length ℓ do
        D[i, j] ← max{D[i + 1, j], D[i, j − 1]}; record an attaining skip
        if A[i] = A[j] then also consider 2 + D[i + 1, j − 1]
        record pair if that case is chosen
follow recorded choices from [1, n]; return the recovered palindrome
```

Theorem 91.1. PalindromicSubsequence *solves* Longest Palindromic Subsequence.

Proof. We prove that $D[i, j]$ is the longest palindromic-subsequence length in $A[i..j]$. We use induction on interval length.

Base cases. Empty intervals have length zero; singletons have length one.

Induction step. For an interval of length at least two, there are two cases:

(i) If an endpoint is omitted, the subsequence belongs to $[i + 1, j]$ or $[i, j - 1]$, whose optimum is known by induction.

(ii) If both endpoints are used, their characters must agree. Removing them leaves an interior palindrome, giving bound $2 + D[i + 1, j - 1]$.

Every candidate is attainable: keep a palindrome from a skipped interval, or surround an interior palindrome by the equal endpoints. Thus the maximum is exact, proving the claim. All dependencies are shorter intervals and are computed first. Recovery follows attaining choices, so it returns a palindrome of optimum length $D[1, n]$. □

Theorem 91.2. PalindromicSubsequence *takes* $O(n^2)$ *time.*

Proof. There are $O(n^2)$ intervals with at most three constant-time candidates each. Recovery visits at most n states; collect left and right halves in lists and reverse one list once, for $O(n)$ time. The total is $O(n^2)$. □

Problem · p. 20

92. Shortest Common Supersequence

For suffixes, initialize $D[n+1, j] = m - j + 1$ and $D[i, m+1] = n - i + 1$.

Algorithm CommonSupersequence($A[1..n], B[1..m]$)

```
initialize boundary entries
for i = n, ..., 1 and j = m, ..., 1 do
    consider 1 + D[i + 1, j], 1 + D[i, j + 1]
    if A[i] = B[j], also consider 1 + D[i + 1, j + 1]
    store the minimum and an attaining case
trace from (1, 1), emitting the character consumed by each chosen case
append the remaining suffix at a boundary; return the output string
```

Theorem 92.1. CommonSupersequence *solves* Shortest Common Supersequence.

Proof. We prove that $D[i, j]$ is the shortest common-supersequence length of $A[i..n]$ and $B[j..m]$. We use induction on the total remaining length.

Base case. With one suffix empty, the other suffix is a shortest common supersequence, giving the initialized length.

Induction step. Fix embeddings of both suffixes in a shortest common supersequence. Its first character cannot be unused, or it could be deleted. It serves the front of A, the front of B, or both; the last case requires equality. Removing that character leaves the corresponding smaller suffix problem, giving one of the three lower bounds in the recurrence. Conversely, each candidate prepends the appropriate character to a smaller common supersequence and is feasible. The induction hypothesis thus makes the minimum exact. The table computes every dependency first.

Recovery follows attaining cases and then appends the remaining suffix. It contains both inputs in order and has optimum length $D[1, 1]$. □

Theorem 92.2. CommonSupersequence *takes* $O(nm)$ *time.*

Proof. There are nm interior entries, each comparing at most three candidates. Boundaries and recovery cost $O(n + m)$, using constant-time list appends. Since lengths are positive, the total is $O(nm)$. □

Problem · p. 20

93. Common Subsequence of Three Strings

Algorithm ThreeCommon($A[1..n], B[1..m], C[1..\ell]$)

```
1 set D[i, j, k] ← 0 whenever one suffix is empty
2 for i = n, ..., 1, j = m, ..., 1, k = ℓ, ..., 1 do
3     D[i, j, k] ← max{D[i + 1, j, k], D[i, j + 1, k], D[i, j, k + 1]}
4     if A[i] = B[j] = C[k] then
5         D[i, j, k] ← max{D[i, j, k], 1 + D[i + 1, j + 1, k + 1]}
6 return D[1, 1, 1]
```

Theorem 93.1. ThreeCommon *solves* Common Subsequence of Three Strings.

Proof. We prove that $D[i, j, k]$ is the longest common-subsequence length of the three indicated suffixes. We use induction on their total length.

Base case. An empty suffix permits only length zero.

Induction step. When all three suffixes are nonempty, there are two cases:

(i) If some front position is omitted, the subsequence is feasible after skipping that position, giving one of the three skip candidates.

(ii) If all front positions are used, represent the subsequence by increasing triples of matching positions. Each front position is earliest in its string, so all three occur in the first triple. Their characters agree, and deleting that triple leaves the smaller suffix problem of line 5.

Every candidate also extends to a common subsequence of the original suffixes. Thus taking the maximum is exact by induction. All dependencies are computed first, and $D[1, 1, 1]$ is the required length. □

Theorem 93.2. ThreeCommon *takes* $O(nm\ell)$ *time.*

Proof. The full table has $(n+1)(m+1)(\ell+1) = O(nm\ell)$ entries for positive input lengths. Boundary initialization and each four-candidate transition take constant time per entry. Returning the last value is constant-time, giving $O(nm\ell)$. □

Problem · p. 20

94. Edit Distance with Unequal Costs

Write $[P] = 1$ when P holds and zero otherwise. All substitutions between unequal characters have the same cost s.

Algorithm WeightedEdit$(A[1..n], B[1..m], d, a, s)$

```
1 set D[n+1, j] ← a(m − j + 1) and D[i, m+1] ← d(n − i + 1)
2 for i = n, ..., 1 and j = m, ..., 1 do
3     consider d + D[i+1, j], a + D[i, j+1], and s[A[i] ≠ B[j]] + D[i+1, j+1]
4     store the minimum in D[i, j] and an attaining case
5 trace cases from (1, 1), emitting deletion, insertion, or paired columns
6 append the remaining gap columns; return (D[1, 1], alignment)
```

Theorem 94.1. WeightedEdit *solves* Edit Distance with Unequal Costs.

Proof. We prove that $D[i, j]$ is the minimum alignment cost for the two suffixes. We use induction on their total remaining length.

Base cases. An empty suffix forces gap columns for the other suffix, giving exactly the initialized boundary values.

Induction step. The first column is a deletion, an insertion, or a character pair. Removing it leaves the corresponding smaller state; conversely, that column can precede any alignment of the smaller state. The induction hypothesis therefore makes the minimum of the three candidates exact. All dependencies are computed first, and recovery follows attaining choices to give an alignment of cost $D[1, 1]$.

We claim that minimum alignment cost equals minimum edit cost. We prove both inequalities:

(i) An alignment gives an edit sequence of the same cost.

(ii) Conversely, track original character identities through any edit sequence. Deleted originals require deletions, surviving inserted characters require insertions, and changed surviving originals require substitutions. These are distinct charges. Pair surviving originals with their final characters and use gaps for the others. Uniform nonnegative costs make this alignment no more expensive than the edit sequence.

The equivalence shows that the recovered alignment attains minimum edit cost. □

Theorem 94.2. WeightedEdit *takes* $O(nm)$ *time.*

Proof. There are nm interior entries, each using three constant-time candidates. Boundaries and recovery cost $O(n + m)$; columns are appended to a list. For positive input lengths the total is $O(nm)$. □

Problem · p. 20

95. Closest Substring by Edit Distance

$D[i, j]$ is the smallest distance from $A[1..i]$ to a substring ending at position j. An empty substring may end at any position.

Algorithm ClosestSubstring($A[1..n], B[1..m]$)

```
set D[0, j] ← 0 for all j and D[i, 0] ← i for all i
for i = 1, ..., n and j = 1, ..., m do
    D[i, j] ← min{1 + D[i − 1, j], 1 + D[i, j − 1], [A[i] ≠ B[j]] + D[i − 1, j − 1]}
    record an attaining case
choose b ∈ {0, ..., m} minimizing D[n, b]
trace cases backwards from (n, b) until reaching (0, k)
return (B[k + 1..b], D[n, b])
```

Theorem 95.1. ClosestSubstring *solves* Closest Substring by Edit Distance.

Proof. We prove that $D[i, j]$ is the minimum alignment cost between $A[1..i]$ and a substring ending at j, allowing an empty substring. We use induction on $i + j$.

Base cases. An empty prefix of A can match an empty substring ending anywhere at cost zero. At $j = 0$, all i characters must be deleted.

Induction step. A nonempty alignment ends in a deletion of $A[i]$, an insertion of $B[j]$, or a paired column. Removing this column gives the corresponding smaller state; conversely, each candidate extends a feasible smaller alignment. Successive deletions also represent an empty substring. The induction hypothesis therefore proves the minimum exact, and all its dependencies are computed first.

The equivalence of alignments and edit sequences identifies these costs with edit distance. Every substring has an ending position b, so minimizing $D[n, b]$ finds the global optimum. Backtracking to $(0, k)$ consumes exactly $B[k + 1..b]$ and attains that cost, proving the output correct. □

Theorem 95.2. ClosestSubstring *takes* $O(nm)$ *time.*

Proof. Each of nm entries takes constant time. Initializing boundaries, choosing b, and tracing at most $n + m$ steps take $O(n + m)$. The bound is therefore $O(nm)$. □

Problem · p. 21

96. Editing with Insertions and Deletions

Algorithm InsertDeleteDistance($A[1..n], B[1..m]$)
1 $Z \leftarrow$ CommonSubsequence(A, B)
2 **return** $n + m - 2|Z|$

Theorem 96.1. InsertDeleteDistance *solves* Editing with Insertions and Deletions.

Proof. We prove that the minimum edit count is $n + m - 2L$, where L is the longest common-subsequence length. We prove a lower bound and an attaining construction.

Lower bound. Suppose k original characters survive an edit sequence. They occur in order in both strings, so $k \leq L$. At least $n - k$ deletions remove the other originals, and at least $m - k$ insertions supply the other final characters. The total is at least $n + m - 2L$.

Attainment. Keep a longest common subsequence in A, delete the other $n - L$ characters, and insert the other $m - L$ characters of B. This uses exactly the claimed number of edits.

The correctness of CommonSubsequence gives $|Z| = L$, so the returned formula is the optimum. □

Theorem 96.2. InsertDeleteDistance *takes* $O(nm)$ *time.*

Proof. CommonSubsequence takes $O(nm)$. Obtaining $|Z|$ costs at most $O(n+m)$, and the final arithmetic is constant-time. These fit $O(nm)$. □

Problem · p. 21

97. Maximum-Weight Independent Set on a Tree

Root T and write $\mathrm{ch}(v)$ for the children of v. Empty sums are zero.

Algorithm WeightedTreeSet(T, w)

```
root T at r by DFS; process children before parents
for each v in this order do
    D_1[v] ← w(v) + Σ_{u∈ch(v)} D_0[u]
    D_0[v] ← Σ_{u∈ch(v)} max{D_0[u], D_1[u]}
recover from r in a state attaining max{D_0[r], D_1[r]}
in state 1, output v and give every child state 0
in state 0, give each child a state attaining its larger entry
return the output vertices
```

Theorem 97.1. WeightedTreeSet *solves* Maximum-Weight Independent Set on a Tree.

Proof. We prove that $D_b[v]$ is the maximum weight of an independent set in v's subtree, with v included exactly when $b = 1$. We use induction on subtree size.

Base case. A leaf has values $D_0[v] = 0$ and $D_1[v] = w(v)$.

Induction step. For an internal vertex v, consider the two states:

(i) If v is included, every child must be omitted, giving $w(v) + \sum_u D_0[u]$ by induction.

(ii) If v is omitted, each child may use either state, giving $\sum_u \max\{D_0[u], D_1[u]\}$.

Different child subtrees have no connecting edges, so these choices are independent and each sum is attainable. This proves the claim, and postorder computes dependencies first. Recovery combines attaining child sets under the same rules, preserving independence and weight. Choosing the better root state therefore gives an unrestricted optimum. □

Theorem 97.2. WeightedTreeSet *takes* $O(n)$ *time.*

Proof. A tree has $n - 1$ edges. DFS constructs child lists and postorder in $O(n)$ time. Each parent–child edge is used a constant number of times in the sums and recovery, and each vertex is output at most once. The total is $O(n)$. □

Problem · p. 21

98. Minimum-Weight Vertex Cover on a Tree

Root the tree. State $b = 1$ means that the subtree root is in the cover.

Algorithm TreeCover(T, c)

```
construct children lists and a postorder by DFS
for each vertex v in postorder do
    D_0[v] ← Σ_{u∈ch(v)} D_1[u]
    D_1[v] ← c(v) + Σ_{u∈ch(v)} min{D_0[u], D_1[u]}
recover a minimum-cost state at the root
output a vertex precisely in state 1
give its children state 1 if it is in state 0, otherwise minimizing states
return the output set
```

Theorem 98.1. TreeCover *solves* Minimum-Weight Vertex Cover on a Tree.

Proof. We prove that $D_b[v]$ is the minimum cost of a cover of all edges within v's subtree, with v included exactly when $b = 1$. We use induction on subtree size.

Base case. A leaf has no internal edges, so the two costs are zero and $c(v)$.

Induction step. For an internal vertex v, consider the two states:

(i) If v is omitted, each child must cover its edge to v, forcing state one and total $\sum_u D_1[u]$.

(ii) If v is included, all edges to children are covered, so each child may use a cheaper state. The total is $c(v) + \sum_u \min\{D_0[u], D_1[u]\}$.

Child subtrees have independent remaining constraints. The induction hypothesis therefore makes both sums exact. Postorder computes them in valid order. Recovery follows minimizing states, covers every tree edge, and attains the chosen cost. The cheaper root state is consequently an optimum cover. □

Theorem 98.2. TreeCover *takes* $O(n)$ *time.*

Proof. DFS and postorder take $O(n)$ time. Summed over all vertices, child lists have $n - 1$ entries. Each is scanned a constant number of times in computation and recovery. Thus total time is $O(n)$. □

Problem · p. 21

99. Counting Independent Sets of a Tree

All operations on counts are modulo p; an empty product is one.

Algorithm CountTreeSets(T, p)

```
root T at r by DFS and obtain postorder
for each v in postorder do
    C_1[v] ← ∏_{u∈ch(v)} C_0[u] mod p
    C_0[v] ← ∏_{u∈ch(v)} (C_0[u] + C_1[u]) mod p
return (C_0[r] + C_1[r]) mod p
```

Theorem 99.1. CountTreeSets *solves* Counting Independent Sets of a Tree.

Proof. We prove that $C_b[v]$ counts independent subsets of v's subtree with v included exactly when $b = 1$, modulo p. We use induction on subtree size.

Base case. Each state of a leaf has one subset, as given by the empty products.

Induction step. For an internal vertex v, consider the two states:

(i) If v is included, each child must be omitted, giving $C_0[u]$ choices for that child's subtree.

(ii) If v is omitted, either child state is allowed. The two classes are disjoint, giving $C_0[u] + C_1[u]$ choices.

There are no edges between child subtrees, so their choices combine independently and bijectively into subtree subsets. The product rule and induction prove the formulas. Postorder computes dependencies first, and modular arithmetic preserves the residues. The two root states partition all independent sets, so their sum is the required answer. □

Theorem 99.2. CountTreeSets *takes* $O(n)$ *time.*

Proof. DFS costs $O(n)$. The products perform constant work for each parent–child edge. Reduction after every addition or multiplication keeps counts within a constant number of words under the stated model. There are $n - 1$ such edges, so the total is $O(n)$. □

Problem · p. 21

100. Height-Bounded Optimal Search Tree

Let $F(i, j) = \sum_{k=i}^{j} f_k$, available from prefix sums. Use $+\infty$ for impossible states.

Algorithm HEIGHTLIMITEDBST$(x[1..n], f[1..n], h)$

```
1 compute frequency prefix sums; set all empty-interval entries to 0
2 set D[0, i, j] ← +∞ for every nonempty interval
3 for b = 1, ..., h and every nonempty interval [i, j] do
4     D[b, i, j] ← F(i, j) + min_{i≤r≤j}(D[b − 1, i, r − 1] + D[b − 1, r + 1, j])
5     record an attaining root r
6 if D[h, 1, n] = +∞ then return impossible
7 build the recorded root and recursively build its two intervals with level b − 1
8 return the resulting tree
```

Theorem 100.1. HEIGHTLIMITEDBST *solves* Height-Bounded Optimal Search Tree.

Proof. We prove that $D[b, i, j]$ is the minimum search cost for keys $i..j$ at height at most b, or $+\infty$ if impossible. We use induction on b.

Base cases. Empty intervals have cost zero at every height. No nonempty tree fits height zero.

Induction step. A nonempty tree has some root x_r and ordered subtrees of height at most $b - 1$. Every search compares at the root, contributing $F(i, j)$; the remaining cost is the sum of its subtree costs. By induction this is at least the candidate for r. Conversely, attaching attaining subtrees realizes every finite candidate at height at most b. Minimizing over r proves the claim. All dependencies are in the preceding height layer.

A final infinite value therefore means impossibility. Otherwise recovery builds the recorded roots and attaining children. The same decomposition shows that it respects the height bound and attains $D[h, 1, n]$. □

Theorem 100.2. HEIGHTLIMITEDBST *takes* $O(hn^3)$ *time.*

Proof. Prefix sums take $O(n)$. There are $O(hn^2)$ states, each trying at most n roots at constant cost per root. Recovery builds n nodes in $O(n)$ time. Thus total time is $O(hn^3)$. □

Problem · p. 22

101. Optimal Alphabetic Tree

Use prefix sums for $W(i,j) = \sum_{k=i}^{j} w_k$. A full binary tree has two children at every internal node.

Algorithm AlphabeticTree($w[1..n]$)

```
compute prefix sums; set D[i, i] ← 0
for interval lengths ℓ = 2, ..., n do
    for each [i, j] of length ℓ do
        choose r ∈ {i, ..., j − 1} minimizing D[i, r] + D[r + 1, j]
        D[i, j] ← W(i, j) + D[i, r] + D[r + 1, j]; record r
build a leaf for a singleton, or an internal node with the two recorded interval trees
return the tree for [1, n]
```

Theorem 101.1. AlphabeticTree *solves* Optimal Alphabetic Tree.

Proof. We prove that $D[i,j]$ is the minimum weighted leaf depth of a full ordered binary tree for leaves $i..j$. We use induction on interval length.

Base case. A singleton is a leaf at depth zero and has cost zero.

Induction step. The root's two nonempty subtrees split the ordered leaves after some r. Attaching them raises every leaf depth by one, adding $W(i,j)$. The induction hypothesis bounds their remaining costs below by $D[i,r] + D[r+1,j]$. Conversely, attaching optimal subtrees attains this candidate. Taking the minimum over all splits proves the claim.

Shorter intervals are computed first. Recovery builds the recorded split at each interval, so it produces a full tree in the prescribed leaf order and attains the optimum $D[1,n]$. □

Theorem 101.2. AlphabeticTree *takes* $O(n^3)$ *time.*

Proof. There are $O(n^2)$ intervals and at most n candidate splits per interval. Prefix sums give each interval weight in constant time. Recovery creates n leaves and $n-1$ internal nodes, costing $O(n)$. Total time is $O(n^3)$. □

Problem · p. 22

102. Search Trees with Unsuccessful Queries

An unsuccessful search stops at a missing child; only comparisons with stored keys are charged. Define $W(i,j) = \sum_{k=i}^{j} f_k + \sum_{k=i-1}^{j} q_k$.

Algorithm SearchTreeWithGaps($x[1..n], f[1..n], q[0..n]$)

```
1 compute prefix sums of f, q; set D[i, i − 1] ← 0
2 for increasing nonempty interval lengths do
3     for each interval [i, j] do
4         choose r ∈ {i, . . . , j} minimizing D[i, r − 1] + D[r + 1, j]
5         D[i, j] ← W(i, j) + D[i, r − 1] + D[r + 1, j]; record r
6 build each recorded root with its two interval subtrees; return the tree
```

Theorem 102.1. SearchTreeWithGaps *solves* Search Trees with Unsuccessful Queries.

Proof. We prove that $D[i,j]$ is the minimum additional comparison cost for keys $i..j$ and gaps $i-1..j$. We use induction on the number of keys.

Base case. An empty interval stores no key and makes no further comparison, so its cost is zero.

Induction step. Every query in a nonempty interval compares once with root r, contributing $W(i,j)$. Queries for r stop. The remaining queries split into keys $i..r-1$ and gaps $i-1..r-1$ on the left, and keys $r+1..j$ and gaps $r..j$ on the right. By induction their least additional costs are $D[i,r-1]$ and $D[r+1,j]$. Conversely, attaching attaining subtrees realizes their sum. Minimizing over r proves the recurrence and the claim.

Shorter intervals are computed first. Recovery through the recorded roots constructs a valid search tree attaining $D[1,n]$, hence an optimum. □

Theorem 102.2. SearchTreeWithGaps *takes* $O(n^3)$ *time.*

Proof. Prefix sums make $W(i,j)$ available in constant time. The $O(n^2)$ intervals each try at most n roots; initialization and tree construction take $O(n)$. The total is $O(n^3)$. □

Problem · p. 22

103. Cheapest Walk with an Edge Limit

An infinite entry means that no such walk exists. Record either stay or an incoming edge.

Algorithm BoundedWalk(G, w, s, t, k)

```
1 D[0, s] ← 0; set all other entries of row zero to +∞
2 for b = 1, ..., k do
3     copy row b − 1 into row b and record stay
4     for each edge (u, v) do
5         if D[b − 1, u] + w(u, v) < D[b, v], update D[b, v] and record (u, v)
6 if D[k, t] = +∞ then return impossible
7 trace from (k, t) to row zero, collecting recorded edges; return their reverse
```

Theorem 103.1. BoundedWalk *solves* Cheapest Walk with an Edge Limit.

Proof. We prove that $D[b, v]$ is the minimum weight of an s–v walk using at most b edges, or $+\infty$ if none exists. We use induction on b.

Base case. At zero edges, only the empty walk at s is possible.

Induction step. For $b > 0$, consider the following possibilities:

(i) If the walk uses fewer than b edges, the optimum is already represented by $D[b-1, v]$.

(ii) If the walk uses exactly b edges, remove its last edge (u, v). The remaining walk has $b - 1$ edges and weight at least $D[b-1, u]$. Conversely, appending (u, v) to an attaining predecessor walk gives a feasible walk of weight $D[b-1, u] + w(u, v)$.

The minimum of these candidates is exact by induction. No restriction on negative cycles is needed because the edge bound is finite. Each row depends only on its predecessor. A finite final entry traces through attaining choices to s in row zero, giving an optimum walk with at most k edges; an infinite entry means impossibility. □

Theorem 103.2. BoundedWalk *takes* $O(k(n + m))$ *time.*

Proof. Every one of k layers copies n entries and scans m edges, for $O(n+m)$ time. Tracing takes $O(k)$ and initialization takes $O(n)$. Since $k \geq 1$, the total is $O(k(n + m))$. □

Problem · p. 22

104. Shortest Paths in a Weighted DAG

Algorithm DAGDistances(G, w, s)

```
1 run DFSTimes(G); order vertices by decreasing finish time
2 set d[s] ← 0 and d[v] ← +∞ for v ≠ s
3 for each vertex u in that order do
4     for each outgoing edge (u, v) do d[v] ← min{d[v], d[u] + w(u, v)}
5 return d
```

Theorem 104.1. DAGDistances *solves* Shortest Paths in a Weighted DAG.

Proof. We prove that, when v is processed, $d[v]$ is its shortest-path distance from s. Decreasing DFS finish order is topological. We use induction on position in this order.

Base case. The first vertex has no incoming edge. Its only possible path from s is the empty path if it equals s, as initialized.

Induction step. Every nonempty path to v has a final edge (u, v) from an earlier vertex. The induction hypothesis gives its best prefix weight $d[u]$. Conversely, extending an attaining prefix by (u, v) gives a path, since a repetition would create a cycle. The incoming updates therefore consider exactly all attainable final-edge candidates before v is processed. If no reachable predecessor exists, the initial label is correct. No nonempty path returns to s in a DAG, so its label remains zero.

Thus every final label is the required distance, including infinity for unreachable vertices. Negative edge weights do not affect the argument. □

Theorem 104.2. DAGDistances *takes* $O(n + m)$ *time.*

Proof. DFSTimes takes $O(n + m)$. Finish times lie in $[1, 2n]$, so buckets yield their order in $O(n)$ time. Initialization costs $O(n)$ and each outgoing edge is relaxed once. The total is $O(n + m)$. □

Problem · p. 22

105. Distances to a Destination

Algorithm DistancesToTarget(G, w, t)
1 construct G^R by replacing each (u, v) with (v, u) of the same weight
2 **return** BellmanFord(G^R, w^R, t)

Theorem 105.1. DistancesToTarget *solves* Distances to a Destination.

Proof. We prove that reversal is a weight-preserving bijection between v–t paths in G and t–v paths in G^R. Reversing a path interchanges its endpoints without changing its edges' weights, and reversing again recovers the original. This proves the claim directly.

Reversal also preserves cycle weights, so G^R has no negative cycle. The correctness of BellmanFord therefore gives its distances from t, including infinity for missing paths. By the bijection these are exactly the required distances to t in G. □

Theorem 105.2. DistancesToTarget *takes* $O(n + nm)$ *time.*

Proof. Reversal takes $O(n+m)$ time. BellmanFord takes $O(n+nm)$. Since $n \geq 1$, the latter bound includes reversal, giving $O(n + nm)$. □

Problem · p. 23

106. A Reachable Negative Cycle

Algorithm ReachableNegative(G, w, s)

```
1 R ← DFS(G, s) using outgoing adjacency lists
2 build the subgraph H induced by reached vertices, retaining original labels
3 C ← NegativeCycle(H, w|_H)
4 return its failure report, or C with original vertex labels
```

Theorem 106.1. ReachableNegative *solves* A Reachable Negative Cycle.

Proof. We prove that negative cycles in H are exactly the negative cycles of G reachable from s. The correctness of DFS identifies precisely the reachable vertices.

($\Rightarrow$) Every cycle of H is a cycle of G; its vertices are reachable from s, and its weight is unchanged.

($\Leftarrow$) Every vertex and edge of a reachable cycle is retained in H, with unchanged weight.

The correctness of NegativeCycle finds a negative cycle exactly when H contains one. The claim justifies both the failure report and the recovered original cycle. □

Theorem 106.2. ReachableNegative *takes* $O(n(n+m))$ *time.*

Proof. DFS and construction cost $O(n+m)$. If H has $n' \le n$ vertices and $m' \le m$ edges, NegativeCycle costs $O(n'(n'+m')) \subseteq O(n(n+m))$. Recovery is linear in the cycle length and fits that bound. □

Problem · p. 23

107. Vertices with Unboundedly Small Walk Weights

$D[b, v]$ is the minimum weight of an s–v walk using at most b edges. Reachability from a set is implemented by initializing one queue with that set.

Algorithm UnboundedVertices(G, w, s)

```
1 set D[0, s] ← 0 and D[0, v] ← +∞ for v ≠ s
2 for b = 1, ..., n do
3     copy row b − 1 to row b
4     for each (u, v) ∈ E do D[b, v] ← min{D[b, v], D[b − 1, u] + w(u, v)}
5 Q ← {v : D[n, v] < D[n − 1, v]}
6 search outgoing edges starting from every vertex of Q
7 return all reached vertices
```

Theorem 107.1. UnboundedVertices *solves* Vertices with Unboundedly Small Walk Weights.

Proof. We prove that the reported vertices are exactly those with unboundedly small walk weights from s. First, we claim that $D[b, v]$ is the minimum weight of an s–v walk using at most b edges. We use induction on b.

Base case. Row zero represents only the empty walk at s.

Induction step. A walk with at most b edges either uses fewer edges or has a final edge appended to a walk from the preceding row. The induction hypothesis makes the recurrence exact in both cases.

We now establish the two directions of the main claim:

($\Rightarrow$) Suppose a vertex is reported. It is reachable from an improved entry, which is attained by an n-edge walk. Such a walk repeats a vertex. If a simple cycle on it had nonnegative weight, removing it would contradict improvement over row $n - 1$. Thus a reachable negative cycle lies on a route to the improved vertex. Repeating that cycle before following the route and search path gives arbitrarily small weights at the reported vertex.

($\Leftarrow$) Suppose a vertex has unboundedly small walk weights. A reachable negative cycle must lie on a route to it; otherwise cycle removal leaves one of finitely many simple paths of no larger weight. All row-$n - 1$ labels on that cycle are finite. For a contradiction, suppose none improves. Every cycle edge then satisfies $D[n - 1, v] \leq D[n - 1, u] + w(u, v)$. Summing makes the cycle weight nonnegative, a contradiction. Some cycle vertex therefore belongs to Q, so the search reaches the given vertex. □

Theorem 107.2. UnboundedVertices *takes* $O(n(n + m))$ *time.*

Proof. There are n layers, each taking $O(n + m)$ time. Comparing the last two layers costs $O(n)$; the final search visits each vertex once and scans each edge once. Their total is $O(n(n + m))$. □

Problem · p. 23

108. A Cycle below a Mean-Weight Threshold

Algorithm LowMeanCycle(G, w, a, b)
1 **for each** edge e **do** $w'(e) \leftarrow b\,w(e) - a$
2 **return** NegativeCycle(G, w')

Theorem 108.1. LowMeanCycle *solves* A Cycle below a Mean-Weight Threshold.

Proof. We prove that a cycle has mean weight below a/b exactly when its transformed weight is negative. For a cycle C with $k > 0$ edges, direct calculation gives

$$\sum_{e \in C} w'(e) = b \sum_{e \in C} w(e) - ak.$$

Since $b, k > 0$, this is negative exactly when $\frac{1}{k} \sum_{e \in C} w(e) < a/b$, proving the claim.

The correctness of NegativeCycle therefore finds a desired cycle exactly when one exists, and any returned cycle has the required mean. □

Theorem 108.2. LowMeanCycle *takes* $O(n(n+m))$ *time.*

Proof. Transforming the weights takes $O(m)$ time, using arithmetic on a constant number of words. NegativeCycle takes $O(n(n+m))$ time on the unchanged graph, which also includes the transformation cost. □

Problem · p. 23

109. Transitive Closure

Algorithm TransitiveClosure(G)

```
1 set D[0, i, j] ← (i = j or (i, j) ∈ E) for all pairs
2 for k = 1, ..., n and i, j = 1, ..., n do
3     D[k, i, j] ← D[k − 1, i, j] or (D[k − 1, i, k] and D[k − 1, k, j])
4 return D[n, ·, ·]
```

Theorem 109.1. TransitiveClosure *solves* Transitive Closure.

Proof. We prove that $D[k, i, j]$ is true exactly when an i–j path exists with internal vertices in $\{1, \ldots, k\}$. We use induction on k.

Base case. With no internal vertices, only the empty path or a direct edge is possible, as initialized.

Induction step. For $k > 0$, there are two cases:

(i) If the path avoids k internally, such paths are represented by $D[k-1, i, j]$.

(ii) If the path uses k internally, splitting at its unique occurrence gives paths represented by $D[k-1, i, k]$ and $D[k-1, k, j]$. Conversely, concatenating paths for these entries gives a walk; deleting repeated portions gives a path without introducing any internal vertex outside $\{1, \ldots, k\}$.

These cases and the induction hypothesis prove the Boolean recurrence. Each layer uses the completed preceding layer. At $k = n$ all internal vertices are permitted, so the returned layer is exactly reachability. □

Theorem 109.2. TransitiveClosure *takes* $O(n^3)$ *time.*

Proof. Initialize an $n \times n$ Boolean matrix, then scan the edges to set their entries, in $O(n^2 + m) = O(n^2)$ time. The n^3 layer entries each use constant work. Returning the final layer gives total time $O(n^3)$. □

Problem · p. 23

110. Minimum-Bottleneck Paths

Algorithm BottleneckDistances(G, w)

```
initialize D[0, i, i] ← 0 and all off-diagonal entries to +∞
for each edge (i, j) do D[0, i, j] ← w(i, j)
for k = 1, ..., n and i, j = 1, ..., n do
    D[k, i, j] ← min{D[k − 1, i, j], max{D[k − 1, i, k], D[k − 1, k, j]}}
return D[n, ·, ·]
```

Theorem 110.1. BottleneckDistances *solves* Minimum-Bottleneck Paths.

Proof. We prove that $D[k, i, j]$ is the minimum bottleneck among i–j paths with internal vertices at most k, or infinity if none exists. We use induction on k.

Base case. Empty paths have bottleneck zero and direct paths have their edge weight, as initialized.

Induction step. For $k > 0$, there are two cases:

(i) If the path avoids k internally, the preceding-layer optimum is the first candidate.

(ii) If the path uses k internally, splitting the path at k bounds its bottleneck below by the maximum of the two preceding-layer optima. Conversely, concatenate attaining paths for those entries. Removing repeated portions cannot increase the maximum edge weight, giving a permitted path with bottleneck no greater than this candidate.

Thus the minimum of the candidates is both a lower bound and attainable up to improvement, proving equality and the claim. Dependencies use the previous layer. The final layer permits every internal vertex and gives the required optima. □

Theorem 110.2. BottleneckDistances *takes* $O(n^3)$ *time.*

Proof. Initialization costs $O(n^2 + m) = O(n^2)$. There are n^3 constant-time transitions, followed by a constant-time return of the final layer. The total is $O(n^3)$. □

Problem · p. 24

111. Minimum-Weight Directed Cycle

Algorithm LightestCycle(G, w)
1 $d \leftarrow$ FloydWarshall(G, w); $z \leftarrow +\infty$
2 **for each** edge (u, v) **do** $z \leftarrow \min\{z, w(u, v) + d[v, u]\}$
3 **if** $z = +\infty$ **then return** no cycle
4 **return** z

Theorem 111.1. LightestCycle *solves* Minimum-Weight Directed Cycle.

Proof. The correctness of FloydWarshall gives all shortest-path distances. We prove that the minimum finite candidate is the minimum cycle weight.

(i) For every cycle, choose an edge (u, v). Its remaining return path has weight at least $d[v, u]$, so the algorithm's minimum is no larger than that cycle's weight.

(ii) With no negative cycle, a shortest return walk can have its cycles removed without increasing weight, giving a simple shortest v–u path. Since $u \neq v$, adjoining (u, v) forms a directed cycle of weight $w(u, v) + d[v, u]$.

The two inequalities prove the claim. They also show that no finite candidate exists exactly when there is no cycle, justifying the failure report. □

Theorem 111.2. LightestCycle *takes $O(n^3)$ time.*

Proof. FloydWarshall takes $O(n^3)$. The final scan costs $O(m)$, and $m \leq n^2$ for a simple graph. Their sum is $O(n^3)$. □

Problem · p. 24

112. Minimum-Weight Hamiltonian Path

Let $U = V \setminus \{s\}$. Use subset masks and constant-time weight lookup with the exponential-table word convention.

Algorithm HamiltonianPath(G, w, s, t)

```
for each v ∈ U do D[{v}, v] ← w(s, v)
for increasing set sizes 2, ..., |U| do
    for each set S of that size and v ∈ S do
        D[S, v] ← min_{u∈S\{v}}(D[S \ {v}, u] + w(u, v))
        record an attaining predecessor u
trace predecessors from (U, t), removing the last vertex at each step
return s followed by the recovered vertices in forward order
```

Theorem 112.1. HamiltonianPath *solves* Minimum-Weight Hamiltonian Path.

Proof. We prove that $D[S, v]$ is the minimum weight of a path from s to v visiting exactly $\{s\} \cup S$. We use induction on $|S|$.

Base case. For $S = \{v\}$, the direct edge is the only such path.

Induction step. Every larger path has a penultimate vertex $u \in S \setminus \{v\}$. Removing its final edge gives the corresponding smaller state, so its weight is at least that state's optimum plus $w(u, v)$. Conversely, appending v to an attaining smaller path is feasible because v has not been visited. The minimum over u is therefore exact by induction.

Increasing set size computes all dependencies first. Stored predecessors attain the entries. The state (U, t) specifies all vertices and the required endpoints, so tracing it returns an optimum Hamiltonian path. □

Theorem 112.2. HamiltonianPath *takes* $O(n^2 2^n)$ *time.*

Proof. There are $O(n2^n)$ states, each examining at most n predecessors. Subset masks fit the address-word convention for the exponential table; grouping them by cardinality costs $O(n2^n)$. Weight lookup preparation costs $O(n^2)$ and recovery $O(n)$. Total time is $O(n^2 2^n)$. □

Problem · p. 24

113. Travelling Salesperson with Precedence Constraints

Let $R[v]$ be the set of required predecessors of v. Subsets use the exponential-table word convention, so a subset-containment test is a word operation.

Algorithm PrecedenceTour(G, w, R)

```
1 if R[1] ≠ ∅ then return impossible
2 initialize all D[S, v] ← +∞; D[{1}, 1] ← 0
3 for sets S ∋ 1 in increasing size and v ∈ S with finite D[S, v] do
4     for each u ∉ S with R[u] ⊆ S do
5         relax D[S ∪ {u}, u] with D[S, v] + w(v, u); store its predecessor
6 choose v ≠ 1 minimizing D[V, v] + w(v, 1)
7 if all candidates are infinite, return impossible
8 trace the attaining path, append its return to 1, and return the tour
```

Theorem 113.1. PrecedenceTour *solves* Travelling Salesperson with Precedence Constraints.

Proof. We prove that $D[S, v]$ is the minimum cost of a legal partial path starting at 1, visiting exactly S, and ending at v, or infinity if none exists. We use induction on $|S|$.

Base case. The starting visit is legal exactly when $R[1]$ is empty. Then $D[\{1\}, 1] = 0$ is the only initial feasible state.

Induction step. A new vertex u may be appended exactly when $R[u] \subseteq S$, because all its required predecessors must have been visited. Every permitted extension of a legal prefix is legal and repeats no vertex. Conversely, deleting the last visit of any legal path gives a legal shorter prefix whose extension is considered. The induction hypothesis therefore makes the minimum of incoming candidates exact.

All transitions go to larger sets. Closing a feasible full path back to 1 gives a legal tour, and every legal tour arises this way. Thus minimizing and tracing a final candidate gives an optimum, or correctly detects impossibility. □

Theorem 113.2. PrecedenceTour *takes* $O(n^2 2^n)$ *time.*

Proof. There are $O(n2^n)$ states and at most n extensions per state, each using constant-time mask and weight operations. Constructing predecessor masks and the weight matrix costs $O(n^2)$; subset enumeration costs $O(n2^n)$; recovery costs $O(n)$. The total is $O(n^2 2^n)$. □

Problem · p. 24

114. Bottleneck Travelling Salesperson

Fix start 1, put $U = V \setminus \{1\}$, and use subset masks with the exponential-table word convention.

Algorithm BottleneckTour(G, w)

```
set D[{v}, v] ← w(1, v) for every v ∈ U
for increasing set sizes, each S ⊆ U of size at least two, and v ∈ S do
    D[S, v] ← min_{u∈S\{v}} max{D[S \ {v}, u], w(u, v)}
    store an attaining predecessor
choose v minimizing max{D[U, v], w(v, 1)}
trace its predecessors, prepend 1, and close back to 1; return the tour
```

Theorem 114.1. BottleneckTour *solves* Bottleneck Travelling Salesperson.

Proof. We prove that $D[S, v]$ is the minimum bottleneck of a path from 1 to v visiting exactly $\{1\} \cup S$. We use induction on $|S|$.

Base case. A singleton uses its direct edge, giving the initialized value.

Induction step. A larger path has some penultimate vertex $u \in S \setminus \{v\}$. Its bottleneck is the maximum of its prefix bottleneck and $w(u, v)$. By induction this is at least the displayed candidate for u; an attaining prefix can be extended by the unvisited v to realize that candidate. Minimizing over u proves the claim.

Increasing set size computes dependencies first, and recorded predecessors attain the entries. Every tour is a full path followed by its closing edge. The final maximum includes exactly that edge's contribution, so selecting and recovering a minimum candidate gives an optimum bottleneck tour. □

Theorem 114.2. BottleneckTour *takes* $O(n^2 2^n)$ *time.*

Proof. There are $O(n2^n)$ states and at most n constant-time candidates per state. Subset enumeration and grouping take $O(n2^n)$; matrix preparation takes $O(n^2)$; recovery takes $O(n)$. Hence the total is $O(n^2 2^n)$. □

Problem · p. 24

Chapter 15

Max Flow and Min Cut

115. Flow with Several Sources and Sinks

Set $M = \sum_{e \in E} c(e)$. Keep original edges and their identities in the auxiliary network.

Algorithm SeveralTerminals(G, c, A, B)

```
1 add new vertices s, t to a copy of G
2 add (s, a) of capacity M for every a ∈ A
3 add (b, t) of capacity M for every b ∈ B
4 (f, S) ← EdmondsKarp(H, c_H, s, t)
5 return f restricted to the original edges
```

Theorem 115.1. SeveralTerminals *solves* Flow with Several Sources and Sinks.

Proof. We prove that the original instance has a feasible flow of value F if and only if the auxiliary network does.

($\Rightarrow$) Given a feasible original flow, put each source's net outflow on its new incoming arc and each sink's net inflow on its new outgoing arc. Each amount is at most the sum M of all original capacities. Conservation of total net flow gives a feasible auxiliary flow with the same value.

($\Leftarrow$) Given a feasible auxiliary flow, restrict it to original edges. At a source a, original net outflow equals $f(s, a) \geq 0$; at a sink b, original net inflow equals $f(b, t) \geq 0$. Other vertices conserve flow. The original objective is exactly the auxiliary value.

The claim proves equality of optima. The correctness of EdmondsKarp gives an optimum auxiliary flow, so its restriction is optimal. □

Theorem 115.2. SeveralTerminals *takes* $O(n(n+m)^2)$ *time.*

Proof. The auxiliary graph has $n + 2$ vertices and at most $m + n$ arcs, built in $O(n + m)$ time. Its flow takes $O(n(n+m)^2)$ time. Restriction costs $O(m)$, within this bound. □

Problem · p. 25

116. Flow with Vertex Capacities

For each internal vertex use copies $v_{\text{in}}, v_{\text{out}}$; leave s, t unsplit.

Algorithm VertexCapacitatedFlow(G, c, b, s, t)

```
1 add (v_in, v_out) of capacity b(v) for every internal v
2 replace every original edge (u, v) by (out(u), in(v)) of capacity c(u, v)
3 (f, S) ← EdmondsKarp(H, c_H, s, t)
4 return the flows on the arcs corresponding to original edges
```

Theorem 116.1. VertexCapacitatedFlow *solves* Flow with Vertex Capacities.

Proof. We prove that the original network has a feasible flow of value F if and only if the split network does.

($\Rightarrow$) Given a feasible original flow, put the total inflow at an internal vertex v on its split arc. Conservation makes this also its total outflow, so both copies conserve flow. The vertex bound ensures the split capacity is respected; original edge capacities and the value are unchanged.

($\Leftarrow$) In a feasible split-graph flow, an entrance has only its split arc outgoing, and an exit has only that arc incoming. Conservation makes original inflow and outflow both equal to the split-arc flow, which is at most $b(v)$. Contraction therefore gives a feasible original flow with unchanged value.

The claim gives equality of optima. The correctness of EdmondsKarp then proves the returned restriction optimal. □

Theorem 116.2. VertexCapacitatedFlow *takes* $O(n(n+m)^2)$ *time.*

Proof. Splitting gives at most $2n$ vertices and $m+n$ arcs. Construction and contraction take $O(n+m)$ time. EdmondsKarp takes $O(n(n+m)^2)$ time on this network, which dominates the scans. □

Problem · p. 25

117. Circulation with Lower Bounds

Put $b(v) = \sum_{e \text{ into } v} \ell(e) - \sum_{e \text{ out of } v} \ell(e)$. A positive $b(v)$ requires extra net outflow.

Algorithm BoundedCirculation(G, ℓ, u)

```
give each original arc residual capacity u(e) − ℓ(e); add new s, t
for b(v) > 0, add (s, v) of capacity b(v)
for b(v) < 0, add (v, t) of capacity −b(v)
B ← Σ_{v:b(v)>0} b(v); (g, S) ← EdmondsKarp(H, c_H, s, t)
if |g| < B then return impossible
return f(e) = ℓ(e) + g(e) on original edges
```

Theorem 117.1. BoundedCirculation *solves* Circulation with Lower Bounds.

Proof. We prove that a feasible circulation exists exactly when the auxiliary network has a flow of value B. Each original lower bound contributes once positively and once negatively to the imbalances, so $\sum_v b(v) = 0$.

($\Rightarrow$) For a feasible f, put $g = f - \ell$ on original edges. It satisfies the residual capacities and has exactly the imbalances $b(v)$. Saturating the new arcs restores conservation and gives an auxiliary flow of value B.

($\Leftarrow$) A value-B flow saturates all arcs out of s and into t, since both sets of capacities sum to B. Conservation gives $\text{out}_g(v) - \text{in}_g(v) = b(v)$. Adding the lower bounds cancels this imbalance, and the residual capacities give $\ell(e) \le \ell(e) + g(e) \le u(e)$. Thus $f = \ell + g$ is feasible.

The correctness of EdmondsKarp and this equivalence justify both outputs. □

Theorem 117.2. BoundedCirculation *takes* $O(n(n+m)^2)$ *time.*

Proof. Imbalances and the network are constructed in $O(n+m)$ time. There are $n+2$ vertices and at most $m+n$ arcs. The flow call costs $O(n(n+m)^2)$; recovering f costs $O(m)$. This proves the bound. □

Problem · p. 25

118. One Unit of Additional Capacity

Residual arcs retain their original edge and direction, including parallel arcs.

Algorithm IncreaseCapacity(G, c, f, s, t, e)

```
1 c(e) ← c(e) + 1; construct the residual graph for f under the new capacities
2 search for a simple residual s–t path P
3 if no such path exists then return f
4 add one on each forward original edge of P and subtract one on each backward edge
5 return the updated flow
```

Theorem 118.1. IncreaseCapacity *solves* One Unit of Additional Capacity.

Proof. Let the old optimum be F. We first prove that the new optimum is at most $F + 1$. By the Max-flow/min-cut theorem, an old cut has capacity F. Increasing one capacity raises this cut's capacity by at most one. Every feasible flow is bounded by every cut capacity, proving the claim.

(i) If there is no residual path, the old flow remains feasible. By *A certificate of optimality*, it is still maximum.

(ii) If a residual path is found, integral residual capacities on the path are positive, hence at least one. Sending one unit respects edge bounds, and incoming and outgoing changes cancel at each internal path vertex. The new flow is feasible with value $F + 1$, attaining the claimed upper bound.

□

Theorem 118.2. IncreaseCapacity *takes $O(n + m)$ time.*

Proof. Residual construction and graph search take $O(n + m)$ time. A simple path has at most $n - 1$ arcs, so updating it costs $O(n)$. Including the capacity change and output gives $O(n + m)$. □

Problem · p. 25

119. Checking a Maximum-Flow Claim

Algorithm CheckFlow(G, c, f, s, t)

```
check 0 ≤ f(e) ≤ c(e) for each edge; return a violated edge if found
accumulate net outflow at every vertex
if an internal vertex has nonzero net outflow, return that violation
construct the residual graph and search from s, storing parent arcs
if t is reached, return its recovered augmenting path
return the reached vertex set as a minimum-cut certificate
```

Theorem 119.1. CheckFlow *solves* Checking a Maximum-Flow Claim.

Proof. We prove that each possible certificate justifies the corresponding answer. There are three cases:

(i) Edge bounds and conservation away from s, t are exactly the feasibility conditions. Any violation returned by the first scans is therefore valid; otherwise the assignment is feasible.

(ii) A recovered residual path has positive residual capacity on every arc. The *Augmentation* lemma permits an increase in flow value, proving that the given flow is not maximum.

(iii) If t is unreachable, the reached set S has no outgoing residual arc. Every original edge leaving S is saturated, and every original edge entering it has zero flow. The *Flow across a cut* identity gives $|f| = c(S)$. Since any flow is bounded by any cut, this certifies both a maximum flow and a minimum cut.

□

Theorem 119.2. CheckFlow *takes* $O(n + m)$ *time.*

Proof. The feasibility scans use $O(n + m)$ time. The residual graph has at most $2m$ arcs and is built and searched in $O(n + m)$. Recovering a simple path or listing the reached set takes $O(n)$. Total time is $O(n + m)$. □

Problem · p. 26

120. Is the Minimum Cut Unique?

Algorithm UniqueCut(G, c, f, s, t)

```
1 construct the residual graph H
2 A ← vertices reachable from s in H
3 B ← vertices reachable from t in H^R
4 if A = V \ B then return unique
5 return not unique, with cuts A and V \ B
```

Theorem 120.1. UniqueCut *solves* Is the Minimum Cut Unique?

Proof. We prove that every minimum cut lies between A and $V \setminus B$, and that both of these sets are minimum cuts.

First, we claim that a cut is minimum exactly when it has no outgoing residual arc. The *Flow across a cut* identity expresses $c(S) - |f|$ as unused capacity on edges leaving S plus flow on edges entering S. All terms are nonnegative, so equality holds exactly when no residual arc leaves S. The Max-flow/min-cut theorem proves the subclaim.

There is no residual s–t path, so $A \cap B$ is empty. Reachability closes A under outgoing residual arcs. Also, a residual arc from outside B into B would give its tail a path to t. Thus neither A nor $V \setminus B$ has an outgoing residual arc. Both contain s and exclude t, so both are minimum cuts.

Any minimum cut is closed under residual reachability, so it must contain A. It excludes B, since including a vertex of B would force it to include t. This proves the main claim. If the two extremes agree, they are the unique minimum cut; otherwise they are two distinct minimum cuts, as returned. □

Theorem 120.2. UniqueCut *takes* $O(n + m)$ *time.*

Proof. Residual construction, reversal, and the two searches each cost $O(n + m)$. Comparing or returning the two vertex sets costs $O(n)$. Their sum is $O(n + m)$. □

Problem · p. 26

121. Edge-Disjoint Paths between Terminal Sets

Algorithm TerminalEdgePaths(G, A, B)

```
1 give original edges capacity one and add new s, t
2 add (s, a) for a ∈ A and (b, t) for b ∈ B, each of capacity m
3 (f, S) ← FordFulkerson(H, c_H, s, t)
4 P ← ExtractFlowPaths(H, f, s, t)
5 delete the first and last auxiliary edges of each path; return P
```

Theorem 121.1. TerminalEdgePaths *solves* Edge-Disjoint Paths between Terminal Sets.

Proof. We prove that the maximum auxiliary flow value equals the largest number of required paths. We prove both inequalities by conversions.

(i) Extend k original edge-disjoint paths through the new terminals. Original edges have unit use, and each new arc is used at most $k \le m$ times. The last bound holds because disjoint terminal sets force each path to use an original edge. This gives a feasible value-k flow.

(ii) FordFulkerson returns an integral maximum flow. ExtractFlowPaths returns one path per unit of flow. Removing auxiliary ends leaves paths with the required endpoints. Each original edge is used at most once because its capacity is one.

The conversions prove the claim, and the returned collection attains it. □

Theorem 121.2. TerminalEdgePaths *takes* $O((F+1)(n+m))$ *time.*

Proof. The new network has $n+2$ vertices and at most $m+n$ arcs, and its maximum flow value is exactly F. Flow and extraction each take $O((F+1)(n+m))$. Construction costs $O(n+m)$ and stripping paths costs $O(Fn)$, within the same bound. □

Problem · p. 26

122. Edge-Disjoint Paths to Distinct Destinations

Algorithm DistinctDestinations(G, s, T)

```
1 give original edges capacity one; add a new sink t
2 add (v, t) of capacity one for every v ∈ T
3 (f, S) ← FordFulkerson(H, c_H, s, t)
4 P ← ExtractFlowPaths(H, f, s, t)
5 remove the final edge of each path; return P
```

Theorem 122.1. DistinctDestinations *solves* Edge-Disjoint Paths to Distinct Destinations.

Proof. We prove that feasible path collections and integral auxiliary flows have the same maximum value.

(i) Extend each original path by the final arc from its destination to t. Distinct destinations use distinct final arcs, and original edges are disjoint, so this is a feasible flow of the collection's size.

(ii) ExtractFlowPaths returns as many paths as the integral flow value, possibly leaving cycles. Unit capacities forbid shared original edges and reuse of a final arc. Removing the final arcs therefore gives edge-disjoint paths with distinct destinations.

These conversions prove the claim. The correctness and integrality of FordFulkerson supply a maximum flow, so the output collection is largest. □

Theorem 122.2. DistinctDestinations *takes* $O(n(n+m))$ *time.*

Proof. The auxiliary network has $n+1$ vertices and at most $m+n$ arcs. Its flow value is at most $|T| \le n$. Flow and extraction take $O(n(n+m))$. Construction and stripping the paths fit this bound. □

Problem · p. 26

123. Minimum Directed Edge Separator

Algorithm DirectedEdgeSeparator(G, s, t)

```
1 set c(e) ← 1 for all original edges
2 (f, S) ← FordFulkerson(G, c, s, t)
3 return all original edges from S to V \ S
```

Theorem 123.1. DirectedEdgeSeparator *solves* Minimum Directed Edge Separator.

Proof. The correctness of FordFulkerson gives a minimum cut S. We prove feasibility and optimality of its outgoing edges.

Feasibility. Every s–t path must leave S, so deleting those edges separates the terminals. Their number is $c(S)$ under unit capacities.

Optimality. For any separator R, let U be the vertices reachable from s after deleting R. Then $t \notin U$, and every original edge leaving U belongs to R. Hence $|R| \geq c(U) \geq c(S)$. The returned separator attains this lower bound and is minimum. □

Theorem 123.2. DirectedEdgeSeparator *takes* $O(n(n+m))$ *time.*

Proof. The unit-capacity source cut has at most $n-1$ edges in a simple directed graph, so the maximum flow value is at most $n-1$. FordFulkerson therefore takes $O(n(n+m))$ time. Assigning capacities and listing the cut edges take $O(m)$, within this bound. □

Problem · p. 26

124. Vertex-Disjoint Paths between Terminal Sets

Every vertex v has an entrance v_{in} and exit v_{out}.

Algorithm TerminalVertexPaths(G, A, B)

```
1 add (v_in, v_out) of capacity one for each v
2 replace each original edge (u, v) by (u_out, v_in) of capacity one
3 add new s, t, arcs (s, a_in) for a ∈ A, and (b_out, t) for b ∈ B
4 give these terminal arcs capacity one; (f, S) ← FordFulkerson(H, c_H, s, t)
5 P ← ExtractFlowPaths(H, f, s, t)
6 remove auxiliary endpoints and contract split pairs; return the resulting paths
```

Theorem 124.1. TerminalVertexPaths *solves* Vertex-Disjoint Paths between Terminal Sets.

Proof. We prove that the auxiliary maximum flow value equals the largest feasible vertex-disjoint path count.

(i) Given k feasible paths, expand each through its vertex split arcs and the new terminal arcs. Vertex-disjointness prevents reuse of every split arc, including endpoints, and also prevents reuse of original or terminal arcs. Thus k paths give a feasible integral flow of value k.

(ii) The maximum flow is integral. Its value equals the number of paths returned by ExtractFlowPaths. Every visit to an original vertex uses its unit split arc, so that vertex occurs in at most one path. Removing auxiliary endpoints and contracting split pairs therefore gives a feasible original collection of the same size.

The conversions prove the claim and optimality of the returned collection. □

Theorem 124.2. TerminalVertexPaths *takes* $O(n(n+m))$ *time.*

Proof. The network has $2n + 2$ vertices and $O(n + m)$ arcs, with maximum flow at most $\min\{|A|, |B|\} \leq n$. FordFulkerson and ExtractFlowPaths therefore take $O(n(n+m))$ time. Construction and contraction fit the same bound. □

Problem · p. 27

125. Minimum Internal Vertex Separator

Leave s, t unsplit and put $M = n + 1$.

Algorithm VertexSeparator(G, s, t)

```
1 split each internal vertex with an entrance-to-exit arc of capacity one
2 replace each original edge by its exit-to-entrance arc of capacity M
3 (f, S) ← FordFulkerson(H, c_H, s, t)
4 return Z = {v ≠ s, t : v_in ∈ S, v_out ∉ S}
```

Theorem 125.1. VertexSeparator *solves* Minimum Internal Vertex Separator.

Proof. We first prove that the minimum auxiliary cut has capacity less than M. Removing all $n - 2$ internal split arcs separates the terminals, since there is no direct s–t edge. The remaining reachable set defines a cut of capacity at most $n - 2 < M$. Thus the minimum cut returned by FordFulkerson crosses only unit split arcs, and its capacity is $|Z|$.

Feasibility. An original path avoiding Z would expand into an auxiliary path avoiding every outgoing arc of this cut. Such a path cannot reach t from s, so Z is a separator.

Optimality. For any original separator X, remove its split arcs. The auxiliary terminals become disconnected. The remaining reachable set has outgoing arcs only among the $|X|$ removed unit arcs, so the minimum-cut capacity is at most $|X|$. The returned Z meets this lower bound and is therefore minimum. □

Theorem 125.2. VertexSeparator *takes* $O(n(n + m))$ *time.*

Proof. The split graph has $O(n)$ vertices and $O(n+m)$ arcs. Deleting all internal split arcs gives a cut of capacity at most $n-2$, bounding the maximum flow. Thus FordFulkerson takes $O(n(n + m))$ time. All other scans take $O(n + m)$. □

Problem · p. 27

126. Edge-Disjoint Paths with Vertex Limits

Leave s, t unsplit.

Algorithm LimitedVertexPaths(G, b, s, t)

```
1 split every internal v with an entrance-to-exit arc of capacity b(v)
2 replace every original edge by an exit-to-entrance arc of capacity one
3 (f, S) ← FordFulkerson(H, c_H, s, t)
4 P ← ExtractFlowPaths(H, f, s, t)
5 contract each split pair in these paths; return the original paths
```

Theorem 126.1. LimitedVertexPaths *solves* Edge-Disjoint Paths with Vertex Limits.

Proof. We prove that feasible original path collections and integral auxiliary flows have the same maximum value.

(i) Given a feasible path collection, expand each path through split vertices. Edge-disjointness respects unit original-edge capacities, and the number of visits to v is at most $b(v)$, respecting its split capacity. The resulting integral flow has value equal to the path count.

(ii) ExtractFlowPaths returns as many paths as the integral flow value. Original unit arcs are used at most once, and the split arc of v is used at most $b(v)$ times. Contracting split pairs therefore gives paths satisfying exactly the required edge and vertex limits.

The correctness and integrality of FordFulkerson supply a maximum flow. The claim shows that the recovered collection is optimal. □

Theorem 126.2. LimitedVertexPaths *takes* $O(n(n+m))$ *time.*

Proof. There are $O(n)$ auxiliary vertices and $O(n+m)$ arcs. The source remains unsplit with at most $n-1$ outgoing unit arcs, so the maximum flow is at most $n-1$. Flow and extraction cost $O(n(n+m))$. Construction and contraction fit that bound. □

Problem · p. 27

127. Edge Connectivity of an Undirected Graph

Algorithm GlobalEdgeCut(G)

```
choose r ∈ V; replace every edge by two opposite unit-capacity arcs, obtaining H
for each v ≠ r do
    (f_v, S_v) ← FordFulkerson(H, 1, r, v)
choose v minimizing the number of original edges crossing S_v
return those crossing original edges
```

Theorem 127.1. GlobalEdgeCut *solves* Edge Connectivity of an Undirected Graph.

Proof. For each vertex set, exactly one arc of each crossing undirected edge leaves it. Thus its auxiliary cut capacity equals its original crossing-edge count. The correctness of FordFulkerson makes each S_v a smallest cut separating r and v.

Lower bound. Any successful deletion set leaves some v outside the component of r. Every original edge leaving that component was deleted, so the deletion set is at least as large as the minimum r–v cut. The least computed cut is therefore no larger than any successful deletion set.

Attainment. Deleting the original edges crossing a chosen S_v disconnects r from v. The returned set is feasible and attains the lower bound, proving optimality. □

Theorem 127.2. GlobalEdgeCut *takes* $O(n^2(n+m))$ *time.*

Proof. The bidirected network has n vertices and $2m$ arcs, with flow value at most $\deg(r) \le n-1$. Each call takes $O(n(n+m))$. There are $n-1$ calls. Construction and scanning all cuts take $O(n(n+m))$ additional time, so the total is $O(n^2(n+m))$. □

Problem · p. 27

128. Undirected Paths with Edge-Use Capacities

Multiplicities encode repeated identical paths without listing them individually.

Algorithm CapacitatedPaths(G, c, s, t)

```
1 replace each edge by opposite arcs, both of capacity c(e); obtain H
2 (g, S) ← EdmondsKarp(H, c_H, s, t); cancel opposing flows on each edge
3 F ← |g|; P ← [ ]
4 while F > 0 do
5     find a simple s–t path P in the positive-g arcs
6     a ← min{F, min_{e∈P} g(e)}; append (P, a) to P
7     subtract a from the flow on P; F ← F − a
8 return P with directions forgotten
```

Theorem 128.1. CapacitatedPaths *solves* Undirected Paths with Edge-Use Capacities.

Proof. We prove that the extracted paths attain the auxiliary maximum flow value while respecting every edge-use capacity. Orienting any feasible path multiset towards t gives a flow of the same total multiplicity, so attaining this upper bound proves optimality. EdmondsKarp gives an integral maximum flow. Cancelling opposing flow preserves its value and conservation, leaving at most one used direction per edge.

We claim that the remaining flow is feasible with value F, extracted multiplicities plus F equal the initial optimum, and extracted use of each arc plus its remaining flow does not exceed its initial flow. We use induction on the number of extractions.

Base case. No paths have been extracted, so the claim holds.

Induction step. When $F > 0$, the *A positive-flow path* lemma supplies P. Subtracting a preserves nonnegativity and conservation, reduces the value by a, and adds that same amount to the extracted multiplicities and arc uses. Thus all parts of the claim are preserved.

Every nonfinal extraction makes an arc zero, and no arc becomes positive again. The final extraction uses a positive arc, so there are at most m extractions. At $F = 0$, the claim gives a feasible representation of the optimum with at most m path-multiplicity pairs. □

Theorem 128.2. CapacitatedPaths *takes* $O(n + m + nm^2)$ *time.*

Proof. The network has n vertices and $2m$ arcs, so EdmondsKarp costs $O(n+m+nm^2)$. Construction and cancellation cost $O(n+m)$. At most m extractions each cost $O(n+m)$; for $m \geq 1$ this is within $O(nm^2)$, and for $m = 0$ there are none. The bound follows without expanding multiplicities. □

Problem · p. 27

129. Two Edge-Disjoint Routes or a Small Separator

Algorithm TwoRoutes(G, s, t)

```
form the bidirected unit-capacity network H; initialize zero flow f
repeat twice
    build the residual graph and search from s
    if t is unreachable, return the original edges crossing the reached set
    augment one unit along a recovered simple residual path
cancel opposite flows; extract two paths using ExtractFlowPaths
return those paths with directions forgotten
```

Theorem 129.1. TwoRoutes *solves* Two Edge-Disjoint Routes or a Small Separator.

Proof. The *Augmentation* lemma preserves integral feasibility and increases the value by one. We distinguish the two outputs.

(i) If a search fails, fewer than two units have been sent. The reached set has no outgoing residual arc, so the *Flow across a cut* identity makes its capacity zero or one. In the bidirected unit network, exactly one arc of each crossing undirected edge leaves the set. The returned edges therefore separate the terminals and have the promised size.

(ii) If both searches succeed, the flow has value two. Cancelling equal opposing flow preserves feasibility and value. The correctness of ExtractFlowPaths ensures that it returns two paths. Unit capacities prevent reuse of an arc, and cancellation prevents opposite use of an original edge. Forgetting directions gives two edge-disjoint undirected paths.

□

Theorem 129.2. TwoRoutes *takes* $O(n + m)$ *time.*

Proof. The bidirected and residual graphs have $O(n)$ vertices and $O(m)$ arcs. Only two searches and augmentations occur. Cancellation and extracting two paths take $O(n + m)$. All construction and output work has the same bound, giving $O(n + m)$. □

Problem · p. 28

130. Bipartite Matching with Vertex Quotas

Algorithm QuotaMatching(G, X, Y, b)

```
1 add a source s and sink t
2 add (s, x) of capacity b(x) for x ∈ X and (y, t) of capacity b(y) for y ∈ Y
3 orient each original edge from X to Y with capacity one
4 (f, S) ← EdmondsKarp(H, c_H, s, t)
5 return the original edges whose middle arcs carry one unit
```

Theorem 130.1. QuotaMatching *solves* Bipartite Matching with Vertex Quotas.

Proof. We prove that the maximum feasible edge-set size equals the maximum auxiliary flow value.

(i) Given a feasible edge set, send one unit through s, x, y, t for each chosen edge xy. Endpoint quotas enforce outer capacities, and distinct chosen edges respect unit middle capacities. The flow value is the set size.

(ii) Integral middle flows are zero or one. Conservation at x and y bounds the number of selected incident edges by $b(x)$ and $b(y)$. The selected edges are feasible and their number is the flow value.

These conversions prove the claim. The correctness and integrality of EdmondsKarp supply a maximum flow, so the returned edge set is optimal. □

Theorem 130.2. QuotaMatching *takes* $O(n(n+m)^2)$ *time.*

Proof. The network has $n + 2$ vertices and $n + m$ arcs, built in $O(n + m)$ time. EdmondsKarp takes $O(n(n + m)^2)$ time. Reading off the chosen edges costs $O(m)$, within that bound. □

Problem · p. 28

131. Bipartite Matching with Required Edges

Algorithm RequiredMatching(G, X, Y, R)

```
1 if two edges of R share an endpoint, return impossible
2 mark all endpoints of R and build the graph H on the unmarked vertices
3 M ← BipartiteMatching(H, X ∩ V(H), Y ∩ V(H))
4 return R ∪ M, restoring original vertex labels
```

Theorem 131.1. RequiredMatching *solves* Bipartite Matching with Required Edges.

Proof. If two required edges share an endpoint, no matching can contain them. Otherwise R is a matching. We prove that an edge set $J \supseteq R$ is a matching of G if and only if $J \setminus R$ is a matching of H. The sizes then differ by $|R|$.

($\Rightarrow$) In a matching containing R, every additional edge avoids all endpoints of R, so deleting R leaves a matching in H.

($\Leftarrow$) A matching in H has no endpoint incident to R, so its union with R is a matching.

The correctness of BipartiteMatching gives a largest residual matching, including the empty matching when appropriate. The claim makes the returned union a largest matching containing R. □

Theorem 131.2. RequiredMatching *takes* $O(n(n+m))$ *time.*

Proof. Marking endpoints and constructing H take $O(n+m)$ time. On $n' \le n$ vertices and $m' \le m$ edges, BipartiteMatching takes $O(n'(n'+m')) \subseteq O(n(n+m))$ time. Restoring labels and appending R take $O(n+m)$, proving the bound. □

Problem · p. 28

132. Matching Covering Specified Left Vertices

The full network has unit-capacity arcs $s \to X$, $X \to Y$ for original edges, and $Y \to t$. Residual paths are simple.

Algorithm PRIORITYMATCHING(G, X, Y, Q)

```
1 compute M_0 ← BIPARTITEMATCHING(G[Q ∪ Y], Q, Y)
2 if |M_0| < |Q| then return impossible
3 build the full matching network and initialize its flow to the unit paths for M_0
4 while a residual s–t path exists do augment one unit
5 return the original edges with middle-arc flow one
```

Theorem 132.1. PRIORITYMATCHING *solves* Matching Covering Specified Left Vertices.

Proof. A matching covering Q restricts to a size-$|Q|$ matching in $G[Q \cup Y]$, and any such matching covers Q. The correctness of BIPARTITEMATCHING therefore justifies the initial failure test. We prove that the current flow remains integral and feasible, with every source arc (s, q) for $q \in Q$ saturated. We use induction on the number of augmentations.

Base case. The unit paths for M_0 give a feasible integral flow and saturate every required source arc, since M_0 covers Q.

Induction step. The *Augmentation* lemma preserves integral feasibility. Reducing a saturated (s, q) would require its reverse residual arc and a revisit to s, impossible on a simple augmenting path. Thus the required source arcs stay saturated.

With no residual path, *A certificate of optimality* gives maximum flow. Unit outer capacities make the integral middle flows a matching of the same size, and the claim ensures it covers Q. It is globally maximum, so it is also maximum under the requirement. Each augmentation increases value by one, and the unit source capacities bound it by $|X|$. □

Theorem 132.2. PRIORITYMATCHING *takes* $O(n(n+m))$ *time.*

Proof. The initial matching takes $O(n(n+m))$. The full network has $n+2$ vertices and $n+m$ arcs. At most n augmentations and a final search each cost $O(n+m)$. Construction and output are linear in that network size, giving $O(n(n+m))$. □

Problem · p. 28

133. Minimum-Weight Bipartite Vertex Cover

Set $M = 1 + \sum_{v \in V} c(v)$.

Algorithm WeightedBipartiteCover(G, X, Y, c)

```
1 add new s, t, arcs (s, x) of capacity c(x), and (y, t) of capacity c(y)
2 for each original edge xy, add (x, y) of capacity M
3 (f, S) ← EdmondsKarp(H, c_H, s, t)
4 return (X \ S) ∪ (Y ∩ S)
```

Theorem 133.1. WeightedBipartiteCover *solves* Minimum-Weight Bipartite Vertex Cover.

Proof. The source-only cut has capacity at most $\sum_v c(v) < M$, so a minimum cut crosses no middle arc. We prove that C is a vertex cover if and only if its cut $S = \{s\} \cup (X \setminus C) \cup (Y \cap C)$ crosses no middle arc.

($\Rightarrow$) For any cover C, take $S = \{s\} \cup (X \setminus C) \cup (Y \cap C)$. A crossing middle arc would be an uncovered edge, so none exists. The cut capacity is exactly the cost of C.

($\Leftarrow$) If S crosses no middle arc, no edge has $x \in S$ and $y \notin S$. Thus $C = (X \setminus S) \cup (Y \cap S)$ covers every edge. Its vertices correspond exactly to crossing outer arcs, so its cost is the cut capacity.

The minimum values are therefore equal. EdmondsKarp returns a minimum cut, so the resulting cover is optimal. □

Theorem 133.2. WeightedBipartiteCover *takes* $O(n(n+m)^2)$ *time.*

Proof. The network has $n + 2$ vertices and $n + m$ arcs. Construction and recovery take $O(n + m)$ time; EdmondsKarp takes $O(n(n+m)^2)$ time. Their sum gives the stated bound. □

Problem · p. 28

134. Maximum-Weight Bipartite Independent Set

Write $W = \sum_v w(v)$ and $M = W + 1$.

Algorithm BipartiteIndependentWeight(G, X, Y, w)

```
1 add new s, t, arcs (s, x) of capacity w(x), and (y, t) of capacity w(y)
2 for each original edge xy, add (x, y) of capacity M
3 (f, S) ← EdmondsKarp(H, c_H, s, t)
4 return I = (X ∩ S) ∪ (Y \ S)
```

Theorem 134.1. BipartiteIndependentWeight *solves* Maximum-Weight Bipartite Independent Set.

Proof. The source-only cut has capacity at most $W < M$, so a minimum cut crosses no middle arc. We prove that I is independent if and only if its cut $S = \{s\} \cup (X \cap I) \cup (Y \setminus I)$ crosses no middle arc.

($\Rightarrow$) For independent I, a crossing middle arc would join two vertices of I, so none exists. This cut has capacity $W - w(I)$.

($\Leftarrow$) Suppose S crosses no middle arc. If both endpoints of an edge belonged to $I = (X \cap S) \cup (Y \setminus S)$, its middle arc would cross the cut. Thus I is independent. Excluded vertices correspond exactly to crossing outer arcs, giving $w(I) = W - c_H(S)$.

A heavier independent set would therefore give a smaller cut. The correctness of EdmondsKarp supplies a minimum cut, making the returned set optimal. □

Theorem 134.2. BipartiteIndependentWeight *takes* $O(n(n+m)^2)$ *time.*

Proof. Construction creates $n+2$ vertices and $n+m$ arcs in $O(n+m)$ time. EdmondsKarp takes $O(n(n+m)^2)$ time, and reading off I costs $O(n)$. The total is $O(n(n+m)^2)$. □

Problem · p. 28

135. Vertex Cover with Required Vertices

Algorithm RequiredCover(G, X, Y, R)

```
1 build H = G[V \ R], retaining original vertex labels
2 C ← BipartiteVertexCover(H, X \ R, Y \ R)
3 return R ∪ C with original labels
```

Theorem 135.1. RequiredCover *solves* Vertex Cover with Required Vertices.

Proof. For $C \supseteq R$, we prove that C is a cover of G if and only if $C \setminus R$ is a cover of H. The sizes then differ by $|R|$.

($\Rightarrow$) Edges not incident to R are precisely the edges of H. The remaining vertices of any original cover must cover them.

($\Leftarrow$) A cover of H united with R covers all edges: R covers its incident edges, and the residual cover handles the others.

The correctness of BipartiteVertexCover gives a smallest cover of H, using the empty cover if H is empty. The fixed additional size $|R|$ and the claim show that the returned union is optimal. □

Theorem 135.2. RequiredCover *takes* $O(n(n+m))$ *time.*

Proof. BipartiteVertexCover takes $O(n'(n'+m'))$ time on n' vertices and m' edges. Since $n' \le n$ and $m' \le m$, this is $O(n(n+m))$. Construction and label mapping take $O(n+m)$ time; forming the union adds $O(n)$. This proves the bound. □

Problem · p. 29

136. Possible First Place with Ties Allowed

Algorithm TiedWinner($w[1..n], g[1..n, 1..n], x$)
1 copy w into w'; $w'_x \leftarrow w_x + 1$
2 **return** PossibleWinner(w', g, x)

Theorem 136.1. TiedWinner *solves* Possible First Place with Ties Allowed.

Proof. We prove that exactly the same allocations make x weakly first in the original instance and strictly first in the modified instance. Fix any allocation with original final totals W_i. Only the designated team's total changes, to $W_x + 1$. By integrality,

$$W_i \leq W_x \quad \Longleftrightarrow \quad W_i < W_x + 1.$$

Applying this to every other team proves the claim directly.

The correctness of PossibleWinner therefore decides and finds exactly the required allocations. Its allocation needs no change on return, because no actual remaining game was added. □

Theorem 136.2. TiedWinner *takes* $O(n^6)$ *time.*

Proof. Copying wins costs $O(n)$. The call to PossibleWinner takes $O(n^6)$; the game matrix is unchanged. Hence the total is $O(n^6)$. □

Problem · p. 29

137. League Winner with Prescribed Results

Algorithm PRESCRIBEDWINNER($w[1..n], g[1..n, 1..n], h[1..n, 1..n], x$)

```
1 w'_i ← w_i + Σ_j h_ij for every team i
2 g'_ij ← g_ij − h_ij − h_ji for every pair
3 call POSSIBLEWINNER(w', g', x)
4 if it reports impossibility, return impossible
5 if its allocation is a, return the allocation h + a
```

Theorem 137.1. PRESCRIBEDWINNER *solves* League Winner with Prescribed Results.

Proof. We prove that an allocation obeys the original counts and prescriptions if and only if it consists of the fixed wins h plus a residual allocation.

($\Rightarrow$) Removing the prescribed wins h from a permitted allocation leaves exactly the remaining counts g'.

($\Leftarrow$) Adding h to a residual allocation satisfies exactly the original counts and prescriptions.

These inverse operations give a bijection. For every team,

$$w_i + \sum_j (h_{ij} + a_{ij}) = w'_i + \sum_j a_{ij},$$

so final totals, and hence strict first place, are preserved.

The correctness of POSSIBLEWINNER gives a residual winning allocation exactly when one exists. The bijection justifies the failure report or the returned allocation $h + a$. □

Theorem 137.2. PRESCRIBEDWINNER *takes* $O(n^6)$ *time.*

Proof. Computing w', g' and adding allocations cost $O(n^2)$ time. POSSIBLEWINNER takes $O(n^6)$. Counts remain compressed rather than expanded into individual games, giving total time $O(n^6)$. □

Problem · p. 29

138. League Completion under Win Caps

For each pair $i < j$ with $g_{ij} > 0$, use a game vertex p_{ij}.

Algorithm CappedLeague$(w[1..n], g[1..n, 1..n], U[1..n])$

```
1 b_i ← U_i − w_i; if any b_i < 0, return impossible
2 add (s, p_ij) of capacity g_ij for each nonempty game pair
3 add (p_ij, i) and (p_ij, j), each of capacity g_ij
4 add (i, t) of capacity b_i for every team; R ← Σ_{i<j} g_ij
5 (f, S) ← EdmondsKarp(H, c_H, s, t)
6 if |f| < R, return impossible
7 return the game-to-team flows as allocated wins, using zero for absent pairs
```

Theorem 138.1. CappedLeague *solves* League Completion under Win Caps.

Proof. A negative budget already violates a cap. Otherwise we prove that a feasible allocation exists exactly when the auxiliary network has an integral flow of value R.

($\Rightarrow$) Given a feasible allocation, route each game's win through its pair vertex to the winning team and then to t. All source arcs are saturated, giving value R. The team capacities hold because final wins do not exceed the caps.

($\Leftarrow$) A value-R flow saturates every source arc. Conservation and integrality divide each g_{ij} into whole-number wins for its two teams. Team arcs enforce the budgets, so these wins respect every cap.

The correctness and integrality of EdmondsKarp supply a maximum integral flow. The claim makes both the test and recovery correct. □

Theorem 138.2. CappedLeague *takes* $O(n^6)$ *time.*

Proof. There are $O(n^2)$ auxiliary vertices and arcs. Construction and recovery take $O(n^2)$ time. EdmondsKarp takes $O(n^2 + n^2(n^2)^2) = O(n^6)$ time. Game counts stay compressed, so this bound is independent of their sum. □

Problem · p. 29

139. Projects with Prerequisites

Set $P = \sum_{v:w(v)>0} w(v)$ and $M = P + 1$.

Algorithm PrerequisiteProjects(G, w)

```
1 add new source s and sink t
2 add (s, v) of capacity w(v) when w(v) > 0
3 add (v, t) of capacity −w(v) when w(v) < 0
4 give every prerequisite arc capacity M
5 (f, S) ← EdmondsKarp(H, c_H, s, t)
6 return S ∩ V
```

Theorem 139.1. PrerequisiteProjects *solves* Projects with Prerequisites.

Proof. We prove that minimum cuts yield prerequisite-closed sets, and their capacity is P minus selected profit. The cut $\{s\}$ has capacity $P < M$. Thus a minimum cut crosses no prerequisite arc, making its project set closed under every requirement, including chains and cycles.

Conversely, every closed set A gives the cut $\{s\} \cup A$ with no crossing prerequisite arc. Its capacity is

$$\sum_{v \notin A,\, w(v)>0} w(v) + \sum_{v \in A,\, w(v)<0} (-w(v)) = P - \sum_{v \in A} w(v).$$

This proves the claim and the converse construction. Since P is fixed, minimizing cut capacity maximizes feasible profit. The correctness of EdmondsKarp therefore proves the returned selection optimal. □

Theorem 139.2. PrerequisiteProjects *takes* $O(n(n+m)^2)$ *time.*

Proof. The network has $n + 2$ vertices and at most $n + m$ arcs, built in $O(n + m)$ time. EdmondsKarp takes $O(n(n + m)^2)$ time. Reading the selected projects costs $O(n)$, within that bound. □

Problem · p. 29

140. Required and Forbidden Projects

Put $P = \sum_{w(v)>0} w(v)$ and $M = 1 + \sum_v |w(v)|$. Keep parallel auxiliary arcs separate.

Algorithm ConstrainedProjects(G, w, R, F)

```
1 add new s, t; add (s, v) of capacity w(v) for positive profits
2 add (v, t) of capacity −w(v) for negative profits
3 give each prerequisite arc capacity M
4 add (s, r) of capacity M for r ∈ R and (v, t) of capacity M for v ∈ F
5 (f, S) ← EdmondsKarp(H, c_H, s, t)
6 if c_H(S) ≥ M, return impossible
7 return S ∩ V
```

Theorem 140.1. ConstrainedProjects *solves* Required and Forbidden Projects.

Proof. We prove that a permitted selection exists exactly when the minimum cut has capacity below M, and that a minimum cut then gives maximum profit.

Feasibility. We prove both directions.

($\Rightarrow$) A permitted project set is prerequisite-closed, contains R, and avoids F. Its cut crosses no capacity-M arc, and its ordinary-arc capacity is at most $\sum_v |w(v)| < M$. The minimum cut therefore also has capacity below M.

($\Leftarrow$) A cut of capacity below M crosses no forcing or prerequisite arc. Its project set is therefore prerequisite-closed, contains R, and avoids F, giving a permitted selection.

Optimality. For a feasible project set A, the crossing arcs charge unchosen positive profits and chosen negative profits, giving $c_H(\{s\} \cup A) = P - \sum_{v \in A} w(v)$. Since P is fixed, minimizing cut capacity maximizes profit.

The correctness of EdmondsKarp supplies a minimum cut. The first part justifies the infeasibility test, and the second proves that a returned feasible selection is optimal. □

Theorem 140.2. ConstrainedProjects *takes* $O(n(n+m)^2)$ *time.*

Proof. The network has $n + 2$ vertices and $O(n + m)$ arcs, including the forcing arcs. Construction and cut-capacity computation cost $O(n + m)$ time. EdmondsKarp takes $O(n(n + m)^2)$ time; extraction adds $O(n)$. The total is $O(n(n + m)^2)$. □

Problem · p. 30

141. Projects with Penalties for Missing Prerequisites

Write $P = \sum_{v:w(v)>0} w(v)$.

Algorithm PenalizedProjects(G, w, p)

```
1 add a source s and sink t
2 add (s, v) of capacity w(v) for positive profits
3 add (v, t) of capacity −w(v) for negative profits
4 give each original arc (u, v) capacity p(u, v)
5 (f, S) ← EdmondsKarp(H, c_H, s, t)
6 return S ∩ V
```

Theorem 141.1. PenalizedProjects *solves* Projects with Penalties for Missing Prerequisites.

Proof. We prove that every project set A corresponds to a cut whose capacity is P minus its net profit. Its unique corresponding cut is $S = \{s\} \cup A$. Source and sink arcs contribute $P - \sum_{v \in A} w(v)$. A project arc crosses exactly when its tail is chosen and its head is not, contributing exactly that missing-prerequisite penalty. Hence

$$c_H(S) = P - \left(\sum_{v \in A} w(v) - \sum_{u \in A,\, v \notin A} p(u, v) \right).$$

This direct calculation proves the claim. Conversely every cut determines exactly one project set, and P is independent of the choice. Thus minimizing cut capacity is equivalent to maximizing net profit. The correctness of EdmondsKarp proves that the returned set is optimal. □

Theorem 141.2. PenalizedProjects *takes* $O(n(n+m)^2)$ *time.*

Proof. The network has $n + 2$ vertices and at most $n + m$ arcs. Building it and extracting the project set take $O(n + m)$ time. The call takes $O(n(n+m)^2)$, which also bounds the remaining work. □

Problem · p. 30

List of Problems

Each row gives the pages of the problem statement, hint, and solution. Click a title or page number to go directly to it.

Problem		Page	Hint	Solution
1.	Stable Sorting	2	32	55
2.	Merging Several Arrays	2	32	56
3.	Multiset Intersection	2	32	57
4.	Significant Inversions	2	32	58
5.	Weighted Inversions	2	32	59
6.	Disagreement between Rankings	3	33	60
7.	Circular Maximum Subarray	3	33	61
8.	Two Disjoint Maximum Subarrays	3	33	62
9.	Maximum Submatrix	3	33	63
10.	Selection with Repetitions	3	33	64
11.	Multiple Selection	4	33	65
12.	Weighted Median	4	34	66
13.	Closest Pair with Equal Coordinates	4	34	67
14.	Bichromatic Closest Pair on Parallel Lines	4	34	68
15.	Manhattan Closest Pair	4	34	69
16.	Polynomial Multiplication	5	34	70
17.	Truncated Integer Multiplication	5	34	71
18.	Signed Integer Multiplication	5	35	72
19.	Rectangular Matrix Multiplication	5	35	73
20.	Matrix Powers	5	35	74
21.	Boolean Matrix Multiplication	6	35	75
22.	Maximum-Weight Independent Set	6	35	76
23.	Counting Independent Sets	6	35	77
24.	Minimum Vertex Cover	6	35	78
25.	Reachability from a Set	7	36	79
26.	Reachability after Deletions	7	36	80
27.	Even-Length Reachability	7	36	81
28.	Multiple-Source Shortest Paths	7	36	82
29.	Counting Shortest Paths	8	36	83
30.	Edges on Shortest Paths	8	36	84
31.	Component Weights	8	37	85
32.	Connecting a Graph	8	37	86
33.	Clique Components	8	37	87

Problem		Page	Hint	Solution
34.	Extending a Partial Two-Colouring	9	37	88
35.	Counting Two-Colourings	9	37	89
36.	Parity Constraints	9	37	90
37.	Path Recovery	9	37	91
38.	Subtree Sizes	9	38	92
39.	Ancestor Queries	9	38	93
40.	Components after Vertex Deletion	10	38	94
41.	Cut Vertices in a Disconnected Graph	10	38	95
42.	Separated Pairs after Vertex Deletion	10	38	96
43.	Edges on Cycles	10	38	97
44.	Components after Removing All Cut Edges	10	38	98
45.	Unavoidable Edges	10	38	99
46.	Directed Path Recovery	11	39	100
47.	Avoiding Specified Vertices	11	39	101
48.	Shortest Directed Path	11	39	102
49.	Strong Connectivity with a Witness	11	39	103
50.	Strongly Connected Component of a Vertex	11	39	104
51.	A Sparse Strongly Connected Subgraph	11	39	105
52.	Directed Cycle Witness	12	39	106
53.	Acyclicity after One Edge Insertion	12	39	107
54.	Longest Directed Path in a DAG	12	40	108
55.	Condensation Graph	12	40	109
56.	Universal Source Vertices	12	40	110
57.	Minimum Starting Set	12	40	111
58.	Scheduling with Turnaround Time	13	41	112
59.	Scheduling with a Required Interval	13	41	113
60.	Stabbing Intervals	13	41	114
61.	Coloring Half-Open Intervals	13	41	115
62.	Maximum Simultaneous Intervals	14	41	116
63.	Interval Coloring with Demands	14	41	117
64.	Weighted Total Completion Time	14	42	118
65.	Minimum Total Waiting Time	14	42	119
66.	Maximum Lateness	14	42	120
67.	Nearest Facility	15	42	121
68.	One Free Edge	15	42	122
69.	Vertex Tolls	15	42	123
70.	Minimum Spanning Forest	15	42	124
71.	MST after One Edge Insertion	15	42	125
72.	Minimum Bottleneck Spanning Tree	16	43	126
73.	Maximum Spanning Tree	16	43	127
74.	MST with Required Edges	16	43	128

Problem		Page	Hint	Solution
75.	Threshold Connectivity Queries	16	43	129
76.	Scheduling at Most k Intervals	17	44	130
77.	Weighted Scheduling with a Required Interval	17	44	131
78.	Weighted Scheduling with Turnaround Time	17	44	132
79.	Equal-Sum Partition	17	44	133
80.	Most Balanced Partition	18	44	134
81.	Subset Sum with Fewest Items	18	44	135
82.	Unbounded Knapsack	18	45	136
83.	Exact-Weight Knapsack	18	45	137
84.	Knapsack with Two Resource Limits	18	45	138
85.	Maximum-Weight Increasing Subsequence	19	45	139
86.	Longest Bitonic Subsequence	19	45	140
87.	Counting Longest Increasing Subsequences	19	45	141
88.	Longest Nondecreasing Subsequence	19	46	142
89.	Fewest Deletions to Make an Array Increasing	19	46	143
90.	Nesting Rectangles	20	46	144
91.	Longest Palindromic Subsequence	20	46	145
92.	Shortest Common Supersequence	20	46	146
93.	Common Subsequence of Three Strings	20	46	147
94.	Edit Distance with Unequal Costs	20	47	148
95.	Closest Substring by Edit Distance	21	47	149
96.	Editing with Insertions and Deletions	21	47	150
97.	Maximum-Weight Independent Set on a Tree	21	47	151
98.	Minimum-Weight Vertex Cover on a Tree	21	47	152
99.	Counting Independent Sets of a Tree	21	47	153
100.	Height-Bounded Optimal Search Tree	22	48	154
101.	Optimal Alphabetic Tree	22	48	155
102.	Search Trees with Unsuccessful Queries	22	48	156
103.	Cheapest Walk with an Edge Limit	22	48	157
104.	Shortest Paths in a Weighted DAG	22	48	158
105.	Distances to a Destination	23	48	159
106.	A Reachable Negative Cycle	23	48	160
107.	Vertices with Unboundedly Small Walk Weights	23	48	161
108.	A Cycle below a Mean-Weight Threshold	23	49	162
109.	Transitive Closure	23	49	163
110.	Minimum-Bottleneck Paths	24	49	164
111.	Minimum-Weight Directed Cycle	24	49	165
112.	Minimum-Weight Hamiltonian Path	24	49	166
113.	Travelling Salesperson with Precedence Constraints	24	49	167
114.	Bottleneck Travelling Salesperson	24	49	168
115.	Flow with Several Sources and Sinks	25	50	169

Problem	Page	Hint	Solution
116. Flow with Vertex Capacities	25	50	170
117. Circulation with Lower Bounds	25	50	171
118. One Unit of Additional Capacity	25	50	172
119. Checking a Maximum-Flow Claim	26	50	173
120. Is the Minimum Cut Unique?	26	50	174
121. Edge-Disjoint Paths between Terminal Sets	26	51	175
122. Edge-Disjoint Paths to Distinct Destinations	26	51	176
123. Minimum Directed Edge Separator	26	51	177
124. Vertex-Disjoint Paths between Terminal Sets	27	51	178
125. Minimum Internal Vertex Separator	27	51	179
126. Edge-Disjoint Paths with Vertex Limits	27	51	180
127. Edge Connectivity of an Undirected Graph	27	51	181
128. Undirected Paths with Edge-Use Capacities	27	51	182
129. Two Edge-Disjoint Routes or a Small Separator	28	52	183
130. Bipartite Matching with Vertex Quotas	28	52	184
131. Bipartite Matching with Required Edges	28	52	185
132. Matching Covering Specified Left Vertices	28	52	186
133. Minimum-Weight Bipartite Vertex Cover	28	52	187
134. Maximum-Weight Bipartite Independent Set	28	52	188
135. Vertex Cover with Required Vertices	29	52	189
136. Possible First Place with Ties Allowed	29	53	190
137. League Winner with Prescribed Results	29	53	191
138. League Completion under Win Caps	29	53	192
139. Projects with Prerequisites	29	53	193
140. Required and Forbidden Projects	30	53	194
141. Projects with Penalties for Missing Prerequisites	30	53	195

Index

Page numbers link to problem statements. Each statement links to its hint and solution. Related variants are grouped by subject.

A

Ancestor queries, 9

Articulation vertices. *See* Cut vertices

B

Baseball elimination. *See* League completion

BFS. *See* Reachability; Shortest paths

Bipartite matching
- Required edges, 28
- Specified left vertices, 28
- Vertex quotas, 28

Bipartiteness
- Counting two-colourings, 9
- Parity constraints, 9
- Partial colourings, 9

Bottleneck optimization
- Paths, 24
- Spanning trees, 16
- Travelling salesperson, 24

Breadth-first search. *See* Reachability; Shortest paths

Bridges. *See* Cut edges

C

Circulations
- Lower bounds, 25

Closest pairs
- Equal coordinates, 4
- Manhattan distance, 4
- Parallel lines, bichromatic, 4

Colouring. *See* Bipartiteness; Intervals

Common subsequences
- Palindromic subsequences, 20
- Shortest common supersequence, 20
- Three strings, 20

Connected components
- Clique components, 8
- Connecting a graph, 8
- Vertex weights, 8

Counting
- Independent sets, 6
- Independent sets of a tree, 21
- Longest increasing subsequences, 19
- Shortest paths, 8
- Two-colourings, 9

Cut edges
- Components after removal, 10
- Edges on cycles, 10
- Unavoidable edges, 10

Cut vertices
- Component counts after deletion, 10
- Disconnected graphs, 10
- Separated pairs, 10

Cycles
- Directed cycle witness, 12
- Edges on cycles, 10
- Mean-weight threshold, 23
- Minimum-weight directed cycle, 24
- Reachable negative cycle, 23

D

DAGs. *See* Directed acyclic graphs

Depth-first search
- Ancestor queries, 9
- Path recovery, 9
- Subtree sizes, 9

DFS. *See* Depth-first search

Directed acyclic graphs
- Edge insertion, 12
- Longest paths, 12
- Shortest weighted paths, 22

Directed reachability
- Avoiding specified vertices, 11
- Path recovery, 11
- Shortest paths, 11
- Transitive closure, 23

Universal source vertices, 12

Disjoint paths
Distinct destinations, 26
Edge-use capacities, 27
Terminal sets, edge-disjoint, 26
Terminal sets, vertex-disjoint, 27
Two undirected routes, 28
Vertex-use limits, 27

E

Edit distance
Closest substring, 21
Insertions and deletions only, 21
Unequal operation costs, 20

H

Hamiltonian paths. *See* Travelling salesperson

I

Independent sets
Counting, 6
Counting on trees, 21
Maximum weight, 6
Maximum weight in bipartite graphs, 28
Maximum weight on trees, 21

Integer multiplication
Signed integers, 5
Truncated products, 5

Intervals
Colouring half-open intervals, 13
Colouring with demands, 14
Maximum simultaneous intervals, 14
Required interval, 13, 17
Stabbing, 13
Turnaround time, 13, 17

Inversions
Disagreement between rankings, 3
Significant inversions, 2
Weighted inversions, 2

K

Knapsack
Exact weight, 18
Two resource limits, 18
Unbounded multiplicities, 18

L

LCS. *See* Common subsequences

League completion
First place with ties, 29
Prescribed results, 29
Win caps, 29

LIS. *See* Longest increasing subsequences

Longest increasing subsequences
Bitonic subsequences, 19
Counting, 19
Fewest deletions, 19
Maximum weight, 19
Nesting rectangles, 20
Nondecreasing subsequences, 19

M

Matching. *See* Bipartite matching

Matrix operations
Boolean multiplication, 6
Maximum submatrix, 3
Powers, 5
Rectangular multiplication, 5

Maximum flow
Checking a claimed maximum, 26
Increasing one capacity, 25
Several sources and sinks, 25
Vertex capacities, 25

Maximum subarrays
Circular arrays, 3
Maximum submatrix, 3
Two disjoint subarrays, 3

Median and selection
Multiple ranks, 4
Repeated entries, 3
Weighted median, 4

Minimum cuts
Directed edge separator, 26
Internal vertex separator, 27
Two routes or a small separator, 28
Undirected edge connectivity, 27
Uniqueness, 26

MST. *See* Spanning trees

N

Negative cycles
Mean-weight threshold, 23
Reachable from a source, 23
Unbounded walk weights, 23

P

Palindromes, 20

Parity
Constraints, 9
Even-length walks, 7

Partitions. *See* Subset sum

Polynomial multiplication, 5

Precedence constraints
Travelling salesperson, 24

Prerequisites. *See* Project selection

Project selection
Prerequisites, 29
Required and forbidden projects, 30
Soft prerequisites, 30

R

Reachability
After edge deletions, 7
Even-length walks, 7
From a set, 7
Transitive closure, 23

Rectangles
Maximum-sum submatrix, 3
Nesting, 20

S

Scheduling
At most k weighted intervals, 17
Maximum lateness, 14
Required interval, 13, 17
Total waiting time, 14
Turnaround time, 13, 17
Weighted total completion time, 14

Search trees
Height bound, 22
Optimal alphabetic tree, 22
Unsuccessful queries, 22

Separators. *See* Minimum cuts

Shortest paths
Counting, 8
Directed, unweighted, 11
Distances to one destination, 23
Edge-limited walks, 22
Edges on shortest paths, 8
Multiple sources, 7
Nearest facility, 15
One free edge, 15
Vertex tolls, 15
Weighted DAGs, 22

Sorting and merging
Merging several arrays, 2
Multiset intersection, 2
Stable sorting, 2

Spanning trees
Edge insertion, 15
Maximum weight, 16
Minimum bottleneck, 16
Minimum spanning forest, 15
Required edges, 16

Strong connectivity
Component of one vertex, 11
Condensation graph, 12
Minimum starting set, 12
Sparse spanning subgraph, 11
Universal source vertices, 12
Witness, 11

Subset sum
Equal-sum partition, 17
Fewest items, 18
Most balanced partition, 18

Supersequences
Shortest common supersequence, 20

T

Threshold connectivity, 16

Transitive closure, 23

Travelling salesperson
Bottleneck objective, 24
Hamiltonian path, 24
Precedence constraints, 24

TSP. *See* Travelling salesperson

V

Vertex capacities
Flow networks, 25
Matching quotas, 28
Path-use limits, 27

Vertex covers
Minimum cardinality, 6
Required vertices, 29
Weighted bipartite graphs, 28
Weighted trees, 21